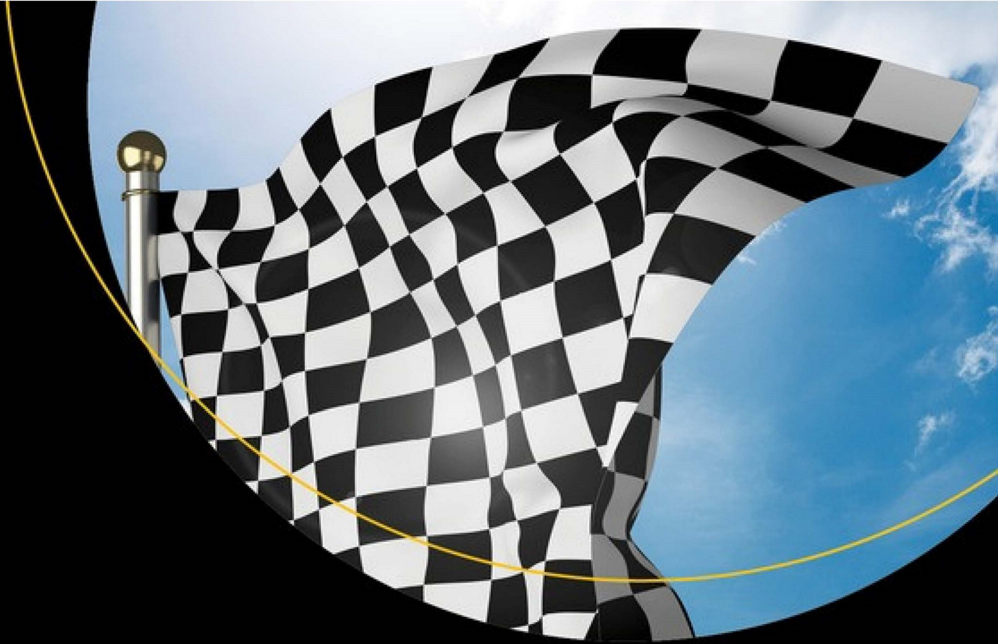

Pro Go

Полное руководство
по программированию надежного
и эффективного программного
обеспечения с использованием Golang

Адам Фриман

Apress®

Адам Фриман

Pro Go
Полное руководство по программированию надежного и эффективного программного обеспечения с использованием Golang

Apress®

ISBN 978-1-4842-7354-8 e-ISBN 978-1-4842-7355-5

Посвящается моей любимой жене Джеки Гриффит.
(А также Арахису.)

Любой исходный код или другие дополнительные материалы, на которые ссылается автор в этой книге, доступны читателям на GitHub. Для получения более подробной информации посетите сайт www.apress.com/source-code.

Оглавление

Часть I: Понимание языка Go

Глава 1: Ваше первое приложение Go

- Настройка сцены
- Установка средств разработки
  - Установка Git
  - Выбор редактора кода
- Создание проекта
- Определение типа данных и коллекции
- Создание HTML-шаблонов
  - Загрузка шаблонов
- Создание обработчиков HTTP и сервера
- Написание функции обработки формы
  - Обработка данных формы
- Добавление проверки данных
- Резюме

Глава 2: Go в контексте

- Почему вам стоит изучать Go?
- В чем подвох?
- Это действительно настолько плохо?
- Что вы должны знать?
- Какова структура этой книги?
  - Часть 1: Понимание языка Go
  - Часть 2: Использование стандартной библиотеки Go
  - Часть 3: Применение Go
- Что не охватывает эта книга?

Что делать, если вы нашли ошибку в книге?
Много ли примеров?
Какое программное обеспечение вам нужно для примеров?
На каких платформах будут работать примеры?
Что делать, если у вас возникли проблемы с примерами?
Где взять пример кода?
Почему некоторые примеры имеют странное форматирование?
Как связаться с автором?
Что, если мне действительно понравилась эта книга?
Что, если эта книга меня разозлила, и я хочу пожаловаться?
Резюме
Глава 3: Использование инструментов Go
Использование команды Go
Создание проекта Go
Понимание объявления пакета
Понимание оператора импорта
Понимание функции
Понимание оператора кода
Компиляция и запуск исходного кода
Очистка
Использование команды go run
Определение модуля
Отладка кода Go
Подготовка к отладке
Использование отладчика
Использование подключаемого модуля редактора Delve

Линтинг Go-кода
Использование линтера
Отключение правил линтера
Исправление распространенных проблем в коде Go
Форматирование кода Go
Резюме
Глава 4. Основные типы, значения и указатели
Подготовка к этой главе
Использование стандартной библиотеки Go
Понимание основных типов данных
Понимание литеральных значений
Использование констант
Понимание нетипизированных констант
Определение нескольких констант с помощью одного оператора
Пересмотр литеральных значений
Использование переменных
Пропуск типа данных переменной
Пропуск присвоения значения переменной
Определение нескольких переменных с помощью одного оператора
Использование краткого синтаксиса объявления переменных
Использование пустого идентификатора
Понимание указателей
Определение указателя
Следование указателю

Понимание нулевых значений указателя
Указывание на указатели
Понимание того, почему указатели полезны
Резюме
Глава 5: Операции и преобразования
Подготовка к этой главе
Понимание операторов Go
Понимание операторов Go
Объединение строк
Понимание операторов сравнения
Понимание логических операторов
Преобразование, анализ и форматирование значений
Выполнение явных преобразований типов
Преобразование значений с плавающей запятой в целые числа
Парсинг из строк
Форматирование значений как строк
Резюме
Глава 6: Управление потоком
Подготовка к этой главе
Понимание управления потоком выполнения
Использование ключевого слова else
Использование ключевого слова else
Понимание области действия оператора if
Использование оператора инициализации с оператором if
Использование циклов for

Включение условия в цикл

Использование операторов инициализации и завершения

Продолжение цикла

Перечисление последовательностей

Использование операторов switch

Сопоставление нескольких значений

Принудительный переход к следующему оператору case

Предоставление пункта по умолчанию

Использование оператора инициализации

Исключение значения сравнения

Использование операторов меток

Резюме

Глава 7: Использование массивов, срезов и карт

Подготовка к этой главе

Работа с массивами

Использование литерального синтаксиса массива

Понимание типов массивов

Понимание значений массива

Сравнение массивов

Перечисление массива

Работа со срезами

Добавление элементов в срез

Добавление одного среза к другому

Создание срезов из существующих массивов

Указание емкости при создании среза из массива

Создание срезов из других срезов

Использование функции копирования
Удаление элементов среза
Перечисление срезов
Сортировка срезов
Сравнение срезов
Получение массива, лежащего в основе среза
Работа с картами
Использование литерального синтаксиса карты
Проверка элементов в карте
Удаление объектов с карты
Перечисление содержимого карты
Понимание двойной природы строк
Преобразование строки в руны
Перечисление строк
Резюме
Глава 8: Определение и использование функций
Подготовка к этой главе
Определение простой функции
Определение и использование параметров функции
Пропуск типов параметров
Пропуск имен параметров
Определение и использование результатов функции
Использование указателей в качестве параметров функций
Определение и использование результатов функции
Возврат функцией нескольких результатов
Использование ключевого слова defer

Резюме

Глава 9: Использование типов функций

Подготовка к этой главе

Понимание типов функций

Понимание сравнения функций и нулевого типа

Использование функций в качестве аргументов

Использование функций в качестве результатов

Создание псевдонимов функциональных типов

Использование литерального синтаксиса функции

Понимание области действия функциональной переменной

Непосредственное использование значений функций

Понимание замыкания функции

Резюме

Глава 10: Определение структур

Подготовка к этой главе

Определение и использование структуры

Создание структурных значений

Использование значения структуры

Частичное присвоение значений структуры

Использование позиций полей для создания значений структуры

Определение встроенных полей

Сравнение значений структуры

Определение анонимных типов структур

Создание массивов, срезов и карт, содержащих структурные значения

Понимание структур и указателей

Понимание удобного синтаксиса указателя структуры

Понимание указателей на значения

Понимание функций конструктора структуры

Использование типов указателей для полей структуры

Понимание нулевого значения для структур и указателей на структуры

Резюме

Глава 11: Использование методов и интерфейсов

Подготовка к этой главе

Определение и использование методов

Определение параметров метода и результатов

Понимание перегрузки метода

Понимание получателей указателей и значений

Определение методов для псевдонимов типов

Размещение типов и методов в отдельных файлах

Определение и использование интерфейсов

Определение интерфейса

Реализация интерфейса

Использование интерфейса

Понимание эффекта приемников метода указателя

Сравнение значений интерфейса

Выполнение утверждений типа

Тестирование перед выполнением утверждения типа

Включение динамических типов

Использование пустого интерфейса

Использование пустого интерфейса для параметров функций

Резюме
Глава 12: Создание и использование пакетов
Подготовка к этой главе
Понимание файла модуля
Создание пользовательского пакета
Использование пользовательского пакета
Понимание управления доступом к пакетам
Добавление файлов кода в пакеты
Разрешение конфликтов имен пакетов
Создание вложенных пакетов
Использование функций инициализации пакета
Использование внешних пакетов
Управление внешними пакетами
Резюме
Глава 13: Тип и состав интерфейса
Подготовка к этой главе
Понимание композиции типов
Определение базового типа
Типы композиций
Создание цепочки вложенных типов
Использование нескольких вложенных типов в одной и той же структуре
Понимание, когда продвижение не может быть выполнено
Понимание композиции и интерфейсов
Использование композиции для реализации интерфейсов
Составление интерфейсов

Резюме
Глава 14. Использование горутин и каналов
Подготовка к этой главе
Понимание того, как Go выполняет код
Создание дополнительных горутин
Возврат результатов из горутин
Отправка результата с использованием канала
Получение результата с использованием канала
Работа с каналами
Координация каналов
Отправка и получение неизвестного количества значений
Ограничение направления канала
Использование операторов select
Получение без блокировки
Прием с нескольких каналов
Отправка без блокировки
Отправка на несколько каналов
Резюме
Глава 15. Обработка ошибок
Подготовка к этой главе
Работа с исправимыми ошибками
Генерация ошибок
Сообщение об ошибках через каналы
Использование удобных функций обработки ошибок
Работа с неисправимыми ошибками
Восстановление после паники

Паника после восстановления

Восстановление после паники в горутинах

Резюме

Часть II: Использование стандартной библиотеки Go

Глава 16. Обработка строк и регулярные выражения

Подготовка к этой главе

Обработка строк

Сравнение строк

Преобразование регистра строк

Работа с регистром символов

Проверка строк

Манипулирование строками

Обрезка строк

Изменение строк

Построение и генерация строк

Использование регулярных выражений

Компиляция и повторное использование шаблонов

Разделение строк с помощью регулярного выражения

Использование подвыражений

Замена подстрок с помощью регулярного выражения

Резюме

Глава 17: Форматирование и сканирование строк

Подготовка к этой главе

Написание строк

Форматирование строк

Понимание глаголов форматирования

Использование глаголов форматирования общего назначения

Использование команд целочисленного форматирования

Использование глаголов форматирования значений с плавающей запятой

Использование глаголов форматирования строк и символов

Использование глагола форматирования логических значений

Использование глагола форматирования указателя

Сканирование строк

Работа с символами новой строки

Использование другого источника строк

Использование шаблона сканирования

Резюме

Глава 18: Математические функции и сортировка данных

Подготовка к этой главе

Работа с числами

Генерация случайных чисел

Сортировка данных

Сортировка числовых и строковых срезов

Поиск отсортированных данных

Сортировка пользовательских типов данных

Резюме

Глава 19: Даты, время и продолжительность

Подготовка к этой главе

Представление дат и времени

Представление дат и времени

Представление продолжительности
Использование функций времени для горутин и каналов
Перевод горутины в сон
Отсрочка выполнения функции
Получение уведомлений по времени
Получение повторяющихся уведомлений
Резюме
Глава 20:Чтение и запись данных
Подготовка к этой главе
Понимание средств чтения и записи
Понимание средств чтения
Понимание средств записи
Использование служебных функций для программ чтения и записи
Использование специализированных средств чтения и записи
Использование пайпов
Объединение нескольких средств чтения
Объединение нескольких средств записи
Повторение данных чтения во Writer
Ограничение чтения данных
Буферизация данных
Использование дополнительных методов буферизованного чтения
Выполнение буферизованной записи
Форматирование и сканирование с помощью средств чтения и записи
Сканирование значений из считывателя

Запись отформатированных строк в Writer
Использование Replacer с Writer
Резюме
Глава 21: Работа с данными JSON
Подготовка к этой главе
Чтение и запись данных JSON
Кодирование данных JSON
Декодирование данных JSON
Резюме
Глава 22: Работа с файлами
Подготовка к этой главе
Чтение файлов
Использование функции удобства чтения
Использование файловой структуры для чтения файла
Запись в файлы
Использование функции удобства записи
Использование файловой структуры для записи в файл
Запись данных JSON в файл
Использование удобных функций для создания новых файлов
Работа с путями к файлам
Управление файлами и каталогами
Изучение файловой системы
Определение существования файла
Поиск файлов с помощью шаблона
Обработка всех файлов в каталоге
Резюме

Глава 23: Использование HTML и текстовых шаблонов
Подготовка к этой главе
Создание HTML-шаблонов
Загрузка и выполнение шаблонов
Понимание действий шаблона
Создание текстовых шаблонов
Резюме
Глава 24: Создание HTTP-серверов
Подготовка к этой главе
Создание простого HTTP-сервера
Создание прослушивателя и обработчика HTTP
Проверка запроса
Фильтрация запросов и генерация ответов
Использование удобных функций ответа
Использование обработчика удобной маршрутизации
Поддержка HTTPS-запросов
Создание статического HTTP-сервера
Создание статического маршрута к файлу
Использование шаблонов для генерации ответов
Ответ с данными JSON
Обработка данных формы
Чтение данных формы из запросов
Чтение составных форм
Чтение и настройка файлов cookie
Резюме
Глава 25: Создание HTTP-клиентов

Подготовка к этой главе
Отправка простых HTTP-запросов
Отправка POST-запросов
Настройка запросов HTTP-клиента
Использование удобных функций для создания запроса
Работа с файлами cookie
Управление перенаправлениями
Создание составных форм
Резюме
Глава 26: Работа с базами данных
Подготовка к этой главе
Подготовка базы данных
Установка драйвера базы данных
Открытие базы данных
Выполнение операторов и запросов
Запрос нескольких строк
Выполнение операторов с заполнителями
Выполнение запросов для отдельных строк
Выполнение других запросов
Использование подготовленных операторов
Использование транзакций
Использование рефлексии для сканирования данных в структуру
Резюме
Глава 27: Использование рефлексии
Подготовка к этой главе

- Понимание необходимости рефлексии
- Использование рефлексии
    - Использование основных функций типа
    - Использование базовых возможностей Value
- Определение типов
    - Идентификация байтовых срезов
- Получение базовых значений
- Установка Value с использованием рефлексии
    - Установка одного Value с помощью другого
- Сравнение Value
    - Использование удобной функции сравнения
- Преобразование значений
    - Преобразование числовых типов
- Создание новых значений
- Резюме

Глава 28: Использование рефлексии, часть 2

- Подготовка к этой главе
- Работа с указателями
    - Работа со значениями указателя
- Работа с типами массивов и срезов
- Работа со значениями массива и среза
    - Перечисление срезов и массивов
    - Создание новых срезов из существующих срезов
    - Создание, копирование и добавление элементов в срезы
- Работа с типами карт
- Работа со значениями карты

Установка и удаление значений карты
Создание новых карт
Работа с типами структур
Обработка вложенных полей
Поиск поля по имени
Проверка тегов структуры
Создание типов структур
Работа со структурными значениями
Установка значений поля структуры
Резюме
Глава 29: Использование рефлексии, часть 3
Подготовка к этой главе
Работа с типами функций
Работа со значениями функций
Создание и вызов новых типов функций и значений
Работа с методами
Вызов методов
Работы с интерфейсами
Получение базовых значений из интерфейсов
Изучение методов интерфейса
Работа с типами каналов
Работа со значениями канала
Создание новых типов и значений каналов
Выбор из нескольких каналов
Резюме
Глава 30: Координация горутин

Подготовка к этой главе
Использование групп ожидания
Использование взаимного исключения
Использование мьютекса чтения-записи
Использование условий для координации горутин
Обеспечение однократного выполнения функции
Использование контекстов
Отмена запроса
Установка крайнего срока
Предоставление данных запроса
Резюме
Глава 31. Модульное тестирование, бенчмаркинг и логирование
Подготовка к этой главе
Использование тестирования
Запуск модульных тестов
Управление выполнением теста
Код бенчмаркинга
Удаление установки из теста
Выполнение суббенчмаркингов
Журналирование ланных
Создание пользовательских регистраторов
Резюме
Часть III: Применение Go
Глава 32: Создание веб-платформы
Создание проекта
Создание некоторых основных функций платформы

Создание системы ведения журнала
Создание системы конфигурации
Управление службами с внедрением зависимостей
Определение жизненных циклов сервиса
Определение внутренних сервисных функций
Определение функций регистрации службы
Определение функций разрешения службы
Регистрация и использование сервисов
Резюме
Глава 33. ПО промежуточного слоя, шаблоны и обработчики
Создание конвейера запросов
Определение интерфейса компонента промежуточного программного обеспечения
Создание конвейера запросов
Создание базовых компонентов
Создание HTTP-сервера
Настройка приложения
Оптимизация разрешения сервиса
Создание HTML-ответов
Создание макета и шаблона
Реализация выполнения шаблона
Создание и использование службы шаблонов
Знакомство с обработчиками запросов
Генерация URL-маршрутов
Подготовка значений параметров для метода обработчика
Сопоставление запросов с маршрутами

Резюме
Глава 34: Действия, сеансы и авторизация
Представляем результаты действий
Определение общих результатов действий
Обновление заполнителей для использования результатов действий
Вызов обработчиков запросов из шаблонов
Обновление обработки запросов
Настройка приложения
Создание URL-адресов из маршрутов
Создание службы генератора URL
Определение альтернативных маршрутов
Проверка данных запроса
Выполнение проверки данных
Добавление сеансов
Отсрочка записи данных ответа
Создание интерфейса сеанса, службы и промежуточного программного обеспечения
Создание обработчика, использующего сеансы
Настройка приложения
Добавление авторизации пользователя
Определение основных типов авторизации
Реализация интерфейсов платформы
Реализация контроля доступа
Реализация функций заполнителя приложения
Создание обработчика аутентификации
Настройка приложения

- Резюме

Глава 35: SportsStore: настоящее приложение

- Создание проекта SportsStore
  - Настройка приложения
- Запуск модели данных
  - Отображение списка продуктов
  - Реализация (временного) репозитория
- Отображение списка продуктов
  - Создание шаблона и макета
  - Настройка приложения
- Добавление пагинации
- Стилизация содержимого шаблона
  - Установка CSS-файла Bootstrap
  - Обновление макета
  - Стилизация содержимого шаблона
- Добавление поддержки фильтрации категорий
  - Обновление обработчика запросов
  - Создание обработчика категории
  - Отображение навигации по категориям в шаблоне списка товаров
  - Регистрация обработчика и обновление псевдонимов
- Резюме

Глава 36: SportsStore: корзина и база данных

- Создание корзины покупок
  - Определение модели корзины и репозитория
  - Создание обработчика запроса корзины

Добавление товаров в корзину
Настройка приложения
Добавление виджета «Сводка корзины»
Использование репозитория базы данных
Создание типов репозиториев
Открытие базы данных и загрузка команд SQL
Определение начального числа и операторов инициализации
Определение основных запросов
Определение постраничных запросов
Определение службы репозитория SQL
Настройка приложения для использования репозитория SQL
Резюме
Глава 37: SportsStore: оформление заказа и администрирование
Создание процесса оформления заказа
Определение модели
Расширение репозитория
Отключение временного репозитория
Определение методов и команд репозитория
Создание обработчика запросов и шаблонов
Создание функций администрирования
Создание функции администрирования продукта
Создание функции администрирования категорий
Резюме
Глава 38: SportsStore: отделка и развертывание
Завершение функций администрирования

Расширение репозитория

Реализация обработчиков запросов

Создание шаблонов

Ограничение доступа к функциям администрирования

Создание пользовательского хранилища и обработчика запросов

Настройка приложения

Создание веб-службы

Подготовка к развертыванию

Установка сертификатов

Настройка приложения

Сборка приложения

Установка рабочего стола Docker

Creating the Docker Configuration Files

Резюме

Об авторе

Адам Фриман

Опытный ИТ-специалист, который занимал руководящие должности в ряде компаний, в последнее время — технический директор и главный операционный директор глобального банка. Теперь на пенсии, он тратит свое время на написание книг и бег на длинные дистанции.

О техническом рецензенте

Фабио Клаудио Ферраккиати

Является старшим консультантом и старшим аналитиком/ разработчиком, использующим технологии Microsoft. Он работает на BluArancio (www.bluarancio.com). Он является сертифицированным разработчиком решений Microsoft для .NET, сертифицированным разработчиком приложений Microsoft для .NET, сертифицированным специалистом Microsoft, а также плодовитым автором и техническим обозревателем. За последние десять лет он написал статьи для итальянских и международных журналов и стал соавтором более десяти книг по различным компьютерным темам.

Часть I
Понимание языка Go

1. Ваше первое приложение Go

Лучший способ начать работу с Go — сразу приступить к делу. В этой главе я объясню, как подготовить среду разработки Go, а также создать и запустить простое веб-приложение. Цель этой главы — получить представление о том, на что похоже написание на Go, поэтому не беспокойтесь, если вы не понимаете всех используемых функций языка. Все, что вам нужно знать, подробно объясняется в последующих главах.

Настройка сцены

Представьте, что подруга решила устроить вечеринку в канун Нового года и попросила меня создать веб-приложение, которое позволяет ее приглашенным в электронном виде отвечать на вопросы. Она попросила эти ключевые функции:

- Домашняя страница с информацией о вечеринке
- Форма, которую можно использовать для RSVP, которая будет отображать страницу благодарности
- Проверка заполнения формы RSVP
- Сводная страница, которая показывает, кто придет на вечеринку

В этой главе я создаю проект Go и использую его для создания простого приложения, которое содержит все эти функции.

> **Подсказка**
> Вы можете загрузить пример проекта для этой главы — и для всех остальных глав этой книги — с `https://github.com/apress/pro-go`. См. Главу 2 о том, как получить помощь, если у вас возникнут проблемы с запуском примеров.

Установка средств разработки

Первым шагом является установка инструментов разработки Go. Перейдите на `https://golang.org/dl` и загрузите установочный файл для вашей операционной системы. Установщики доступны для Windows, Linux и macOS. Следуйте инструкциям по установке, которые можно найти по адресу `https://golang.org/doc/install` для вашей платформы. Когда вы завершите установку, откройте командную строку и выполните команду, показанную в листинге 1-1, которая подтвердит, что инструменты Go были установлены, распечатав версию пакета.

> **ОБНОВЛЕНИЯ ЭТОЙ КНИГИ**
> Go активно разрабатывается, и существует постоянный поток новых выпусков, а это значит, что к тому времени, когда вы будете читать эту книгу, может быть доступна более поздняя версия. Go имеет прекрасную политику поддержки совместимости, поэтому у вас не должно возникнуть проблем с примерами из этой книги, даже в более поздних версиях. Если у вас возникнут проблемы, см. репозиторий этой книги на GitHub, `https://github.com/apress/pro-go`, где я буду публиковать бесплатные обновления, устраняющие критические изменения.
>
> Для меня (и для Apress) обновление такого рода является продолжающимся экспериментом, и оно продолжает развиваться — не в последнюю очередь потому, что я не знаю, что будет

содержать будущие версии Go. Цель состоит в том, чтобы продлить жизнь этой книги, дополнив содержащиеся в ней примеры.

Я не даю никаких обещаний относительно того, какими будут обновления, какую форму они примут или как долго я буду их выпускать, прежде чем включить их в новое издание этой книги. Пожалуйста, будьте непредвзяты и проверяйте репозиторий этой книги при выпуске новых версий. Если у вас есть идеи о том, как можно улучшить обновления, напишите мне по адресу adam@adam-freeman.com и дайте мне знать.

```
go version
```

Листинг 1-1 Проверка установки Go

Текущая версия на момент написания статьи — 1.17.1, что приводит к следующему выводу на моем компьютере с Windows:

```
go version go1.17.1 windows/amd64
```

Неважно, видите ли вы другой номер версии или другую информацию об операционной системе — важно то, что команда `go` работает и выдает результат.

Установка Git

Некоторые команды Go полагаются на систему контроля версий Git. Перейдите на `https://git-scm.com` и следуйте инструкциям по установке для вашей операционной системы.

Выбор редактора кода

Единственный другой шаг — выбрать редактор кода. Файлы исходного кода Go представляют собой обычный текст, что означает, что вы можете использовать практически любой редактор. Однако некоторые редакторы предоставляют специальную поддержку для Go. Наиболее популярным выбором является Visual Studio Code, который можно использовать бесплатно и который поддерживает новейшие функции языка Go. Visual Studio Code — это редактор, который я рекомендую, если у вас еще нет предпочтений. Visual Studio Code можно загрузить с `http://code.visualstudio.com`, и существуют установщики для всех популярных операционных систем. Вам будет предложено установить расширения Visual Studio Code для Go, когда вы начнете работу над проектом в следующем разделе.

Если вам не нравится код Visual Studio, вы можете найти список доступных опций по адресу `https://github.com/golang/go/wiki/IDEsAndTextEditorPlugins`. Для выполнения примеров из этой книги не требуется специального редактора кода, и все задачи, необходимые для создания и компиляции проектов, выполняются в командной строке.

Создание проекта

Откройте командную строку, перейдите в удобное место и создайте папку с именем `partyinvites`. Перейдите в папку `partyinvites` и выполните команду, показанную в листинге 1-2, чтобы запустить новый проект Go.

```
go mod init partyinvites
```

Листинг 1-2 Запуск проекта Go

Команда `go` используется почти для каждой задачи разработки, как я объясню в Главе 3. Эта команда создает файл с именем `go.mod`, который используется для отслеживания пакетов, от которых зависит проект, а также может использоваться для публикации проекта, если необходимо.

Файлы кода Go имеют расширение .go. Используйте выбранный вами редактор для создания файла с именем main.go в папке partyinvites с содержимым, показанным в листинге 1-3. Если вы используете Visual Studio Code и впервые редактируете файл Go, вам будет предложено установить расширения, поддерживающие язык Go.

```
package main

import "fmt"

func main() {
    fmt.Println("TODO: add some features")
}
```

Листинг 1-3 Содержимое файла main.go в папке partyinvites

Синтаксис Go будет вам знаком, если вы использовали любой C или C-подобный язык, например C# или Java. В этой книге я подробно описываю язык Go, но вы можете многое понять, просто взглянув на ключевые слова и структуру кода в листинге 1-3.

Функции сгруппированы в *пакеты* (package), поэтому в листинге 1-3 есть оператор пакета. Зависимости пакетов создаются с помощью оператора импорта, который позволяет получить доступ к функциям, которые они используют, в файле кода. Операторы сгруппированы в функции, которые определяются с помощью ключевого слова func. В листинге 1-3 есть одна функция, которая называется main. Это *точка входа* для приложения, что означает, что это точка, с которой начнется выполнение, когда приложение будет скомпилировано и запущено.

Функция main содержит один оператор кода, который вызывает функцию с именем Println, предоставляемую пакетом с именем fmt. Пакет fmt является частью обширной стандартной библиотеки Go, описанной во второй части этой книги. Функция Println выводит строку символов.

Хотя детали могут быть незнакомы, назначение кода в листинге 1-3 легко понять: когда приложение выполняется, оно выводит простое сообщение. Запустите команду, показанную в листинге 1-4, в папке partyinvites, чтобы скомпилировать и выполнить проект. (Обратите внимание, что в этой команде после слова run стоит точка.)

```
go run .
```

Листинг 1-4 Компиляция и выполнение проекта

Команда go run полезна во время разработки, поскольку выполняет задачи компиляции и выполнения за один шаг. Приложение выдает следующий вывод:

```
TODO: add some features
```

Если вы получили ошибку компилятора, вероятно, причина в том, что вы не ввели код точно так, как показано в листинге 1-3. Go настаивает на том, чтобы код определялся определенным образом. Вы можете предпочесть, чтобы открывающие фигурные скобки отображались на отдельной строке, и вы могли автоматически отформатировать код таким образом, как показано в листинге 1-5.

```
package main

import "fmt"

func main()
{
    fmt.Println("TODO: add some features")
}
```

Листинг 1-5 Ставим фигурную скобку на новую строку в файле main.go в папке partyinvites

Запустите команду, показанную в листинге 1-4, для компиляции проекта, и вы получите следующие ошибки:

```
# partyinvites
.\main.go:5:6: missing function body
.\main.go:6:1: syntax error: unexpected semicolon or newline before {
```

Go настаивает на определенном стиле кода и необычным образом обрабатывает распространенные элементы кода, такие как точки с запятой. Подробности синтаксиса Go описаны в следующих главах, но сейчас важно точно следовать приведенным примерам, чтобы избежать ошибок.

Определение типа данных и коллекции

Следующим шагом является создание пользовательского типа данных, который будет представлять ответы RSVP, как показано в листинге 1-6.

```
package main

import "fmt"

type Rsvp struct {
    Name, Email, Phone string
    WillAttend bool
}

func main() {
    fmt.Println("TODO: add some features");
}
```

Листинг 1-6 Определение типа данных в файле main.go в папке partyinvites

Go позволяет определять пользовательские типы и присваивать им имена с помощью ключевого слова `type`. В листинге 1-6 создается тип данных `struct` с именем `Rsvp`. Структуры позволяют группировать набор связанных значений. Структура `Rsvp` определяет четыре поля, каждое из которых имеет имя и тип данных. Типы данных, используемые полями `Rsvp`, — `string` и `bool`, которые являются встроенными типами для представления строки символов и логических значений. (Встроенные типы Go описаны в главе 4.)

Далее мне нужно собрать вместе значения `Rsvp`. В последующих главах я объясню, как использовать базу данных в приложении Go, но для этой главы будет достаточно хранить ответы в памяти, что означает, что ответы будут потеряны при остановке приложения.

Go имеет встроенную поддержку массивов фиксированной длины, массивов переменной длины (известных как *срезы*) и *карт* (словарей), содержащих пары ключ-значение. В листинге 1-7 создается срез, что является хорошим выбором, когда количество сохраняемых значений заранее неизвестно.

```
package main

import "fmt"

type Rsvp struct {
    Name, Email, Phone string
    WillAttend bool
}
```

```
var responses = make([]*Rsvp, 0, 10)

func main() {
    fmt.Println("TODO: add some features");
}
```

Листинг 1-7 Определение среза в файле main.go в папке partyinvites

Этот новый оператор основан на нескольких функциях Go, которые проще всего понять, если начать с конца оператора и прорабатывать в обратном направлении.

Go предоставляет встроенные функции для выполнения общих операций с массивами, срезами и картами. Одной из таких функций является `make`, которая используется в листинге 1-7 для инициализации нового среза. Последние два аргумента функции `make` — это начальный размер и начальная емкость.

```
...
var responses = make([]*Rsvp, 0, 10)
...
```

Я указал ноль для аргумента размера, чтобы создать пустой срез. Размеры срезов изменяются автоматически по мере добавления новых элементов, а начальная емкость определяет, сколько элементов можно добавить, прежде чем размер среза нужно будет изменить. В этом случае к срезу можно добавить десять элементов, прежде чем его размер нужно будет изменить.

Первый аргумент метода `make` указывает тип данных, для хранения которого будет использоваться срез:

```
...
var responses = make([]*Rsvp, 0, 10)
...
```

Квадратные скобки `[]` обозначают срез. Звездочка `*` обозначает указатель. Часть типа `Rsvp` обозначает тип структуры, определенный в листинге 1-6. В совокупности `[]*Rsvp` обозначает срез указателей на экземпляры структуры `Rsvp`.

Вы, возможно, вздрогнули от термина *указатель*, если вы пришли к Go из C# или Java, которые не позволяют использовать указатели напрямую. Но вы можете расслабиться, потому что Go не допускает операций над указателями, которые могут создать проблемы для разработчика. Как я объясню в главе 4, использование указателей в Go определяет только то, копируется ли значение при его использовании. Указав, что мой срез будет содержать указатели, я говорю Go не создавать копии моих значений `Rsvp`, когда я добавляю их в срез.

Остальная часть оператора присваивает инициализированный срез переменной, чтобы я мог использовать его в другом месте кода:

```
...
var responses = make([]*Rsvp, 0, 10)
...
```

Ключевое слово `var` указывает, что я определяю новую переменную, которой присваивается имя `responses`. Знак равенства, `=`, является оператором присваивания Go и устанавливает значение переменной `responses` для вновь созданного среза. Мне не нужно указывать тип переменной responses, потому что компилятор Go выведет его из присвоенного ей значения.

Создание HTML-шаблонов

Go поставляется с обширной стандартной библиотекой, которая включает поддержку HTML-шаблонов. Добавьте файл с именем `layout.html` в папку `partyinvites` с содержимым,

показанным в листинге 1-8.

```
<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>Let's Party!</title>
    <link href=
       "https://cdnjs.cloudflare.com/ajax/libs/bootstrap/5.1.1/css/bootstrap.min.css"
            rel="stylesheet">
</head>
<body class="p-2">
    {{ block "body" . }} Content Goes Here {{ end }}
</body>
</html>
```

Листинг 1-8 Содержимое файла layout.html в папке partyinvites

Этот шаблон будет макетом, содержащим содержимое, общее для всех ответов, которые будет создавать приложение. Он определяет базовый HTML-документ, включая элемент `link` (ссылки), указывающий таблицу стилей из CSS-фреймворка Bootstrap, которая будет загружаться из сети распространения контента (CDN). Я продемонстрирую, как обслуживать этот файл из папки в главе 24, но для простоты в этой главе я использовал CDN. Пример приложения по-прежнему будет работать в автономном режиме, но вы увидите элементы HTML без стилей, показанных на рисунках.

Двойные фигурные скобки в листинге 1-8, `{{` и `}}`, используются для вставки динамического содержимого в выходные данные, созданные шаблоном. Используемое здесь выражение `block` (блок) определяет содержимое заполнителя, которое будет заменено другим шаблоном во время выполнения.

Чтобы создать содержимое, которое будет приветствовать пользователя, добавьте файл с именем `welcome.html` в папку `partyinvites` с содержимым, показанным в листинге 1-9.

```
{{ define "body"}}

    <div class="text-center">
        <h3> We're going to have an exciting party!</h3>
        <h4>And YOU are invited!</h4>
        <a class="btn btn-primary" href="/form">
            RSVP Now
        </a>
    </div>

{{ end }}
```

Листинг 1-9 Содержимое файла welcome.html в папке partyinvites

Чтобы создать шаблон, который позволит пользователю дать свой ответ на RSVP, добавьте файл с именем `form.html` в папку partyinvites с содержимым, показанным в листинге 1-10.

```
{{ define "body"}}

<div class="h5 bg-primary text-white text-center m-2 p-2">RSVP</div>

{{ if gt (len .Errors) 0}}
    <ul class="text-danger mt-3">
        {{ range .Errors }}
            <li>{{ .  }}</li>
        {{ end }}
```

```
    </ul>
{{ end }}

<form method="POST" class="m-2">
    <div class="form-group my-1">
        <label>Your name:</label>
        <input name="name" class="form-control" value="{{.Name}}" />
    </div>
    <div class="form-group my-1">
        <label>Your email:</label>
        <input name="email" class="form-control" value="{{.Email}}" />
    </div>
    <div class="form-group my-1">
        <label>Your phone number:</label>
        <input name="phone" class="form-control" value="{{.Phone}}" />
    </div>
    <div class="form-group my-1">
        <label>Will you attend?</label>
        <select name="willattend" class="form-select">
            <option value="true" {{if .WillAttend}}selected{{end}}>
                Yes, I'll be there
            </option>
            <option value="false" {{if not .WillAttend}}selected{{end}}>
                No, I can't come
            </option>
        </select>
    </div>
    <button class="btn btn-primary mt-3" type="submit">
        Submit RSVP
    </button>
</form>

{{ end }}
```

Листинг 1-10 Содержимое файла form.html в папке partyinvites

Чтобы создать шаблон, который будет представлен посетителям, добавьте файл с именем `thanks.html` в папку `partyinvites` с содержимым, показанным в листинге 1-11.

```
{{ define "body"}}

<div class="text-center">
    <h1>Thank you, {{ . }}!</h1>
    <div> It's great that you're coming. The drinks are already in the fridge!</div>
    <div>Click <a href="/list">here</a> to see who else is coming.</div>
</div>

{{ end }}
```

Листинг 1-11 Содержимое файла thanks.html в папке partyinvites

Чтобы создать шаблон, который будет отображаться при отклонении приглашения, добавьте файл с именем `sorry.html` в папку `partyinvites` с содержимым, показанным в листинге 1-12.

```
{{ define "body"}}

<div class="text-center">
    <h1>It won't be the same without you, {{ . }}!</h1>
    <div>Sorry to hear that you can't make it, but thanks for letting us know.</div>
    <div>
```

```
        Click <a href="/list">here</a> to see who is coming,
        just in case you change your mind.
    </div>
</div>

{{ end }}
```

Листинг 1-12 Содержимое файла sorry.html в папке partyinvites

Чтобы создать шаблон, отображающий список участников, добавьте файл с именем `list.html` в папку `partyinvites` с содержимым, показанным в листинге 1-13.

```
{{ define "body"}}

<div class="text-center p-2">
    <h2>Here is the list of people attending the party</h2>
    <table class="table table-bordered table-striped table-sm">
        <thead>
            <tr><th>Name</th><th>Email</th><th>Phone</th></tr>
        </thead>
        <tbody>
            {{ range . }}
                {{ if .WillAttend }}
                    <tr>
                        <td>{{ .Name }}</td>
                        <td>{{ .Email }}</td>
                        <td>{{ .Phone }}</td>
                    </tr>
                {{ end }}
            {{ end }}
        </tbody>
    </table>
</div>

{{ end }}
```

Листинг 1-13 Содержимое файла list.html в папке partyinvites

Загрузка шаблонов

Следующим шагом является загрузка шаблонов, чтобы их можно было использовать для создания контента, как показано в листинге 1-14. Я собираюсь написать код, чтобы сделать это поэтапно, объясняя, что делает каждое изменение по ходу дела. (Вы можете увидеть подсветку ошибок в редакторе кода, но это будет устранено, когда я добавлю новые операторы кода в более поздние списки.)

```
package main

import (
    "fmt"
    "html/template"
)

type Rsvp struct {
    Name, Email, Phone string
    WillAttend bool
}

var responses = make([]*Rsvp, 0, 10)
var templates = make(map[string]*template.Template, 3)
```

```
func loadTemplates() {
    // TODO - load templates here
}

func main() {
    loadTemplates()
}
```

Листинг 1-14 Загрузка шаблонов из файла main.go в папку partyinvites

Первое изменение относится к оператору *импорта* `import` и объявляет зависимость от функций, предоставляемых пакетом `html/template`, который является частью стандартной библиотеки Go. Этот пакет поддерживает загрузку и отображение HTML-шаблонов и подробно описан в главе 23.

Следующий новый оператор создает переменную с именем `templates`. Тип значения, присваиваемого этой переменной, выглядит сложнее, чем есть на самом деле:

```
...
var templates = make(map[string]*template.Template, 3)
...
```

Ключевое слово `map` обозначает карту, тип ключа которой указывается в квадратных скобках, за которым следует тип значения. Тип ключа для этой карты — `string`, а тип значения — `*template.Template`, что означает указатель на структуру `Template`, определенную в пакете шаблона. Когда вы импортируете пакет, для доступа к его функциям используется последняя часть имени пакета. В этом случае доступ к функциям, предоставляемым пакетом `html/template`, осуществляется с помощью шаблона, и одной из этих функций является структура с именем `Template`. Звездочка указывает на указатель, что означает, что карта использует `string` ключи, используемые для хранения указателей на экземпляры структуры `Template`, определенной пакетом `html/template`.

Затем я создал новую функцию с именем `loadTemplates`, которая пока ничего не делает, но будет отвечать за загрузку файлов HTML, определенных в предыдущих листингах, и их обработку для создания значений `*template.Template`, которые будут храниться на карте. Эта функция вызывается внутри функции `main`. Вы можете определять и инициализировать переменные непосредственно в файлах кода, но самые полезные функции языка можно реализовать только внутри функций.

Теперь мне нужно реализовать функцию `loadTemplates`. Каждый шаблон загружается с макетом, как показано в листинге 1-15, что означает, что мне не нужно повторять базовую структуру HTML-документа в каждом файле.

```
package main

import (
    "fmt"
    "html/template"
)

type Rsvp struct {
    Name, Email, Phone string
    WillAttend bool
}

var responses = make([]*Rsvp, 0, 10)

var templates = make(map[string]*template.Template, 3)
```

```
func loadTemplates() {
    templateNames := [5]string { "welcome", "form", "thanks", "sorry", "list" }
    for index, name := range templateNames {
        t, err := template.ParseFiles("layout.html", name + ".html")
        if (err == nil) {
            templates[name] = t
            fmt.Println("Loaded template", index, name)
        } else {
            panic(err)
        }
    }
}

func main() {
    loadTemplates()
}
```

Листинг 1-15 Загрузка шаблонов из файла main.go в папку partyinvites

Первый оператор в теле `loadTemplates` определяет переменные, используя краткий синтаксис Go, который можно использовать только внутри функций. Этот синтаксис определяет имя, за которым следует двоеточие (`:`), оператор присваивания (`=`) и затем значение:

```
...
templateNames := [5]string { "welcome", "form", "thanks", "sorry", "list" }
...
```

Этот оператор создает переменную с именем `templateNames`, и ее значение представляет собой массив из пяти строковых значений, которые выражены с использованием литеральных значений. Эти имена соответствуют именам файлов, определенных ранее. Массивы в Go имеют фиксированную длину, и массив, присвоенный переменной `templateNames`, может содержать только пять значений.

Эти пять значений перечисляются в цикле `for` с использованием ключевого слова `range`, например:

```
...
for index, name := range templateNames {
...
```

Ключевое слово `range` используется с ключевым словом `for` для перечисления массивов, срезов и карт. Операторы внутри цикла `for` выполняются один раз для каждого значения в источнике данных, которым в данном случае является массив, и этим операторам присваиваются два значения для работы:

```
...
for index, name := range templateNames {
...
```

Переменной `index` присваивается позиция значения в массиве, который в настоящее время перечисляется. Переменной `name` присваивается значение в текущей позиции. Тип первой переменной всегда `int`, это встроенный тип данных Go для представления целых чисел. Тип другой переменной соответствует значениям, хранящимся в источнике данных. Перечисляемый в этом цикле массив содержит строковые значения, что означает, что переменной `name` будет присвоена строка в позиции в массиве, указанной значением индекса.

Первый оператор в цикле `for` загружает шаблон:

```
...
t, err := template.ParseFiles("layout.html", name + ".html")
...
```

Пакет html/templates предоставляет функцию ParseFiles, которая используется для загрузки и обработки HTML-файлов. Одной из самых полезных и необычных возможностей Go является то, что функции могут возвращать несколько результирующих значений. Функция ParseFiles возвращает два результата: указатель на значение template.Template и *ошибку*, которая является встроенным типом данных для представления ошибок в Go. Краткий синтаксис для создания переменных используется для присвоения этих двух результатов переменным, например:

```
...
t, err := template.ParseFiles("layout.html", name + ".html")
...
```

Мне не нужно указывать типы переменных, которым присваиваются результаты, потому что они уже известны компилятору Go. Шаблон присваивается переменной с именем t, а ошибка присваивается переменной с именем err. Это распространенный шаблон в Go, и он позволяет мне определить, был ли загружен шаблон, проверив, равно ли значение err nil, что является нулевым значением Go:

```
...
t, err := template.ParseFiles("layout.html", name + ".html")
if (err == nil) {
    templates[name] = t
    fmt.Println("Loaded template", index, name)
} else {
    panic(err)
}
...
```

Если err равен nil, я добавляю на карту пару ключ-значение, используя значение name в качестве ключа и *template.Tempate, назначенный t в качестве значения. Go использует стандартную нотацию индекса для присвоения значений массивам, срезам и картам.

Если значение err не равно nil, то что-то пошло не так. В Go есть функция panic, которую можно вызвать при возникновении неисправимой ошибки. Эффект вызова panic может быть разным, как я объясню в главе 15, но для этого приложения он будет иметь эффект записи трассировки стека и прекращения выполнения.

Скомпилируйте и запустите проект с помощью команды go run.; вы увидите следующий вывод по мере загрузки шаблонов:

```
Loaded template 0 welcome
Loaded template 1 form
Loaded template 2 thanks
Loaded template 3 sorry
Loaded template 4 list
```

Создание обработчиков HTTP и сервера

Стандартная библиотека Go включает встроенную поддержку создания HTTP-серверов и обработки HTTP-запросов. Во-первых, мне нужно определить функции, которые будут вызываться, когда пользователь запрашивает путь URL-адреса по умолчанию для приложения,

который будет `/`, и когда им предоставляется список участников, который будет запрошен с путем URL-адреса `/list`, как показано в листинге 1-16.

```
package main

import (
    "fmt"
    "html/template"
    "net/http"
)

type Rsvp struct {
    Name, Email, Phone string
    WillAttend bool
}

var responses = make([]*Rsvp, 0, 10)
var templates = make(map[string]*template.Template, 3)

func loadTemplates() {
    templateNames := [5]string { "welcome", "form", "thanks", "sorry", "list" }
    for index, name := range templateNames {
        t, err := template.ParseFiles("layout.html", name + ".html")
        if (err == nil) {
            templates[name] = t
            fmt.Println("Loaded template", index, name)
        } else {
            panic(err)
        }
    }
}

func welcomeHandler(writer http.ResponseWriter, request *http.Request) {
    templates["welcome"].Execute(writer, nil)
}

func listHandler(writer http.ResponseWriter, request *http.Request) {
    templates["list"].Execute(writer, responses)
}

func main() {
    loadTemplates()

    http.HandleFunc("/", welcomeHandler)
    http.HandleFunc("/list", listHandler)
}
```

Листинг 1-16 Определение обработчиков начальных запросов в файле main.go в папке partyinvites

Функциональность для работы с HTTP-запросами определена в пакете `net/http`, который является частью стандартной библиотеки Go. Функции, обрабатывающие запросы, должны иметь определенную комбинацию параметров, например:

```
...
func welcomeHandler(writer http.ResponseWriter, request *http.Request) {
...
```

Второй аргумент — это указатель на экземпляр структуры Request, определенной в пакете net/http, который описывает обрабатываемый запрос. Первый аргумент — это пример интерфейса, поэтому он не определен как указатель. Интерфейсы определяют набор методов, которые может реализовать любой тип структуры, что позволяет писать код для использования любого типа, реализующего эти методы, которые я подробно объясню в главе 11.

Одним из наиболее часто используемых интерфейсов является Writer, который используется везде, где можно записывать данные, такие как файлы, строки и сетевые подключения. Тип ResponseWriter добавляет дополнительные функции, относящиеся к работе с ответами HTTP.

Go имеет умный, хотя и необычный подход к интерфейсам и абстракции, следствием которого является то, что ResponseWriter, полученный функциями, определенными в листинге 1-16, может использоваться любым кодом, который знает, как записывать данные с использованием интерфейса Writer. Это включает в себя метод Execute, определенный типом *Template, который я создал при загрузке шаблонов, что упрощает использование вывода от рендеринга шаблона в ответе HTTP:

```
...
templates["list"].Execute(writer, responses)
...
```

Этот оператор считывает *template.Template из карты, назначенной переменной templates, и вызывает определенный им метод Execute. Первый аргумент — это ResponseWriter, куда будут записываться выходные данные ответа, а второй аргумент — это значение данных, которое можно использовать в выражениях, содержащихся в шаблоне.

Пакет net/http определяет функцию HandleFunc, которая используется для указания URL-адреса и обработчика, который будет получать соответствующие запросы. Я использовал HandleFunc для регистрации своих новых функций-обработчиков, чтобы они реагировали на URL-пути / и /list:

```
...
http.HandleFunc("/", welcomeHandler)
http.HandleFunc("/list", listHandler)
...
```

Я продемонстрирую, как можно настроить процесс отправки запросов в последующих главах, но стандартная библиотека содержит базовую систему маршрутизации URL-адресов, которая будет сопоставлять входящие запросы и передавать их функции-обработчику для обработки. Я не определил все функции обработчика, необходимые приложению, но их достаточно, чтобы начать обработку запросов с помощью HTTP-сервера, как показано в листинге 1-17.

```
package main

import (
    "fmt"
    "html/template"
    "net/http"
)

type Rsvp struct {
    Name, Email, Phone string
    WillAttend bool
}

var responses = make([]*Rsvp, 0, 10)
var templates = make(map[string]*template.Template, 3)
```

```
func loadTemplates() {
    templateNames := [5]string { "welcome", "form", "thanks", "sorry", "list" }
    for index, name := range templateNames {
        t, err := template.ParseFiles("layout.html", name + ".html")
        if (err == nil) {
            templates[name] = t
            fmt.Println("Loaded template", index, name)
        } else {
            panic(err)
        }
    }
}

func welcomeHandler(writer http.ResponseWriter, request *http.Request) {
    templates["welcome"].Execute(writer, nil)
}

func listHandler(writer http.ResponseWriter, request *http.Request) {
    templates["list"].Execute(writer, responses)
}

func main() {
    loadTemplates()

    http.HandleFunc("/", welcomeHandler)
    http.HandleFunc("/list", listHandler)

    err := http.ListenAndServe(":5000", nil)
    if (err != nil) {
        fmt.Println(err)
    }
}
```

Листинг 1-17 Создание HTTP-сервера в файле main.go в папке partyinvites

Новые операторы создают HTTP-сервер, который прослушивает запросы через порт 5000, указанный первым аргументом функции `ListenAndServe`. Второй аргумент равен `nil`, что говорит серверу, что запросы должны обрабатываться с использованием функций, зарегистрированных с помощью функции `HandleFunc`. Запустите команду, показанную в листинге 1-18, в папке `partyinvites`, чтобы скомпилировать и выполнить проект.

```
go run .
```

Листинг 1-18 Компиляция и выполнение проекта

Откройте новый веб-браузер и запросите URL-адрес `http://localhost:5000`, что даст ответ, показанный на рисунке 1-1. (Если вы используете Windows, вам может быть предложено подтвердить разрешение брандмауэра Windows, прежде чем запросы смогут быть обработаны сервером. Вам нужно будет предоставлять одобрение каждый раз, когда вы используете команду `go run .` в этой главе. В последующих главах представлен простой сценарий PowerShell для решения этой проблемы.)

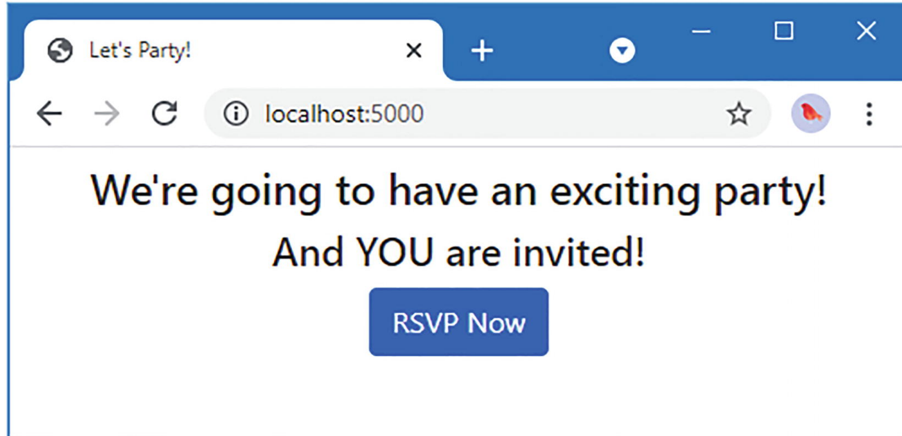

Рисунок 1-1 Обработка HTTP-запросов

Нажмите Ctrl+C, чтобы остановить приложение, как только вы подтвердите, что оно может дать ответ.

Написание функции обработки формы

Нажатие кнопки **RSVP Now** не имеет никакого эффекта, поскольку для URL-адреса `/form`, на который он нацелен, нет обработчика. В листинге 1-19 определяется новая функция-обработчик и начинается реализация функций, необходимых приложению.

```
package main

import (
    "fmt"
    "html/template"
    "net/http"
)

type Rsvp struct {
    Name, Email, Phone string
    WillAttend bool
}

var responses = make([]*Rsvp, 0, 10)
var templates = make(map[string]*template.Template, 3)

func loadTemplates() {
    templateNames := [5]string { "welcome", "form", "thanks", "sorry", "list" }
    for index, name := range templateNames {
        t, err := template.ParseFiles("layout.html", name + ".html")
        if (err == nil) {
            templates[name] = t
            fmt.Println("Loaded template", index, name)
        } else {
            panic(err)
        }
    }
}

func welcomeHandler(writer http.ResponseWriter, request *http.Request) {
    templates["welcome"].Execute(writer, nil)
```

```
}

func listHandler(writer http.ResponseWriter, request *http.Request) {
    templates["list"].Execute(writer, responses)
}

type formData struct {
    *Rsvp
    Errors []string
}

func formHandler(writer http.ResponseWriter, request *http.Request) {
    if request.Method == http.MethodGet {
        templates["form"].Execute(writer, formData {
            Rsvp: &Rsvp{}, Errors: []string {},
        })
    }
}

func main() {
    loadTemplates()

    http.HandleFunc("/", welcomeHandler)
    http.HandleFunc("/list", listHandler)
    http.HandleFunc("/form", formHandler)

    err := http.ListenAndServe(":5000", nil)
    if (err != nil) {
        fmt.Println(err)
    }
}
```

Листинг 1-19 Добавление функции обработчика форм в файл main.go в папке partyinvites

Шаблон `form.html` ожидает получить определенную структуру данных значений данных для отображения своего содержимого. Для представления этой структуры я определил новый тип структуры с именем `formData`. Структуры Go могут быть больше, чем просто группа полей «имя-значение», и одна из предоставляемых ими функций — поддержка создания новых структур с использованием существующих структур. В этом случае я определил структуру `formData`, используя указатель на существующую структуру `Rsvp`, например:

```
...
type formData struct {
    *Rsvp
    Errors []string
}
...
```

В результате структуру `formData` можно использовать так, как будто она определяет поля `Name`, `Email`, `Phone` и `WillAttend` из структуры `Rsvp`, и я могу создать экземпляр структуры `formData`, используя существующее значение `Rsvp`. Звездочка обозначает указатель, что означает, что я не хочу копировать значение `Rsvp` при создании значения `formData`.

Новая функция-обработчик проверяет значение поля `request.Method`, которое возвращает тип полученного HTTP-запроса. Для GET-запросов выполняется шаблон `form`, например:

```
...
if request.Method == http.MethodGet {
    templates["form"].Execute(writer, formData {
```

```
        Rsvp: &Rsvp{}, Errors: []string {},
    })
...
```

Нет данных для использования при ответе на запросы GET, но мне нужно предоставить шаблон с ожидаемой структурой данных. Для этого я создаю экземпляр структуры `formData`, используя значения по умолчанию для ее полей:

```
...
templates["form"].Execute(writer, formData {
        Rsvp: &Rsvp{}, Errors: []string {},
    })
...
```

В Go нет ключевого слова `new`, а значения создаются с помощью фигурных скобок, при этом значения по умолчанию используются для любого поля, для которого значение не указано. Поначалу такой оператор может быть трудно разобрать, но он создает структуру `formData` путем создания нового экземпляра структуры `Rsvp` и создания среза строк, не содержащего значений. Амперсанд (символ `&`) создает указатель на значение:

```
...
templates["form"].Execute(writer, formData {
        Rsvp: &Rsvp{}, Errors: []string {},
    })
...
```

Структура `formData` была определена так, чтобы ожидать указатель на значение `Rsvp`, которое мне позволяет создать амперсанд. Запустите команду, показанную в листинге 1-20, в папке `partyinvites`, чтобы скомпилировать и выполнить проект.

```
go run .
```

Листинг 1-20 Компиляция и выполнение проекта

Откройте новый веб-браузер, запросите URL-адрес `http://localhost:5000` и нажмите кнопку **RSVP Now**. Новый обработчик получит запрос от браузера и отобразит HTML-форму, показанную на рисунке 1-2.

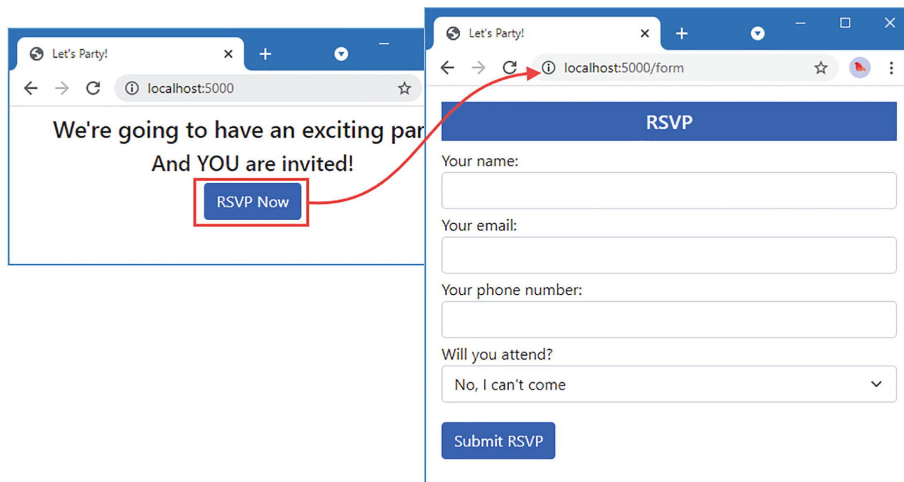

Рисунок 1-2 Отображение HTML-формы

Обработка данных формы

Теперь мне нужно обработать POST-запросы и прочитать данные, которые пользователь ввел в форму, как показано в листинге 1-21. В этом листинге показаны только изменения функции formHandler; остальная часть файла main.go остается неизменной.

```
...
func formHandler(writer http.ResponseWriter, request *http.Request) {
    if request.Method == http.MethodGet {
        templates["form"].Execute(writer, formData {
            Rsvp: &Rsvp{}, Errors: []string {},
        })
    } else if request.Method == http.MethodPost {
        request.ParseForm()
        responseData := Rsvp {
            Name: request.Form["name"][0],
            Email: request.Form["email"][0],
            Phone: request.Form["phone"][0],
            WillAttend: request.Form["willattend"][0] == "true",
        }

        responses = append(responses, &responseData)

        if responseData.WillAttend {
            templates["thanks"].Execute(writer, responseData.Name)
        } else {
            templates["sorry"].Execute(writer, responseData.Name)
        }
    }
}
...
```

Листинг 1-21 Обработка данных формы в файле main.go в папке partyinvites

Метод ParseForm обрабатывает данные формы, содержащиеся в HTTP-запросе, и заполняет карту, доступ к которой можно получить через поле Form. Затем данные формы используются для создания значения Rsvp:

```
...
responseData := Rsvp {
    Name: request.Form["name"][0],
    Email: request.Form["email"][0],
    Phone: request.Form["phone"][0],
    WillAttend: request.Form["willattend"][0] == "true",
}
...
```

Этот оператор демонстрирует, как структура создается со значениями для ее полей, в отличие от значений по умолчанию, которые использовались в листинге 1-19. HTML-формы могут включать несколько значений с одним и тем же именем, поэтому данные формы представлены в виде среза значений. Я знаю, что для каждого имени будет только одно значение, и я обращаюсь к первому значению в срезе, используя стандартную нотацию индекса с отсчетом от нуля, которую используют большинство языков.

Создав значение Rsvp, я добавляю его в срез, присвоенный переменной responses:

```
...
responses = append(responses, &responseData)
...
```

Функция append используется для добавления значения к срезу. Обратите внимание, что я использую амперсанд для создания указателя на созданное значение Rsvp. Если бы я не использовал указатель, то мое значение Rsvp дублировалось бы при добавлении в срез.

Остальные операторы используют значение поля WillAttend для выбора шаблона, который будет представлен пользователю.

Запустите команду, показанную в листинге 1-22, в папке partyinvites, чтобы скомпилировать и выполнить проект.

```
go run .
```

Листинг 1-22 Компиляция и выполнение проекта

Откройте новый веб-браузер, запросите URL-адрес http://localhost:5000 и нажмите кнопку **RSVP Now**. Заполните форму и нажмите кнопку **Submit RSVP**; вы получите ответ, выбранный на основе значения, которое вы выбрали с помощью элемента выбора HTML. Щелкните ссылку в ответе, чтобы просмотреть сводку ответов, полученных приложением, как показано на рисунке 1-3.

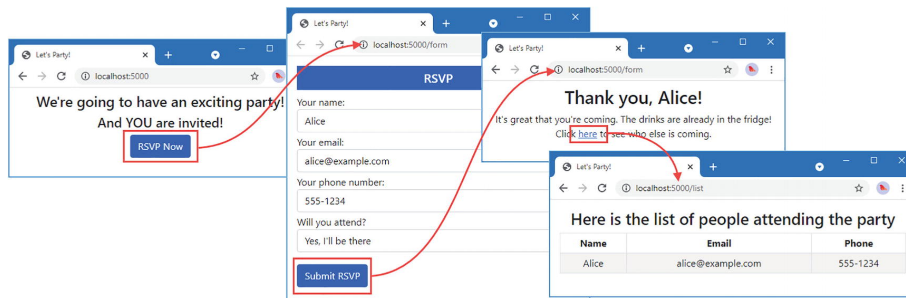

Рисунок 1-3 Обработка данных формы

Добавление проверки данных

Все, что требуется для завершения приложения, — это некоторая базовая проверка, чтобы убедиться, что пользователь заполнил форму, как показано в листинге 1-23. В этом листинге показаны изменения в функции formHandler, а остальная часть файла main.go осталась неизменной.

```
...
func formHandler(writer http.ResponseWriter, request *http.Request) {
    if request.Method == http.MethodGet {
        templates["form"].Execute(writer, formData {
            Rsvp: &Rsvp{}, Errors: []string {},
        })
    } else if request.Method == http.MethodPost {
        request.ParseForm()
        responseData := Rsvp {
            Name: request.Form["name"][0],
            Email: request.Form["email"][0],
            Phone: request.Form["phone"][0],
            WillAttend: request.Form["willattend"][0] == "true",
        }

        errors := []string {}
        if responseData.Name == "" {
            errors = append(errors, "Please enter your name")
        }
```

```
        if responseData.Email == "" {
            errors = append(errors, "Please enter your email address")
        }
        if responseData.Phone == "" {
            errors = append(errors, "Please enter your phone number")
        }
        if len(errors) > 0 {
            templates["form"].Execute(writer, formData {
                Rsvp: &responseData, Errors: errors,
            })
        } else {
            responses = append(responses, &responseData)
            if responseData.WillAttend {
                templates["thanks"].Execute(writer, responseData.Name)
            } else {
                templates["sorry"].Execute(writer, responseData.Name)
            }
        }
    }
}
...
```

Листинг 1-23 Проверка данных формы в файле main.go в папке partyinvites

Приложение получит пустую строку (`""`) из запроса, если пользователь не предоставит значение для поля формы. Новые операторы в листинге 1-23 проверяют поля `Name`, `EMail` и `Phone` и добавляют сообщение к срезу строк для каждого поля, не имеющего значения. Я использую встроенную функцию `len`, чтобы получить количество значений в срезе ошибок, и если есть ошибки, я снова визуализирую содержимое шаблона `form`, включая сообщения об ошибках в данных, которые получает шаблон. Если ошибок нет, то используется шаблон `thanks` или `sorry`.

Запустите команду, показанную в листинге 1-24, в папке `partyinvites`, чтобы скомпилировать и выполнить проект.

```
go run .
```

Листинг 1-24 Компиляция и выполнение проекта

Откройте новый веб-браузер, запросите URL-адрес `http://localhost:5000` и нажмите кнопку **RSVP Now**. Нажмите кнопку **Submit RSVP**, не вводя никаких значений в форму; вы увидите предупреждающие сообщения, как показано на рисунке 1-4. Введите некоторые данные в форму и отправьте ее снова, и вы увидите окончательное сообщение.

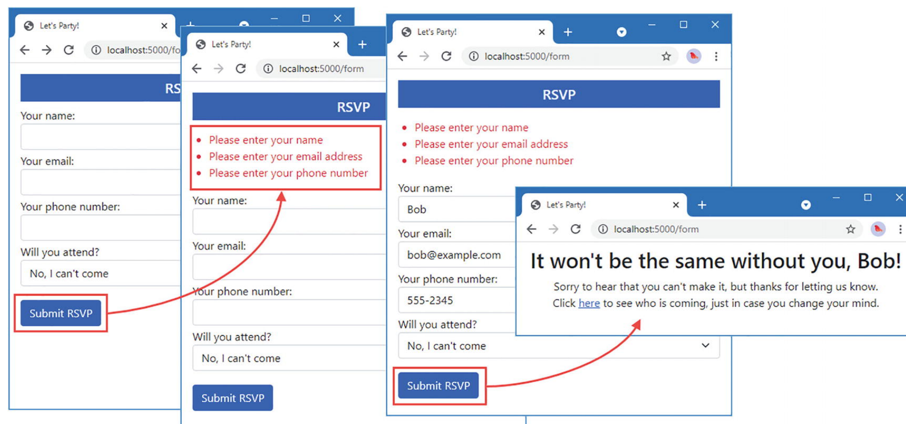

Рисунок 1-4 Проверка данных

Резюме

В этой главе я установил пакет Go и использовал содержащиеся в нем инструменты для создания простого веб-приложения, используя только один файл кода и несколько основных шаблонов HTML. Теперь, когда вы увидели Go в действии, следующая глава поместит эту книгу в контекст.

2. Включение Go в контекст

Go, часто называемый *Golang*, — это язык, первоначально разработанный в Google, который начал получать широкое распространение. Go синтаксически похож на C, но имеет безопасные указатели, автоматическое управление памятью и одну из самых полезных и хорошо написанных стандартных библиотек, с которыми мне приходилось сталкиваться.

Почему вам стоит изучать Go?

Go можно использовать практически для любых задач программирования, но лучше всего он подходит для разработки серверов или систем. Обширная стандартная библиотека включает поддержку наиболее распространенных задач на стороне сервера, таких как обработка HTTP-запросов, доступ к базам данных SQL и рендеринг шаблонов HTML. Он имеет отличную поддержку многопоточности, а комплексная система отражения позволяет писать гибкие API для платформ и фреймворков.

Go поставляется с полным набором инструментов разработки, а также имеется хорошая поддержка редактора, что упрощает создание качественной среды разработки.

Go является кроссплатформенным, что означает, что вы можете писать, например, в Windows и развертывать на серверах Linux. Или, как я показываю в этой книге, вы можете упаковать свое приложение в контейнеры Docker для простого развертывания на общедоступных платформах хостинга.

В чем подвох?

Go может быть трудным для изучения, и это язык с «мнением», что может разочаровать его использование. Эти мнения варьируются от проницательных до раздражающих. Проницательные мнения делают Go свежим и приятным опытом, например, позволяя функциям

возвращать несколько результатов, чтобы одно значение не должно было представлять как успешные, так и неудачные результаты. В Go есть несколько выдающихся функций, в том числе интуитивно понятная поддержка многопоточности, которые обагатили бы многие другие языки.

Раздражающие мнения превращают написание Go в затяжной спор с компилятором, что-то вроде спора о программировании «и еще кое-что…». Если ваш стиль кодирования не совпадает с мнением дизайнеров Go, вы можете ожидать появления множества ошибок компилятора. Если, как и я, вы пишете код в течение длительного времени и у вас есть укоренившиеся привычки, перенятые со многих языков, то вы разработаете новые и инновационные ругательства, которые будете использовать, когда компилятор неоднократно отвергает ваш код для выражений, которые бы компилировались на любом другом основном языке программирования за последние 30 лет.

Кроме того, у Go есть определенный уклон в сторону системного программирования и разработки на стороне сервера. Например, есть пакеты, которые обеспечивают поддержку разработки пользовательского интерфейса, но это не та область, в которой Go сияет, и есть лучшие альтернативы.

Это действительно настолько плохо?

Не откладывай. Go превосходен, и его стоит изучить, если вы работаете над системным программированием или проектами по разработке серверов. Go обладает инновационными и эффективными функциями. Опытный разработчик Go может писать сложные приложения, прилагая на удивление мало усилий и кода.

Изучайте Go, зная, что это требует усилий. Пишите на Go, зная, что когда вы и разработчики языка расходитесь во мнениях, их предпочтения превалируют.

Что вы должны знать?

Это продвинутая книга, написанная для опытных разработчиков. Эта книга не учит программированию, и вам потребуется разбираться в смежных темах, таких как HTML, чтобы следовать всем примерам.

Какова структура этой книги?

Эта книга разделена на три части, каждая из которых охватывает набор связанных тем.

Часть 1: Понимание языка Go

В первой части этой книги я описываю средства разработки Go и язык Go. Я опишу встроенные типы данных, покажу, как можно создавать собственные типы, и расскажу о таких функциях, как управление потоком, обработка ошибок и параллелизм. Эти главы включают некоторые функции из стандартной библиотеки Go, где они необходимы для поддержки объяснения возможностей языка или где они выполняют задачи, тесно связанные с описываемыми функциями языка.

Часть 2: Использование стандартной библиотеки Go

Во второй части этой книги я описываю наиболее полезные пакеты из обширной стандартной библиотеки Go. Вы узнаете о функциях форматирования строк, чтения и записи данных; создание HTTP-серверов и клиентов; использование баз данных; и использование мощной поддержки для рефлексии.

Часть 3: Применение Go

В третьей части этой книги я использую Go для создания пользовательской среды веб-приложений, которая является основой для интернет-магазина SportsStore. В этой части книги показано, как Go и его стандартная библиотека могут использоваться вместе для решения проблем, возникающих в реальных проектах. Примеры в первой и второй части этой книги сфокусированы на применение отдельных функций, а цель третьей части — показать использование функций в комбинации.

Что не охватывает эта книга?

Эта книга не охватывает все пакеты, предоставляемые стандартной библиотекой Go, которая, как уже отмечалось, обширна. Кроме того, есть некоторые функции языка Go, которые я пропустил, поскольку

они бесполезны в основной разработке. Функции, которые я описал в этой книге, нужны большинству читателей в большинстве ситуаций.

Пожалуйста, свяжитесь со мной и дайте мне знать, если есть функция, которую я не описал, которую вы хотите изучить. Я сохраню список и включу наиболее востребованные темы в следующий выпуск.

Что делать, если вы нашли ошибку в книге?

Вы можете сообщать мне об ошибках по электронной почте adam@adam-freeman.com, хотя я прошу вас сначала проверить список опечаток/исправлений для этой книги, который вы можете найти в репозитории книги на GitHub по адресу https://github.com/apress/pro-go, если о проблеме уже сообщалось.

Я добавляю ошибки, которые могут запутать читателей, особенно проблемы с примерами кода, в файл опечаток/исправлений в репозитории GitHub с благодарностью первому читателю, сообщившему об этом. Я также веду список менее серьезных проблем, которые обычно означают ошибки в тексте, окружающем примеры, и я использую их, когда пишу новое издание.

Много ли примеров?

Есть *масса* примеров. Лучший способ учиться — на примерах, и я собрал в этой книге столько примеров, сколько смог. Чтобы облегчить следование примерам, я принял простое соглашение, которому следую, когда это возможно. Когда я создаю новый файл, я перечисляю его полное содержимое, как показано в листинге 2-1. Все листинги кода включают имя файла в заголовке листинга вместе с папкой, в которой его можно найти.

```
package store

type Product struct {
    Name, Category string
    price float64
}
```

```
func (p *Product) Price(taxRate float64) float64 {
    return p.price + (p.price * taxRate)
}
```

Листинг 2-1 Содержимое файла product.go в папке store

Этот листинг взят из главы 13. Не беспокойтесь о том, что он делает; просто имейте в виду, что это полный листинг, в котором показано все содержимое файла, а в заголовке указано, как называется файл и где он находится в проекте.

Когда я вношу изменения в код, я выделяю измененные операторы жирным шрифтом, как показано в листинге 2-2.

```
package store

type Product struct {
    Name, Category string
    price float64
}

func NewProduct(name, category string, price float64)
*Product {
    return &Product{ name, category, price }
}

func (p *Product) Price(taxRate float64) float64 {
    return p.price + (p.price * taxRate)
}
```

Листинг 2-2 Определение конструктора в файле product.go в папке store

Этот список взят из более позднего примера, который требует изменения в файле, созданном в листинге 2-1. Чтобы помочь вам следовать примеру, изменения выделены жирным шрифтом.

Некоторые примеры требуют небольших изменений в большом файле. Чтобы не тратить место на перечисление неизмененных частей файла, я просто показываю изменяющуюся область, как показано в листинге 2-3. Вы можете сказать, что этот список показывает только часть файла, потому что он начинается и заканчивается многоточием (...).

```
...
```

```
func queryDatabase(db *sql.DB) {
    rows, err := db.Query("SELECT * from Products")
    if (err == nil) {
        for (rows.Next()) {
            var id, category int
            var name int
            var price float64
                scanErr := rows.Scan(&id, &name, &category,
&price)
            if (scanErr == nil) {
                    Printfln("Row: %v %v %v %v", id, name,
category, price)
            } else {
                Printfln("Scan error: %v", scanErr)
                break
            }
        }
    } else {
        Printfln("Error: %v", err)
    }
}
...
```

Листинг 2-3 Несовпадающее сканирование в файле main.go в папке data

В некоторых случаях мне нужно внести изменения в разные части одного и того же файла, и в этом случае я опускаю некоторые элементы или операторы для краткости, как показано в листинге 2-4. В этом листинге добавлены новые операторы использования и определены дополнительные методы для существующего файла, большая часть которых не изменилась и была исключена из листинга.

```
package main

import "database/sql"

// ...код пропущен для краткости...

func insertAndUseCategory(db *sql.DB, name string, productIDs
...int) (err error) {
    tx, err := db.Begin()
    updatedFailed := false
```

```
    if (err == nil) {
                                         catResult,    err    :=
tx.Stmt(insertNewCategory).Exec(name)
        if (err == nil) {
            newID, _ := catResult.LastInsertId()
                                           preparedStatement   :=
tx.Stmt(changeProductCategory)
            for _, id := range productIDs {
                                          changeResult,  err   :=
preparedStatement.Exec(newID, id)
                if (err == nil) {
                    changes, _ := changeResult.RowsAffected()
                    if (changes == 0) {
                        updatedFailed = true
                        break
                    }
                }
            }
        }
    }
    if (err != nil || updatedFailed) {
        Printfln("Aborting transaction %v", err)
        tx.Rollback()
    } else {
        tx.Commit()
    }
    return
}
```

Листинг 2-4 Использование транзакции в файле main.go в папке data

Это соглашение позволяет мне упаковать больше примеров, но это означает, что может быть трудно найти конкретный метод. С этой целью главы в этой книге начинаются со сводной таблицы, описывающей содержащиеся в ней методы, а большинство глав в первой части и второй части содержат краткие справочные таблицы, в которых перечислены методы, используемые для реализации конкретной функции.

Какое программное обеспечение вам нужно для примеров?

Единственное программное обеспечение, необходимое для разработки на Go, описано в главе 1. Я устанавливаю некоторые сторонние пакеты в последующих главах, но их можно получить с помощью уже настроенной вами команды `go`. Я использую Docker контейнеры в части 3, но это необязательно.

На каких платформах будут работать примеры?

Все примеры были протестированы на Windows и Linux (в частности, на Ubuntu 20.04), и все сторонние пакеты поддерживают эти платформы. Go поддерживает другие платформы, и примеры должны работать на этих платформах, но я не могу помочь, если у вас возникнут проблемы с примерами из этой книги.

Что делать, если у вас возникли проблемы с примерами?

Первое, что нужно сделать, это вернуться к началу главы и начать заново. Большинство проблем вызвано случайным пропуском шага или неполным применением изменений, показанных в листинге. Обратите особое внимание на листинг кода, выделенный жирным шрифтом, который показывает необходимые изменения.

Затем проверьте список опечаток/исправлений, который включен в репозиторий книги на GitHub. Технические книги сложны, и ошибки неизбежны, несмотря на все мои усилия и усилия моих редакторов. Проверьте список ошибок, чтобы найти список известных ошибок и инструкции по их устранению.

Если у вас все еще есть проблемы, загрузите проект главы, которую вы читаете, из GitHub-репозитория книги, `https://github.com/apress/pro-go`, и сравните его со своим проектом. Я создаю код для репозитория GitHub, прорабатывая

каждую главу, поэтому в вашем проекте должны быть одни и те же файлы с одинаковым содержимым.

Если вы по-прежнему не можете заставить примеры работать, вы можете связаться со мной по адресу `adam@adam-freeman.com` для получения помощи. Пожалуйста, укажите в письме, какую книгу вы читаете и какая глава/пример вызывает проблему. Номер страницы или список кодов всегда полезны. Пожалуйста, помните, что я получаю много писем и могу не ответить сразу.

Где взять пример кода?

Вы можете загрузить примеры проектов для всех глав этой книги с `https://github.com/apress/pro-go`.

Почему некоторые примеры имеют странное форматирование?

Go имеет необычный подход к форматированию, что означает, что операторы могут быть разбиты на несколько строк только в определенных точках. Это не проблема в редакторе кода, но вызывает проблемы с печатной страницей, которая имеет определенную ширину. Некоторые примеры, особенно в последних главах, требуют длинных строк кода, которые неудобно отформатированы, чтобы их можно было использовать в книге.

Как связаться с автором?

Вы можете написать мне по адресу `adam@adam-freeman.com`. Прошло несколько лет с тех пор, как я впервые опубликовал адрес электронной почты в своих книгах. Я не был полностью уверен, что это была хорошая идея, но я рад, что сделал это. Я получаю электронные письма со всего мира от читателей, работающих или обучающихся в каждой отрасли, и, во всяком случае, по большей части электронные письма позитивны, вежливы, и их приятно получать.

Я стараюсь отвечать быстро, но получаю много писем, а иногда получаю невыполненные работы, особенно когда пытаюсь закончить

книгу. Я всегда стараюсь помочь читателям, которые застряли с примером в книге, хотя я прошу вас выполнить шаги, описанные ранее в этой главе, прежде чем связываться со мной.

Хотя я приветствую электронные письма читателей, есть некоторые общие вопросы, на которые всегда будет ответ «нет». Я боюсь, что я не буду писать код для вашего нового стартапа, помогать вам с поступлением в колледж, участвовать в споре о дизайне вашей команды разработчиков или учить вас программировать.

Что, если мне действительно понравилась эта книга?

Пожалуйста, напишите мне по адресу adam@adam-freeman.com и дайте мне знать. Всегда приятно получать известия от довольных читателей, и я ценю время, затрачиваемое на отправку этих писем. Написание этих книг может быть трудным, и эти электронные письма обеспечивают существенную мотивацию, чтобы упорствовать в деятельности, которая иногда может казаться невозможной..

Что, если эта книга меня разозлила, и я хочу пожаловаться?

Вы по-прежнему можете написать мне по адресу adam@adam-freeman.com, и я все равно постараюсь вам помочь. Имейте в виду, что я могу помочь только в том случае, если вы объясните, в чем проблема и что вы хотите, чтобы я с ней сделал. Вы должны понимать, что иногда единственным выходом является признание того, что я не писатель для вас, и что мы удовлетворитесь только тогда, когда вы вернете эту книгу и выберете другую. Я тщательно обдумаю все, что вас расстроило, но после 25 лет написания книг я пришел к выводу, что не всем нравится читать книги, которые я люблю писать.

Резюме

В этой главе я изложил содержание и структуру этой книги. Лучший способ изучить Go — написать код, и в следующей главе я опишу

инструменты, которые Go предоставляет именно для этого.

3. Использование инструментов Go

В этой главе я описываю инструменты разработки Go, большинство из которых были установлены как часть пакета Go в главе 1. Я описываю базовую структуру проекта Go, объясняю, как компилировать и выполнять код Go, и показываю, как установить и использовать отладчик для приложений Go. Я также описываю инструменты Go для линтинга и форматирования.

> **Подсказка**
> Вы можете загрузить пример проекта для этой главы — и для всех остальных глав этой книги — с `https://github.com/apress/pro-go`. См. главу 2 о том, как получить помощь, если у вас возникнут проблемы с запуском примеров.

Использование команды Go

Команда `go` предоставляет доступ ко всем функциям, необходимым для компиляции и выполнения кода Go, и используется в этой книге. Аргумент, используемый с командой `go`, определяет операцию, которая будет выполнена, например, аргумент `run`, используемый в главе 1, который компилирует и выполняет исходный код Go. Команда `go` поддерживает большое количество аргументов; Таблица 3-1 описывает наиболее полезные из них.

Таблица 3-1 Используемые аргументы в команде go

Аргументы	Описание
`build`	Команда `go build` компилирует исходный код в текущем каталоге и создает исполняемый файл, как описано в разделе «Компиляция и запуск исходного кода».
`clean`	Команда `go clean` удаляет выходные данные, созданные командой `go build`, включая исполняемый файл и любые временные файлы, созданные во время сборки, как описано в разделе «Компиляция и запуск исходного кода».

Аргументы	Описание
`doc`	Команда `go doc` генерирует документацию из исходного кода. Смотрите простой пример в разделе «Линтинг кода Go».
`fmt`	Команда `go fmt` обеспечивает согласованный отступ и выравнивание в файлах исходного кода, как описано в разделе «Форматирование кода Go».
`get`	Команда `go get` загружает и устанавливает внешние пакеты, как описано в главе 12.
`install`	Команда `go install` загружает пакеты и обычно используется для установки пакетов инструментов, как показано в разделе «Отладка кода Go».
`help`	Команда `go help` отображает справочную информацию по другим функциям Go. Например, команда `go help build` отображает информацию об аргументе `build`.
`mod`	Команда `go mod` используется для создания модуля Go и управления им, как показано в разделе «Определение модуля» и более подробно описано в главе 12.
`run`	Команда `go run` создает и выполняет исходный код в указанной папке без создания исполняемого вывода, как описано в разделе «Использование команды go run».
`test`	Команда `go test` выполняет модульные тесты, как описано в Uлаве 31.
`version`	Команда `go version` выводит номер версии Go.
`vet`	Команда `go vet` обнаруживает распространенные проблемы в коде Go, как описано в разделе «Устранение распространенных проблем в коде Go».

Создание проекта Go

Проекты Go не имеют сложной структуры и быстро настраиваются. Откройте новую командную строку и создайте папку с именем `tools` в удобном месте. Добавьте файл с именем `main.go` в папку инструментов с содержимым, показанным в листинге 3-1.

```
package main

import "fmt"

func main() {
    fmt.Println("Hello, Go")
}
```

Листинг 3-1 Содержимое файла main.go в папке tools

Я подробно расскажу о языке Go в последующих главах, но для начала на рисунке 3-1 показаны ключевые элементы файла `main.go`.

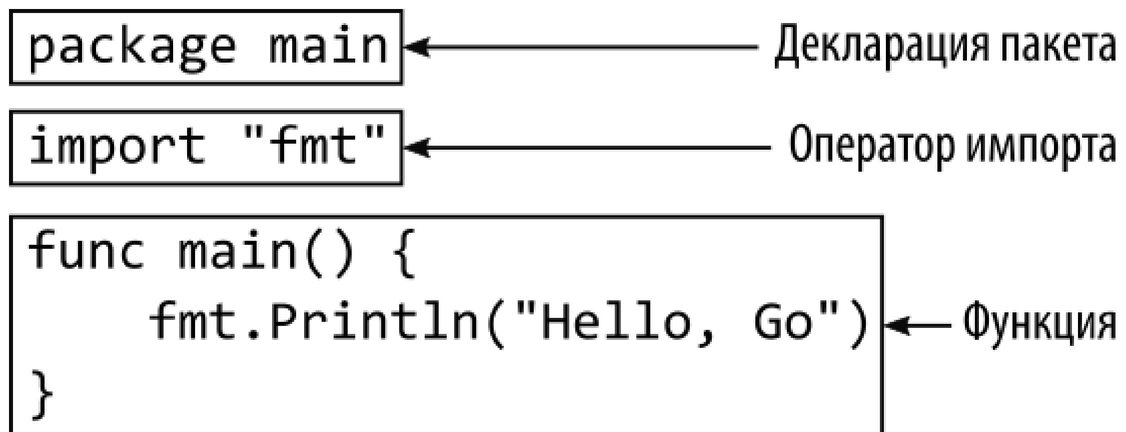

Рисунок 3-1 Ключевые элементы в файле кода

Понимание объявления пакета

Первый оператор — это объявление пакета. Пакеты используются для группировки связанных функций, и каждый файл кода должен объявлять пакет, к которому принадлежит его содержимое. В объявлении пакета используется ключевое слово `package`, за которым следует имя пакета, как показано на рисунке 3-2. Оператор в этом файле указывает пакет с именем `main`.

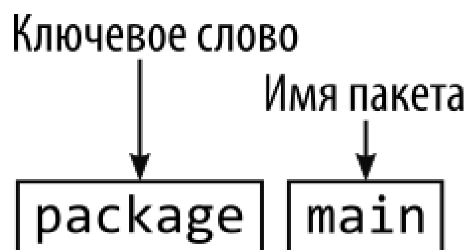

Рисунок 3-2 Указание пакета для файла кода

Понимание оператора импорта

Следующий оператор — это оператор импорта, который используется для объявления зависимостей от других пакетов. За ключевым словом `import` следует имя пакета, заключенное в двойные кавычки, как показано на рисунке 3-3. Оператор `import` в листинге 3-1 задает пакет с именем `fmt`, который является встроенным пакетом Go для чтения и записи форматированных строк (подробно описанный в главе 17).

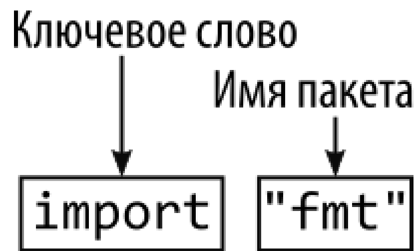

Рисунок 3-3 Объявление зависимости пакета

> **Подсказка**
> Полный список встроенных пакетов Go доступен по адресу `https://golang.org/pkg`.

Понимание функции

Остальные операторы в файле `main.go` определяют функцию с именем `main`. Я подробно описываю функции в главе 8, но функция `main` особенная. Когда вы определяете функцию с именем `main` в пакете с именем `main`, вы создаете точку входа, с которой начинается выполнение в приложении командной строки. Рисунок 3-4 иллюстрирует структуру функции `main`.

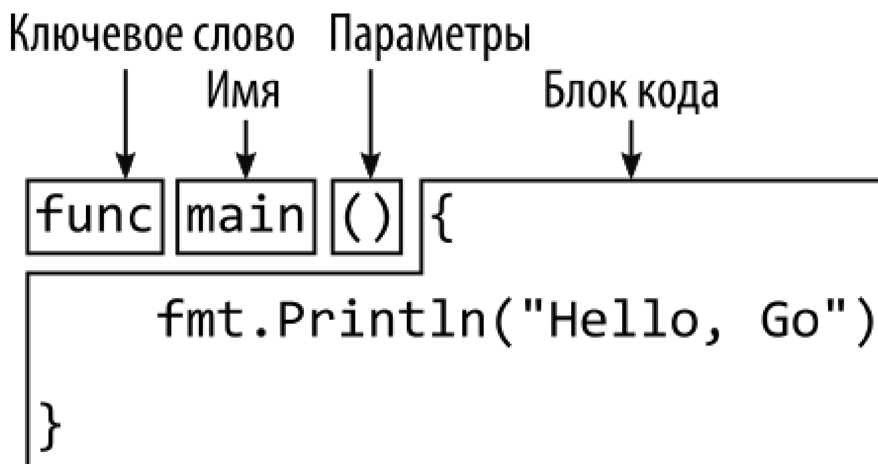

Рисунок 3-4 Структура функции main

Базовая структура функций Go аналогична другим языкам. Ключевое слово `func` обозначает функцию, за которым следует имя функции, которое в данном примере — `main`.

Функция в листинге 3-1 не определяет никаких параметров, что обозначено пустыми скобками и не дает результата. Я опишу более

сложные функции в следующих примерах, но этой простой функции достаточно для начала.

Блок кода функции содержит операторы, которые будут выполняться при вызове функции. Поскольку функция `main` является точкой входа, она будет вызываться автоматически при выполнении скомпилированного вывода проекта.

Понимание оператора кода

Функция `main` содержит один оператор кода. Когда вы объявляете зависимость от пакета с помощью оператора `import`, результатом является ссылка на пакет, которая обеспечивает доступ к функциям пакета. По умолчанию ссылке на пакет назначается имя пакета, так что функции, предоставляемые пакетом `fmt`, например, доступны через ссылку на пакет `fmt`, как показано на рисунке 3-5.

Рисунок 3-5 Доступ к функциям пакета

Этот оператор вызывает функцию с именем `Println`, предоставляемую пакетом `fmt`. Эта функция записывает строку в стандартный вывод, что означает, что она будет отображаться в консоли при сборке и выполнении проекта в следующем разделе.

Для доступа к функции используется имя пакета, за которым следует точка, а затем функция: `fmt.Println`. Этой функции передается один аргумент — строка, которая будет записана.

ИСПОЛЬЗОВАНИЕ ТОЧКИ С ЗАПЯТОЙ В КОДЕ GO

В Go необычный подход к точкам с запятой: они необходимы для завершения операторов кода, но не требуются в файлах исходного кода. Вместо этого инструменты сборки Go выясняют, куда должны

идти точки с запятой, когда они обрабатывают файлы, действуя так, как будто они были добавлены разработчиком.

В результате точки с запятой можно использовать в файлах исходного кода Go, но они не обязательны и обычно опускаются.

Некоторые странности возникают, если вы не следуете ожидаемому стилю кода Go. Например, вы получите ошибки компилятора, если попытаетесь поместить открывающую фигурную скобку для функции или цикла for на следующей строке, например:

```
package main

import "fmt"

func main()
{
    fmt.Println("Hello, Go")
}
```

Ошибки сообщают о неожиданной точке с запятой и отсутствующем теле функции. Это связано с тем, что инструменты Go автоматически вставили точку с запятой следующим образом:

```
package main

import "fmt"

func main();
{
    fmt.Println("Hello, Go")
}
```

Сообщения об ошибках имеют больше смысла, когда вы понимаете, почему они возникают, хотя может быть сложно приспособиться к ожидаемому формату кода, если это ваше предпочтительное размещение фигурной скобки.

В этой книге я пытался следовать соглашению об отсутствии точки с запятой, но я десятилетиями пишу код на языках, требующих точки с запятой, поэтому вы можете найти случайный пример, когда я добавлял точки с запятой исключительно по привычке. Команда go fmt, которую я описываю в разделе «Форматирование кода Go»,

удалит точки с запятой и устранит другие проблемы с форматированием.

Компиляция и запуск исходного кода

Команда `go build` компилирует исходный код Go и создает исполняемый файл. Запустите команду, показанную в листинге 3-2, в папке `tools`, чтобы скомпилировать код.

```
go build main.go
```

Листинг 3-2 Использование компилятора

Компилятор обрабатывает инструкции в файле `main.go` и создает исполняемый файл, который называется `main.exe` в Windows и `main` на других платформах. (Компилятор начнет создавать файлы с более удобными именами, как только я добавлю модули в раздел «Определение модуля».)

Запустите команду, показанную в листинге 3-3, в папке `tools`, чтобы запустить исполняемый файл.

```
./main
```

Листинг 3-3 Запуск скомпилированного исполняемого файла

Точка входа проекта — функция с именем `main` в пакете, который тоже называется `main` — выполняется и выдает следующий результат:

```
Hello, Go
```

НАСТРОЙКА КОМПИЛЯТОРА GO

Поведение компилятора Go можно настроить с помощью дополнительных аргументов, хотя для большинства проектов достаточно настроек по умолчанию. Двумя наиболее полезными являются `-a`, вызывающая полную пересборку даже для неизмененных файлов, и `-o`, указывающая имя скомпилированного выходного файла. Используйте команду `go help build`, чтобы увидеть полный список доступных опций. По умолчанию компилятор создает исполняемый файл, но доступны и другие

выходные данные — подробности см. на странице https://golang.org/cmd/go/#hdr-Build_modes.

Очистка

Чтобы удалить выходные данные процесса компиляции, запустите команду, показанную в листинге 3-4, в папке `tools`.

```
go clean main.go
```

Листинг 3-4 Очистка

Скомпилированный исполняемый файл, созданный в предыдущем разделе, удаляется, остается только файл исходного кода.

Использование команды go run

Обычно разработка выполняется с помощью команды `go run`. Запустите команду, показанную в листинге 3-5, в папке `tools`.

```
go run main.go
```

Листинг 3-5 Использование команды go run

Файл компилируется и выполняется за один шаг, без создания исполняемого файла в папке инструментов. Создается исполняемый файл, но во временной папке, из которой он затем запускается. (Именно эта серия временных местоположений заставляла брандмауэр Windows запрашивать разрешение каждый раз, когда в главе 1 использовалась команда `go run`. Каждый раз, когда запускалась команда, исполняемый файл создавался в новой временной папке и который казался совершенно новым файлом для брандмауэра.)

Команда в листинге 3-5 выводит следующий результат:

```
Hello, Go
```

Определение модуля

В предыдущем разделе было показано, что вы можете начать работу, просто создав файл кода, но более распространенным подходом является создание модуля Go, что является обычным первым шагом при запуске нового проекта. Создание модуля Go позволяет проекту легко

использовать сторонние пакеты и может упростить процесс сборки. Запустите команду, показанную в листинге 3-6, в папке `tools`.

```
go mod init tools
```

Листинг 3-6 Создание модуля

Эта команда добавляет файл с именем `go.mod` в папку `tools`. Причина, по которой большинство проектов начинается с команды `go mod init`, заключается в том, что она упрощает процесс сборки. Вместо указания конкретного файла кода проект может быть построен и выполнен с использованием точки, указывающей проект в текущем каталоге. Запустите команду, показанную в листинге 3-7, в папке инструментов, чтобы скомпилировать и выполнить содержащийся в ней код, не указывая имя файла кода.

```
go run .
```

Листинг 3-7 Компиляция и выполнение проекта

Файл `go.mod` можно использовать и по-другому, как показано в следующих главах, но я начинаю все примеры в оставшейся части книги с команды `go mod init`, чтобы упростить процесс сборки.

Отладка кода Go

Стандартный отладчик для приложений Go называется Delve. Это сторонний инструмент, но он хорошо поддерживается и рекомендуется командой разработчиков Go. Delve поддерживает Windows, macOS, Linux и FreeBSD. Чтобы установить пакет Delve, откройте новую командную строку и выполните команду, показанную в листинге 3-8.

> **Подсказка**
> См. `https://github.com/go-delve/delve/tree/master/Documentation/installation` для получения подробных инструкций по установке для каждой платформы. Для выбранной операционной системы может потребоваться дополнительная настройка.

```
go install github.com/go-delve/delve/cmd/dlv@latest
```

Листинг 3-8 Установка пакета отладчика

Команда `go install` загружает и устанавливает пакет и используется для установки таких инструментов, как отладчики. Аналогичная команда — `go get` — выполняет аналогичную задачу для пакетов, предоставляющих функции кода, которые должны быть включены в приложение, как показано в главе 12.

Чтобы убедиться, что отладчик установлен, выполните команду, показанную в листинге 3-9.

```
dlv version
```

Листинг 3-9 Запуск отладчика

Если вы получаете сообщение об ошибке, что команда `dlv` не может быть найдена, попробуйте указать путь напрямую. По умолчанию команда `dlv` будет установлена в папку `~/go/bin` (хотя это можно переопределить, задав переменную среды `GOPATH`), как показано в листинге 3-10.

```
~/go/bin/dlv
```

Листинг 3-10 Запуск отладчика с путем

Если пакет был установлен правильно, вы увидите вывод, аналогичный следующему, хотя вы можете увидеть другой номер версии и идентификатор сборки:

```
Delve Debugger
Version: 1.7.1
Build: $Id: 3bde2354aafb5a4043fd59838842c4cd4a8b6f0b $
```

ОТЛАДКА С ФУНКЦИЕЙ PRINTLN

Мне нравятся такие отладчики, как Delve, но я использую их только для решения проблем, которые не могу решить с помощью своего основного метода отладки: функции `Println`. Я использую `Println`, потому что это быстро, просто и надежно, а также потому, что большинство ошибок (по крайней мере, в моем коде) возникают из-за того, что функция не получила ожидаемого значения или из-за

того, что конкретный оператор не выполняется, когда я ожидаю. Эти простые проблемы легко диагностируются с помощью записи сообщения в консоль.

Если вывод моих сообщений Println не помогает, я запускаю отладчик, устанавливаю точку останова и выполняю свой код. Даже тогда, как только я понимаю причину проблемы, я склонен возвращаться к операторам Println, чтобы подтвердить свою теорию.

Многие разработчики не хотят признавать, что они находят отладчики неудобными или запутанными, и в конечном итоге все равно тайно используют Println. Отладчики сбивают с толку, и нет ничего постыдного в использовании всех имеющихся в вашем распоряжении инструментов. Функция Println и отладчик являются взаимодополняющими инструментами, и важно то, что ошибки исправляются независимо от того, как это делается.

Подготовка к отладке

В файле main.go недостаточно кода для отладки. Добавьте операторы, показанные в листинге 3-11, чтобы создать цикл, который будет распечатывать ряд числовых значений.

```
package main

import "fmt"

func main() {
    fmt.Println("Hello, Go")
    for i := 0; i < 5; i++ {
        fmt.Println(i)
    }
}
```

Листинг 3-11 Добавление цикла в файл main.go в папке tools

Я описываю синтаксис for в главе 6, но для этой главы мне просто нужны операторы кода, чтобы продемонстрировать, как работает отладчик. Скомпилируйте и выполните код с помощью команды go run. команда; вы получите следующий вывод:

```
Hello, Go
```

```
0
1
2
3
4
```

Использование отладчика

Чтобы запустить отладчик, выполните команду, показанную в листинге 3-12, в папке `tools`.

```
dlv debug main.go
```

Листинг 3-12 Запуск отладчика

Эта команда запускает текстовый клиент отладки, который поначалу может сбивать с толку, но становится чрезвычайно мощным, как только вы привыкнете к тому, как он работает. Первым шагом является создание точки останова, что делается путем указания местоположения в коде, как показано в листинге 3-13.

```
break bp1 main.main:3
```

Листинг 3-13 Создание точки останова

Команда `break` создает точку останова. Аргументы задают имя точки останова и расположение. Расположение можно указать по-разному, но расположение, используемое в листинге 3-13, определяет пакет, функцию в этом пакете и строку внутри этой функции, как показано на рисунке 3-6.

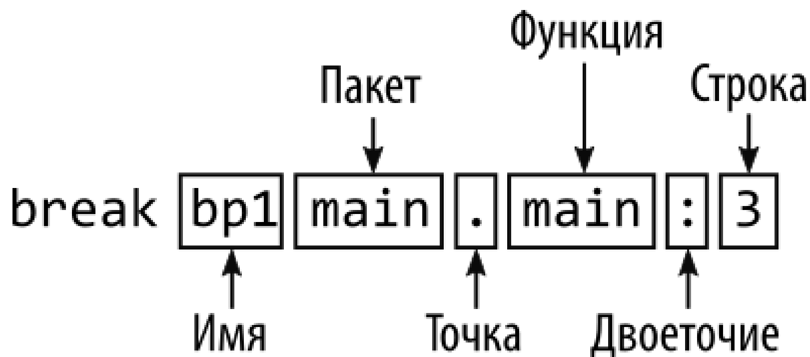

Рисунок 3-6 Указание расположения точки останова

Имя точки останова — `bp1`, а ее местоположение указывает на третью строку основной функции в основном пакете. Отладчик отображает следующее подтверждающее сообщение:

```
Breakpoint 1 set at 0x697716 for main.main()
c:/tools/main.go:8
```

Далее я собираюсь создать условие для точки останова, чтобы выполнение было остановлено только тогда, когда указанное выражение оценивается как `true` (истинное). Введите в отладчик команду, показанную в листинге 3-14, и нажмите клавишу Return.

```
condition bp1 i == 2
```

Листинг 3-14 Указание условия точки останова в отладчике

Аргументы команды `condition` задают точку останова и выражение. Эта команда сообщает отладчику, что точка останова с именем `bp1` должна остановить выполнение только тогда, когда выражение `i == 2` истинно. Чтобы начать выполнение, введите команду, показанную в листинге 3-15, и нажмите клавишу Return. The arguments for the `condition` command specify a breakpoint and an expression. This command tells the debugger that the breakpoint named `bp1` should halt execution only when the expression `i == 2` is true. To start execution, enter the command shown in Listing 3-15 and press Return.

```
continue
```

Листинг 3-15 Запуск выполнения в отладчике

Отладчик начинает выполнять код, выдавая следующий результат:

```
Hello, Go
0
1
```

Выполнение останавливается, когда выполняется условие, указанное в листинге 3-15, и отладчик отображает код и точку остановки выполнения, которую я выделил жирным шрифтом:

```
> [bp1] main.main() c:/tools/main.go:8 (hits goroutine(1):1
```

```
total:1) (PC: 0x207716)
     3: import "fmt"
     4:
     5: func main() {
     6:     fmt.Println("Hello, Go")
     7:     for i := 0; i < 5; i++ {
=>   8:         fmt.Println(i)
     9:     }
    10: }
```

Отладчик предоставляет полный набор команд для проверки и изменения состояния приложения, наиболее полезные из которых показаны в Таблице 3-2. (Полный набор команд, поддерживаемых отладчиком, см. на странице `https://github.com/go-delve/delve`.)

Таблица 3-2 Полезные команды состояния отладчика

Команда	Описание
`print <expr>`	Эта команда оценивает выражение и отображает результат. Его можно использовать для отображения значения (`print i`) или выполнить более сложный тест (`print i > 0`).
`set <variable> = <value>`	Эта команда изменяет значение указанной переменной.
`locals`	Эта команда выводит значения всех локальных переменных.
`whatis <expr>`	Эта команда выводит тип указанного выражения, например `whatis i`. Я описываю типы Go в главе 4.

Запустите команду, показанную в листинге 3-16, чтобы отобразить текущее значение переменной с именем `i`.

```
print i
```

Листинг 3-16 Печать значения в отладчике

Отладчик отображает ответ `2`, который является текущим значением переменной и соответствует условию, которое я указал для точки останова в листинге 3-16. Отладчик предоставляет полный набор команд для управления выполнением, наиболее полезные из которых показаны в Таблице 3-3.

Таблица 3-3 Полезные команды отладчика для управления выполнением

Команда	Описание
`continue`	Эта команда возобновляет выполнение приложения.
`next`	This command moves to the next statement.
`step`	Эта команда переходит в текущий оператор.
`stepout`	Эта команда выходит за пределы текущего оператора.
`restart`	Эта команда перезапускает процесс. Используйте команду `continue`, чтобы начать выполнение.
`exit`	Эта команда закрывает отладчик.

Введите команду `continue`, чтобы возобновить выполнение, что приведет к следующему выводу:

```
2
3
4
Process 3160 has exited with status 0
```

Условие, которое я указал для точки останова, больше не выполняется, поэтому программа работает до тех пор, пока не завершится. Используйте команду `exit`, чтобы выйти из отладчика и вернуться в командную строку.

Использование подключаемого модуля редактора Delve

Delve также поддерживается рядом подключаемых модулей редактора, которые создают возможности отладки на основе пользовательского интерфейса для Go. Полный список подключаемых модулей можно найти по адресу `https://github.com/go-delve/delve`, но один из лучших способов отладки Go/Delve предоставляется Visual Studio Code и устанавливается автоматически при установке языковых инструментов для Go.

Если вы используете Visual Studio Code, вы можете создавать точки останова, щелкая в поле редактора кода, и запускать отладчик с помощью команды «Запустить отладку» в меню «Выполнить».

Если вы получили сообщение об ошибке или вам было предложено выбрать среду, откройте файл `main.go` для редактирования, щелкните любой оператор кода в окне редактора и снова выберите команду «Запустить отладку».

Я не собираюсь подробно описывать процесс отладки с помощью Visual Studio Code или любого другого редактора, но на рисунке 3-7 показан отладчик после остановки выполнения в условной точке останова, воссоздающий пример командной строки из предыдущего раздела.

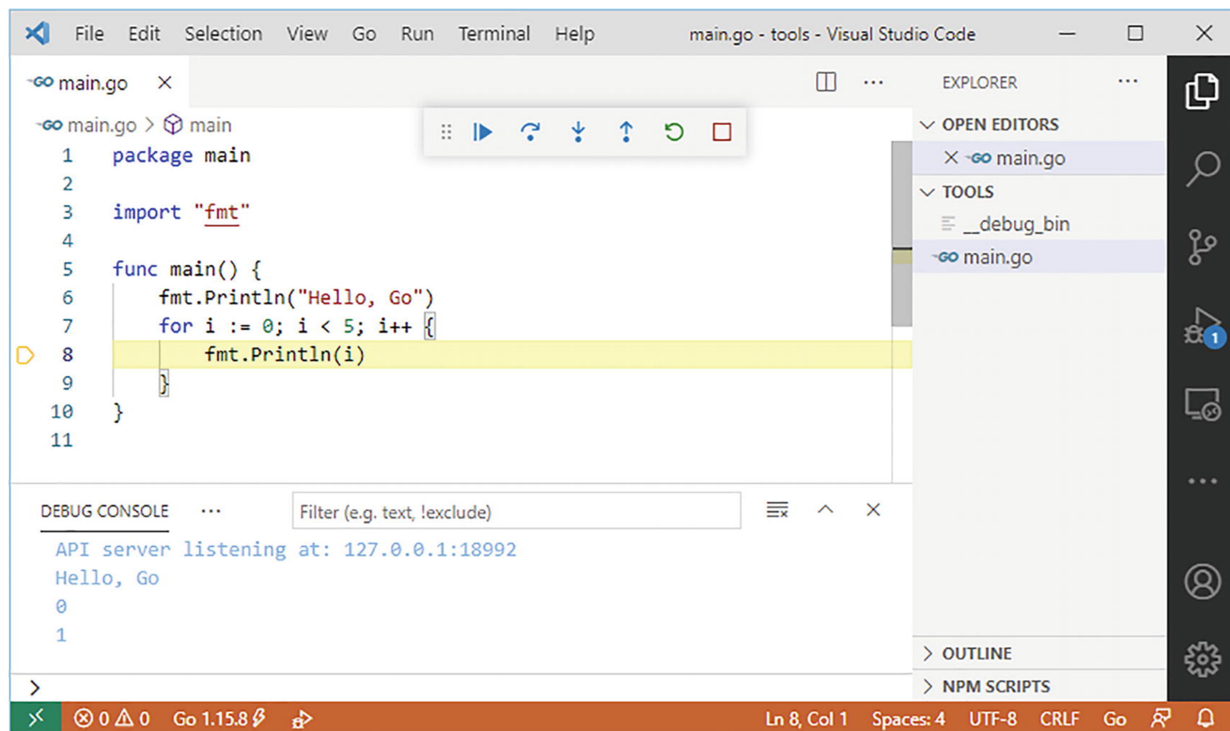

Рисунок 3-7 Использование подключаемого модуля редактора Delve

Линтинг Go-кода

Линтер — это инструмент, проверяющий файлы кода с помощью набора правил, описывающих проблемы, вызывающие путаницу, приводящие к неожиданным результатам или снижающие читабельность кода. Наиболее широко используемый линтер для Go называется `golint`, который применяет правила, взятые из двух источников. Первый — это документ Effective Go, созданный Google (`https://golang.org/doc/effective_go.html`), который содержит советы по написанию ясного и лаконичного кода Go. Второй источник — это коллекция комментариев из обзоров кода (`https://github.com/golang/go/wiki/CodeReviewComments`).

Проблема с golint заключается в том, что он не предоставляет параметров конфигурации и всегда будет применять все правила, что может привести к тому, что предупреждения, которые вам небезразличны, могут быть потеряны в длинном списке предупреждений для правил, которые вам не нужны. Я предпочитаю использовать revive пакет линтера, который является прямой заменой golint, но с поддержкой контроля применяемых правил. Чтобы установить пакет восстановления, откройте новую командную строку и выполните команду, показанную в листинге 3-17.

```
go install github.com/mgechev/revive@latest
```

Листинг 3-17 Установка пакета линтера

РАДОСТЬ И ПЕЧАЛЬ ЛИНТИНГА

Линтеры могут быть мощным инструментом во благо, особенно в команде разработчиков с разным уровнем навыков и опыта. Линтеры могут обнаруживать распространенные проблемы и незаметные ошибки, которые приводят к непредвиденному поведению или долгосрочным проблемам обслуживания. Мне нравится этот вид линтинга, и мне нравится запускать свой код в процессе линтинга после того, как я завершил основную функцию приложения или до того, как я передам свой код в систему контроля версий.

Но линтеры также могут быть инструментом разделения и борьбы, когда правила используются для обеспечения соблюдения личных предпочтений одного разработчика во всей команде. Обычно это делается под лозунгом «мнения». Логика в том, что разработчики тратят слишком много времени на споры о разных стилях кодирования, и всем лучше, если их заставят писать одинаково.

Мой опыт показывает, что разработчики просто найдут, о чем поспорить, и что навязывание стиля кода часто является просто предлогом, чтобы сделать предпочтения одного человека обязательными для всей команды разработчиков.

В этой главе я не использовал популярный пакет golint, потому что в нем нельзя отключить отдельные правила. Я уважаю твердое мнение разработчиков golint, но использование golint заставляет меня чувствовать, что у меня постоянный спор с кем-то, кого я даже

не знаю, что почему-то хуже, чем постоянный спор с одним разработчиком в команде, который расстраивается из-за отступов.

Мой совет — используйте линтинг экономно и сосредоточьтесь на проблемах, которые вызовут настоящие проблемы. Дайте отдельным разработчикам свободу самовыражения и сосредоточьтесь только на вопросах, которые имеют заметное влияние на проект. Это противоречит самоуверенному идеалу Go, но я считаю, что производительность не достигается рабским соблюдением произвольных правил, какими бы благими намерениями они ни были.

Использование линтера

Файл `main.go` настолько прост, что линтеру не составит труда его выделить. Добавьте операторы, показанные в листинге 3-18, которые являются допустимым кодом Go, который не соответствует правилам, применяемым линтером.

```
package main

import "fmt"

func main() {
    PrintHello()
    for i := 0; i < 5; i++ {
        PrintNumber(i)
    }
}

func PrintHello() {
    fmt.Println("Hello, Go")
}

func PrintNumber(number int) {
    fmt.Println(number)
}
```

Листинг 3-18 Добавление утверждений в файл main.go в папку tools

Сохраните изменения и используйте командную строку для запуска команды, показанной в листинге 3-19. (Как и в случае с командой `dlv`,

для запуска этой команды вам может потребоваться указать путь go/bin в вашей домашней папке.)

```
revive
```

Листинг 3-19 Запуск линтера

Линтер проверяет файл main.go и сообщает о следующей проблеме:

```
main.go:12:1: exported function PrintHello should have comment or be unexported
main.go:16:1: exported function PrintNumber should have comment or be unexported
```

Как я объясню в главе 12, функции, имена которых начинаются с заглавной буквы, считаются экспортируемыми и доступными для использования за пределами пакета, в котором они определены. По соглашению для экспортируемых функций предоставляется описательный комментарий. Линтер пометил факт отсутствия комментариев для функций PrintHello и PrintNumber. Листинг 3-20 добавляет комментарий к одной из функций.

```
package main

import "fmt"

func main() {
    PrintHello()
    for i := 0; i < 5; i++ {
        PrintNumber(i)
    }
}

func PrintHello() {
    fmt.Println("Hello, Go")
}

// This function writes a number using the fmt.Println function
func PrintNumber(number int) {
    fmt.Println(number)
}
```

Листинг 3-20 Добавление комментария в файл main.go в папке tools

Запустите команду `revive` еще раз; вы получите другую ошибку для функции `PrintNumber`:

```
main.go:12:1: exported function PrintHello should have comment or be unexported
main.go:16:1: comment on exported function PrintNumber should be of the form "PrintNumber ..."
```

Некоторые правила линтера специфичны по своим требованиям. Комментарий в листинге 3-20 не принимается, поскольку в Effective Go указано, что комментарии должны содержать предложение, начинающееся с имени функции, и должны давать краткий обзор назначения функции, как описано на https://golang.org/doc/effective_go.html#commentary. Листинг 3-21 исправляет комментарий, чтобы он следовал требуемой структуре.

```
package main

import "fmt"

func main() {
    PrintHello()
    for i := 0; i < 5; i++ {
        PrintNumber(i)
    }
}

func PrintHello() {
    fmt.Println("Hello, Go")
}

// PrintNumber writes a number using the fmt.Println function
func PrintNumber(number int) {
    fmt.Println(number)
}
```

Листинг 3-21 Редактирование комментария в файле main.go в папке

Запустите команду `revive` еще раз; линтер завершится без сообщений об ошибках для функции `PrintNumber`, хотя для функции

PrintHello все равно будет выдано предупреждение, поскольку у нее нет комментария.

ПОНИМАНИЕ ДОКУМЕНТАЦИИ GO

Причина, по которой линтер так строго относится к комментариям, заключается в том, что они используются командой go doc, которая генерирует документацию из комментариев исходного кода. Подробную информацию о том, как используется команда go doc, можно найти по адресу https://blog.golang.org/godoc, но вы можете запустить команду go doc -all в папке tools, чтобы быстро продемонстрировать, как она использует комментарии для документирования пакета.

Отключение правил линтера

Пакет revive можно настроить с помощью комментариев в файлах кода, отключив одно или несколько правил для разделов кода. В листинге 3-22 я использовал комментарии, чтобы отключить правило, вызывающее предупреждение для функции PrintNumber.

```
package main

import "fmt"

func main() {
    PrintHello()
    for i := 0; i < 5; i++ {
        PrintNumber(i)
    }
}

// revive:disable:exported

func PrintHello() {
    fmt.Println("Hello, Go")
}

// revive:enable:exported

// PrintNumber writes a number using the fmt.Println function
func PrintNumber(number int) {
```

```
    fmt.Println(number)
}
```

Листинг 3-22 Отключение правила Linter для функции в файле main.go в папке tools

Синтаксис, необходимый для управления линтером, таков: `revive`, за которым следует двоеточие, `enable` (включить) или `disable` (отключить) и, возможно, еще одно двоеточие и имя правила линтера. Так, например, комментарий `revive:disable:exported` не позволяет линтеру применить правило с именем `exported`, которое генерирует предупреждения. Комментарий `revive:disable:exported` включает правило, чтобы оно применялось к последующим операторам в файле кода.

Вы можете найти список правил, поддерживаемых линтером, по адресу https://github.com/mgechev/revive#available-rules. Кроме того, вы можете опустить имя правила из комментария, чтобы управлять применением всех правил.

Создание конфигурационного файла линтера

Использование комментариев к коду полезно, когда вы хотите подавить предупреждения для определенной области кода, но при этом применить правило в другом месте проекта. Если вы вообще не хотите применять правило, вы можете использовать файл конфигурации в TOML-формате. Добавьте в папку `tools` файл с именем `revive.toml`, содержимое которого показано в листинге 3-23.

> **Подсказка**
> Формат TOML предназначен специально для файлов конфигурации и описан на странице `https://toml.io/en`. Полный набор параметров настройки восстановления описан на странице https://github.com/mgechev/revive#configuration.

```
ignoreGeneratedHeader = false
severity = "warning"
confidence = 0.8
errorCode = 0
warningCode = 0

[rule.blank-imports]
```

```
[rule.context-as-argument]
[rule.context-keys-type]
[rule.dot-imports]
[rule.error-return]
[rule.error-strings]
[rule.error-naming]
#[rule.exported]
[rule.if-return]
[rule.increment-decrement]
[rule.var-naming]
[rule.var-declaration]
[rule.package-comments]
[rule.range]
[rule.receiver-naming]
[rule.time-naming]
[rule.unexported-return]
[rule.indent-error-flow]
[rule.errorf]
```

Листинг 3-23 Содержимое файла vanilla.toml в папке tools

Это конфигурация `revive` по умолчанию, описанная на https://github.com/mgechev/revive#recommended-configuration, за исключением того, что я поставил символ `#` перед записью, которая включает правило `exported`. В листинге 3-24 я удалил комментарии из файла `main.go`, которые больше не требуются для проверки линтера.

```
package main

import "fmt"

func main() {
    PrintHello()
    for i := 0; i < 5; i++ {
        PrintNumber(i)
    }
}

func PrintHello() {
    fmt.Println("Hello, Go")
}

func PrintNumber(number int) {
```

```
    fmt.Println(number)
}
```

Листинг 3-24 Удаление комментариев из файла main.go в папке tools

Чтобы использовать линтер с файлом конфигурации, выполните команду, показанную в листинге 3-25, в папке `tools`.

```
revive -config revive.toml
```

Листинг 3-25 Запуск линтера с конфигурационным файлом

Вывода не будет, потому что единственное правило, вызвавшее ошибку, отключено.

ЛИНТИНГ В РЕДАКТОРЕ КОДА

Некоторые редакторы кода автоматически поддерживают анализ кода. Например, если вы используете Visual Studio Code, анализ выполняется в фоновом режиме, а проблемы помечаются как предупреждения. Код линтера Visual Studio по умолчанию время от времени меняется; на момент написания статьи это `staticcheck`, который можно настроить, но ранее он был `golint`, а это не так.

Линтер легко заменить на `revive`, используя параметр настройки Preferences ➤ Extensions ➤ Go ➤ Lint Tool. Если вы хотите использовать пользовательский файл конфигурации, используйте параметр конфигурации Lint Flags, чтобы добавить флаг со значением `-config=./revive.toml`, который выберет файл `vanilla.toml`.

Исправление распространенных проблем в коде Go

Команда `go vet` идентифицирует операторы, которые могут быть ошибочными. В отличие от линтера, который часто фокусируется на вопросах стиля, команда `go vet` находит код, который компилируется, но, вероятно, не будет выполнять то, что задумал разработчик.

Мне нравится команда `go vet`, потому что она выявляет ошибки, которые не замечают другие инструменты, хотя анализаторы не замечают каждую ошибку и иногда выделяют код, который не является

проблемой. В листинге 3-26 я добавил в файл `main.go` оператор, намеренно вносящий ошибку в код.

```
package main

import "fmt"

func main() {
    PrintHello()
    for i := 0; i < 5; i++ {
        i = i
        PrintNumber(i)
    }
}

func PrintHello() {
    fmt.Println("Hello, Go")
}

func PrintNumber(number int) {
    fmt.Println(number)
}
```

Листинг 3-26 Добавление заявления в файл main.go в папке tools

Новый оператор присваивает переменной `i` саму себя, что разрешено компилятором Go, но, скорее всего, будет ошибкой. Чтобы проанализировать код, используйте командную строку для запуска команды, показанной в листинге 3-27, в папке `tools`.

```
go vet main.go
```

Листинг 3-27 Анализ кода

Команда `go vet` проверит операторы в файле `main.go` и выдаст следующее предупреждение:

```
# _/C_/tools
.\main.go:8:9: self-assignment of i to i
```

Предупреждения, выдаваемые командой `go vet`, указывают место в коде, где была обнаружена проблема, и предоставляют описание проблемы.

Команда `go vet` применяет к коду несколько анализаторов, и вы можете увидеть список анализаторов на странице https://golang.org/cmd/vet. Вы можете выбрать отдельные анализаторы для включения или отключения, но может быть трудно определить, какой анализатор сгенерировал конкретное сообщение. Чтобы выяснить, какой анализатор отвечает за предупреждение, запустите команду, показанную в листинге 3-28, в папке `tools`.

```
go vet -json main.go
```

Листинг 3-28 Идентификация анализатора

Аргумент `json` генерирует вывод в формате JSON, который группирует предупреждения по анализатору, например:

```
# _/C_/tools {
    "_/C_/tools": {
        "assign": [
            {
                "posn": "C:\\tools\\main.go:8:9",
                "message": "self-assignment of i to i"
            }
        ]
    }
}
```

Использование этой команды показывает, что анализатор с именем `assign` отвечает за предупреждение, сгенерированное для файла `main.go`. Когда имя известно, анализатор можно включить или отключить, как показано в листинге 3-29.

```
go vet -assign=false
go vet -assign
```

Листинг 3-29 Выбор анализаторов

Первая команда в листинге 3-29 запускает все анализаторы, кроме `assign`, анализатора, выдавшего предупреждение для оператора самоназначения. Вторая команда запускает только анализатор `assign`.

ПОНИМАНИЕ, ЧТО ДЕЛАЕТ КАЖДЫЙ АНАЛИЗАТОР

Может быть трудно понять, что ищет каждый анализатор `go vet`. Я считаю модульные тесты, которые команда Go написала для анализаторов, полезными, поскольку они содержат примеры искомых типов проблем. Тесты находятся на `https://github.com/golang/go/tree/master/src/cmd/vet/testdata`.

Некоторые редакторы, в том числе Visual Studio Code, отображают сообщения от `go vet` в окне редактора, как показано на рисунке 3-8, что позволяет легко воспользоваться преимуществами анализа без необходимости явного запуска команды.

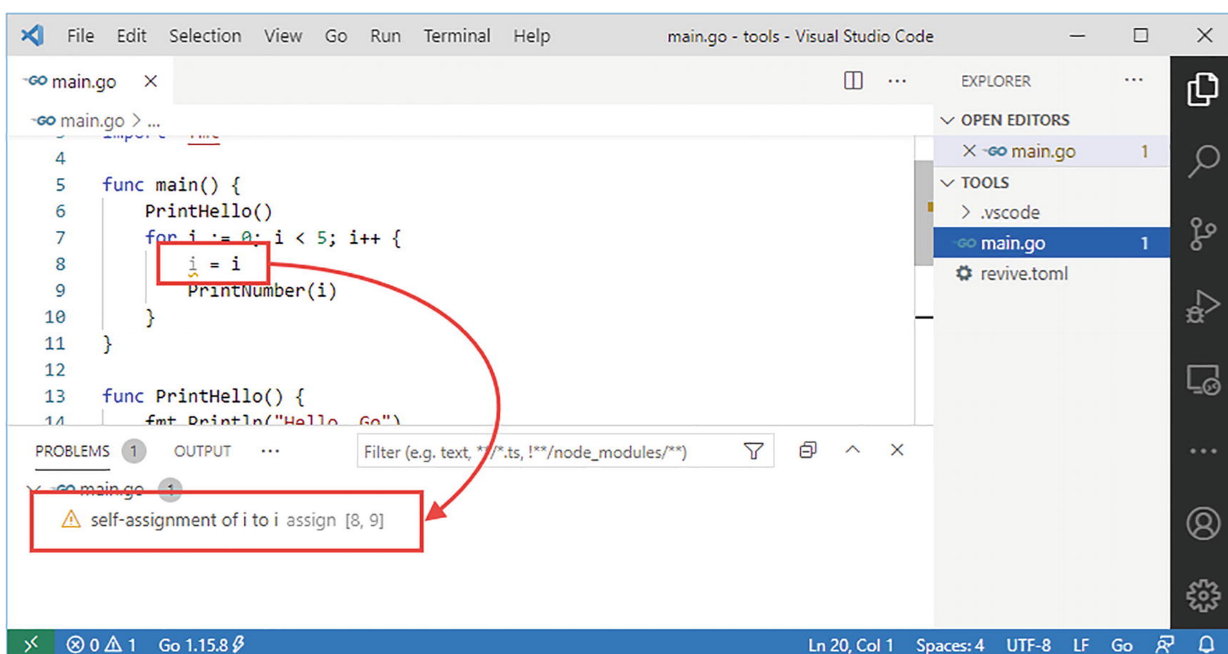

Рисунок 3-8 Потенциальная проблема с кодом в редакторе кода

Visual Studio Code помечает ошибку в окне редактора и отображает подробности в окне «Проблемы». Анализ с помощью `go vet` включен по умолчанию, вы можете отключить эту функцию с помощью элемента конфигурации Настройки ➤ Расширения ➤ Go ➤ Vet On Save.

Форматирование кода Go

Команда `go fmt` форматирует файлы исходного кода Go для согласованности. Нет параметров конфигурации для изменения

форматирования, применяемого командой go fmt, которая преобразует код в стиль, указанный командой разработчиков Go. Наиболее очевидными изменениями являются использование табуляции для отступов, последовательное выравнивание комментариев и устранение ненужных точек с запятой. В листинге 3-30 показан код с несогласованными отступами, смещенными комментариями и точками с запятой там, где они не требуются.

> **Подсказка**
> Вы можете обнаружить, что ваш редактор автоматически форматирует код, когда он вставляется в окно редактора или когда файл сохраняется.

```
package main

import "fmt"

func main() {
    PrintHello  ()
           for i := 0; i < 5; i++ { // loop with a counter
           PrintHello(); // print out a message
            PrintNumber(i); // print out the counter
   }
}

func PrintHello  () {
      fmt.Println("Hello, Go");
}

func PrintNumber  (number int) {
  fmt.Println(number);
}
```

Листинг 3-30 Создание задач форматирования в файле main.go в папке tools

Запустите команду, показанную в листинге 3-31, в папке tools, чтобы переформатировать код.

```
go fmt main.go
```

Листинг 3-31 Форматирование исходного кода

Средство форматирования удалит точки с запятой, отрегулирует отступ и выровняет комментарии, создав следующий отформатированный код:

```
package main

import "fmt"

func main() {
    PrintHello()
    for i := 0; i < 5; i++ { // loop with a counter
        PrintHello()   // print out a message
        PrintNumber(i) // print out the counter
    }
}

func PrintHello() {
    fmt.Println("Hello, Go")
}

func PrintNumber(number int) {
    fmt.Println(number)
}
```

Я не использовал `go fmt` для примеров в этой книге, потому что использование вкладок вызывает проблемы с макетом на печатной странице. Я должен использовать пробелы для отступов, чтобы код выглядел должным образом при печати книги, и они заменяются вкладками с помощью `go fmt`.

Резюме

В этой главе я представил инструменты, которые используются для разработки Go. Я объяснил, как компилировать и выполнять исходный код, как отлаживать код Go, как использовать линтер, как форматировать исходный код и как находить распространенные проблемы. В следующей главе я начну описывать возможности языка Go, начиная с основных типов данных.

4. Основные типы, значения и указатели

В этой главе я начинаю описывать язык Go, сосредоточившись на основных типах данных, прежде чем перейти к тому, как они используются для создания констант и переменных. Я также представляю поддержку Go для указателей. Указатели могут быть источником путаницы, особенно если вы переходите к Go с таких языков, как Java или C#, и я описываю, как работают указатели Go, демонстрирую, почему они могут быть полезны, и объясняю, почему их не следует бояться.

Функции, предоставляемые любым языком программирования, предназначены для совместного использования, что затрудняет их постепенное внедрение. Некоторые примеры в этой части книги основаны на функциях, описанных ниже. Эти примеры содержат достаточно подробностей, чтобы обеспечить контекст, и включают ссылки на ту часть книги, где можно найти дополнительную информацию. В Таблице 4-1 показаны основные функции Go в контексте.

Таблица 4-1 Помещение базовых типов, значений и указателей в контекст

Вопрос	**Ответ**
Кто они такие?	Типы данных используются для хранения основных значений, общих для всех программ, включая числа, строки и значения `true`/`false`. Эти типы данных можно использовать для определения постоянных и переменных значений. Указатели — это особый тип данных, в котором хранится адрес памяти.
Почему они полезны?	Базовые типы данных полезны сами по себе для хранения значений, но они также являются основой, на которой могут быть определены более сложные типы данных, как я объясню в главе 10. Указатели полезны, потому что они позволяют программисту решить, является ли значение следует копировать при использовании.
Как они используются?	Базовые типы данных имеют собственные имена, такие как `int` и `float64`, и могут использоваться с ключевыми словами `const` и `var`. Указатели создаются с помощью оператора адреса `&`.

Вопрос	Ответ
Есть ли подводные камни или ограничения?	Go не выполняет автоматическое преобразование значений, за исключением особой категории значений, известных как *нетипизированные константы*.
Есть ли альтернативы?	Нет альтернатив основным типам данных, которые используются при разработке Go.

Таблица 4-2 резюмирует главу.

Таблица 4-2 Краткое содержание главы

Проблема	Решение	Листинг
Использовать значение напрямую	Используйте значение литерала	6
Определение константы	Используйте ключевое слово `const`	7, 10
Определите константу, которую можно преобразовать в связанный тип данных	Создать нетипизированную константу	8, 9, 11
Определить переменную	Используйте ключевое слово `var` или используйте короткий синтаксис объявления	12-21
Предотвращение ошибок компилятора для неиспользуемой переменной	Используйте пустой идентификатор	22, 23
Определить указатель	Используйте оператор адреса	24, 25, 29–30
Значение по указателю	Используйте звездочку с именем переменной-указателя	26–28, 31

Подготовка к этой главе

Чтобы подготовиться к этой главе, откройте новую командную строку, перейдите в удобное место и создайте каталог с именем `basicFeatures`. Запустите команду, показанную в листинге 4-1, чтобы создать файл `go.mod` для проекта.

```
go mod init basicfeatures
```

Листинг 4-1 Создание проекта примера

Добавьте файл с именем `main.go` в папку `basicFeatures` с содержимым, показанным в листинге 4-2.

Подсказка
Вы можете загрузить пример проекта для этой главы — и для всех остальных глав этой книги — с `https://github.com/apress/pro-go`. См. главу 2 о том, как получить помощь, если у вас возникнут проблемы с запуском примеров.

```
package main

import (
    "fmt"
    "math/rand"
)

func main() {
    fmt.Println(rand.Int())
}
```

Листинг 4-2 Содержимое файла main.go в папке basicFeatures

Используйте командную строку для запуска команды, показанной в листинге 4-3, в папке `basicFeatures`.

```
go run .
```

Листинг 4-3 Запуск примера проекта

Код в файле `main.go` будет скомпилирован и выполнен, что приведет к следующему результату:

```
5577006791947779410
```

Вывод кода всегда будет одним и тем же значением, даже если оно создается пакетом случайных чисел, как я объясню в главе 18.

Использование стандартной библиотеки Go

Go предоставляет широкий набор полезных функций через свою стандартную библиотеку — этот термин используется для описания встроенного API. Стандартная библиотека Go представлена в виде

набора *пакетов*, являющихся частью установщика Go, используемого в главе 1.

Я описываю способ создания и использования пакетов Go в главе 12, но некоторые примеры основаны на пакетах из стандартной библиотеки, и важно понимать, как они используются.

Каждый пакет в стандартной библиотеке объединяет набор связанных функций. Код в листинге 4-2 использует два пакета: пакет `fmt` предоставляет возможности для форматирования и записи строк, а пакет `math/rand` работает со случайными числами.

Первым шагом в использовании пакета является определение оператора `import`. Рисунок 4-1 иллюстрирует оператор импорта, используемый в листинге 4-2.

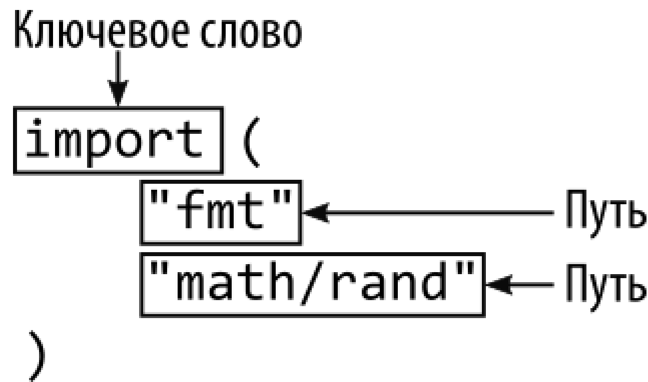

Рисунок 4-1 Импорт пакета

В операторе импорта есть две части: ключевое слово `import` и пути к пакетам. Пути сгруппированы в круглых скобках, если импортируется более одного пакета.

Оператор `import` создает ссылку на пакет, через которую можно получить доступ к функциям, предоставляемым пакетом. Имя ссылки на пакет — это последний сегмент пути к пакету. Путь к пакету `fmt` состоит только из одного сегмента, поэтому ссылка на пакет будет `fmt`. В пути `math/rand` есть два сегмента — `math` и `rand`, поэтому ссылка на пакет будет `rand`. (Я объясню, как выбрать собственное имя ссылки на пакет, в главе 12.)

Пакет `fmt` определяет функцию `Println`, которая записывает значение в стандартный вывод, а пакет `math/rand` определяет функцию `Int`, которая генерирует случайное целое число. Чтобы получить доступ к этим функциям, я использую их ссылку на пакет, за

которой следует точка и затем имя функции, как показано на рисунке 4-2.

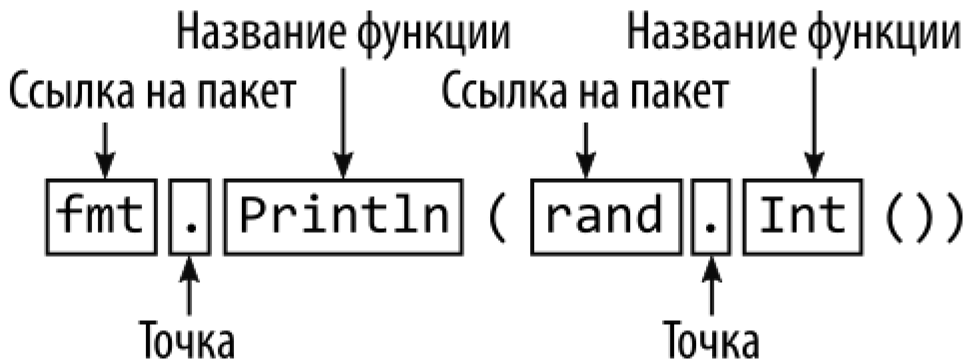

Рисунок 4-2 Использование ссылки на пакет

> **Подсказка**
> Список пакетов стандартной библиотеки Go доступен по адресу `https://golang.org/pkg`. Наиболее полезные пакеты описаны во второй части.

Связанная с этим функция, предоставляемая пакетом `fmt`, — это возможность составлять строки путем объединения статического содержимого со значениями данных, как показано в листинге 4-4.

```
package main

import (
    "fmt"
    "math/rand"
)

func main() {
    fmt.Println("Value:", rand.Int())
}
```

Листинг 4-4 Составление строки в файле main.go в папке basicFeatures

Ряд значений, разделенных запятыми, переданных в функцию `Println`, объединяются в одну строку, которая затем записывается в стандартный вывод. Чтобы скомпилировать и выполнить код,

используйте командную строку для запуска команды, показанной в листинге 4-5, в папке basicFeatures.

```
go run .
```

Листинг 4-5 Запуск примера проекта

Код в файле main.go будет скомпилирован и выполнен, что приведет к следующему результату:

```
Value: 5577006791947779410
```

Есть более полезные способы составления строк, которые я описываю в второй части, но это простой и полезный для меня способ предоставления вывода в примерах.

Понимание основных типов данных

Go предоставляет набор основных типов данных, которые описаны в Таблице 4-3. В следующих разделах я опишу эти типы и объясню, как они используются. Эти типы являются основой разработки Go, и многие характеристики этих типов будут знакомы из других языков.

Таблица 4-3 Основные типы данных Go

Имя	Описание
int	Этот тип представляет целое число, которое может быть положительным или отрицательным. Размер типа int зависит от платформы и может быть либо 32, либо 64 бита. Существуют также целые типы, которые имеют определенный размер, например int8, int16, int32 и int64, но следует использовать тип int, если вам не нужен определенный размер.
uint	Этот тип представляет положительное целое число. Размер типа uint зависит от платформы и может составлять 32 или 64 бита. Существуют также целочисленные типы без знака, которые имеют определенный размер, например uint8, uint16, uint32 и uint64, но следует использовать тип uint, если вам не нужен определенный размер.
byte	Этот тип является псевдонимом для uint8 и обычно используется для представления байта данных.
float32, float64	Эти типы представляют числа с дробью. Эти типы выделяют 32 или 64 бита для хранения значения.
complex64, complex128	Эти типы представляют числа, которые имеют действительные и мнимые компоненты. Эти типы выделяют 64 или 128 бит для хранения значения.

Имя	Описание
`bool`	Этот тип представляет булеву истину со значениями `true` и `false`.
`string`	Этот тип представляет собой последовательность символов.
`rune`	Этот тип представляет одну кодовую точку Unicode. Юникод сложен, но, грубо говоря, это представление одного символа. Тип `rune` является псевдонимом для `int32`.

КОМПЛЕКСНЫЕ ЧИСЛА В GO

Как отмечено в Таблице 4-3, в Go есть встроенная поддержка комплексных чисел, у которых есть действительные и мнимые части. Я помню, как узнал о комплексных числах в школе и быстро забыл о них, пока не начал читать спецификацию языка Go. В этой книге я не описываю использование комплексных чисел, потому что они используются только в определенных областях, таких как электротехника. Вы можете узнать больше о комплексных числах на странице `https://en.wikipedia.org/wiki/Complex_number`.

Понимание литеральных значений

Значения Go могут быть выражены буквально, где значение определяется непосредственно в файле исходного кода. Обычное использование литеральных значений включает операнды в выражениях и аргументы функций, как показано в листинге 4-6.

Подсказка

Обратите внимание, что я закомментировал пакет `math/rand` из оператора `import` в листинге 4-6. Ошибка в Go — импортировать пакет, который не используется.

```
package main

import (
    "fmt"
    //"math/rand"
)

func main() {
    fmt.Println("Hello, Go")
```

```
    fmt.Println(20 + 20)
    fmt.Println(20 + 30)
}
```

Листинг 4-6 Использование литеральных значений в файле main.go в папке basicFeatures

Первый оператор в функции `main` использует строковый литерал, который обозначается двойными кавычками, в качестве аргумента функции `fmt.Println`. Другие операторы используют литеральные значения `int` в выражениях, результаты которых используются в качестве аргумента функции `fmt.Println`. Скомпилируйте и выполните код, и вы увидите следующий вывод:

```
Hello, Go
40
50
```

Вам не нужно указывать тип при использовании буквального значения, потому что компилятор выведет тип на основе способа выражения значения. Для быстрого ознакомления в Таблице 4-4 приведены примеры литеральных значений для основных типов.

Таблица 4-4 Примеры литерального значения

Тип	Примеры
`int`	`20`, `-20`. Значения также могут быть выражены в шестнадцатеричной (`0x14`), восьмеричной (`0o24`) и двоичной записи (`0b0010100`).
`unit`	Нет литералов `uint`. Все литеральные целые числа обрабатываются как значения `int`.
`byte`	Байтовых литералов нет. Байты обычно выражаются как целочисленные литералы (например, `101`) или литералы выполнения (`'e'`), поскольку тип `byte` является псевдонимом для типа `uint8`.
`float64`	`20.2`, `-20.2`, `1.2e10`, `1.2e-10`. Значения также могут быть выражены в шестнадцатеричном представлении (`0x2p10`), хотя показатель степени выражается десятичными цифрами.
`bool`	`true`, `false`.
`string`	`"Hello"`. Последовательности символов, экранированные обратной косой чертой, интерпретируются, если значение заключено в двойные кавычки (`"Hello\n"`). Escape-последовательности не интерпретируются, если значение заключено в обратные кавычки (`` `Hello\n` ``).
`rune`	`'A'`, `'\n'`, `'\u00A5'`, `'¥'`.Символы, глифы и escape-последовательности заключаются в одинарные кавычки (символ `'`).

Использование констант

Константы — это имена для определенных значений, что позволяет использовать их многократно и согласованно. В Go есть два способа определения констант: типизированные константы и нетипизированные константы. В листинге 4-7 показано использование типизированных констант.

```
package main

import (
    "fmt"
    //"math/rand"
)

func main() {
    const price float32 = 275.00
    const tax float32 = 27.50
    fmt.Println(price + tax)
}
```

Листинг 4-7 Определение типизированных констант в файле main.go в папке basicFeatures

Типизированные константы определяются с помощью ключевого слова `const`, за которым следует имя, тип и присвоенное значение, как показано на рисунке 4-3.

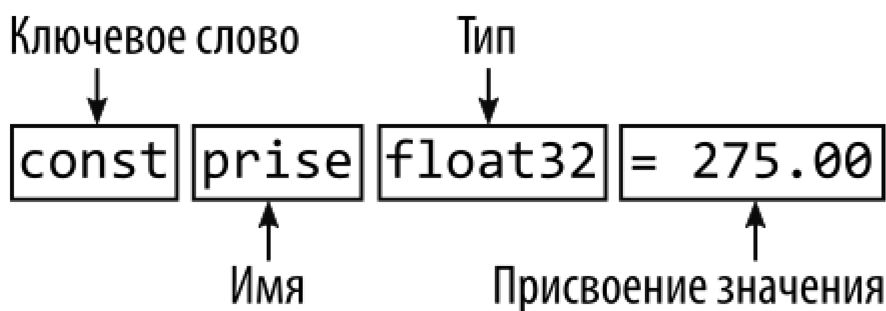

Рисунок 4-3 Определение типизированной константы

Этот оператор создает float32 именованную константу `price`, значение которой равно `275.00`. Код в листинге 4-7 создает две константы и использует их в выражении, которое передается функции

fmt.Println. Скомпилируйте и запустите код, и вы получите следующий вывод:

```
302.5
```

Понимание нетипизированных констант

Go имеет строгие правила в отношении типов данных и не выполняет автоматических преобразований типов, что может усложнить общие задачи программирования, как показано в листинге 4-8.

```
package main

import (
    "fmt"
    //"math/rand"
)

func main() {
    const price float32 = 275.00
    const tax float32 = 27.50
    const quantity int = 2
    fmt.Println("Total:", quantity * (price + tax))
}
```

Листинг 4-8 Смешивание типов данных в файле main.go в папке basicFeatures

Тип новой константы — int, что является подходящим выбором, например, для количества, которое может представлять только целое количество продуктов. Константа используется в выражении, переданном функции fmt.Println для расчета общей цены. Но компилятор сообщает о следующей ошибке при компиляции кода:

```
.\main.go:12:26: invalid operation: quantity * (price + tax)
(mismatched types int and float32)
```

Большинство языков программирования автоматически преобразовали бы типы, чтобы можно было вычислить выражение, но более строгий подход Go означает, что типы int и float32 нельзя смешивать. Функция нетипизированных констант упрощает работу с

константами, поскольку компилятор Go будет выполнять ограниченное автоматическое преобразование, как показано в листинге 4-9.

```
package main

import (
    "fmt"
    //"math/rand"
)

func main() {
    const price float32 = 275.00
    const tax float32 = 27.50
    const quantity = 2
    fmt.Println("Total:", quantity * (price + tax))
}
```

Листинг 4-9 UИспользование нетипизированной константы в файле main.go в папке basicFeatures

Нетипизированная константа определяется без типа данных, как показано на рисунок 4-4.

Рисунок 4-4 Определение нетипизированной константы

Отсутствие типа при определении константы `quantity` сообщает компилятору Go, что он должен быть более гибким в отношении типа константы. Когда выражение, переданное функции `fmt.Println`, оценивается, компилятор Go преобразует значение `quantity` в `float32`. Скомпилируйте и выполните код, и вы получите следующий вывод:

```
Total: 605
```

Нетипизированные константы будут преобразованы, только если значение может быть представлено в целевом типе. На практике это означает, что вы можете смешивать нетипизированные целые и числовые значения с плавающей запятой, но преобразования между другими типами данных должны выполняться явно, как я описываю в главе 5.

ПОНИМАНИЕ IOTA

Ключевое слово iota можно использовать для создания серии последовательных нетипизированных целочисленных констант без необходимости присваивать им отдельные значения. Вот пример iota:

```
...
const (
    Watersports = iota
    Soccer
    Chess
)
...
```

Этот шаблон создает серию констант, каждой из которых присваивается целочисленное значение, начиная с нуля. Вы можете увидеть примеры iota в третьей части.

Определение нескольких констант с помощью одного оператора

Один оператор может использоваться для определения нескольких констант, как показано в листинге 4-10.

```
package main

import (
    "fmt"
    //"math/rand"
)

func main() {
    const price, tax float32 = 275, 27.50
```

```
    const quantity, inStock = 2, true
    fmt.Println("Total:", quantity * (price + tax))
    fmt.Println("In stock: ", inStock)
}
```

Листинг 4-10 Определение нескольких констант в файле main.go в папке basicFeatures

За ключевым словом `const` следует список имен, разделенных запятыми, знак равенства и список значений, разделенных запятыми, как показано на рисунке 4-5. Если указан тип, все константы будут созданы с этим типом. Если тип опущен, то создаются нетипизированные константы, и тип каждой константы будет выведен из ее значения.

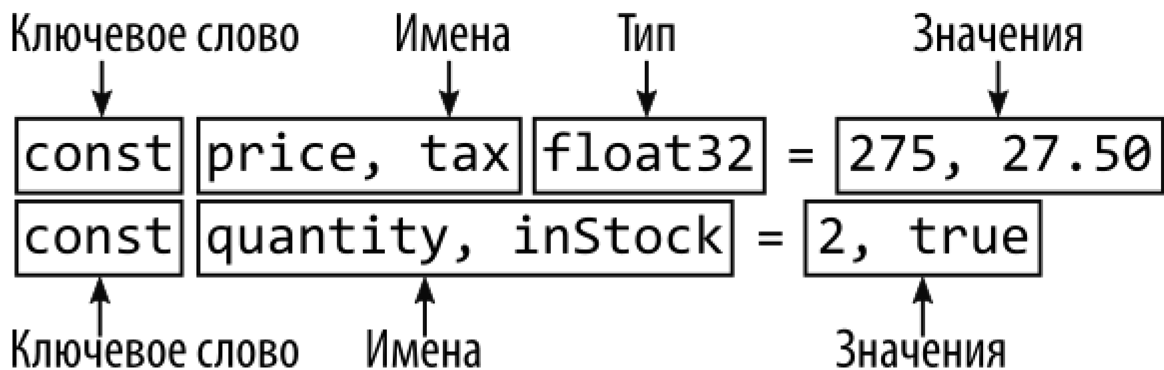

Рисунок 4-5 Определение нескольких констант

Компиляция и выполнение кода из листинга 4-10 приводит к следующему результату:

```
Total: 605
In stock:  true
```

Пересмотр литеральных значений

Нетипизированные константы могут показаться странной функцией, но они значительно облегчают работу с Go, и вы обнаружите, что полагаетесь на эту функцию, часто не осознавая этого, потому что литеральные значения — это нетипизированные константы, а это означает, что вы можете использовать литеральные значения в выражениях. и полагайтесь на компилятор для обработки несоответствующих типов, как показано в листинге 4-11.

```
package main

import (
    "fmt"
    //"math/rand"
)

func main() {
    const price, tax float32 = 275, 27.50
    const quantity, inStock = 2, true
    fmt.Println("Total:", 2 * quantity * (price + tax))
    fmt.Println("In stock: ", inStock)
}
```

Листинг 4-11 Использование литерального значения в файле main.go в папке basicFeatures

Выделенное выражение использует буквальное значение `2`, которое является значением `int`, как описано в Таблице 4-4, вместе с двумя значениями `float32`. Поскольку значение `int` может быть представлено как `float32`, значение будет преобразовано автоматически. При компиляции и выполнении этот код выдает следующий результат:

```
Total: 1210
In stock:  true
```

Использование переменных

Переменные определяются с помощью ключевого слова `var`, и, в отличие от констант, значение, присвоенное переменной, можно изменить, как показано в листинге 4-12.

```
package main

import "fmt"

func main() {
    var price float32 = 275.00
    var tax float32 = 27.50
    fmt.Println(price + tax)
    price = 300
```

```
    fmt.Println(price + tax)
}
```

Листинг 4-12 Использование констант в файле main.go в папке basicFeatures

Переменные объявляются с использованием ключевого слова `var`, имени, типа и присвоения значения, как показано на рисунке 4-6.

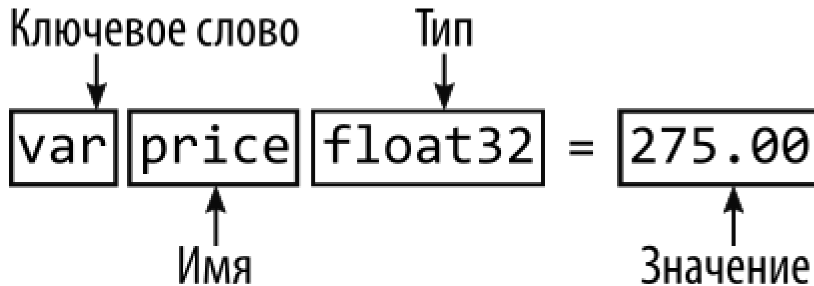

Рисунок 4-6 Определение перменной

Листинг 4-12 определяет переменные `price` и `tax`, которым присвоены значения `float32`. Новое значение переменной цены присваивается с помощью знака равенства, который является оператором присваивания Go, как показано на рисунке 4-7. (Обратите внимание, что я могу присвоить значение 300 переменной с плавающей запятой. Это потому, что буквальное значение `300` является нетипизированной константой, которая может быть представлена как значение `float32`.)

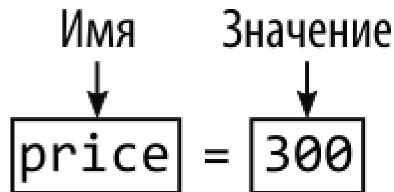

Рисунок 4-7 Присвоение нового значения переменной

Код в листинге 4-12 записывает две строки в стандартный вывод с помощью функции `fmt.Println`, производя следующий вывод после компиляции и выполнения кода:

```
302.5
327.5
```

Пропуск типа данных переменной

Компилятор Go может вывести тип переменных на основе начального значения, что позволяет опустить тип, как показано в листинге 4-13.

```
package main

import "fmt"

func main() {
    var price = 275.00
    var price2 = price
    fmt.Println(price)
    fmt.Println(price2)
}
```

Листинг 4-13 Пропуск типа переменной в файле main.go в папке basicFeatures

Переменная определяется с помощью ключевого слова `var`, имени и присваивания значения, но тип опускается, как показано на рисунке 4-8. Значение переменной может быть установлено с использованием буквального значения или имени константы или другой переменной. В листинге значение переменной `price` устанавливается с использованием литерального значения, а значение `price2` устанавливается равным текущему значению `price`.

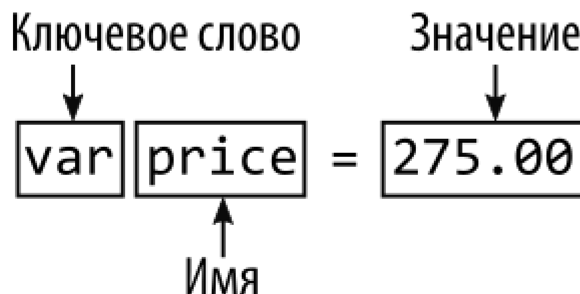

Рисунок 4-8 Определение переменной без указания типа

Компилятор выведет тип из значения, присвоенного переменной. Компилятор проверит буквальное значение, присвоенное `price`, и выведет его тип как `float64`, как описано в Таблице 4-4. Тип `price2` также будет выведен как `float64`, поскольку его значение устанавливается с использованием значения цены. Код в листинге 4-13 выдает следующий результат при компиляции и выполнении:

```
275
```

```
275
```

Отсутствие типа не имеет такого же эффекта для переменных, как для констант, и компилятор Go не позволит смешивать разные типы, как показано в листинге 4-14.

```
package main

import "fmt"

func main() {
    var price = 275.00
    var tax float32 = 27.50
    fmt.Println(price + tax)
}
```

Листинг 4-14 Смешивание типов данных в файле main.go в папке basicFeatures

Компилятор всегда будет определять тип буквенных значений с плавающей запятой как `float64`, что не соответствует типу `float32` переменной `tax`. Строгое соблюдение типов в Go означает, что компилятор выдает следующую ошибку при компиляции кода:

```
.\main.go:10:23: invalid operation: price + tax (mismatched
types float64 and float32)
```

Чтобы использовать переменные `price` и `tax` в одном выражении, они должны иметь один и тот же тип или быть конвертируемыми в один и тот же тип. Я объясню различные способы преобразования типов в главе 5.

Пропуск присвоения значения переменной

Переменные могут быть определены без начального значения, как показано в листинге 4-15.

```
package main

import "fmt"

func main() {
    var price float32
```

```
    fmt.Println(price)
    price = 275.00
    fmt.Println(price)
}
```

Листинг 4-15 Определение переменной без начального значения в файле main.go в папке basicFeatures

Переменные определяются с помощью ключевого слова `var`, за которым следуют имя и тип, как показано на рисунке 4-9. Тип нельзя опустить, если нет начального значения.

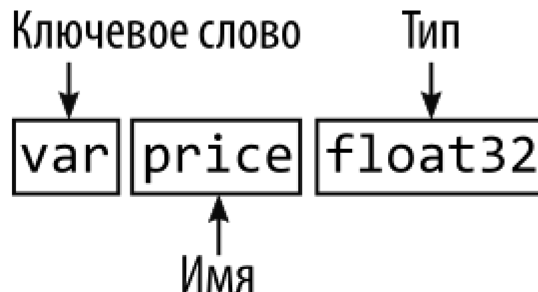

Рисунок 4-9 Определение переменной без начального значения в файле main.go в папке basicFeatures

Переменным, определенным таким образом, присваивается *нулевое значение* для указанного типа, как описано в Таблице 4-5.

Таблица 4-5 Нулевые значения для основных типов данных

Type	Zero Value
`int`	`0`
`unit`	`0`
`byte`	`0`
`float64`	`0`
`bool`	`false`
`string`	`""` (пустая строка)
`rune`	`0`

Нулевое значение для числовых типов равно нулю, что можно увидеть, скомпилировав и выполнив код. Первое значение, отображаемое в выходных данных, — это нулевое значение, за

которым следует значение, назначенное явно в последующем операторе:

```
0
275
```

Определение нескольких переменных с помощью одного оператора

Один оператор может использоваться для определения нескольких переменных, как показано в листинге 4-16.

```
package main

import "fmt"

func main() {
    var price, tax = 275.00, 27.50
    fmt.Println(price + tax)
}
```

Листинг 4-16 Определение переменных в файле main.go в папке basicFeatures

Это тот же подход, который используется для определения констант, и начальное значение, присвоенное каждой переменной, используется для определения ее типа. Тип должен быть указан, если начальные значения не присвоены, как показано в листинге 4-17, и все переменные будут созданы с использованием указанного типа и им будет присвоено нулевое значение.

```
package main

import "fmt"

func main() {
    var price, tax float64
    price = 275.00
    tax = 27.50
    fmt.Println(price + tax)
}
```

Листинг 4-17 Определение переменных без начальных значений в файле main.go в папке basicFeatures

Листинг 4-16 и листинг 4-17 дают одинаковый результат при компиляции и выполнении:

```
302.5
```

Использование краткого синтаксиса объявления переменных

Краткое объявление переменной обеспечивает сокращение для объявления переменных, как показано в листинге 4-18.

```
package main

import "fmt"

func main() {
    price := 275.00
    fmt.Println(price)
}
```

Листинг 4-18 Использование синтаксиса краткого объявления переменных в файле main.go в папке basicFeatures

В сокращенном синтаксисе указывается имя переменной, двоеточие, знак равенства и начальное значение, как показано на рисунке 4-10. Ключевое слово `var` не используется, и тип данных не может быть указан.

Рисунок 4-10 Синтаксис короткого объявления переменных

Код в листинге 4-18 выдает следующий результат после компиляции и выполнения кода:

```
275
```

Несколько переменных могут быть определены с помощью одного оператора путем создания списков имен и значений, разделенных запятыми, как показано в листинге 4-19.

```
package main

import "fmt"

func main() {
    price, tax, inStock := 275.00, 27.50, true
    fmt.Println("Total:", price + tax)
    fmt.Println("In stock:", inStock)
}
```

Листинг 4-19 Определение нескольких переменных в файле main.go в папке basicFeatures

В сокращенном синтаксисе типы не указаны, что означает, что можно создавать переменные разных типов, полагаясь на то, что компилятор выведет типы из значений, присвоенных каждой переменной. Код в листинге 4-19 выдает следующий результат при компиляции и выполнении:

```
Total: 302.5
In stock: true
```

Синтаксис короткого объявления переменных можно использовать только внутри функций, таких как `main` функция в листинге 4-19. Функции Go подробно описаны в главе 8.

Использование краткого синтаксиса переменных для переопределения переменных

Go обычно не позволяет переопределять переменные, но делает ограниченное исключение, когда используется короткий синтаксис. Чтобы продемонстрировать поведение по умолчанию, в листинге 4-20 ключевое слово `var` используется для определения переменной с тем же именем, что и уже существующая в той же функции.

```
package main
```

```
import "fmt"

func main() {
    price, tax, inStock := 275.00, 27.50, true
    fmt.Println("Total:", price + tax)
    fmt.Println("In stock:", inStock)

    var price2, tax = 200.00, 25.00
    fmt.Println("Total 2:", price2 + tax)
}
```

Листинг 4-20 Переопределение переменной в файле main.go в папке basicFeatures

Первый новый оператор использует ключевое слово `var` для определения переменных с именами `price2` и `tax`. В функции `main` уже есть переменная с именем `tax`, что вызывает следующую ошибку при компиляции кода:

```
.\main.go:10:17: tax redeclared in this block
```

Однако переопределение переменной разрешено, если используется короткий синтаксис, как показано в листинге 4-21, если хотя бы одна из других определяемых переменных еще не существует и тип переменной не изменяется.

```
package main

import "fmt"

func main() {
    price, tax, inStock := 275.00, 27.50, true
    fmt.Println("Total:", price + tax)
    fmt.Println("In stock:", inStock)

    price2, tax := 200.00, 25.00
    fmt.Println("Total 2:", price2 + tax)
}
```

Листинг 4-21 Использование краткого синтаксиса в файле main.go в папке basicFeatures

Скомпилируйте и запустите проект, и вы увидите следующий вывод:

```
Total: 302.5
In stock: true
Total 2: 225
```

Использование пустого идентификатора

В Go запрещено определять переменную и не использовать ее, как показано в листинге 4-22.

```
package main

import "fmt"

func main() {
     price, tax, inStock, discount := 275.00, 27.50, true,
true
    var salesPerson = "Alice"
    fmt.Println("Total:", price + tax)
    fmt.Println("In stock:", inStock)
}
```

Листинг 4-22 Определение неиспользуемых переменных в файле main.go в папке basicFeatures

В листинге определены переменные с именами `discount` и `salesperson`, ни одна из которых не используется в остальной части кода. При компиляции кода сообщается следующая ошибка:

```
.\main.go:6:26: discount declared but not used
.\main.go:7:9: salesPerson declared but not used
```

Один из способов решить эту проблему — удалить неиспользуемые переменные, но это не всегда возможно. Для таких ситуаций Go предоставляет *пустой идентификатор*, который используется для обозначения значения, которое не будет использоваться, как показано в листинге 4-23.

```
package main

import "fmt"
```

```
func main() {
    price, tax, inStock, _ := 275.00, 27.50, true, true
    var _ = "Alice"
    fmt.Println("Total:", price + tax)
    fmt.Println("In stock:", inStock)
}
```

Листинг 4-23 Использование пустого идентификатора в файле main.go в папке basicFeatures

Пустым идентификатором является символ подчеркивания (символ _), и его можно использовать везде, где использование имени создаст переменную, которая впоследствии не будет использоваться. Код в листинге 4-23 при компиляции и выполнении выдает следующий результат:

```
Total: 302.5
In stock: true
```

Это еще одна особенность, которая кажется необычной, но она важна при использовании функций в Go. Как я объясню в главе 8, функции Go могут возвращать несколько результатов, и пустой идентификатор полезен, когда вам нужны некоторые из этих значений результата, но не другие.

Понимание указателей

Указатели часто неправильно понимают, особенно если вы пришли к Go с такого языка, как Java или C#, где указатели используются за кулисами, но тщательно скрыты от разработчика. Чтобы понять, как работают указатели, лучше всего начать с понимания того, что делает Go, когда указатели не используются, как показано в листинге 4-24.

> **Подсказка**
> Последний пример в этом разделе обеспечивает простую демонстрацию того, чем могут быть полезны указатели, а не просто объясняет, как они используются.

```
package main
```

```
import "fmt"

func main() {

    first := 100
    second := first

    first++

    fmt.Println("First:", first)
    fmt.Println("Second:", second)
}
```

Листинг 4-24 Определение переменных в файле main.go в папке basicFeatures

Код в листинге 4-24 выдает следующий результат при компиляции и выполнении:

```
First: 101
Second: 100
```

Код в листинге 4-24 создает две переменные. Значение переменной с именем `first` устанавливается с помощью строкового литерала. Значение переменной с именем `second` устанавливается с использованием значения `first`, например:

```
...
first := 100
second := first
...
```

Go копирует текущее значение `first` при создании `second`, после чего эти переменные не зависят друг от друга. Каждая переменная является ссылкой на отдельную ячейку памяти, где хранится ее значение, как показано на рисунке 4-11.

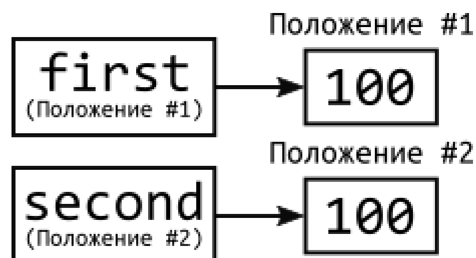

Рисунок 4-11 Независимые значения

Когда я использую оператор `++` для увеличения переменной `first` в листинге 4-24, Go считывает значение в ячейке памяти, связанной с переменной, увеличивает значение и сохраняет его в той же ячейке памяти. Значение, присвоенное переменной `second`, остается прежним, поскольку изменение влияет только на значение, сохраненное переменной `first`, как показано на рисунке 4-12.

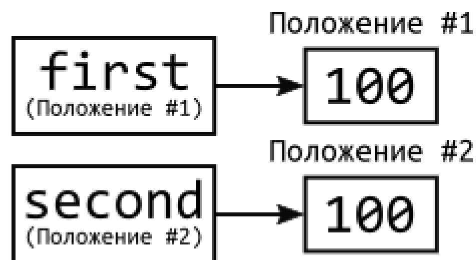

Рисунок 4-12 Изменение значения

ПОНИМАНИЕ АРИФМЕТИКИ УКАЗАТЕЛЕЙ

Указатели имеют плохую репутацию из-за арифметики указателей. Указатели сохраняют ячейки памяти в виде числовых значений, что означает, что ими можно манипулировать с помощью арифметических операторов, обеспечивая доступ к другим ячейкам памяти. Например, вы можете начать с местоположения, указывающего на значение `int`; увеличить значение на количество битов, используемых для хранения `int`; и прочитайте соседнее значение. Это может быть полезно, но может привести к неожиданным результатам, таким как попытка доступа к неправильному расположению или расположению за пределами памяти, выделенной программе.

Go не поддерживает арифметику указателей, что означает, что указатель на одно местоположение нельзя использовать для получения других местоположений. Компилятор сообщит об ошибке, если вы попытаетесь выполнить арифметические действия с помощью указателя.

Определение указателя

Указатель — это переменная, значением которой является адрес памяти. В листинге 4-25 определяется указатель.

```
package main

import "fmt"

func main() {

    first := 100
    var second *int = &first

    first++

    fmt.Println("First:", first)
    fmt.Println("Second:", second)
}
```

Листинг 4-25 Определение указателя в файле main.go в папке basicFeatures

Указатели определяются с помощью амперсанда (символа `&`), известного как *оператор адреса*, за которым следует имя переменной, как показано на рисунке 4-13.

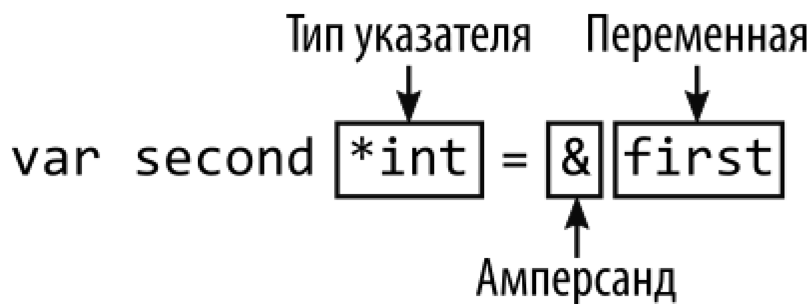

Рисунок 4-13 Определение указателя

Указатели такие же, как и другие переменные в Go. У них есть тип и значение. Значением переменной `second` будет адрес памяти, используемый Go для хранения значения переменной `first`. Скомпилируйте и выполните код, и вы увидите такой вывод:

```
First: 101
Second: 0xc000010088
```

Вы увидите разные выходные данные в зависимости от того, где Go решил сохранить значение для переменной `first`. Конкретное место в памяти не имеет значения, интерес представляют отношения между переменными, показанные на рисунке 4-14.

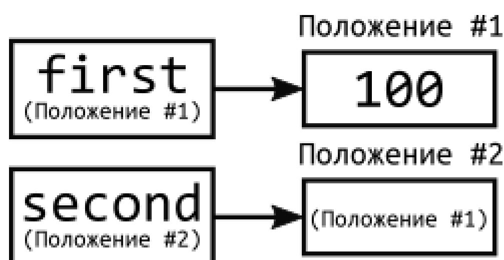

Рисунок 4-14 Указатель и его расположение в памяти

Тип указателя основан на типе переменной, из которой он создан, с префиксом звездочки (символ `*`). Тип переменной с именем `second` — `*int`, потому что она была создана путем применения оператора адреса к переменной `first`, значение которой равно `int`. Когда вы видите тип `*int`, вы знаете, что это переменная, значением которой является адрес памяти, в котором хранится переменная типа `int`.

Тип указателя фиксирован, потому что все типы Go фиксированы, а это означает, что когда вы создаете указатель, например, на `int`, вы меняете значение, на которое он указывает, но вы не можете использовать его для указания на адрес памяти, используемый для хранения другого типа, например, `float64`. Это ограничение важно, поскольку в Go указатели — это не просто адреса памяти, а, скорее, адреса памяти, которые могут хранить определенный тип значения.

Следование указателю

Фраза, следование указателю, означает чтение значения по адресу памяти, на который указывает указатель, и это делается с помощью звездочки (символа `*`), как показано в листинге 4-26. Я также использовал короткий синтаксис объявления переменной для указателя в этом примере. Go выведет тип указателя так же, как и с другими типами.

```
package main

import "fmt"
```

```
func main() {

    first := 100
    second := &first

    first++

    fmt.Println("First:", first)
    fmt.Println("Second:", *second)
}
```

Листинг 4-26 Следование указателю в файле main.go в папке basicFeatures

Звездочка сообщает Go, что нужно следовать указателю и получить значение в ячейке памяти, как показано на рисунке 4-15. Это известно как *разыменование* указателя.

Рисунок 4-15 Следование указателю

Код в листинге 4-26 выдает следующий результат при компиляции и выполнении:

```
First: 101
Second: 101
```

Распространенным заблуждением является то, что `first` и `second` переменные имеют одинаковое значение, но это не так. Есть два значения. Существует значение `int`, доступ к которому можно получить, используя переменную с именем `first`. Существует также значение `*int`, в котором хранится место в памяти значения `first`. Можно использовать значение `*int`, которое будет обращаться к сохраненному значению `int`. Но поскольку значение `*int` является значением, его можно использовать само по себе, а это значит, что его

можно присваивать другим переменным, использовать в качестве аргумента для вызова функции и т.д.

Листинг 4-27 демонстрирует первое использование указателя. За указателем следуют, и значение в ячейке памяти увеличивается.

```
package main

import "fmt"

func main() {

    first := 100
    second := &first

    first++
    *second++

    fmt.Println("First:", first)
    fmt.Println("Second:", *second)
}
```

Листинг 4-27 Следование указателю и изменение значения в файле main.go в папке basicFeatures

Этот код производит следующий вывод при компиляции и выполнении:

```
First: 102
Second: 102
```

В листинге 4-28 показано второе использование указателя, то есть его использование в качестве самостоятельного значения и присвоение его другой переменной.

```
package main

import "fmt"

func main() {

    first := 100
    second := &first
```

```
    first++
    *second++

    var myNewPointer *int
    myNewPointer = second
    *myNewPointer++

    fmt.Println("First:", first)
    fmt.Println("Second:", *second)
}
```

Листинг 4-28 Присвоение значения указателя другой переменной в файле main.go в папке basicFeatures

Первый новый оператор определяет новую переменную, которую я создал с ключевым словом `var`, чтобы подчеркнуть, что тип переменной `*int`, что означает указатель на значение `int`. Следующий оператор присваивает значение переменной `second` новой переменной, а это означает, что значения как `second`, так и `myNewPointer` являются расположением в памяти значения `first`. По любому указателю осуществляется доступ к одному и тому же адресу памяти, что означает, что увеличение `myNewPointer` влияет на значение, полученное при переходе по `second` указателю. Скомпилируйте и выполните код, и вы увидите следующий вывод:

```
First: 103
Second: 103
```

Понимание нулевых значений указателя

Указатели, которые определены, но не имеют значения, имеют нулевое значение `nil`, как показано в листинге 4-29.

```
package main

import "fmt"

func main() {

    first := 100
    var second *int
```

```
    fmt.Println(second)
    second = &first
    fmt.Println(second)
}
```

Листинг 4-29 Определение неинициализированного указателя в файле main.go в папке basicFeatures

Указатель second определяется, но не инициализируется значением и выводится с помощью функции fmt.Println. Оператор адреса используется для создания указателя на переменную first, а значение second записывается снова. Код в листинге 4-29 выдает следующий результат при компиляции и выполнении (игнорируйте < и > в результате, который просто обозначает nil функцией Println):

```
<nil>
0xc000010088
```

Ошибка выполнения произойдет, если вы будете пытаться получить значение по указателю, которому не присвоено значение, как показано в листинге 4-30.

```
package main

import "fmt"

func main() {

    first := 100
    var second *int

    fmt.Println(*second)
    second = &first
    fmt.Println(second == nil)
}
```

Листинг 4-30 Следование неинициализированному указателю в файле main.go в папке basicFeatures

Этот код компилируется, но при выполнении выдает следующую ошибку:

```
panic: runtime error: invalid memory address or nil pointer
```

```
dereference
[signal 0xc0000005 code=0x0 addr=0x0 pc=0xec798a]
goroutine 1 [running]:
main.main()
    C:/basicFeatures/main.go:10 +0x2a
exit status 2
```

Указывание на указатели

Учитывая, что указатели хранят ячейки памяти, можно создать указатель, значением которого будет адрес памяти другого указателя, как показано в листинге 4-31.

```
package main

import "fmt"

func main() {

    first := 100
    second := &first
    third := &second

    fmt.Println(first)
    fmt.Println(*second)
    fmt.Println(**third)
}
```

Листинг 4-31 Создание указателя на указатель в файле main.go в папке basicFeatures

Синтаксис для следующих цепочек указателей может быть неудобным. В этом случае необходимы две звездочки. Первая звездочка следует за указателем на ячейку памяти, чтобы получить значение, хранящееся в переменной с именем `second`, которая является значением `*int`. Вторая звездочка следует за указателем с именем `second`, который дает доступ к расположению в памяти значения, сохраненного переменной `first`. Это не то, что вам нужно делать в большинстве проектов, но это дает хорошее подтверждение того, как работают указатели и как вы можете следовать цепочке, чтобы добраться до значения данных. Код в листинге 4-31 выдает следующий результат при компиляции и выполнении:

```
100
100
100
```

Понимание того, почему указатели полезны

Легко потеряться в деталях того, как работают указатели, и упустить из виду, почему они могут быть друзьями программиста. Указатели полезны, потому что они позволяют программисту выбирать между передачей значения и передачей ссылки. В последующих главах есть много примеров, в которых используются указатели, но в завершение этой главы будет полезна быстрая демонстрация. Тем не менее, листинги в этом разделе основаны на функциях, которые объясняются в следующих главах, поэтому вы можете вернуться к этим примерам позже. В листинге 4-32 приведен пример полезной работы со значениями.

```
package main

import (
    "fmt"
    "sort"
)

func main() {

    names := [3]string {"Alice", "Charlie", "Bob"}

    secondName := names[1]

    fmt.Println(secondName)

    sort.Strings(names[:])

    fmt.Println(secondName)
}
```

Листинг 4-32 Работа со значениями в файле main.go в папке basicFeatures

Синтаксис может быть необычным, но этот пример прост. Создается массив из трех строковых значений, и значение в позиции 1 присваивается переменной с именем `secondName`. Значение

переменной secondName записывается в консоль, массив сортируется, и значение переменной secondName снова записывается в консоль. Этот код производит следующий вывод при компиляции и выполнении:

```
Charlie
Charlie
```

Когда создается переменная secondName, значение строки в позиции 1 массива копируется в новую ячейку памяти, поэтому операция сортировки не влияет на это значение. Поскольку значение было скопировано, теперь оно совершенно не связано с массивом, и сортировка массива не влияет на значение переменной secondName.

В листинге 4-33 в примере представлена переменная-указатель.

```
package main

import (
    "fmt"
    "sort"
)

func main() {

    names := [3]string {"Alice", "Charlie", "Bob"}

    secondPosition := &names[1]

    fmt.Println(*secondPosition)

    sort.Strings(names[:])

    fmt.Println(*secondPosition)
}
```

Листинг 4-33 Использование указателя в файле main.go в папке basicFeatures

При создании переменной secondPosition ее значением является адрес памяти, используемый для хранения строкового значения в позиции 1 массива. Когда массив отсортирован, порядок элементов в массиве изменяется, но указатель по-прежнему ссылается на ячейку

памяти для позиции 1, что означает, что следуя указателю возвращается отсортированное значение, производится следующий вывод, после того как код скомпилируется и выполнится:

```
Charlie
Bob
```

Указатель означает, что я могу сохранить ссылку на местоположение 1 таким образом, чтобы обеспечить доступ к текущему значению, отражающему любые изменения, внесенные в содержимое массива. Это простой пример, но он показывает, как указатели предоставляют разработчику выбор между копированием значений и использованием ссылок.

Если вы все еще не уверены в указателях, подумайте, как проблема значения и ссылки решается в других языках, с которыми вы знакомы. C#, например, который я часто использую, поддерживает как структуры, которые передаются по значению, так и классы, экземпляры которых передаются как ссылки. И Go, и C# позволяют мне выбирать, хочу ли я использовать копию или ссылку. Разница в том, что C# заставляет меня выбирать один раз, когда я создаю тип данных, а Go позволяет мне выбирать каждый раз, когда я использую значение. Подход Go более гибкий, но требует большего внимания со стороны программиста.

Резюме

В этой главе я представил основные встроенные типы, предоставляемые Go, которые образуют строительные блоки почти для каждой функции языка. Я объяснил, как определяются константы и переменные, используя как полный, так и краткий синтаксис; продемонстрировано использование нетипизированных констант; описал использование указателей в Go. В следующей главе я опишу операции, которые можно выполнять над встроенными типами данных, и объясню, как преобразовать значение из одного типа в другой.

5. Операции и преобразования

В этой главе я описываю операторы Go, которые используются для выполнения арифметических операций, сравнения значений и создания логических выражений, выдающих `true`/`false` результаты. Я также объясню процесс преобразования значения из одного типа в другой, который можно выполнить, используя комбинацию встроенных функций языка и средств, предоставляемых стандартной библиотекой Go. В Таблице 5-1 операции и преобразования Go показаны в контексте.

Таблица 5-1 Помещение операций и конверсий в контекст

Вопрос	Ответ
Кто они такие?	Основные операции используются для арифметики, сравнения и логической оценки. Функции преобразования типов позволяют выражать значение одного типа в виде другого типа.
Почему они полезны?	Базовые операции необходимы почти для каждой задачи программирования, и трудно написать код, в котором они не используются. Функции преобразования типов полезны, поскольку строгие правила типов Go предотвращают совместное использование значений разных типов.
Как они используются?	Основные операции применяются с использованием операндов, которые аналогичны тем, которые используются в других языках. Преобразования выполняются либо с использованием синтаксиса явного преобразования Go, либо с использованием средств, предоставляемых пакетами стандартной библиотеки Go.
Есть ли подводные камни или ограничения?	Любой процесс преобразования может привести к потере точности, поэтому необходимо следить за тем, чтобы преобразование значения не приводило к результату с меньшей точностью, чем требуется для задачи.
Есть ли альтернативы?	Нет. Функции, описанные в этой главе, являются фундаментальными для разработки Go.

Таблица 5-2 резюмирует главу.

Таблица 5-2 Краткое содержание главы

Проблема	Решение	Листинг

Проблема	Решение	Листинг
Выполнить арифметику	Используйте арифметические операторы	4–7
Объединить строки	Используйте оператор `+`	8
Сравните два значения	Используйте операторы сравнения	9–11
Объединить выражения	Используйте логические операторы	12
Преобразование из одного типа в другой	Выполнить явное преобразование	13–15
Преобразование значения с плавающей запятой в целое число	Используйте функции, определенные пакетом `math`	16
Разобрать строку в другой тип данных	Используйте функции, определенные пакетом `strconv`	17–28
Выразить значение в виде строки	Используйте функции, определенные пакетом `strconv`	29–32

Подготовка к этой главе

Чтобы подготовиться к этой главе, откройте новую командную строку, перейдите в удобное место и создайте каталог с именем `operations`. Запустите команду, показанную в листинге 5-1, чтобы инициализировать проект.

> **Подсказка**
> Вы можете загрузить пример проекта для этой главы — и для всех остальных глав этой книги — с `https://github.com/apress/pro-go`. См. главу 2 о том, как получить помощь, если у вас возникнут проблемы с запуском примеров.

```
go mod init operations
```

Листинг 5-1 Инициализация проекта

Добавьте файл с именем `main.go` в папку `operations` с содержимым, показанным в листинге 5-2.

```
package main

import "fmt"
```

```
func main() {

    fmt.Println("Hello, Operations")
}
```

Листинг 5-2 Содержимое файла main.go в папке operations

Используйте командную строку для запуска команды, показанной в листинге 5-3, в папке `operations`.

```
go run .
```

Листинг 5-3 Запуск примера проекта

Код в файле `main.go` будет скомпилирован и выполнен, что приведет к следующему результату:

```
Hello, Operations
```

Понимание операторов Go

Go предоставляет стандартный набор операторов, в Таблице 5-3 описаны те из них, с которыми вы будете сталкиваться чаще всего, особенно при работе с типами данных, описанными в главе 4.

Таблица 5-3 Основные операторы Go

Оператор	Описание
`+`, `-`, `*`, `/`, `%`	Эти операторы используются для выполнения арифметических операций с числовыми значениями, как описано в разделе «Знакомство с арифметическими операторами». Оператор `+` также можно использовать для объединения строк, как описано в разделе «Объединение строк».
`==`, `!=`, `<`, `<=`, `>`, `>=`	Эти операторы сравнивают два значения, как описано в разделе «Общие сведения об операторах сравнения».
`||`, `&&`, `!`	Это логические операторы, которые применяются к `bool` значениям и возвращают `bool` значение, как описано в разделе «Понимание логических операторов».
`=`, `:=`	Это операторы присваивания. Стандартный оператор присваивания (`=`) используется для установки начального значения при определении константы или переменной или для изменения значения, присвоенного ранее определенной переменной. Сокращенный оператор (`:=`) используется для определения переменной и присвоения значения, как описано в главе 4.

Оператор	Описание
`-=`, `+=`, `++`, `--`	Эти операторы увеличивают и уменьшают числовые значения, как описано в разделе «Использование операторов увеличения и уменьшения».
`&`, `|`, `^`, `&^`, `<<`, `>>`	Это побитовые операторы, которые можно применять к целочисленным значениям. Эти операторы не часто требуются в основной разработке, но вы можете увидеть пример в главе 31, где оператор `|` используется для настройки функций ведения журнала Go.

Понимание операторов Go

Арифметические операторы можно применять к числовым типам данных (`float32`, `float64`, `int`, `uint` и типам, зависящим от размера, описанным в главе 4). Исключением является оператор остатка (`%`), который можно использовать только с целыми числами. Таблица 5-4 описывает арифметические операторы.

Таблица 5-4 Арифметические операторы

Оператор	Описание
`+`	Этот оператор возвращает сумму двух операндов.
`-`	Этот оператор возвращает разницу между двумя операндами.
`*`	Этот оператор возвращает произведение двух операндов.
`/`	Этот оператор возвращает частное двух операторов.
`%`	Этот оператор возвращает остаток от деления, который аналогичен оператору по модулю, предоставляемому другими языками программирования, но может возвращать отрицательные значения, как описано в разделе «Использование оператора остатка».

Значения, используемые с арифметическими операторами, должны быть одного типа (например, все значения `int`) или быть представлены одним и тем же типом, например нетипизированные числовые константы. В листинге 5-4 показано использование арифметических операторов.

```
package main

import "fmt"

func main() {
    price, tax := 275.00, 27.40
```

```
    sum := price + tax
    difference := price - tax
    product := price * tax
    quotient := price / tax

    fmt.Println(sum)
    fmt.Println(difference)
    fmt.Println(product)
    fmt.Println(quotient)
}
```

Листинг 5-4 Использование арифметических операторов в файле main.go в папке operations

Код в листинге 5-4 выдает следующий результат при компиляции и выполнении:

```
302.4
247.6
7535
10.036496350364963
```

Понимание арифметического переполнения

Go позволяет целочисленным значениям переполняться путем переноса, а не сообщать об ошибке. Значения с плавающей запятой переполняются до положительной или отрицательной бесконечности. В листинге 5-5 показаны переполнения для обоих типов данных.

```
package main

import (
    "fmt"
    "math"
)

func main() {

    var intVal = math.MaxInt64
    var floatVal = math.MaxFloat64

    fmt.Println(intVal * 2)
    fmt.Println(floatVal * 2)
```

```
    fmt.Println(math.IsInf((floatVal * 2), 0))
}
```

Листинг 5-5 Переполнение числовых значений в файле main.go в папке operations

Преднамеренно вызвать переполнение проще всего с помощью пакета math, который является частью стандартной библиотеки Go. Я опишу этот пакет более подробно в главе 18, но в этой главе меня интересуют константы, предусмотренные для наименьшего и наибольшего значений, которые может представлять каждый тип данных, а также функция `IsInf`, которая может использоваться для определения того, является ли значение с плавающей запятой достигло бесконечности. В листинге я использую константы `MaxInt64` и `MaxFloat64` для установки значений двух переменных, которые затем переполняются в выражениях, передаваемых функции `fmt.Println`. Листинг производит следующий вывод, когда он компилируется и выполняется:

```
-2
+Inf
true
```

Целочисленное значение переносится, чтобы получить значение `-2`, а значение с плавающей запятой переполняется до `+Inf`, что обозначает положительную бесконечность. Функция `math.IsInf` используется для обнаружения бесконечности.

Использование оператора остатка от деления

Go предоставляет оператор `%`, который возвращает остаток при делении одного целочисленного значения на другое. Его часто ошибочно принимают за оператор по модулю, предоставляемый другими языками программирования, такими как Python, но, в отличие от этих операторов, оператор остатка от деления Go может возвращать отрицательные значения, как показано в листинге 5-6.

```
package main

import (
    "fmt"
    "math"
```

```
)

func main() {
    posResult := 3 % 2
    negResult := -3 % 2
    absResult := math.Abs(float64(negResult))

    fmt.Println(posResult)
    fmt.Println(negResult)
    fmt.Println(absResult)
}
```

Листинг 5-6 Использование оператора остатка в файле main.go в папке operations

Оператор остатка от деления используется в двух выражениях, чтобы продемонстрировать возможность получения положительных и отрицательных результатов. Пакет `math` предоставляет функцию `Abs`, которая возвращает абсолютное значение `float64`, хотя результатом также является `float64`. Код в листинге 5-6 выдает следующий результат при компиляции и выполнении:

```
1
-1
1
```

Использование операторов инкремента и декремента

Go предоставляет набор операторов для увеличения и уменьшения числовых значений, как показано в листинге 5-7. Эти операторы могут применяться к целым числам и числам с плавающей запятой.

```
package main

import (
    "fmt"
//    "math"
)

func main() {
    value := 10.2
    value++
    fmt.Println(value)
    value += 2
```

```
    fmt.Println(value)
    value -= 2
    fmt.Println(value)
    value--
    fmt.Println(value)
}
```

Листинг 5-7 Использование операторов увеличения и уменьшения в файле main.go в папке operations

Операторы ++ и -- увеличивают или уменьшают значение на единицу. += и -= увеличивают или уменьшают значение на указанную величину. Эти операции подвержены описанному ранее поведению переполнения, но в остальном они согласуются с сопоставимыми операторами в других языках, кроме операторов ++ и --, которые могут быть только постфиксными, что означает отсутствие поддержки выражения, такого как --value. Код в листинге 5-7 выдает следующий результат при компиляции и выполнении:

```
11.2
13.2
11.2
10.2
```

Объединение строк

Оператор + можно использовать для объединения строк для получения более длинных строк, как показано в листинге 5-8.

```
package main

import (
    "fmt"
//    "math"
)

func main() {
    greeting := "Hello"
    language := "Go"
    combinedString := greeting + ", " + language

    fmt.Println(combinedString)
}
```

Листинг 5-8 Объединение строк в файле main.go в папке operations

Результатом оператора **+** является новая строка, а код в листинге 5-8 выдает следующий результат при компиляции и выполнении:

```
Hello, Go
```

Go не объединяет строки с другими типами данных, но стандартная библиотека включает функции, которые составляют строки из значений разных типов, как описано в главе 17.

Понимание операторов сравнения

Операторы сравнения сравнивают два значения, возвращая логическое значение `true`, если они совпадают, и `false` в противном случае. Таблица 5-5 описывает сравнение, выполненное каждым оператором.

Таблица 5-5 Операторы сравнения

Оператор	Описание
`==`	Этот оператор возвращает `true`, если операнды равны.
`!=`	Этот оператор возвращает `true`, если операнды не равны.
`<`	Этот оператор возвращает значение `true`, если первый операнд меньше второго операнда.
`>`	Этот оператор возвращает значение `true`, если первый операнд больше второго операнда.
`<=`	Этот оператор возвращает значение `true`, если первый операнд меньше или равен второму операнду.
`>=`	Этот оператор возвращает значение `true`, если первый операнд больше или равен второму операнду.

Значения, используемые с операторами сравнения, должны быть одного типа или должны быть нетипизированными константами, которые могут быть представлены как целевой тип, как показано в листинге 5-9.

```
package main

import (
    "fmt"
//    "math"
```

```
)

func main() {

    first := 100
    const second = 200.00

    equal := first == second
    notEqual := first != second
    lessThan := first < second
    lessThanOrEqual := first <= second
    greaterThan := first > second
    greaterThanOrEqual := first >= second

    fmt.Println(equal)
    fmt.Println(notEqual)
    fmt.Println(lessThan)
    fmt.Println(lessThanOrEqual)
    fmt.Println(greaterThan)
    fmt.Println(greaterThanOrEqual)
}
```

Листинг 5-9 Использование нетипизированной константы в файле main.go в папке operations

Нетипизированная константа представляет собой значение с плавающей запятой, но может быть представлена как целочисленное значение, поскольку дробные числа в нем равны нулю. Это позволяет использовать переменную `first` и константу `second` вместе в сравнениях. Это было бы невозможно, например, для постоянного значения `200.01`, потому что значение с плавающей запятой не может быть представлено как целое число без отбрасывания дробных цифр и создания другого значения. Для этого требуется явное преобразование, как описано далее в этой главе. Код в листинге 5-9 выдает следующий результат при компиляции и выполнении:

```
false
true
true
true
false
false
```

ВЫПОЛНЕНИЕ ТЕРНАРНЫХ СРАВНЕНИЙ

В Go нет тернарного оператора, а это значит, что подобные выражения использовать нельзя:

```
...
max := first > second ? first : second
...
```

Вместо этого один из операторов сравнения, описанных в таблице 5-5, используется с оператором `if`, например:

```
...
var max int
if (first > second) {
    max = first
} else {
    max = second
}
...
```

Этот синтаксис менее лаконичен, но, как и многие функции Go, вы быстро привыкнете работать без троичных выражений.

Сравнение указателей

Указатели можно сравнить, чтобы увидеть, указывают ли они на одну и ту же ячейку памяти, как показано в листинге 5-10.

```
package main

import (
    "fmt"
//    "math"
)

func main() {

    first := 100

    second := &first
    third := &first
```

```
    alpha := 100
    beta := &alpha

    fmt.Println(second == third)
    fmt.Println(second == beta)
}
```

Листинг 5-10 Сравнение указателей в файле main.go в папке operations

Оператор равенства Go (==) используется для сравнения ячеек памяти. В листинге 5-10 указатели с именами second и third указывают на одно и то же место и равны. Указатель с именем beta указывает на другое место в памяти. Код в листинге 5-10 выдает следующий результат при компиляции и выполнении:

```
true
false
```

Важно понимать, что сравниваются области памяти, а не значения, которые они хранят. Если вы хотите сравнить значения, вы должны следовать указателям, как показано в листинге 5-11.

```
package main

import (
    "fmt"
//    "math"
)

func main() {

    first := 100

    second := &first
    third := &first

    alpha := 100
    beta := &alpha

    fmt.Println(*second == *third)
    fmt.Println(*second == *beta)
}
```

Листинг 5-11 Следующие указатели в сравнении в файле main.go в папке operations

Эти сравнения следуют указателям для сравнения значений, хранящихся в указанных ячейках памяти, и производят следующий вывод, когда код компилируется и выполняется:

```
true
true
```

Понимание логических операторов

Логические операторы сравнивают `bool` значения, как описано в таблице 5-6. Результаты, полученные этими операторами, могут быть присвоены переменным или использованы как часть выражения управления потоком, которое я описываю в главе 6.

Таблица 5-6 Логические операторы

Оператор	Описание
`||`	Этот оператор возвращает `true` (истину), если любой из операндов `true`. Если первый операнд `true`, то второй операнд не будет оцениваться.
`&&`	Этот оператор возвращает `true`, если оба операнда `true`. Если первый операнд `false`, то второй операнд не будет оцениваться.
`!`	Этот оператор используется с одним операндом. Он возвращает `true`, если операнд `false`, и `false`, если операнд `true`.

В листинге 5-12 показаны логические операторы, используемые для получения значений, присваиваемых переменным.

```
package main

import (
    "fmt"
//    "math"
)

func main() {

    maxMph := 50
    passengerCapacity := 4
    airbags := true
```

```
    familyCar := passengerCapacity > 2 && airbags
    sportsCar := maxMph > 100 || passengerCapacity == 2
    canCategorize := !familyCar && !sportsCar

    fmt.Println(familyCar)
    fmt.Println(sportsCar)
    fmt.Println(canCategorize)
}
```

Листинг 5-12 Использование логических операторов в файле main.go в папке operations

С логическими операторами можно использовать только логические значения, и Go не будет пытаться преобразовать значение, чтобы получить истинное или ложное значение. Если операнд для логического оператора является выражением, то он оценивается для получения логического результата, который используется при сравнении. Код в листинге 5-12 выдает следующий результат при компиляции и выполнении:

```
true
false
false
```

Go сокращает процесс оценки, когда используются логические операторы, а это означает, что для получения результата оценивается наименьшее количество значений. В случае оператора `&&` оценка останавливается, когда встречается ложное значение. В случае `||` оператор, оценка останавливается, когда встречается истинное значение. В обоих случаях никакое последующее значение не может изменить результат операции, поэтому дополнительные вычисления не требуются.

Преобразование, анализ и форматирование значений

Go не позволяет смешивать типы в операциях и не будет автоматически преобразовывать типы, за исключением случаев нетипизированных констант. Чтобы показать, как компилятор реагирует на смешанные типы данных, в листинге 5-13 содержится

инструкция, которая применяет оператор сложения к значениям разных типов. (Вы можете обнаружить, что ваш редактор кода автоматически исправляет код в листинге 5-13, и вам, возможно, придется отменить исправление, чтобы код в редакторе соответствовал листингу, чтобы увидеть ошибку компилятора.)

```
package main

import (
    "fmt"
//    "math"
)

func main() {

    kayak := 275
    soccerBall := 19.50

    total := kayak + soccerBall

    fmt.Println(total)
}
```

Листинг 5-13 Смешивание типов в операции в файле main.go в папке operations

Литеральные значения, используемые для определения переменных `kayak` и `soccerBall`, приводят к значению `int` и значению `float64`, которые затем используются в операции сложения для установки значения переменной `total`. Когда код будет скомпилирован, будет сообщено о следующей ошибке:

```
.\main.go:13:20: invalid operation: kayak + soccerBall
(mismatched types int and float64)
```

Для такого простого примера я мог бы просто изменить буквальное значение, используемое для инициализации переменной каяка, на `275.00`, что дало бы переменную `float64`. Но в реальных проектах типы редко так просто изменить, поэтому Go предоставляет функции, описанные в следующих разделах.

Выполнение явных преобразований типов

Явное преобразование преобразует значение для изменения его типа, как показано в листинге 5-14.

```
package main

import (
    "fmt"
//    "math"
)

func main() {

    kayak := 275
    soccerBall := 19.50

    total := float64(kayak) + soccerBall

    fmt.Println(total)
}
```

Листинг 5-14 Использование явного преобразования в файле main.go в папке operations

Синтаксис для явных преобразований — `T(x)`, где `T` — это целевой тип, а `x` — это значение или выражение для преобразования. В листинге 5-14 я использовал явное преобразование для получения значения `float64` из переменной `kayak`, как показано на рисунке 5-1.

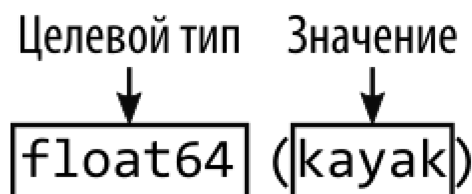

Рисунок 5-1 Явное преобразование типа

Преобразование в значение `float64` означает, что типы в операции сложения согласованы. Код в листинге 5-14 выдает следующий результат при компиляции и выполнении:

```
294.5
```

Понимание ограничений явных преобразований

Явные преобразования можно использовать только в том случае, если значение может быть *представлено* в целевом типе. Это означает, что вы можете выполнять преобразование между числовыми типами и между строками и рунами, но другие комбинации, такие как преобразование значений int в значения bool, не поддерживаются.

Следует соблюдать осторожность при выборе значений для преобразования, поскольку явные преобразования могут привести к потере точности числовых значений или вызвать переполнение, как показано в листинге 5-15.

```
package main

import (
    "fmt"
//    "math"
)

func main() {

    kayak := 275
    soccerBall := 19.50

    total := kayak + int(soccerBall)

    fmt.Println(total)
    fmt.Println(int8(total))

}
```

Листинг 5-15 Преобразование числовых типов в файле main.go в папке operations

Этот листинг преобразует значение float64 в int для операции сложения и, отдельно, преобразует int в int8 (это тип для целого числа со знаком, выделяющего 8 бит памяти, как описано в главе 4). Код выдает следующий результат при компиляции и выполнении:

```
294
38
```

При преобразовании из числа с плавающей запятой в целое дробная часть значения отбрасывается, так что число с плавающей

запятой 19.50 становится int со значением 19. Отброшенная дробь является причиной того, что значение переменной total равно 294 вместо 294.5 произведено в предыдущем разделе.

Значение int8, используемое во втором явном преобразовании, слишком мало для представления значения int 294, поэтому происходит переполнение переменной, как описано в предыдущем разделе «Понимание арифметического переполнения».

Преобразование значений с плавающей запятой в целые числа

Как показано в предыдущем примере, явные преобразования могут привести к неожиданным результатам, особенно при преобразовании значений с плавающей запятой в целые числа. Самый безопасный подход — преобразовать в другом направлении, представляя целые числа и значения с плавающей запятой, но если это невозможно, то math пакет предоставляет набор полезных функций, которые можно использовать для выполнения преобразований контролируемым образом, как описано в таблице 5-7.

Таблица 5-7 Функции в пакете math для преобразования числовых типов

Функция	Описание
Ceil(value)	Эта функция возвращает наименьшее целое число, большее указанного значения с плавающей запятой. Например, наименьшее целое число, большее 27.1, равно 28.
Floor(value)	Эта функция возвращает наибольшее целое число, которое меньше указанного значения с плавающей запятой. Например, наибольшее целое число, меньшее 27.1, равно 28.
Round(value)	Эта функция округляет указанное значение с плавающей запятой до ближайшего целого числа.
RoundToEven(value)	Эта функция округляет указанное значение с плавающей запятой до ближайшего четного целого числа.

Функции, описанные в таблице, возвращают значения float64, которые затем могут быть явно преобразованы в тип int, как показано в листинге 5-16.

```
package main
```

```
import (
    "fmt"
    "math"
)

func main() {

    kayak := 275
    soccerBall := 19.50

    total := kayak + int(math.Round(soccerBall))

    fmt.Println(total)
}
```

Листинг 5-16 Округление значения в файле main.go в папке operations

Функция `math.Round` округляет значение `soccerBall` с `19.5` до `20`, которое затем явно преобразуется в целое число и используется в операции сложения. Код в листинге 5-16 выдает следующий результат при компиляции и выполнении:

```
295
```

Парсинг из строк

Стандартная библиотека Go включает пакет `strconv`, предоставляющий функции для преобразования `string` значений в другие базовые типы данных. Таблица 5-8 описывает функции, которые анализируют строки в другие типы данных.

Таблица 5-8 Функции для преобразования строк в другие типы данных

Функция	Описание
`ParseBool(str)`	Эта функция преобразует строку в логическое значение. Распознаваемые строковые значения: `"true"`, `"false"`, `"TRUE"`, `"FALSE"`, `"True"`, `"False"`, `"T"`, `"F"`, `"0"` и `"1"`.
`ParseFloat(str, size)`	Эта функция анализирует строку в значение с плавающей запятой указанного размера, как описано в разделе «Анализ чисел с плавающей запятой».

Функция	Описание
`ParseInt(str, base, size)`	Эта функция анализирует строку в `int64` с указанным основанием и размером. Допустимые базовые значения: 2 для двоичного, 8 для восьмеричного, 16 для шестнадцатеричного и 10, как описано в разделе «Синтаксический анализ целых чисел».
`ParseUint(str, base, size)`	Эта функция преобразует строку в целое число без знака с указанным основанием и размером.
`Atoi(str)`	Эта функция преобразует строку в целое число с основанием 10 и эквивалентна вызову функции `ParseInt(str, 10, 0)`, как описано в разделе «Использование удобной целочисленной функции».

В листинге 5-17 показано использование функции `ParseBool` для преобразования строк в логические значения.

```
package main

import (
    "fmt"
    "strconv"
)

func main() {

    val1 := "true"
    val2 := "false"
    val3 := "not true"

     bool1, b1err := strconv.ParseBool(val1)
     bool2, b2err := strconv.ParseBool(val2)
     bool3, b3err := strconv.ParseBool(val3)

    fmt.Println("Bool 1", bool1, b1err)
    fmt.Println("Bool 2", bool2, b2err)
    fmt.Println("Bool 3", bool3, b3err)
}
```

Листинг 5-17 Разбор строк в файле main.go в папке operations

Как я объясню в главе 6, функции Go могут выдавать несколько результирующих значений. Функции, описанные в таблице 5-8, возвращают два значения результата: проанализированный результат и ошибку, как показано на рисунке 5-8.

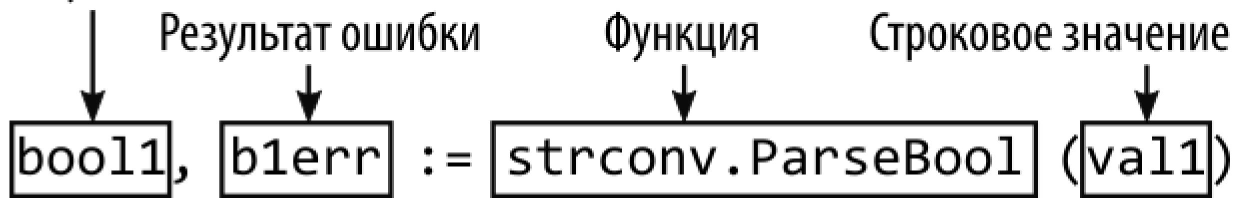

Рисунок 5-2 Разбор строки

Возможно, вы привыкли к языкам, которые сообщают о проблемах, генерируя исключение, которое можно перехватить и обработать с помощью специального ключевого слова, такого как catch. Go работает, присваивая ошибку второму результату, полученному функциями в Таблице 5-8. Если результат ошибки равен нулю, то строка успешно проанализирована. Если результат ошибки не `nil`, то синтаксический анализ завершился неудачно. Вы можете увидеть примеры успешного и неудачного синтаксического анализа, скомпилировав и выполнив код в листинге 5-17, который дает следующий результат:

```
Bool 1 true <nil>
Bool 2 false <nil>
Bool 3 false strconv.ParseBool: parsing "not true": invalid
syntax
```

Первые две строки разбираются на значения `true` и `false`, и результат ошибки для обоих вызовов функции равен `nil`. Третья строка отсутствует в списке распознаваемых значений, описанном в таблице 5-8, и ее нельзя проанализировать. Для этой операции результат ошибки предоставляет подробные сведения о проблеме.

Необходимо соблюдать осторожность, проверяя результат ошибки, потому что другой результат по умолчанию будет равен нулю, когда строка не может быть проанализирована. Если вы не проверите результат ошибки, вы не сможете отличить ложное значение, которое было правильно проанализировано из строки, и нулевое значение, которое было использовано из-за сбоя синтаксического анализа. Проверка на наличие ошибки обычно выполняется с использованием ключевых слов `if`/`else`, как показано в листинге 5-18. Я описываю ключевое слово `if` и связанные с ним функции в главе 6.

```
package main

import (
    "fmt"
    "strconv"
)

func main() {

    val1 := "0"

    bool1, b1err := strconv.ParseBool(val1)

    if b1err == nil {
        fmt.Println("Parsed value:", bool1)
    } else {
        fmt.Println("Cannot parse", val1)
    }
}
```

Листинг 5-18 Проверка на наличие ошибки в файле main.go в папке operations

Блок `if/else` позволяет отличить нулевое значение от успешной обработки строки, которая анализируется до значения `false`. Как я объясняю в главе 6, операторы Go `if` могут определять оператор инициализации, что позволяет вызывать функцию преобразования и проверять ее результаты в одном операторе, как показано в листинге 5-19.

```
package main

import (
    "fmt"
    "strconv"
)

func main() {

    val1 := "0"

    if bool1, b1err := strconv.ParseBool(val1); b1err == nil
{
```

```
        fmt.Println("Parsed value:", bool1)
    } else {
        fmt.Println("Cannot parse", val1)
    }
}
```

Листинг 5-19 Проверка ошибки в отдельном операторе в файле main.go в папке operations

Листинг 5-18 и Листинг 5-19 выдают следующий результат, когда проект компилируется и выполняется:

```
Parsed value: false
```

Разбор целых чисел

Функции `ParseInt` и `ParseUint` требуют основания числа, представленного строкой, и размера типа данных, который будет использоваться для представления проанализированного значения, как показано в листинге 5-20.

```
package main

import (
    "fmt"
    "strconv"
)

func main() {

    val1 := "100"

    int1, int1err := strconv.ParseInt(val1, 0, 8)

    if int1err == nil {
        fmt.Println("Parsed value:", int1)
    } else {
        fmt.Println("Cannot parse", val1)
    }
}
```

Листинг 5-20 Разбор целого числа в файле main.go в папке operations

Первым аргументом функции `ParseInt` является строка для анализа. Второй аргумент — это основание для числа или ноль, чтобы функция могла определить основание по префиксу строки. Последний аргумент — это размер типа данных, которому будет присвоено проанализированное значение. В этом примере я оставил функцию определения основания и указал размер 8.

Скомпилируйте и выполните код из листинга 5-20, и вы получите следующий вывод, показывающий проанализированное целочисленное значение:

```
Parsed value: 100
```

Вы могли бы ожидать, что указание размера изменит тип, используемый для результата, но это не так, и функция всегда возвращает `int64`. Размер указывает только размер данных, в который должно поместиться проанализированное значение. Если строковое значение содержит числовое значение, которое не может быть представлено в пределах указанного размера, то это значение не будет проанализировано. В листинге 5-21 я изменил строковое значение, чтобы оно содержало большее значение.

```
package main

import (
    "fmt"
    "strconv"
)

func main() {

    val1 := "500"

    int1, int1err := strconv.ParseInt(val1, 0, 8)

    if int1err == nil {
        fmt.Println("Parsed value:", int1)
    } else {
        fmt.Println("Cannot parse", val1, int1err)
    }
}
```

Листинг 5-21 Увеличение значения в файле main.go в папке operations

Строка "500" может быть преобразована в целое число, но она слишком велика для представления в виде 8-битного значения, размер которого определяется аргументом ParseInt. Когда код компилируется и выполняется, вывод показывает ошибку, возвращаемую функцией:

```
Cannot parse 500 strconv.ParseInt: parsing "500": value out of range
```

Это может показаться непрямым подходом, но он позволяет Go поддерживать свои правила типов, гарантируя при этом, что вы можете безопасно выполнять явное преобразование результата, если он успешно проанализирован, как показано в листинге 5-22.

```go
package main

import (
    "fmt"
    "strconv"
)

func main() {

    val1 := "100"

    int1, int1err := strconv.ParseInt(val1, 0, 8)

    if int1err == nil {
        smallInt := int8(int1)
        fmt.Println("Parsed value:", smallInt)
    } else {
        fmt.Println("Cannot parse", val1, int1err)
    }
}
```

Листинг 5-22 Явное преобразование результата в файле main.go в папке operations

Указание размера 8 при вызове функции ParseInt позволяет мне выполнить явное преобразование в тип int8 без возможности переполнения. Код в листинге 5-22 выдает следующий результат при компиляции и выполнении:

```
Parsed value: 100
```

Разбор двоичных, восьмеричных и шестнадцатеричных целых чисел

Аргумент `base`, полученный функциями `Parse<Type>`, позволяет анализировать недесятичные числовые строки, как показано в листинге 5-23.

```
package main

import (
    "fmt"
    "strconv"
)

func main() {

    val1 := "100"

    int1, int1err := strconv.ParseInt(val1, 2, 8)

    if int1err == nil {
        smallInt := int8(int1)
        fmt.Println("Parsed value:", smallInt)
    } else {
        fmt.Println("Cannot parse", val1, int1err)
    }
}
```

Листинг 5-23 Анализ двоичного значения в файле main.go в папке operations

Строковое значение `"100"` может быть преобразовано в десятичное значение 100, но оно также может представлять двоичное значение 4. Используя второй аргумент функции `ParseInt`, я могу указать основание `2`, что означает, что строка будет интерпретироваться как двоичное значение. Скомпилируйте и выполните код, и вы увидите десятичное представление числа, проанализированного из двоичной строки:

```
Parsed value: 4
```

Вы можете оставить функции Parse<Type> для определения базы значения с помощью префикса, как показано в листинге 5-24.

```
package main

import (
    "fmt"
    "strconv"
)

func main() {

    val1 := "0b1100100"

    int1, int1err := strconv.ParseInt(val1, 0, 8)

    if int1err == nil {
        smallInt := int8(int1)
        fmt.Println("Parsed value:", smallInt)
    } else {
        fmt.Println("Cannot parse", val1, int1err)
    }
}
```

Листинг 5-24 Использование префикса в файле main.go в папке operations

Функции, описанные в таблице 5-8, могут определять базу анализируемого значения на основе его префикса. Таблица 5-9 описывает набор поддерживаемых префиксов.

Таблица 5-9 Базовые префиксы для числовых строк

Префикс	Описание
0b	Этот префикс обозначает двоичное значение, например 0b1100100.
0o	Этот префикс обозначает восьмеричное значение, например 0o144.
0x	Этот префикс обозначает шестнадцатеричное значение, например 0x64.

Строка в листинге 5-24 имеет префикс 0b, обозначающий двоичное значение. Когда код компилируется и выполняется, создается следующий вывод:

```
Parsed value: 100
```

Использование удобной целочисленной функции

Для многих проектов наиболее распространенной задачей синтаксического анализа является создание значений int из строк, содержащих десятичные числа, как показано в листинге 5-25.

```
package main

import (
    "fmt"
    "strconv"
)

func main() {

    val1 := "100"

    int1, int1err := strconv.ParseInt(val1, 10, 0)

    if int1err == nil {
        var intResult int = int(int1)
        fmt.Println("Parsed value:", intResult)
    } else {
        fmt.Println("Cannot parse", val1, int1err)
    }
}
```

Листинг 5-25 Выполнение общей задачи синтаксического анализа в файле main.go в папке operations

Это настолько распространенная задача, что пакет strconv предоставляет функцию Atoi, которая выполняет синтаксический анализ и явное преобразование за один шаг, как показано в листинге 5-26.

```
package main

import (
    "fmt"
    "strconv"
)

func main() {
```

```
    val1 := "100"

    int1, int1err := strconv.Atoi(val1)

    if int1err == nil {
        var intResult int = int1
        fmt.Println("Parsed value:", intResult)
    } else {
        fmt.Println("Cannot parse", val1, int1err)
    }
}
```

Листинг 5-26 Использование функции удобства в файле main.go в папке operations

Функция `Atoi` принимает только значение для анализа и не поддерживает анализ недесятичных значений. Тип результата — `int` вместо `int64`, создаваемого функцией `ParseInt`. Код в листингах 5-25 и 5-26 выдает следующий результат при компиляции и выполнении:

```
Parsed value: 100
```

Разбор чисел с плавающей запятой

Функция `ParseFloat` используется для анализа строк, содержащих числа с плавающей запятой, как показано в листинге 5-27.

```
package main

import (
    "fmt"
    "strconv"
)

func main() {

    val1 := "48.95"

    float1, float1err := strconv.ParseFloat(val1, 64)

    if float1err == nil {
        fmt.Println("Parsed value:", float1)
    } else {
        fmt.Println("Cannot parse", val1, float1err)
```

```
    }
}
```

Листинг 5-27 Анализ значений с плавающей запятой в файле main.go в папке operations

Первым аргументом функции `ParseFloat` является анализируемое значение. Второй аргумент определяет размер результата. Результатом функции `ParseFloat` является значение `float64`, но если указано `32`, то результат можно явно преобразовать в значение `float32`.

Функция `ParseFloat` может анализировать значения, выраженные с помощью экспоненты, как показано в листинге 5-28.

```
package main

import (
    "fmt"
    "strconv"
)

func main() {

    val1 := "4.895e+01"

    float1, float1err := strconv.ParseFloat(val1, 64)

    if float1err == nil {
        fmt.Println("Parsed value:", float1)
    } else {
        fmt.Println("Cannot parse", val1, float1err)
    }
}
```

Листинг 5-28 Разбор значения с экспонентой в файле main.go в папке operations

Листинги 5-27 и 5-28 дают одинаковый результат при компиляции и выполнении:

```
Parsed value: 48.95
```

Форматирование значений как строк

Стандартная библиотека Go также предоставляет функции для преобразования основных значений данных в строки, которые можно

использовать напрямую или составлять с другими строками. Пакет strconv предоставляет функции, описанные в таблице 5-10.

Таблица 5-10 Функции strconv для преобразования значений в строки

Функция	Описание
FormatBool(val)	Эта функция возвращает строку true или false в зависимости от значения указанного bool значения.
FormatInt(val, base)	Эта функция возвращает строковое представление указанного значения int64, выраженное в указанной системе счисления.
FormatUint(val, base)	Эта функция возвращает строковое представление указанного значения uint64, выраженное в указанной базе.
FormatFloat(val, format, precision, size)	Эта функция возвращает строковое представление указанного значения float64, выраженное с использованием указанного формата, точности и размера.
Itoa(val)	Эта функция возвращает строковое представление указанного значения int, выраженное с использованием базы 10.

Форматирование логических значений

Функция FormatBool принимает bool значение и возвращает строковое представление, как показано в листинге 5-29. Это самая простая из функций, описанных в таблице 5-10, поскольку она возвращает только строки true и false.

```
package main

import (
    "fmt"
    "strconv"
)

func main() {

    val1 := true
    val2 := false

    str1 := strconv.FormatBool(val1)
    str2 := strconv.FormatBool(val2)

    fmt.Println("Formatted value 1: " + str1)
    fmt.Println("Formatted value 2: " + str2)
```

```
}
```

Листинг 5-29 Форматирование логического значения в файле main.go в папке operations

Обратите внимание, что я могу использовать оператор `+` для объединения результата функции `FormatBool` с литеральной строкой, чтобы в функцию `fmt.Println` передавался только один аргумент. Код в листинге 5-29 выдает следующий результат при компиляции и выполнении:

```
Formatted value 1: true
Formatted value 2: false
```

Форматирование целочисленных значений

Функции `FormatInt` и `FormatUint` форматируют целочисленные значения как строки, как показано в листинге 5-30.

```
package main

import (
    "fmt"
    "strconv"
)

func main() {

    val := 275

    base10String := strconv.FormatInt(int64(val), 10)
    base2String := strconv.FormatInt(int64(val), 2)

    fmt.Println("Base 10: " + base10String)
    fmt.Println("Base 2: " + base2String)
}
```

Листинг 5-30 Форматирование целого числа в файле main.go в папке operations

Функция `FormatInt` принимает только значения `int64`, поэтому я выполняю явное преобразование и указываю строки, выражающие значение в десятичном (десятичном) и в двух (двоичном) формате. Код выдает следующий результат при компиляции и выполнении:

```
Base 10: 275
Base 2: 100010011
```

Использование удобной целочисленной функции

Целочисленные значения чаще всего представляются с использованием типа `int` и преобразуются в строки с основанием 10. Пакет `strconv` предоставляет функцию `Itoa`, которая представляет собой более удобный способ выполнения этого конкретного преобразования, как показано в листинге 5-31.

```
package main

import (
    "fmt"
    "strconv"
)

func main() {

    val := 275

    base10String := strconv.Itoa(val)
    base2String := strconv.FormatInt(int64(val), 2)

    fmt.Println("Base 10: " + base10String)
    fmt.Println("Base 2: " + base2String)
}
```

Листинг 5-31 Использование функции удобства в файле main.go в папке operations

Функция `Itoa` принимает значение `int`, которое явно преобразуется в `int64` и передается функции `ParseInt`. Код в листинге 5-31 выводит следующий результат:

```
Base 10: 275
Base 2: 100010011
```

Форматирование значений с плавающей запятой

Для выражения значений с плавающей запятой в виде строк требуются дополнительные параметры конфигурации, поскольку доступны

разные форматы. В листинге 5-32 показана базовая операция форматирования с использованием функции FormatFloat.

```
package main

import (
    "fmt"
    "strconv"
)

func main() {

    val := 49.95

    Fstring := strconv.FormatFloat(val, 'f', 2, 64)
    Estring := strconv.FormatFloat(val, 'e', -1, 64)

    fmt.Println("Format F: " + Fstring)
    fmt.Println("Format E: " + Estring)
}
```

Листинг 5-32 Преобразование числа с плавающей запятой в файле main.go в папке operations

Первым аргументом функции FormatFloat является обрабатываемое значение. Второй аргумент — это byte значение, указывающее формат строки. Байт обычно выражается как литеральное значение руны, и в таблице 5-11 описаны наиболее часто используемые форматы руны. (Как отмечалось в главе 4, тип byte является псевдонимом для uint8 и часто для удобства выражается с помощью руны.)

Таблица 5-11 Обычно используемые параметры формата для форматирования строк с плавающей запятой

Функция	Описание
`f`	Значение с плавающей запятой будет выражено в форме `±ddd.ddd` без экспоненты, например `49.95`.
`e, E`	Значение с плавающей запятой будет выражено в форме `±ddd.ddde±dd`, например, `4.995e+01` или `4.995E+01`. Регистр буквы, обозначающей показатель степени, определяется регистром руны, используемой в качестве аргумента форматирования.

Функция	Описание
g, G	Значение с плавающей запятой будет выражено в формате e/E для больших показателей степени или в формате f для меньших значений.

Третий аргумент функции FormatFloat указывает количество цифр, которые будут следовать за десятичной точкой. Специальное значение -1 можно использовать для выбора наименьшего количества цифр, которое создаст строку, которую можно будет разобрать обратно в то же значение с плавающей запятой без потери точности. Последний аргумент определяет, округляется ли значение с плавающей запятой, чтобы его можно было выразить как значение float32 или float64, используя значение 32 или 64.

Эти аргументы означают, что этот оператор форматирует значение, назначенное переменной с именем val, используя параметр формата f, с двумя десятичными знаками и округляет так, чтобы значение могло быть представлено с использованием типа float64:

```
...
Fstring := strconv.FormatFloat(val, 'f', 2, 64)
...
```

Эффект заключается в форматировании значения в строку, которую можно использовать для представления денежной суммы. Код в листинге 5-32 выдает следующий результат при компиляции и выполнении:

```
Format F: 49.95
Format E: 4.995e+01
```

Резюме

В этой главе я представил операторы Go и показал, как их можно использовать для выполнения арифметических операций, сравнения, конкатенации и логических операций. Я также описал различные способы преобразования одного типа в другой, используя как возможности, встроенные в язык Go, так и функции, входящие в стандартную библиотеку Go. В следующей главе я опишу функции управления потоком выполнения Go.

6. Управление потоком выполнения

В этой главе я описываю возможности Go для управления потоком выполнения. Go поддерживает ключевые слова, общие для других языков программирования, такие как `if`, `for`, `switch` и т. д., но каждое из них имеет некоторые необычные и инновационные функции. Таблица 6-1 помещает функции управления потоком Go в контекст.

Таблица 6-1 Помещение управления потоком в контекст

Вопрос	**Ответ**
Что это?	Управление потоком позволяет программисту выборочно выполнять операторы.
Почему они полезны?	Без управления потоком приложение последовательно выполняет серию операторов кода, а затем завершает работу. Управление потоком позволяет изменять эту последовательность, откладывая выполнение одних операторов и повторяя выполнение других.
Как это используется?	Go поддерживает ключевые слова управления потоком, в том числе `if`, `for` и `switch`, каждое из которых по-разному управляет потоком выполнения.
Есть ли подводные камни или ограничения?	Go вводит необычные функции для каждого из своих ключевых слов управления потоком, которые предлагают дополнительные функции, которые следует использовать с осторожностью.
Есть ли альтернативы?	Нет. Управление потоком — это фундаментальная функция языка.

Таблица 6-2 суммирует главу.

Таблица 6-2 Краткое содержание главы

Проблема	**Решение**	**Листинг**
Условно выполнять операторы	Используйте оператор `if` с необязательными предложениями `else if` и `else` и оператором инициализации	4–10
Повторно выполнить операторы	Используйте цикл `for` с необязательными операторами инициализации и завершения	11–13
Прервать цикл	Используйте ключевое слово `continue` или `break`	14

Проблема	Решение	Листинг
Перечислить последовательность значений	Используйте цикл `for` с ключевым словом `range`	15–18
Выполнение сложных сравнений для условного выполнения операторов	Используйте оператор `switch` с необязательным оператором инициализации	19–21, 23–26
Заставить один оператор `case` переходить в следующий оператор `case`	Используйте ключевое слово `fallthrough`	22
Укажите место, в которое должно перейти выполнение	Использовать метку	27

Подготовка к этой главе

Чтобы подготовиться к этой главе, откройте новую командную строку, перейдите в удобное место и создайте каталог с именем `flowcontrol`. Перейдите в папку управления потоком и выполните команду, показанную в листинге 6-1, чтобы инициализировать проект.

> **Подсказка**
> Вы можете загрузить пример проекта для этой главы — и для всех остальных глав этой книги — с `https://github.com/apress/pro-go`. См. Главу 2 о том, как получить помощь, если у вас возникнут проблемы с запуском примеров.

```
go mod init flowcontrol
```

Листинг 6-1 Инициализация проекта

Добавьте файл с именем `main.go` в папку `flowcontrol` с содержимым, показанным в листинге 6-2.

```
package main

import "fmt"

func main() {

    kayakPrice := 275.00
    fmt.Println("Price:", kayakPrice)
```

```
}
```

Листинг 6-2 Содержимое файла main.go в папке flowcontrol

Используйте командную строку для запуска команды, показанной в листинге 6-3, в папке `flowcontrol`.

```
go run .
```

Листинг 6-3 Запуск примера проекта

Код в файле `main.go` будет скомпилирован и выполнен, что приведет к следующему результату:

```
Price: 275
```

Понимание управления потоком выполнения

Поток выполнения в приложении Go прост для понимания, особенно когда приложение такое же простое, как пример. Операторы, определенные в специальной функции `main`, известной как точка входа приложения, выполняются в том порядке, в котором они определены. После выполнения всех этих операторов приложение завершает работу. Рисунок 6-1 иллюстрирует основной поток.

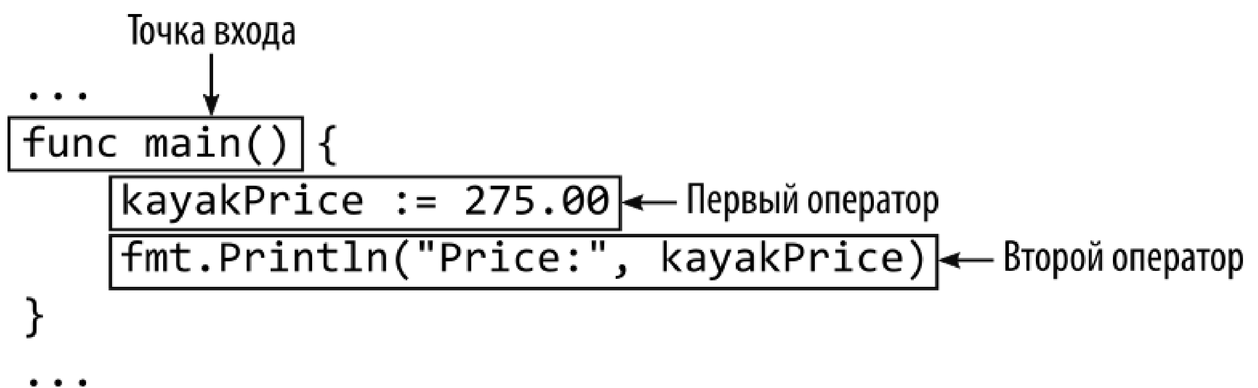

Рисунок 6-1 Поток исполнения

После выполнения каждого оператора поток переходит к следующему оператору, и процесс повторяется до тех пор, пока не останется операторов для выполнения.

Существуют приложения, в которых базовый поток выполнения — это именно то, что требуется, но для большинства приложений

функции, описанные в следующих разделах, используются для управления потоком выполнения для выборочного выполнения инструкций.

Использование операторов if

Оператор `if` используется для выполнения группы операторов только тогда, когда указанное выражение возвращает логическое значение `true` при его оценке, как показано в листинге 6-4.

```
package main

import "fmt"

func main() {

    kayakPrice := 275.00

    if kayakPrice > 100 {
        fmt.Println("Price is greater than 100")
    }
}
```

Листинг 6-4 Использование инструкции if в файле main.go в папке flowcontrol

За ключевым словом `if` следует выражение, а затем группа операторов, которые должны быть выполнены, заключенные в фигурные скобки, как показано на рисунке 6-2.

Рисунок 6-2 Анатомия оператора if

Выражение в листинге 6-4 использует оператор `>` для сравнения значения переменной `kayakPrice` с литеральным постоянным значением `100`. Выражение оценивается как `true`, что означает, что выражение, содержащееся в фигурных скобках, выполняется, что приводит к следующему результату:

```
Price is greater than 100
```

Я обычно заключаю выражение в круглые скобки, как показано в листинге 6-5. Go не требует круглых скобок, но я использую их по привычке.

```
package main

import "fmt"

func main() {

    kayakPrice := 275.00

    if (kayakPrice > 100) {
        fmt.Println("Price is greater than 100")
    }
}
```

Листинг 6-5 Использование скобок в файле main.go в папке flowcontrol

ОГРАНИЧЕНИЯ ПО СИНТАКСИСУ УПРАВЛЕНИЯ ПОТОКОМ

Go менее гибок, чем другие языки, когда речь идет о синтаксисе операторов `if` и других операторов управления потоком. Во-первых, фигурные скобки нельзя опускать, даже если в блоке кода есть только один оператор, то есть такой синтаксис недопустим:

```
...
if (kayakPrice > 100)
    fmt.Println("Price is greater than 100")
...
```

Во-вторых, открывающая фигурная скобка должна стоять в той же строке, что и ключевое слово управления потоком, и не может

появляться в следующей строке, что означает, что этот синтаксис также не разрешен::

```
...
if (kayakPrice > 100)
{
    fmt.Println("Price is greater than 100")
}
...
```

В-третьих, если вы хотите разбить длинное выражение на несколько строк, вы не можете разбить строку после значения или имени переменной:

```
...
if (kayakPrice > 100
        && kayakPrice < 500) {
     fmt.Println("Price is greater than 100 and less than
500")
}
...
```

Компилятор Go сообщит об ошибке для всех этих операторов, и проблема заключается в том, как процесс сборки пытается вставить точки с запятой в исходный код. Изменить такое поведение невозможно, и по этой причине некоторые примеры в этой книге имеют странный формат: некоторые операторы кода содержат больше символов, чем может быть отображено в одной строке на печатной странице, и мне пришлось тщательно разделить операторы, чтобы избежать этой проблемы.

Использование ключевого слова else

Ключевое слово `else` можно использовать для создания дополнительных предложений в операторе `if`, как показано в листинге 6-6.

```
package main

import "fmt"
```

```
func main() {

    kayakPrice := 275.00

    if (kayakPrice > 500) {
        fmt.Println("Price is greater than 500")
    } else if (kayakPrice < 300) {
        fmt.Println("Price is less than 300")
    }
}
```

Листинг 6-6 Использование ключевого слова else в файле main.go в папке flowcontrol

Когда ключевое слово `else` сочетается с ключевым словом `if`, операторы кода в фигурных скобках выполняются только тогда, когда выражение `true`, а выражение в предыдущем предложении `false`, как показано на рисунке 6-3.

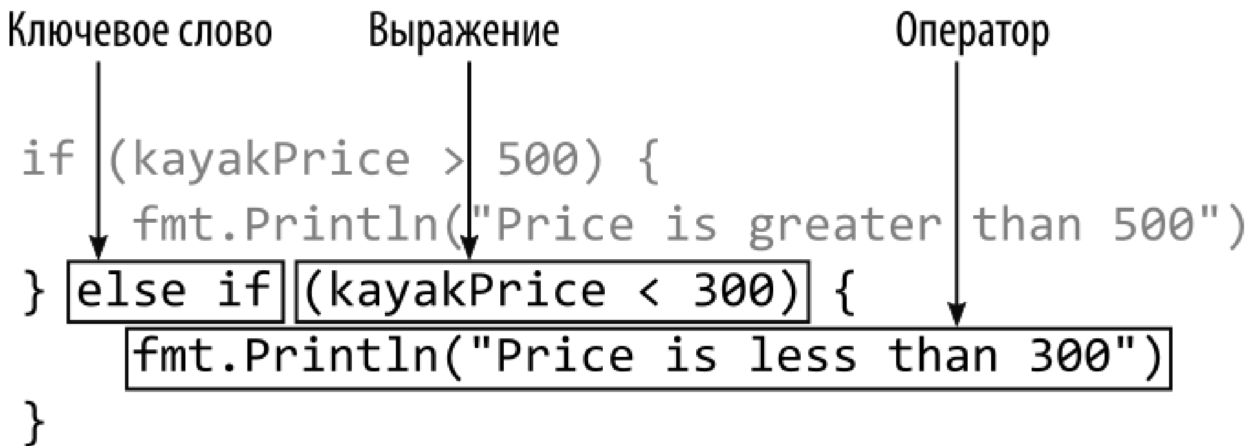

Рисунок 6-3 Предложение else/if в операторе if

В листинге 6-6 выражение, используемое в предложении `if`, дает ложный результат, поэтому выполнение переходит к выражению `else`/`if`, которое дает истинный результат. Код в листинге `6-6` выдает следующий результат при компиляции и выполнении:

```
Price is less than 300
```

Комбинация `else`/`if` может быть повторена для создания последовательности предложений, как показано в листинге `6-7`, каждое из которых будет выполняться только тогда, когда все предыдущие выражения были `false`.

```
package main

import "fmt"

func main() {

    kayakPrice := 275.00

    if (kayakPrice > 500) {
        fmt.Println("Price is greater than 500")
    } else if (kayakPrice < 100) {
        fmt.Println("Price is less than 100")
    } else if (kayakPrice > 200 && kayakPrice < 300) {
        fmt.Println("Price is between 200 and 300")
    }
}
```

Листинг 6-7 Определение нескольких предложений else/if в файле main.go в папке flowcontrol

Выполнение проходит через оператор `if`, оценивая выражения до тех пор, пока не будет получено истинное значение или пока не останется вычисляемых выражений. Код в листинге 6-7 выдает следующий результат при компиляции и выполнении:

```
Price is between 200 and 300
```

Ключевое слово else можно также использовать для создания резервного предложения, операторы которого будут выполняться только в том случае, если все выражения `if` и `else`/`if` в операторе дадут ложные результаты, как показано в листинге 6-8.

```
package main

import "fmt"

func main() {

    kayakPrice := 275.00

    if (kayakPrice > 500) {
        fmt.Println("Price is greater than 500")
    } else if (kayakPrice < 100) {
```

```
        fmt.Println("Price is less than 100")
    } else {
                fmt.Println("Price  not  matched  by  earlier
expressions")
    }
}
```

Листинг 6-8 Создание резервного предложения в файле main.go в папке flowcontrol

Предложение резервного варианта должно быть определено в конце оператора и указывается с помощью ключевого слова `else` без выражения, как показано на рисунке 6-4.

Ключевое слово

Оператор

```
if (kayakPrice > 500) {
   fmt.Println("Price is greater than 500")
} else if (kayakPrice < 100) {
   fmt.Println("Price is less than 100")
} else {
   fmt.Println("Price not matched by earlier expressions54")
}
```

Рисунок 6-4 Резервное предложение в операторе if

Код в листинге 6-8 выдает следующий результат при компиляции и выполнении:

```
Price not matched by earlier expressions
```

Понимание области действия оператора if

Каждое предложение в операторе `if` имеет свою собственную область видимости, что означает, что доступ к переменным возможен только в пределах предложения, в котором они определены. Это также означает, что вы можете использовать одно и то же имя переменной для разных целей в отдельных предложениях, как показано в листинге 6-9.

```
package main

import "fmt"
```

```
func main() {

    kayakPrice := 275.00

    if (kayakPrice > 500) {
        scopedVar := 500
        fmt.Println("Price is greater than", scopedVar)
    } else if (kayakPrice < 100) {
        scopedVar := "Price is less than 100"
        fmt.Println(scopedVar)
    } else {
        scopedVar := false
        fmt.Println("Matched: ", scopedVar)
    }
}
```

Листинг 6-9 Использование области видимости в файле main.go в папке flowcontrol

Каждое предложение в операторе `if` определяет переменную с именем `scopedVar`, и каждая из них имеет свой тип. Каждая переменная является локальной для своего предложения, что означает, что к ней нельзя получить доступ в других предложениях или вне оператора `if`. Код в листинге `6-9` выдает следующий результат при компиляции и выполнении:

```
Matched:  false
```

Использование оператора инициализации с оператором if

Go позволяет оператору `if` использовать оператор инициализации, который выполняется перед вычислением выражения оператора `if`. Оператор инициализации ограничен простым оператором Go, что означает, в общих чертах, что оператор может определять новую переменную, присваивать новое значение существующей переменной или вызывать функцию.

Чаще всего эта функция используется для инициализации переменной, которая впоследствии используется в выражении, как показано в листинге 6-10.

```
package main

import (
```

```
    "fmt"
    "strconv"
)

func main() {

    priceString := "275"

     if kayakPrice, err := strconv.Atoi(priceString); err ==
nil {
        fmt.Println("Price:", kayakPrice)
    } else {
        fmt.Println("Error:", err)
    }
}
```

Листинг 6-10 Использование оператора инициализации в файле main.go в папке flowcontrol

За ключевым словом `if` следует оператор инициализации, затем точка с запятой и вычисляемое выражение, как показано на рисунке 6-5.

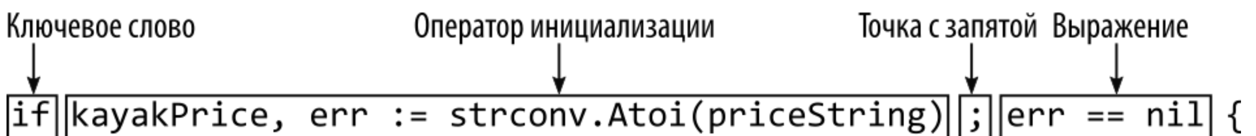

Рисунок 6-5 Использование оператора инициализации

Оператор инициализации в листинге 6-10 вызывает функцию `strconv.Atoi`, описанную в главе 5, для преобразования строки в значение типа `int`. Функция возвращает два значения, которые присваиваются переменным с именами `kayakPrice` и `err`:

```
...
if kayakPrice, err := strconv.Atoi(priceString); err == nil {
...
```

Областью действия переменных, определенных оператором инициализации, является весь оператор `if`, включая выражение. Переменная `err` используется в выражении оператора `if`, чтобы определить, была ли строка проанализирована без ошибок:

```
...
```

```
if kayakPrice, err := strconv.Atoi(priceString); err == nil {
...
```

Переменные также можно использовать в предложении if и любых предложениях else/if и else:

```
...
if kayakPrice, err := strconv.Atoi(priceString); err == nil {
    fmt.Println("Price:", kayakPrice)
} else {
    fmt.Println("Error:", err)
}
...
```

Код в листинге 6-10 выдает следующий результат при компиляции и выполнении:

```
Price: 275
```

ИСПОЛЬЗОВАНИЕ СКОБОК С ПРЕДСТАВИТЕЛЯМИ ИНИЦИАЛИЗАЦИИ

Как я объяснял ранее, я обычно использую круглые скобки для заключения выражений в операторах if. Это по-прежнему возможно при использовании оператора инициализации, но вы должны убедиться, что круглые скобки применяются только к выражению, например:

```
...
if kayakPrice, err := strconv.Atoi(priceString); (err ==
nil) {
...
```

Круглые скобки нельзя применять к инструкции инициализации или заключать обе части инструкции.

Использование циклов for

Ключевое слово for используется для создания циклов, которые многократно выполняют операторы. Самые простые циклы for будут

повторяться бесконечно, если их не прервет ключевое слово break, как показано в листинге 6-11. (Ключевое слово return также может использоваться для завершения цикла.)

```
package main

import (
    "fmt"
    //"strconv"
)

func main() {

    counter := 0
    for {
        fmt.Println("Counter:", counter)
        counter++
        if (counter > 3) {
            break
        }
    }
}
```

Листинг 6-11 Использование базового цикла в файле main.go в папке flowcontrol

За ключевым словом for следуют инструкции для повторения, заключенные в фигурные скобки, как показано на рисунке 6-6. Для большинства циклов одним из операторов будет ключевое слово break, завершающее цикл.

```
Ключевое слово
↓
for {
    fmt.Println("Counter:", counter)
    counter++
    if (counter > 3) {
        break ←——— Ключевое слово
    }
}
```

Рисунок 6-6 Базовый цикл for

Ключевое слово `break` в листинге 6-11 содержится внутри оператора `if`, что означает, что цикл не прерывается до тех пор, пока выражение оператора `if` не даст истинное значение. Код в листинге 6-11 выдает следующий результат при компиляции и выполнении:

```
Counter: 0
Counter: 1
Counter: 2
Counter: 3
```

Включение условия в цикл

Цикл, показанный в предыдущем разделе, представляет собой обычное требование, которое должно повторяться до тех пор, пока не будет достигнуто условие. Это настолько распространенное требование, что условие может быть включено в синтаксис цикла, как показано в листинге 6-12.

```
package main

import (
    "fmt"
    //"strconv"
)
```

```
func main() {

    counter := 0
    for (counter <= 3) {
        fmt.Println("Counter:", counter)
        counter++
        // if (counter > 3) {
        //     break
        // }
    }
}
```

Листинг 6-12 Использование условия цикла в файле main.go в папке flowcontrol

Условие указывается между ключевым словом `for` и открывающей фигурной скобкой, заключающей операторы цикла, как показано на рисунке 6-7. Условия можно заключать в круглые скобки, как показано в примере, но это не обязательно.

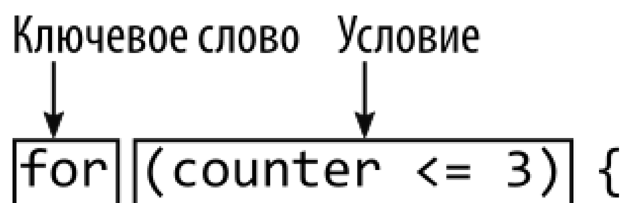

Рисунок 6-7 Условие цикла for

Операторы, заключенные в фигурные скобки, будут выполняться повторно, пока условие оценивается как `true`. В этом примере условие возвращает `true`, пока значение переменной `counter` меньше или равно 3, а код выдает следующие результаты при компиляции и выполнении:

```
Counter: 0
Counter: 1
Counter: 2
Counter: 3
```

Использование операторов инициализации и завершения

Циклы могут быть определены с помощью дополнительных операторов, которые выполняются перед первой итерацией цикла (известные как *оператор инициализации*) и после каждой итерации (*пост оператор*), как показано в листинге 6-13.

Подсказка
Как и в случае с оператором `if`, круглые скобки могут быть применены к условию оператора `for`, но не к операторам инициализации или пост-операторам.

```
package main

import (
    "fmt"
    //"strconv"
)

func main() {

    for counter := 0; counter <= 3; counter++ {
        fmt.Println("Counter:", counter)
        // counter++
    }
}
```

Листинг 6-13 Использование необязательных операторов цикла в файле main.go в папке flowcontrol

Оператор инициализации, условие и пост-оператор разделяются точкой с запятой и следуют за ключевым словом `for`, как показано на рисунке 6-8.

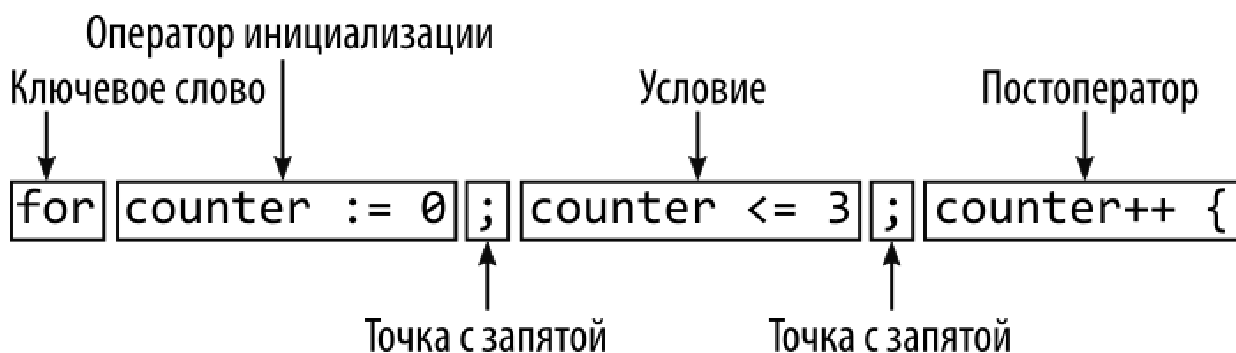

Рисунок 6-8 Цикл for с операторами инициализации и публикации

Выполняется оператор инициализации, после чего оценивается условие. Если условие дает истинный результат, то выполняются операторы, содержащиеся в фигурных скобках, а затем пост-оператор.

Затем условие оценивается снова, и цикл повторяется. Это означает, что оператор инициализации выполняется ровно один раз, а пост-оператор выполняется один раз каждый раз, когда условие дает истинный результат; если условие дает ложный результат при первой оценке, то пост-оператор никогда не будет выполнен. Код в листинге 6-13 выдает следующий результат при компиляции и выполнении:

```
Counter: 0
Counter: 1
Counter: 2
Counter: 3
```

ВОССОЗДАНИЕ ЦИКЛА DO...WHILE

В Go нет цикла `do...while`, который является функцией, предоставляемой другими языками программирования для определения цикла, который выполняется хотя бы один раз, после чего оценивается условие, чтобы определить, требуются ли последующие итерации. Хотя это неудобно, аналогичный результат может быть достигнут с помощью цикла `for`, например:

```
package main

import (
    "fmt"
)

func main() {

    for counter := 0; true; counter++ {
        fmt.Println("Counter:", counter)
        if (counter > 3) {
            break
        }
    }
}
```

Условие для цикла `for` истинно, а последующие итерации управляются оператором `if`, который использует ключевое слово `break` для завершения цикла.

Продолжение цикла

Ключевое слово `continue` можно использовать для прекращения выполнения операторов цикла `for` для текущего значения и перехода к следующей итерации, как показано в листинге 6-14.

```
package main

import (
    "fmt"
    //"strconv"
)

func main() {

    for counter := 0; counter <= 3; counter++ {
        if (counter == 1) {
            continue
        }
        fmt.Println("Counter:", counter)
    }
}
```

Листинг 6-14 Продолжение цикла в файле main.go в папке flowcontrol

Оператор `if` гарантирует, что ключевое слово `continue` будет достигнуто только в том случае, если значение счетчика равно `1`. Для этого значения выполнение не достигнет оператора, вызывающего функцию `fmt.Println`, что приведет к следующему результату при компиляции и выполнении кода:

```
Counter: 0
Counter: 2
Counter: 3
```

Перечисление последовательностей

Ключевое слово `for` можно использовать с ключевым словом `range` для создания циклов, перебирающих последовательности, как показано в листинге 6-15.

```
package main
```

```
import (
    "fmt"
    //"strconv"
)

func main() {

    product := "Kayak"

    for index, character := range product {
                fmt.Println("Index:",  index,  "Character:",
string(character))
    }
}
```

Листинг 6-15 Использование ключевого слова range в файле main.go в папке flowcontrol

В этом примере перечисляется строка, которую цикл `for` обрабатывает как последовательность значений `rune`, каждое из которых представляет символ. Каждая итерация цикла присваивает значения двум переменным, которые обеспечивают текущий индекс в последовательности и значение по текущему индексу, как показано на рисунке 6-9.

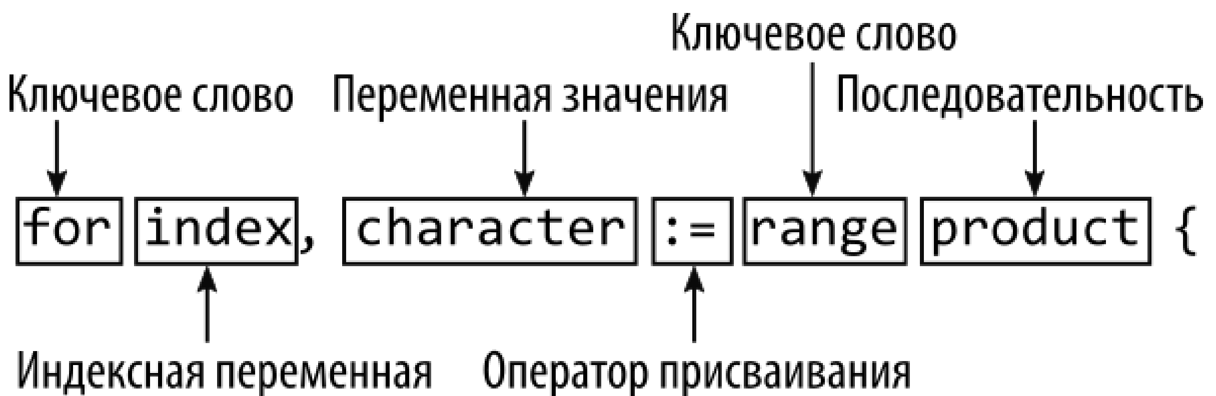

Рисунок 6-9 Перечисление последовательности

Операторы, содержащиеся в фигурных скобках цикла `for`, выполняются один раз для каждого элемента последовательности. Эти операторы могут считывать значения двух переменных, предоставляя доступ к элементам последовательности. В листинге 6-15 это означает, что операторам в цикле предоставляется доступ к отдельным символам,

содержащимся в строке, что приводит к следующему результату при компиляции и выполнении:

```
Index: 0 Character: K
Index: 1 Character: a
Index: 2 Character: y
Index: 3 Character: a
Index: 4 Character: k
```

Получение только индексов или значений при перечислении последовательностей

Go сообщит об ошибке, если переменная определена, но не используется. Вы можете опустить переменную `value` в операторе `for...range`, если вам нужны только значения индекса, как показано в листинге 6-16.

```
package main

import (
    "fmt"
    //"strconv"
)

func main() {

    product := "Kayak"

    for index := range product {
        fmt.Println("Index:", index)
    }
}
```

Листинг 6-16 Получение значений индекса в файле main.go в папке flowcontrol

Цикл `for` в этом примере будет генерировать последовательность значений индекса для каждого символа в строке `product`, производя следующий вывод при компиляции и выполнении:

```
Index: 0
Index: 1
Index: 2
Index: 3
```

```
Index: 4
```

Пустой идентификатор можно использовать, когда вам нужны только значения в последовательности, а не индексы, как показано в листинге 6-17.

```
package main

import (
    "fmt"
    //"strconv"
)

func main() {

    product := "Kayak"

    for _, character := range product {
        fmt.Println("Character:", string(character))
    }
}
```

Листинг 6-17 Получение значений в файле main.go в папке flowcontrol

Пустой идентификатор (символ _) используется для индексной переменной, а обычная переменная используется для значений. Код в листинге 6-17 создает следующий код при компиляции и выполнении:

```
Character: K
Character: a
Character: y
Character: a
Character: k
```

Перечисление встроенных структур данных

Ключевое слово `range` также можно использовать со встроенными структурами данных, предоставляемыми Go — массивами, срезами и картами — все они описаны в главе 7, включая примеры использования ключевых слов `for` и `range`. Для справки в листинге 6-18 показан цикл `for`, использующий ключевое слово `range` для перечисления содержимого массива.

```
package main

import (
    "fmt"
    //"strconv"
)

func main() {

      products  :=  []string  {  "Kayak",  "Lifejacket",  "Soccer
Ball"}

    for index, element:= range products {
        fmt.Println("Index:", index, "Element:", element)
    }
}
```

Листинг 6-18 Перечисление массива в файле main.go в папке flowcontrol

В этом примере используется литеральный синтаксис для определения массивов, которые представляют собой наборы значений фиксированной длины. (В Go также есть встроенные коллекции переменной длины, известные как *срезы*, и карты ключ-значение.) Этот массив содержит три строковых значения, а текущий индекс и элемент присваиваются двум переменным каждый раз, когда выполняется цикл `for`, производя следующий вывод, когда код скомпилирован и выполнен:

```
Index: 0 Element: Kayak
Index: 1 Element: Lifejacket
Index: 2 Element: Soccer Ball
```

Использование операторов switch

Оператор `switch` предоставляет альтернативный способ управления потоком выполнения, основанный на сопоставлении результата выражения с определенным значением, в отличие от оценки истинного или ложного результата, как показано в листинге 6-19. Это может быть краткий способ выполнения множественных сравнений, предоставляющий менее многословную альтернативу сложному оператору `if/elseif/else`.

> **Примечание**
> Оператор `switch` можно также использовать для различения типов данных, как описано в главе 11.

```
package main

import (
    "fmt"
    //"strconv"
)

func main() {

    product := "Kayak"

    for index, character := range product {
        switch (character) {
            case 'K':
                fmt.Println("K at position", index)
            case 'y':
                fmt.Println("y at position", index)
        }
    }
}
```

Листинг 6-19 Использование оператора switch в файле main.go в папке flowcontrol

За ключевым словом `switch` следует значение или выражение, которое дает результат, используемый для сравнения. Сравнения выполняются с серией операторов `case`, каждый из которых определяет значение, как показано на рисунке 6-10.

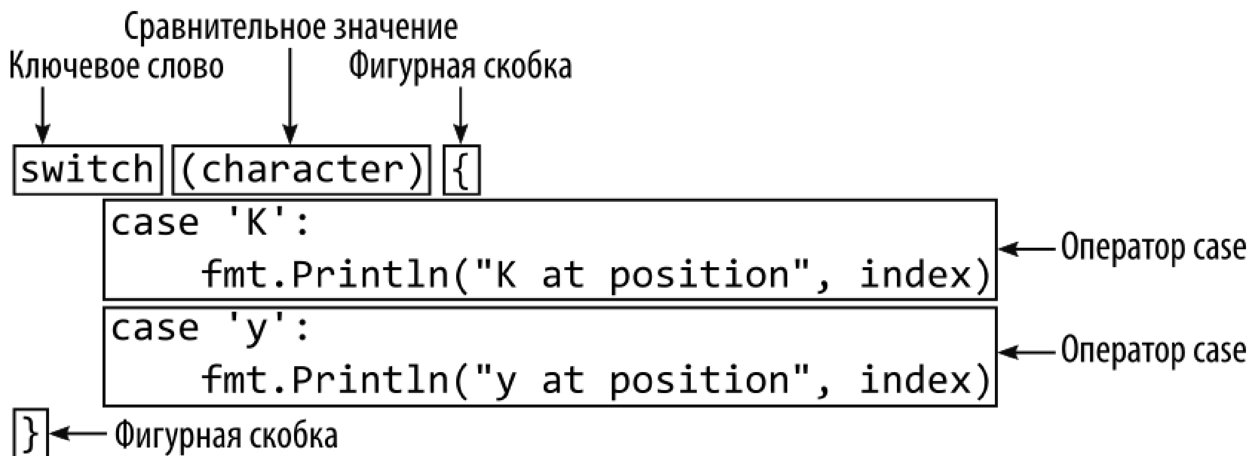

Рисунок 6-10 Базовый оператор switch

В листинге 6-19 оператор `switch` используется для проверки каждого символа, созданного циклом `for`, применяемым к строковому значению, создавая последовательность значений рун, а операторы `case` используются для сопоставления конкретных символов.

За ключевым словом `case` следует значение, двоеточие и один или несколько операторов, которые нужно выполнить, когда значение сравнения совпадает со значением оператора `case`, как показано на рисунке 6-11.

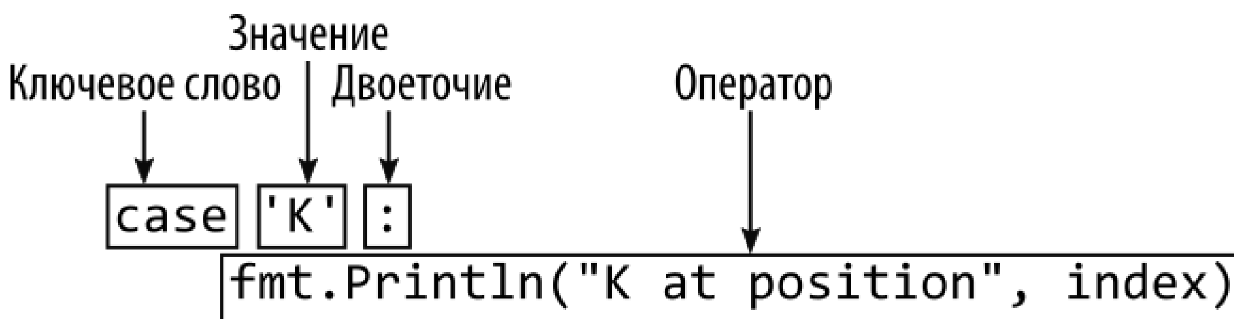

Рисунок 6-11 Анатомия оператора case

Этот оператор `case` соответствует руне `K` и при совпадении выполнит оператор, вызывающий функцию `fmt.Println`. Компиляция и выполнение кода из листинга 6-19 приводит к следующему результату:

```
K at position 0
y at position 2
```

Сопоставление нескольких значений

В некоторых языках операторы switch «проваливаются», что означает, что после того, как оператор case установил совпадение, операторы выполняются до тех пор, пока не будет достигнут оператор break, даже если это означает выполнение операторов из последующего оператора case. Провал часто используется для того, чтобы позволить нескольким операторам case выполнять один и тот же код, но он требует тщательного использования ключевого слова break, чтобы предотвратить неожиданное выполнение выполнения.

Операторы Go switch не выполняются автоматически, но можно указать несколько значений в списке, разделенном запятыми, как показано в листинге 6-20.

```
package main

import (
    "fmt"
    //"strconv"
)

func main() {

    product := "Kayak"

    for index, character := range product {
        switch (character) {
            case 'K', 'k':
                fmt.Println("K or k at position", index)
            case 'y':
                fmt.Println("y at position", index)
        }
    }
}
```

Листинг 6-20 Использование нескольких значений в файле main.go в папке flowcontrol

Набор значений, которым должен соответствовать оператор case, выражается в виде списка, разделенного запятыми, как показано на рисунке 6-12.

Рисунок 6-12 Указание нескольких значений в операторе case

Оператор `case` будет соответствовать любому из указанных значений, производя следующий вывод, когда код в листинге 6-20 компилируется и выполняется:

```
K or k at position 0
y at position 2
K or k at position 4
```

Прекращение выполнения оператора case

Хотя ключевое слово `break` не требуется для завершения каждого оператора `case`, его можно использовать для завершения выполнения операторов до того, как будет достигнут конец оператора `case`, как показано в листинге 6-21.

```
package main

import (
    "fmt"
    //"strconv"
)

func main() {

    product := "Kayak"

    for index, character := range product {
        switch (character) {
            case 'K', 'k':
                if (character == 'k') {
                    fmt.Println("Lowercase k at position",
index)
                    break
```

```
                }
                fmt.Println("Uppercase K at position", index)
            case 'y':
                fmt.Println("y at position", index)
        }
    }
}
```

Листинг 6-21 Использование ключевого слова break в файле main.go в папке flowcontrol

Оператор `if` проверяет, является ли текущая руна `k`, и, если это так, вызывает функцию `fmt.Println`, а затем использует ключевое слово `break`, чтобы остановить выполнение оператора `case`, предотвращая выполнение любых последующих операторов. Листинг 6-21 дает следующий результат при компиляции и выполнении:

```
Uppercase K at position 0
y at position 2
Lowercase k at position 4
```

Принудительный переход к следующему оператору case

Операторы Go `switch` не проваливаются автоматически, но это поведение можно включить с помощью ключевого слова `fallthrough`, как показано в листинге 6-22.

```
package main

import (
    "fmt"
    //"strconv"
)

func main() {

    product := "Kayak"

    for index, character := range product {
        switch (character) {
            case 'K':
                fmt.Println("Uppercase character")
                fallthrough
            case 'k':
```

```
                fmt.Println("k at position", index)
            case 'y':
                fmt.Println("y at position", index)
        }
    }
}
```

Листинг 6-22 Проваливание в файле main.go в папке flowcontrol

Первый оператор case содержит ключевое слово `fallthrough`, что означает, что выполнение продолжится с операторов в следующем операторе `case`. Код в листинге 6-22 выдает следующий результат при компиляции и выполнении:

```
Uppercase character
k at position 0
y at position 2
k at position 4
```

Предоставление пункта по умолчанию

Ключевое слово `default` используется для определения предложения, которое будет выполняться, когда ни один из операторов `case` не соответствует значению оператора `switch`, как показано в листинге 6-23.

```
package main

import (
    "fmt"
    //"strconv"
)

func main() {

    product := "Kayak"

    for index, character := range product {
        switch (character) {
            case 'K', 'k':
                if (character == 'k') {
                    fmt.Println("Lowercase k at position",
index)
```

```
                        break
                    }
                    fmt.Println("Uppercase K at position", index)
            case 'y':
                    fmt.Println("y at position", index)
            default:
                        fmt.Println("Character", string(character),
"at position", index)
        }
    }
}
```

Листинг 6-23 Добавление пункта по умолчанию в файл main.go в папке flowcontrol

Операторы в предложении `default` будут выполняться только для значений, которые не совпадают с оператором `case`. В этом примере символы `K`, `k` и `y` сопоставляются операторам `case`, поэтому предложение `default` будет использоваться только для других символов. Код в листинге 6-23 выдает следующий результат:

```
Uppercase K at position 0
Character a at position 1
y at position 2
Character a at position 3
Lowercase k at position 4
```

Использование оператора инициализации

Оператор `switch` может быть определен с оператором инициализации, который может быть полезным способом подготовки значения сравнения, чтобы на него можно было ссылаться в операторах `case`. В листинге 6-24 показана проблема, характерная для операторов `switch`, где выражение используется для получения значения сравнения.

```
package main

import (
    "fmt"
    //"strconv"
)

func main() {
```

```
    for counter := 0; counter < 20; counter++ {
        switch(counter / 2) {
            case 2, 3, 5, 7:
                fmt.Println("Prime value:", counter / 2)
            default:
                fmt.Println("Non-prime value:", counter / 2)
        }
    }
}
```

Листинг 6-24 Использование выражения в файле main.go в папке flowcontrol

Оператор switch применяет оператор деления к значению переменной counter для получения значения сравнения, а это означает, что та же самая операция должна быть выполнена в операторах case для передачи совпавшего значения в функцию fmt.Println. Дублирования можно избежать с помощью оператора инициализации, как показано в листинге 6-25.

```
package main

import (
    "fmt"
    //"strconv"
)

func main() {

    for counter := 0; counter < 20; counter++ {
        switch val := counter / 2; val {
            case 2, 3, 5, 7:
                fmt.Println("Prime value:", val)
            default:
                fmt.Println("Non-prime value:", val)
        }
    }
}
```

Листинг 6-25 Использование оператора инициализации в файле main.go в папке flowcontrol

Оператор инициализации следует за ключевым словом switch и отделяется от значения сравнения точкой с запятой, как показано на

рисунок 6-13.

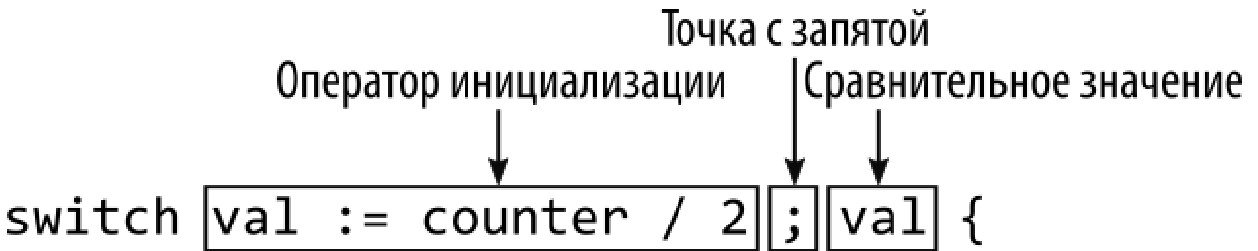

Рисунок 6-13 Оператор инициализации оператора switch

Оператор инициализации создает переменную с именем `val` с помощью оператора деления. Это означает, что `val` можно использовать в качестве значения сравнения, и к нему можно получить доступ в операторах `case`, что позволяет избежать повторения операции. Листинг 6-24 и Листинг 6-25 эквивалентны, и оба выдают следующий результат при компиляции и выполнении:

```
Non-prime value: 0
Non-prime value: 0
Non-prime value: 1
Non-prime value: 1
Prime value: 2
Prime value: 2
Prime value: 3
Prime value: 3
Non-prime value: 4
Non-prime value: 4
Prime value: 5
Prime value: 5
Non-prime value: 6
Non-prime value: 6
Prime value: 7
Prime value: 7
Non-prime value: 8
Non-prime value: 8
Non-prime value: 9
Non-prime value: 9
```

Исключение значения сравнения

Go предлагает другой подход к операторам `switch`, который опускает значение сравнения и использует выражения в операторах `case`. Это

подтверждает идею о том, что операторы switch являются краткой альтернативой операторам if, как показано в листинге 6-26.

```
package main

import (
    "fmt"
    //"strconv"
)

func main() {

    for counter := 0; counter < 10; counter++ {
        switch {
            case counter == 0:
                fmt.Println("Zero value")
            case counter < 3:
                fmt.Println(counter, "is < 3")
            case counter >= 3 && counter < 7:
                fmt.Println(counter, "is >= 3 && < 7")
            default:
                fmt.Println(counter, "is >= 7")
        }
    }
}
```

Листинг 6-26 Использование выражений в операторе switch в файле main.go в папке flowcontrol

Когда значение сравнения опущено, каждый оператор case указывается с условием. При выполнении оператора switch каждое условие оценивается до тех пор, пока одно из них не даст true результат или пока не будет достигнуто необязательное предложение default. Листинг 6-26 производит следующий вывод, когда проект компилируется и выполняется:

```
Zero value
1 is < 3
2 is < 3
3 is >= 3 && < 7
4 is >= 3 && < 7
5 is >= 3 && < 7
6 is >= 3 && < 7
```

```
7 is >= 7
8 is >= 7
9 is >= 7
```

Использование операторов меток

Операторы меток позволяют выполнять переход к другой точке, обеспечивая большую гибкость, чем другие функции управления потоком. В листинге 6-27 показано использование оператора метки.

```
package main

import (
    "fmt"
    //"strconv"
)

func main() {

    counter := 0
    target: fmt.Println("Counter", counter)
    counter++
    if (counter < 5) {
        goto target
    }
}
```

Листинг 6-27 Использование оператора Label в файле main.go в папке flowcontrol

Метки определяются именем, за которым следует двоеточие, а затем обычный оператор кода, как показано на рисунке 6-14. Ключевое слово `goto` используется для перехода к метке.

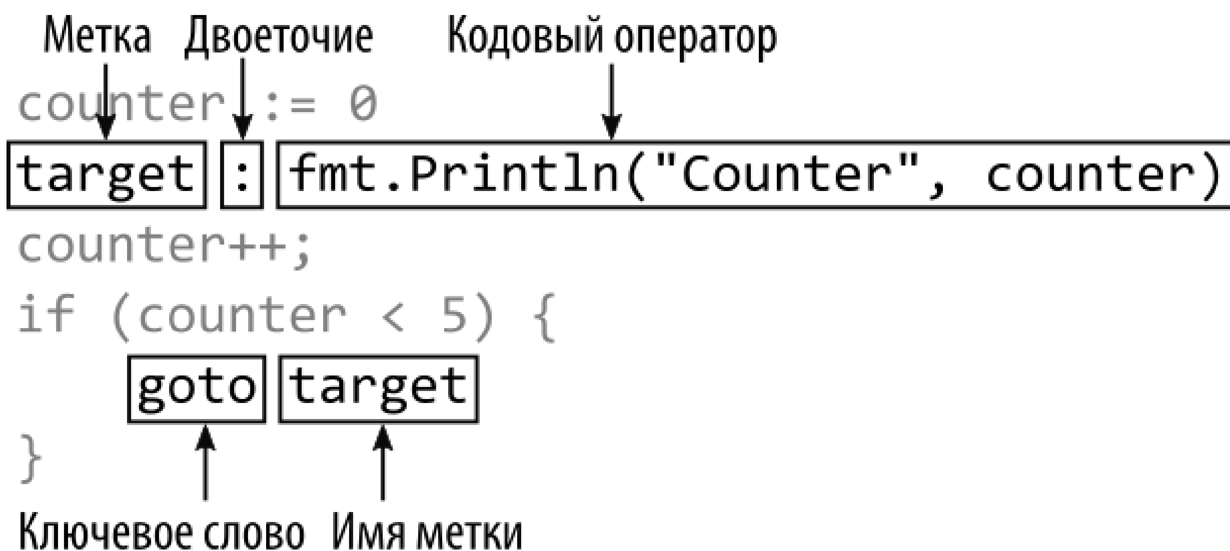

Рисунок 6-14 Маркировка заявления

> **Подсказка**
> Существуют ограничения на то, когда вы можете перейти к метке, например, невозможность перехода к оператору `case` из-за пределов охватывающего его оператора `switch`.

Имя, присвоенное метке в этом примере, — target. Когда выполнение достигает ключевого слова `goto`, оно переходит к оператору с указанной меткой. Эффект представляет собой базовый цикл, который вызывает увеличение значения переменной `counter`, пока оно меньше 5. При компиляции и выполнении листинга 6-27 выводится следующий результат:

```
Counter 0
Counter 1
Counter 2
Counter 3
Counter 4
```

Резюме

В этой главе я описал функции управления потоком Go. Я объяснил, как условно выполнять операторы с операторами `if` и `switch` и как многократно выполнять операторы с помощью цикла `for`. Как показано в этой главе, в Go меньше ключевых слов управления потоком, чем в

других языках, но каждый из них имеет дополнительные функции, такие как операторы инициализации и поддержка ключевого слова `range`. В следующей главе я опишу типы коллекций Go: массив, срез и карта.

7. Использование массивов, срезов и карт

В этой главе я опишу встроенные в Go типы коллекций: массивы, срезы и карты. Эти функции позволяют группировать связанные значения, и, как и в случае с другими функциями, Go использует другой подход к коллекциям по сравнению с другими языками. Я также описываю необычный аспект строковых значений Go, которые можно рассматривать как массивы, но вести себя по-разному в зависимости от того, как используются элементы. Таблица 7-1 помещает массивы, срезы и карты в контекст.

Таблица 7-1 Помещение массивов, срезов и карт в контекст

Вопрос	Ответ
Кто они такие?	Классы коллекций Go используются для группировки связанных значений. В массивах хранится фиксированное количество значений, в срезах хранится переменное количество значений, а в картах хранятся пары ключ-значение.
Почему они полезны?	Эти классы коллекций являются удобным способом отслеживать связанные значения данных.
Как они используются?	Каждый тип коллекции можно использовать с литеральным синтаксисом или с помощью функции `make`.
Есть ли подводные камни или ограничения?	Необходимо соблюдать осторожность, чтобы понять, какое влияние операции, выполняемые над срезами, оказывают на базовый массив, чтобы избежать непредвиденных результатов.
Есть ли альтернативы?	Вам не обязательно использовать какой-либо из этих типов, но это упрощает большинство задач программирования.

Таблица 7-2 суммирует главу.

Таблица 7-2 Краткое содержание главы

Проблема	Решение	Листинг
Хранить фиксированное количество значений	Использовать массив	4–8
Сравнить массивы	Используйте операторы сравнения	9

Проблема	Решение	Листинг
Перечислить массив	Используйте цикл `for` с ключевым словом `range`	10, 11
Хранить переменное количество значений	Используйте срез	12–13, 16, 17, 23
Добавить элемент в срез	Используйте функцию `append`	14–15, 18, 20–22
Создать срез из существующего массива или выберите элементы из среза	Используйте диапазон	19, 24
Скопировать элементы в срез	Используйте функцию `copy`	25, 29
Удалить элементы из среза	Используйте функцию `append` с диапазонами, которые пропускают элементы для удаления	30
Перечислить срез	Используйте цикл `for` с ключевым словом `range`	31
Сортировка элементов в срезе	Используйте пакет `sort`	32
Сравнить срезы	Используйте пакет `reflect`	33, 34
Получить указатель на массив, лежащий в основе среза	Выполните явное преобразование в тип массива, длина которого меньше или равна количеству элементов в срезе.	35
Хранить пары ключ-значение	Используйте карты	36–40
Удалить пару ключ-значение с карты	Используйте функцию `delete`	41
Перечислить содержимое карты	Используйте цикл `for` с ключевым словом `range`	42, 43
Чтение байтовых значений или символов из строки	Используйте строку как массив или выполните явное преобразование к типу `[]rune`	44–48
Перечислить символы в строке	Используйте цикл `for` с ключевым словом `range`	49
Перечислить байты в строке	Выполните явное преобразование в тип `[]byte` и используйте цикл `for` с ключевым словом `range`.	50

Подготовка к этой главе

Чтобы подготовиться к этой главе, откройте новую командную строку, перейдите в удобное место и создайте каталог с именем `collections`. Перейдите в папку `collections` и выполните команду, показанную в листинге 7-1, чтобы инициализировать проект.

```
go mod init collections
```

Листинг 7-1 Инициализация проекта

Добавьте файл с именем `main.go` в папку `collections` с содержимым, показанным в листинге 7-2.

> **Подсказка**
> Вы можете загрузить пример проекта для этой главы — и для всех остальных глав этой книги — с `https://github.com/apress/pro-go`. См. Главу 2 о том, как получить помощь, если у вас возникнут проблемы с запуском примеров.

```
package main

import "fmt"

func main() {

    fmt.Println("Hello, Collections")
}
```

Листинг 7-2 Содержимое файла main.go в папке collections

Используйте командную строку для запуска команды, показанной в листинге 7-3, в папке `collections`.

```
go run .
```

Листинг 7-3 Запуск примера проекта

Код в файле `main.go` будет скомпилирован и выполнен, что приведет к следующему результату::

```
Hello, Collections
```

Работа с массивами

Массивы Go имеют фиксированную длину и содержат элементы одного типа, доступ к которым осуществляется по индексу, как показано в листинге 7-4.

```
package main
```

```
import "fmt"

func main() {

    var names [3]string

    names[0] = "Kayak"
    names[1] = "Lifejacket"
    names[2] = "Paddle"

    fmt.Println(names)
}
```

Листинг 7-4 Определение и использование массивов в файле main.go в папке collections

Типы массивов включают размер массива в квадратных скобках, за которым следует тип элемента, который будет содержать массив, известный как *базовый тип*, как показано на рисунке 7-1. Длина и тип элемента массива не могут быть изменены, а длина массива должна быть указана как константа. (Срезы, описанные далее в этой главе, хранят переменное количество значений.)

Рисунок 7-1 Определение массива

Массив создается и заполняется нулевым значением для типа элемента. В этом примере массив `names` будет заполнен пустой строкой (`""`), которая является нулевым значением для строкового типа. Доступ к элементам массива осуществляется с использованием нотации индекса с отсчетом от нуля, как показано на рисунке 7-2.

Рисунок 7-2 Доступ к элементу массива

Последний оператор в листинге 7-4 передает массив `fmt.Println`, который создает строковое представление массива и записывает его в консоль, производя следующий вывод после компиляции и выполнения кода:

```
[Kayak Lifejacket Paddle]
```

Использование литерального синтаксиса массива

Массивы могут быть определены и заполнены в одном операторе с использованием литерального синтаксиса, показанного в листинге 7-5.

```go
package main

import "fmt"

func main() {

    names := [3]string { "Kayak", "Lifejacket", "Paddle" }

    fmt.Println(names)
}
```

Листинг 7-5 Использование литерального синтаксиса массива в файле main.go в папке collections

За типом массива следуют фигурные скобки, содержащие элементы, которые будут заполнять массив, как показано на рисунке 7-3.

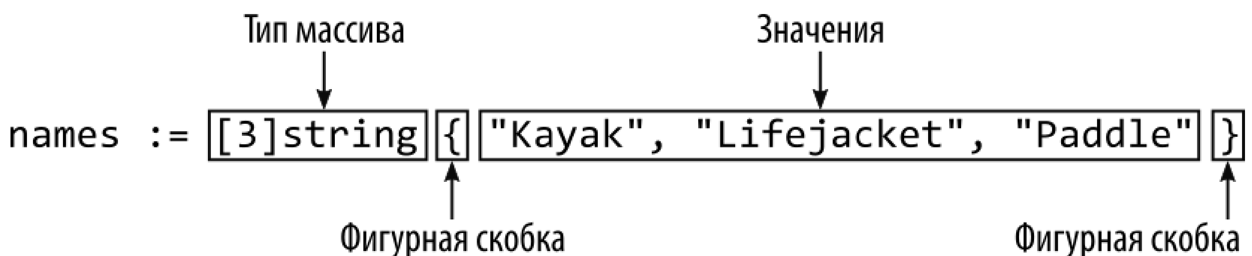

Рисунок 7-3 Синтаксис литерального массива

Подсказка
Количество элементов, указанных с литеральным синтаксисом, может быть меньше емкости массива. Любой позиции в массиве, для

которой не указано значение, будет присвоено нулевое значение для типа массива.

Код в листинге 7-5 выдает следующий результат при компиляции и выполнении:

```
[Kayak Lifejacket Paddle]
```

СОЗДАНИЕ МНОГОМЕРНЫХ МАССИВОВ

Массивы Go являются одномерными, но их можно комбинировать для создания многомерных массивов, например:

```
...
var coords [3][3]int
...
```

Этот оператор создает массив, емкость которого равна 3 и базовый тип которого является массивом `int`, также с емкостью 3, создавая массив значений `int` 3×3. Отдельные значения указываются с использованием двух позиций индекса, например:

```
...
coords[1][2] = 10
...
```

Синтаксис немного неудобен, особенно для массивов с большим количеством измерений, но он функционален и соответствует подходу Go к массивам.

Понимание типов массивов

Тип массива — это комбинация его размера и базового типа. Вот оператор из листинга 7-5, определяющий массив:

```
...
names := [3]string { "Kayak", "Lifejacket", "Paddle" }
...
```

Тип переменной `name` — `[3]string`, что означает массив с базовым типом `string` и емкостью 3. Каждая комбинация базового типа и емкости является отдельным типом, как показано в листинге 7-6.

```
package main

import "fmt"

func main() {

    names := [3]string { "Kayak", "Lifejacket", "Paddle" }

    var otherArray [4]string = names

    fmt.Println(names)
}
```

Листинг 7-6 Работа с типами массивов в файле main.go в папке collections

Базовые типы двух массивов в этом примере одинаковы, но компилятор сообщит об ошибке, даже если емкость `otherArray` достаточна для размещения элементов из массива `names`. Вот ошибка, которую выдает компилятор:

```
.\main.go:9:9: cannot use names (type [3]string) as type
[4]string in assignment
```

ПОЗВОЛЯЕМ КОМПИЛЯТОРУ ОПРЕДЕЛЯТЬ ДЛИНУ МАССИВА

При использовании литерального синтаксиса компилятор может вывести длину массива из списка элементов, например:

```
...
names := [...]string { "Kayak", "Lifejacket", "Paddle" }
...
```

Явная длина заменяется тремя точками (`...`), что указывает компилятору определять длину массива из литеральных значений. Тип переменной `names` по-прежнему `[3]string`, и единственное отличие состоит в том, что вы можете добавлять или удалять литеральные значения, не обновляя при этом явно указанную длину.

Я не использую эту функцию для примеров в этой книге, потому что хочу сделать используемые типы максимально понятными.

Понимание значений массива

Как я объяснял в главе 4, Go по умолчанию работает со значениями, а не со ссылками. Это поведение распространяется на массивы, что означает, что присваивание массива новой переменной копирует массив и копирует содержащиеся в нем значения, как показано в листинге 7-7.

```
package main

import "fmt"

func main() {

    names := [3]string { "Kayak", "Lifejacket", "Paddle" }

    otherArray := names

    names[0] = "Canoe"

    fmt.Println("names:", names)
    fmt.Println("otherArray:", otherArray)
}
```

Листинг 7-7 Присвоение массива новой переменной в файле main.go в папке collections

В этом примере я присваиваю массив `names` новой переменной с именем `otherArray`, а затем изменяю значение нулевого индекса массива `names` перед записью обоих массивов. При компиляции и выполнении код выдает следующий вывод, показывающий, что массив и его содержимое были скопированы:

```
names: [Canoe Lifejacket Paddle]
otherArray: [Kayak Lifejacket Paddle]
```

Указатель можно использовать для создания ссылки на массив, как показано в листинге 7-8.

```
package main
```

```
import "fmt"

func main() {

    names := [3]string { "Kayak", "Lifejacket", "Paddle" }

    otherArray := &names

    names[0] = "Canoe"

    fmt.Println("names:", names)
    fmt.Println("otherArray:", *otherArray)
}
```

Листинг 7-8 Использование указателя на массив в файле main.go в папке collections

Тип переменной `otherArray` — `*[3]string`, обозначающий указатель на массив, способный хранить три строковых значения. Указатель массива работает так же, как и любой другой указатель, и для доступа к содержимому массива необходимо следовать. Код в листинге 7-8 выдает следующий результат при компиляции и выполнении:

```
names: [Canoe Lifejacket Paddle]
otherArray: [Canoe Lifejacket Paddle]
```

Вы также можете создавать массивы, содержащие указатели, что означает, что значения в массиве не копируются при копировании массива. И, как я показал в главе 4, вы можете создавать указатели на определенные позиции в массиве, которые обеспечат доступ к значению в этом месте, даже если содержимое массива изменилось.

Сравнение массивов

Операторы сравнения `==` и `!=` можно применять к массивам, как показано в листинге 7-9.

```
package main

import "fmt"

func main() {

    names := [3]string { "Kayak", "Lifejacket", "Paddle" }
```

```
    moreNames := [3]string { "Kayak", "Lifejacket", "Paddle"
}

    same := names == moreNames

    fmt.Println("comparison:", same)
}
```

Листинг 7-9 Сравнение массивов в файле main.go в папке collections

Массивы равны, если они одного типа и содержат одинаковые элементы в одном и том же порядке. Массивы `names` и `moreNames` равны, потому что оба они являются массивами [3]string и содержат одни и те же строковые значения. Код в листинге 7-9 выдает следующий результат:

```
comparison: true
```

Перечисление массива

Массивы перечисляются с использованием ключевых слов `for` и `range`, как показано в листинге 7-10.

```
package main

import "fmt"

func main() {

    names := [3]string { "Kayak", "Lifejacket", "Paddle" }

    for index, value := range names {
        fmt.Println("Index:", index, "Value:", value)
    }
}
```

Листинг 7-10 Перечисление массива в файле main.go в папке collections

Я подробно описал циклы `for` в главе 6, но при использовании с ключевым словом `range` ключевое слово `for` перечисляет содержимое массива, создавая два значения для каждого элемента по мере перечисления массива, как показано на рисунке 7-4.

}

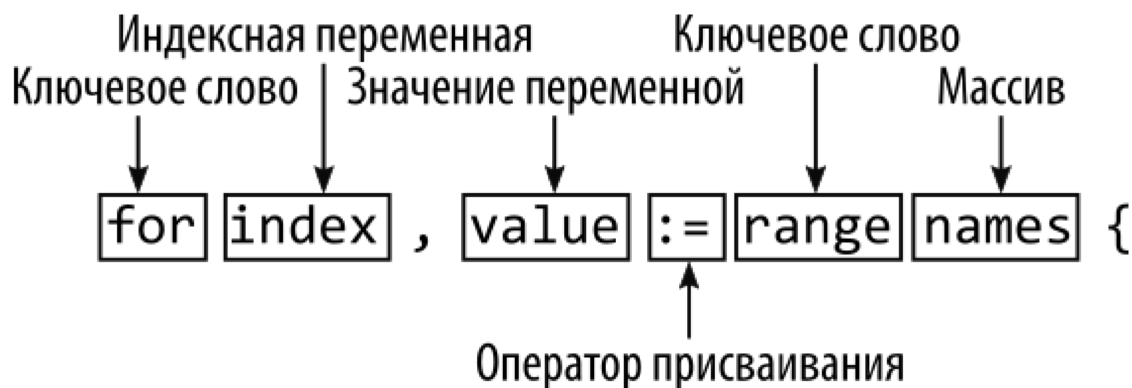

Рисунок 7-4 Перечисление массива

Первое значение, присвоенное переменной `index` в листинге 7-10, соответствует местоположению массива, которое перечисляется. Второе значение, которое присваивается переменной с именем `value` в листинге 7-10, присваивается элементу в текущем местоположении. Листинг производит следующий вывод при компиляции и выполнении:

```
Index: 0 Value: Kayak
Index: 1 Value: Lifejacket
Index: 2 Value: Paddle
```

Go не позволяет определять переменные и не использовать их. Если вам не нужны ни индекс, ни значение, вы можете использовать символ подчеркивания (символ `_`) вместо имени переменной, как показано в листинге 7-11.

```
package main

import "fmt"

func main() {

    names := [3]string { "Kayak", "Lifejacket", "Paddle" }

    for _, value := range names {
        fmt.Println("Value:", value)
    }
}
```

Листинг 7-11 Не использование текущего индекса в файле main.go в папке collections

Подчеркивание известно как *пустой идентификатор* и используется, когда функция возвращает значения, которые впоследствии не используются и для которых не следует назначать имя. Код в листинге 7-11 отбрасывает текущий индекс по мере перечисления массива и выдает следующий результат:

```
Value: Kayak
Value: Lifejacket
Value: Paddle
```

Работа со срезами

Лучше всего рассматривать срезы как массив переменной длины, потому что они полезны, когда вы не знаете, сколько значений вам нужно сохранить, или когда число меняется со временем. Один из способов определить срез — использовать встроенную функцию `make`, как показано в листинге 7-12.

```go
package main

import "fmt"

func main() {

    names := make([]string, 3)

    names[0] = "Kayak"
    names[1] = "Lifejacket"
    names[2] = "Paddle"

    fmt.Println(names)
}
```

Листинг 7-12 Определение среза в файле main.go в папке collections

Функция `make` принимает аргументы, определяющие тип и длину среза, как показано на рисунке 7-5.

```
names := make([]string, 3)
```

Функция → make; Тип → []string; Длина → 3

Рисунок 7-5 Создание среза

Тип среза в этом примере — []string, что означает срез, содержащий строковые значения. Длина не является частью типа среза, потому что размер срезов может варьироваться, как я продемонстрирую позже в этом разделе. Срезы также можно создавать с использованием литерального синтаксиса, как показано в листинге 7-13.

```
package main

import "fmt"

func main() {

    names := []string {"Kayak", "Lifejacket", "Paddle"}

    fmt.Println(names)
}
```

Листинг 7-13 Использование литерального синтаксиса в файле main.go в папке collections

Синтаксис литерала среза подобен тому, который используется для массивов, а начальная длина среза выводится из количества литеральных значений, как показано на рисунке 7-6.

```
names := []string { "Kayak", "Lifejacket", "Paddle" }
```

Тип массива → []string; Предполагаемая длина → "Kayak", "Lifejacket", "Paddle"

Рисунок 7-6 Использование синтаксиса литерала среза

Комбинация типа среза и длины используется для создания массива, который действует как хранилище данных для среза. Срез — это структура данных, которая содержит три значения: указатель на массив, длину среза и емкость среза. Длина среза — это количество элементов, которые он может хранить, а емкость — это количество элементов,

которые могут быть сохранены в массиве. В этом примере и длина, и емкость равны 3, как показано на рисунке 7-7.

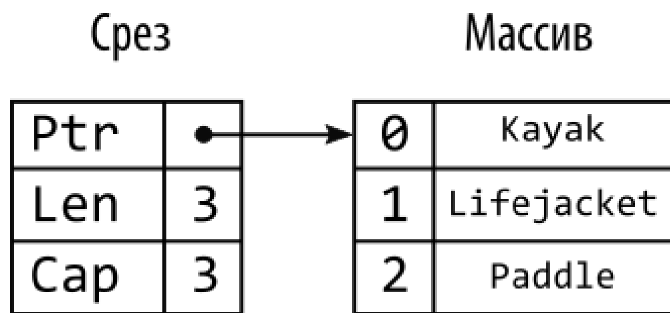

Рисунок 7-7 Срез и его базовый массив

Срезы поддерживают нотацию индекса в стиле массива, которая обеспечивает доступ к элементам базового массива. Хотя рисунке 7-7 представляет собой более реалистичное представление среза, на рисунке 7-8 показано, как срез отображается в свой массив.

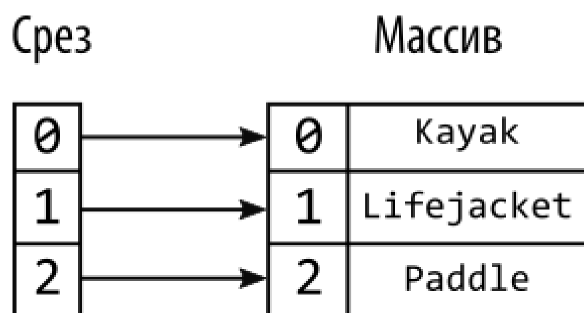

Рисунок 7-8 Срез и его базовый массив

Сопоставление между этим срезом и его массивом простое, но срезы не всегда имеют такое прямое сопоставление со своим массивом, как демонстрируют последующие примеры. Код в листинге 7-12 и листинге 7-13 выдает следующий результат при компиляции и выполнении:

```
[Kayak Lifejacket Paddle]
```

Добавление элементов в срез

Одним из ключевых преимуществ срезов является то, что их можно расширять для размещения дополнительных элементов, как показано в листинге 7-14.

```
package main

import "fmt"

func main() {

    names := []string {"Kayak", "Lifejacket", "Paddle"}

    names = append(names, "Hat", "Gloves")

    fmt.Println(names)
}
```

Листинг 7-14 Добавление элементов к срезу в файле main.go в папке collections

Встроенная функция `append` принимает срез и один или несколько элементов для добавления к срезу, разделенных запятыми, как показано на рисунке 7-9.

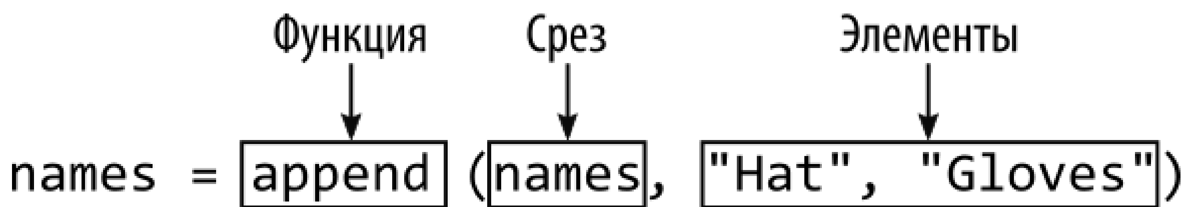

Рисунок 7-9 Добавление элементов в срез

Функция `append` создает массив, достаточно большой для размещения новых элементов, копирует существующий массив и добавляет новые значения. Результатом функции `append` является срез, отображаемый на новый массив, как показано на рисунке 7-10.

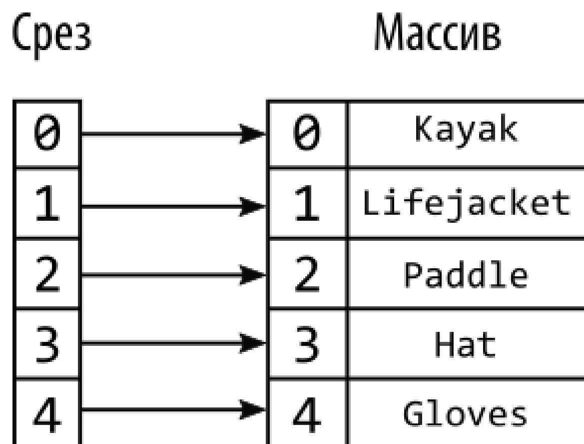

Рисунок 7-10 Результат добавления элементов в срез

Код в листинге 7-14 выдает после компиляции и выполнения следующий вывод, показывающий добавление двух новых элементов в срез:

```
[Kayak Lifejacket Paddle Hat Gloves]
```

Исходный срез и его базовый массив все еще существуют и могут использоваться, как показано в листинге 7-15.

```
package main

import "fmt"

func main() {

    names := []string {"Kayak", "Lifejacket", "Paddle"}

    appendedNames := append(names, "Hat", "Gloves")

    names[0] = "Canoe"

    fmt.Println("names:", names)
    fmt.Println("appendedNames:", appendedNames)
}
```

Листинг 7-15 Добавление элементов к срезу в файле main.go в папке collections

В этом примере результат функции append присваивается другой переменной, в результате чего получается два среза, один из которых был создан из другого. Каждый срез имеет базовый массив, и срезы независимы. Код в листинге 7-15 выдает следующий результат при компиляции и выполнении, показывающий, что изменение значения с использованием одного среза не влияет на другой срез:

```
names: [Canoe Lifejacket Paddle]
appendedNames: [Kayak Lifejacket Paddle Hat Gloves]
```

Выделение дополнительной емкости срезов

Создание и копирование массивов может быть неэффективным. Если вы предполагаете, что вам нужно будет добавлять элементы в срез, вы

можете указать дополнительную емкость при использовании функции `make`, как показано в листинге 7-16.

```
package main

import "fmt"

func main() {

    names := make([]string, 3, 6)

    names[0] = "Kayak"
    names[1] = "Lifejacket"
    names[2] = "Paddle"

    fmt.Println("len:", len(names))
    fmt.Println("cap:", cap(names))
}
```

Листинг 7-16 Выделение дополнительной емкости в файле main.go в папке collections

Как отмечалось ранее, срезы имеют длину и емкость. Длина среза — это количество значений, которые он может содержать в данный момент, а емкость — это количество элементов, которые могут быть сохранены в базовом массиве, прежде чем размер среза должен быть изменен и создан новый массив. Емкость всегда будет не меньше длины, но может быть больше, если с помощью функции `make` была выделена дополнительная емкость. Вызов функции `make` в листинге 7-16 создает срез длиной 3 и емкостью 6, как показано на рисунке 7-11.

Рисунок 7-11 Выделение дополнительной емкости

> **Подсказка**
> Вы также можете использовать функции `len` и `cap` для стандартных массивов фиксированной длины. Обе функции будут возвращать длину массива, так что для массива типа `[3]string`, например, обе

функции вернут 3. См. пример в разделе «Использование функции копирования»..

Встроенные функции len и cap возвращают длину и емкость среза. Код в листинге 7-16 выдает следующий результат при компиляции и выполнении:

```
len: 3
cap: 6
```

В результате базовый массив для среза имеет пространство для роста, как показано на рисунке 7-12.

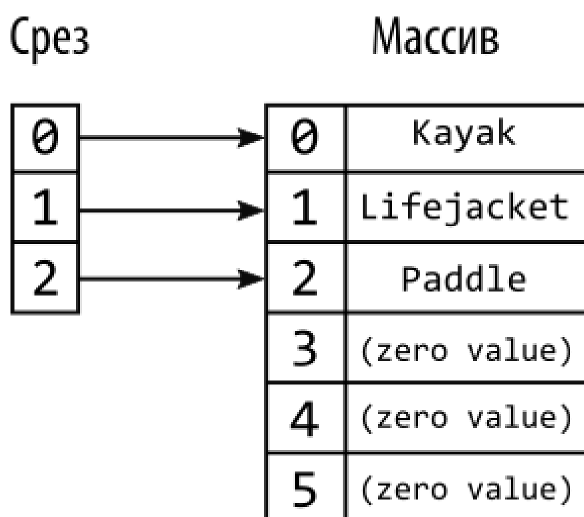

Рисунок 7-12 Срез, базовый массив которого имеет дополнительную емкость

Базовый массив не заменяется, когда функция append вызывается для среза с достаточной емкостью для размещения новых элементов, как показано в листинге 7-17.

Осторожно
Если вы определяете переменную среза, но не инициализируете ее, то результатом будет срез с нулевой длиной и нулевой емкостью, и это вызовет ошибку при добавлении к нему элемента.

```
package main
```

```
import "fmt"

func main() {

    names := make([]string, 3, 6)

    names[0] = "Kayak"
    names[1] = "Lifejacket"
    names[2] = "Paddle"

    appendedNames := append(names, "Hat", "Gloves")

    names[0] = "Canoe"

    fmt.Println("names:",names)
    fmt.Println("appendedNames:", appendedNames)
}
```

Листинг 7-17 Добавление элементов в срез в файле main.go в папке collections

Результатом функции `append` является срез, длина которого увеличилась, но по-прежнему поддерживается тем же базовым массивом. Исходный срез по-прежнему существует и поддерживается тем же массивом, в результате чего теперь есть два представления одного массива, как показано на рисунке 7-13.

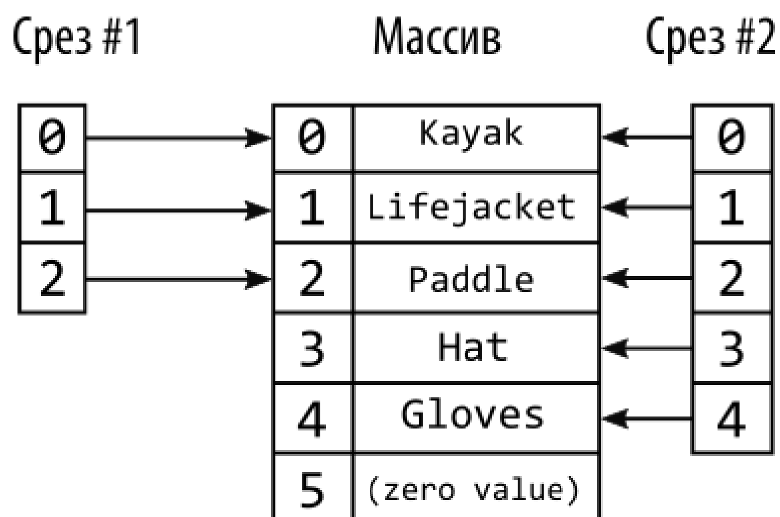

Рисунок 7-13 Несколько срезов, поддерживаемых одним массивом

Поскольку срезы поддерживаются одним и тем же массивом, присвоение нового значения одному срезу влияет и на другой срез, что

можно увидеть в выводе кода в листинге 7-17:

```
names: [Canoe Lifejacket Paddle]
appendedNames: [Canoe Lifejacket Paddle Hat Gloves]
```

Добавление одного среза к другому

Функцию append можно использовать для добавления одного среза к другому, как показано в листинге 7-18.

```
package main

import "fmt"

func main() {

    names := make([]string, 3, 6)

    names[0] = "Kayak"
    names[1] = "Lifejacket"
    names[2] = "Paddle"

    moreNames := []string { "Hat Gloves"}

    appendedNames := append(names, moreNames...)

    fmt.Println("appendedNames:", appendedNames)
}
```

Листинг 7-18 Добавление среза в файл main.go в папку collections

За вторым аргументом следуют три точки (...), которые необходимы, поскольку встроенная функция append определяет переменный параметр, который я описываю в главе 8. Для этой главы достаточно знать, что вы можете добавлять содержимое одного среза в другой срез, пока используются три точки. (Если вы опустите три точки, компилятор Go сообщит об ошибке, потому что он решит, что вы пытаетесь добавить второй срез как одно значение к первому срезу, и знает, что типы не совпадают.) Код в листинге 7-18 производит следующий вывод при компиляции и выполнении:

```
appendedNames: [Kayak Lifejacket Paddle Hat Gloves]
```

Создание срезов из существующих массивов

Срезы можно создавать с использованием существующих массивов, что основано на поведении, описанном в предыдущих примерах, и подчеркивает природу срезов как представлений массивов. В листинге 7-19 определяется массив, который используется для создания срезов.

```
package main

import "fmt"

func main() {

     products := [4]string { "Kayak", "Lifejacket", "Paddle",
"Hat"}

    someNames := products[1:3]
    allNames := products[:]

    fmt.Println("someNames:", someNames)
    fmt.Println("allNames", allNames)
}
```

Листинг 7-19 Создание срезов из существующего массива в файле main.go в папке collections

Переменной `products` назначается стандартный массив фиксированной длины, содержащий строковые значения. Массив используется для создания срезов с использованием диапазона, в котором указаны низкие и высокие значения, как показано на рисунке 7-14.

Рисунок 7-14 Использование диапазона для создания среза из существующего массива

Диапазоны выражены в квадратных скобках, где низкие и высокие значения разделены двоеточием. Первый индекс в срезе устанавливается как наименьшее значение, а длина является результатом наибольшего значения минус наименьшее значение. Это означает, что диапазон `[1:3]` создает диапазон, нулевой индекс которого отображается в индекс 1 массива, а длина равна 2. Как показывает этот пример, срезы не обязательно выравнивать с началом резервного массива.

Начальный индекс и счетчик можно не указывать в диапазоне, чтобы включить все элементы из источника, как показано на рисунке 7-15. (Вы также можете опустить только одно из значений, как показано в последующих примерах.)

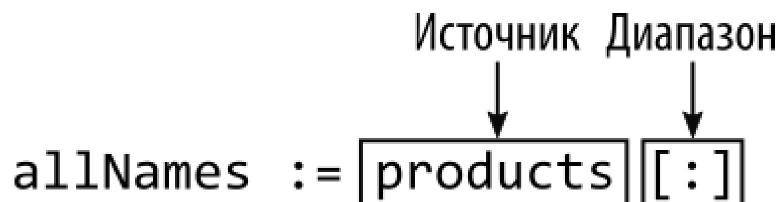

Рисунок 7-15 Диапазон, включающий все элементы

Код в листинге 7-19 создает два среза, каждый из которых поддерживается одним и тем же массивом. Срез `someNames` имеет частичное представление массива, тогда как срез `allNames` представляет собой представление всего массива, как показано на рисунке 7-16.

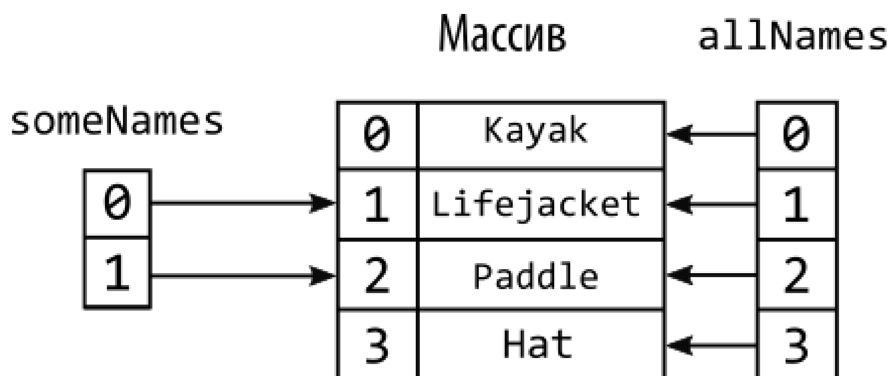

Рисунок 7-16 Создание срезов из существующих массивов

Код в листинге 7-19 выдает следующий результат при компиляции и выполнении:

```
someNames: [Lifejacket Paddle]
allNames [Kayak Lifejacket Paddle Hat]
```

Добавление элементов при использовании существующих массивов для срезов

Связь между срезом и существующим массивом может создавать разные результаты при добавлении элементов.

Как показано в предыдущем примере, можно сместить срез так, чтобы его первая позиция индекса не находилась в начале массива и чтобы его конечный индекс не указывал на последний элемент массива. В листинге 7-19 индекс 0 для среза someNames отображается в индекс 1 массива. До сих пор емкость срезов согласовывалась с длиной базового массива, но это уже не так, поскольку эффект смещения заключается в уменьшении объема массива, который может использоваться срезом. В листинге 7-20 добавлены операторы, записывающие длину и емкость двух срезов.

```
package main

import "fmt"

func main() {

    products := [4]string { "Kayak", "Lifejacket", "Paddle",
"Hat"}

    someNames := products[1:3]
    allNames := products[:]

    fmt.Println("someNames:", someNames)
    fmt.Println("someNames len:", len(someNames), "cap:",
cap(someNames))
    fmt.Println("allNames", allNames)
    fmt.Println("allNames len", len(allNames), "cap:",
cap(allNames))
}
```

Листинг 7-20 Отображение длины и емкости среза в файле main.go в папке collections

Код в листинге 7-20 выдает следующий вывод при компиляции и выполнении, подтверждая эффект среза смещения:

```
someNames: [Lifejacket Paddle]
someNames len: 2 cap: 3
allNames [Kayak Lifejacket Paddle Hat]
allNames len 4 cap: 4
```

Листинг 7-21 добавляет элемент к срезу someNames.

```
package main

import "fmt"

func main() {

    products := [4]string { "Kayak", "Lifejacket", "Paddle",
"Hat"}

    someNames := products[1:3]
    allNames := products[:]

    someNames = append(someNames, "Gloves")

    fmt.Println("someNames:", someNames)
    fmt.Println("someNames len:", len(someNames), "cap:",
cap(someNames))
    fmt.Println("allNames", allNames)
    fmt.Println("allNames len", len(allNames), "cap:",
cap(allNames))
}
```

Листинг 7-21 Добавление элемента к срезу в файле main.go в папке collections

Этот срез может вместить новый элемент без изменения размера, но расположение массива, которое будет использоваться для хранения элемента, уже включено в срез allNames, а это означает, что операция append расширяет срез someNames и изменяет одно из значений, которые можно получить через срез allNames, как показано на рисунке 7-17.

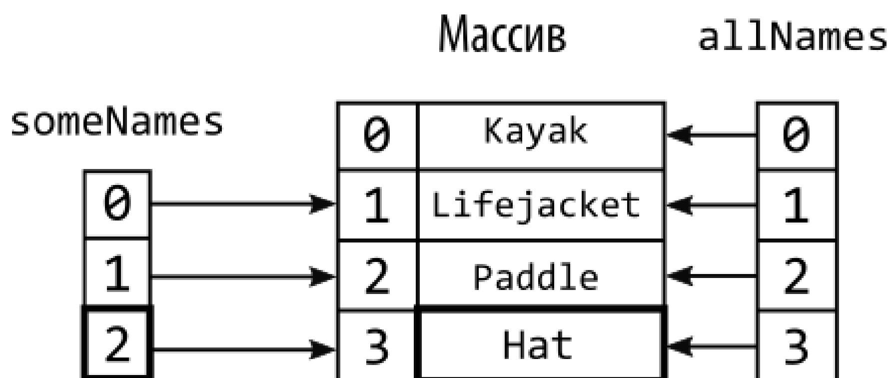

Рисунок 7-17 Добавление элемента в срез

Делаем срезы предсказуемыми

То, как срезы могут совместно использовать массив, вызывает путаницу. Некоторые разработчики ожидают, что срезы будут независимыми, и получают неожиданные результаты, когда значение хранится в массиве, используемом несколькими срезами. Другие разработчики пишут код, который ожидает общие массивы, и получают неожиданные результаты, когда изменение размера разделяет срезы.

Срезы могут показаться непредсказуемыми, но только если обращаться с ними непоследовательно. Мой совет — разделить срезы на две категории, решить, к какой из них относится срез при его создании, и не менять эту категорию.

Первая категория представляет собой представление фиксированной длины в массиве фиксированной длины. Это более полезно, чем кажется, потому что срезы могут быть сопоставлены с определенной областью массива, которую можно выбрать программно. В этой категории вы можете изменять элементы в срезе, но не добавлять новые элементы, что означает, что все срезы, сопоставленные с этим массивом, будут использовать измененные элементы.

Вторая категория представляет собой набор данных переменной длины. Я удостоверяюсь, что каждый срез в этой категории имеет свой собственный резервный массив, который не используется никаким другим срезом. Этот подход позволяет мне свободно добавлять новые элементы в срез, не беспокоясь о влиянии на другие срезы.

Если вы увязли в срезах и не получили ожидаемых результатов, спросите себя, к какой категории относится каждый из ваших срезов и не обрабатываете ли вы срез непоследовательно или создаете срезы из разных категорий из одного и того же исходного массива.

Если вы используете срез в качестве фиксированного представления массива, вы можете ожидать, что несколько срезов дадут вам согласованное представление этого массива, и любые новые значения, которые вы назначите, будут отражены всеми срезами, которые отображаются в измененный элемент.

Этот результат подтверждается выводом, полученным при компиляции и выполнении кода в листинге 7-21:

```
someNames: [Lifejacket Paddle Gloves]
someNames len: 3 cap: 3
allNames [Kayak Lifejacket Paddle Gloves]
allNames len 4 cap: 4
```

Добавление значения `Gloves` к срезу `someNames` изменяет значение, возвращаемое `allNames[3]`, поскольку срезы используют один и тот же массив.

Выходные данные также показывают, что длина и емкость срезов одинаковы, что означает, что больше нет места для расширения среза без создания большего резервного массива. Чтобы подтвердить это поведение, в листинге 7-22 к срезу `someNames` добавляется еще один элемент.

```
package main

import "fmt"

func main() {

     products := [4]string { "Kayak", "Lifejacket", "Paddle",
"Hat"}

    someNames := products[1:3]
    allNames := products[:]

    someNames = append(someNames, "Gloves")
```

```
    someNames = append(someNames, "Boots")

    fmt.Println("someNames:", someNames)
      fmt.Println("someNames len:", len(someNames), "cap:",
cap(someNames))
    fmt.Println("allNames", allNames)
        fmt.Println("allNames len", len(allNames), "cap:",
cap(allNames))
}
```

Листинг 7-22 Добавление еще одного элемента в файл main.go в папке collections

Первый вызов функции `append` расширяет срез `someNames` в существующем базовом массиве. При повторном вызове функции `append` дополнительной емкости не остается, поэтому создается новый массив, содержимое копируется, а два среза поддерживаются разными массивами, как показано на рисунке 7-18.

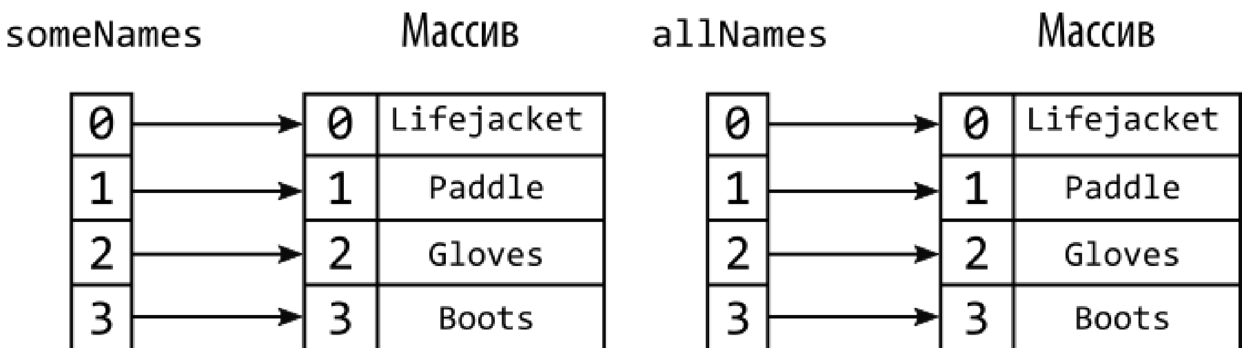

Рисунок 7-18 Изменение размера среза путем добавления элемента

Процесс изменения размера копирует только те элементы массива, которые отображаются срезом, что приводит к повторному выравниванию индексов среза и массива. Код в листинге 7-22 выдает следующий результат при компиляции и выполнении:

```
someNames: [Lifejacket Paddle Gloves Boots]
someNames len: 4 cap: 6
allNames [Kayak Lifejacket Paddle Gloves]
allNames len 4 cap: 4
```

Указание емкости при создании среза из массива

Диапазоны могут включать максимальную емкость, которая обеспечивает некоторую степень контроля над тем, когда массивы

будут дублироваться, как показано в листинге 7-23.

```
package main

import "fmt"

func main() {

    products := [4]string { "Kayak", "Lifejacket", "Paddle",
"Hat"}

    someNames := products[1:3:3]
    allNames := products[:]

    someNames = append(someNames, "Gloves")
    //someNames = append(someNames, "Boots")

    fmt.Println("someNames:", someNames)
    fmt.Println("someNames len:", len(someNames), "cap:",
cap(someNames))
    fmt.Println("allNames", allNames)
    fmt.Println("allNames len", len(allNames), "cap:",
cap(allNames))
}
```

Листинг 7-23 Указание емкости среза в файле main.go в папке collections

Дополнительное значение, известное как *максимальное* значение, указывается после старшего значения, как показано на рисунке 7-19, и должно находиться в пределах границ массива, который нарезается.

Рисунок 7-19 Указание емкости в диапазоне

Максимальное значение не определяет максимальную емкость напрямую. Вместо этого максимальная емкость определяется путем вычитания нижнего значения из максимального значения. В примере максимальное значение равно 3, а минимальное значение равно 1, что означает, что емкость будет ограничена до 2. В результате операция append приводит к изменению размера среза и выделению собственного массива, вместо расширения в существующем массиве, что можно увидеть в выводе кода в листинге 7-23:

```
someNames: [Lifejacket Paddle Gloves]
someNames len: 3 cap: 4
allNames [Kayak Lifejacket Paddle Hat]
allNames len 4 cap: 4
```

Изменение размера среза означает, что значение Gloves, добавляемое к срезу someNames, не становится одним из значений, сопоставленных срезом allNames.

Создание срезов из других срезов

Срезы также можно создавать из других срезов, хотя взаимосвязь между срезами не сохраняется при изменении их размера. Чтобы продемонстрировать, что это значит, в листинге 7-24 создается один срез из другого.

```
package main

import "fmt"

func main() {

     products := [4]string { "Kayak", "Lifejacket", "Paddle",
"Hat"}

    allNames := products[1:]
    someNames := allNames[1:3]

    allNames = append(allNames, "Gloves")
    allNames[1] = "Canoe"

    fmt.Println("someNames:", someNames)
    fmt.Println("allNames", allNames)
```

```
}
```

Листинг 7-24 Создание среза из среза в файле main.go в папке collections

Диапазон, используемый для создания среза `someNames`, применяется к `allNames`, который также является срезом:

```
...
someNames := allNames[1:3]
...
```

Этот диапазон создает срез, который отображается на второй и третий элементы среза `allNames`. Срез `allNames` был создан с собственным диапазоном:

```
...
allNames := products[1:]
...
```

Диапазон создает срез, который отображается на все элементы исходного массива, кроме первого. Эффекты диапазонов суммируются, что означает, что срез `someNames` будет отображен на вторую и третью позиции в массиве, как показано на рисунке 7-20.

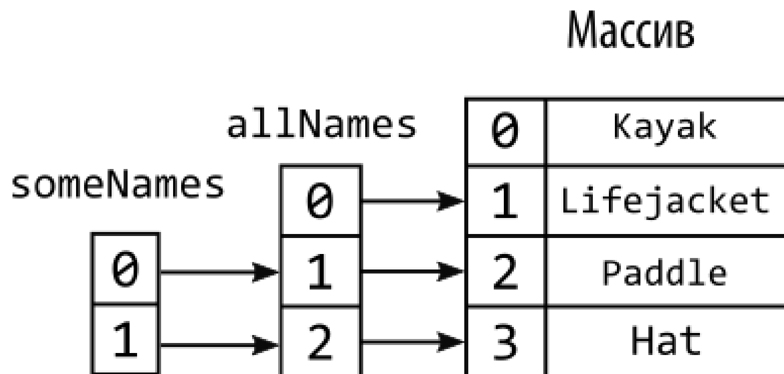

Рисунок 7-20 Создание среза из среза

Использование одного среза для создания другого является эффективным способом переноса положения начального смещения, как это показано на рисунке 7-19. Но помните, что срезы по сути являются указателями на секции массивов, а это значит, что они не могут указывать на другой срез. В действительности диапазоны используются

для определения отображений для срезов, поддерживаемых одним и тем же массивом, как показано на рисунке 7-21.

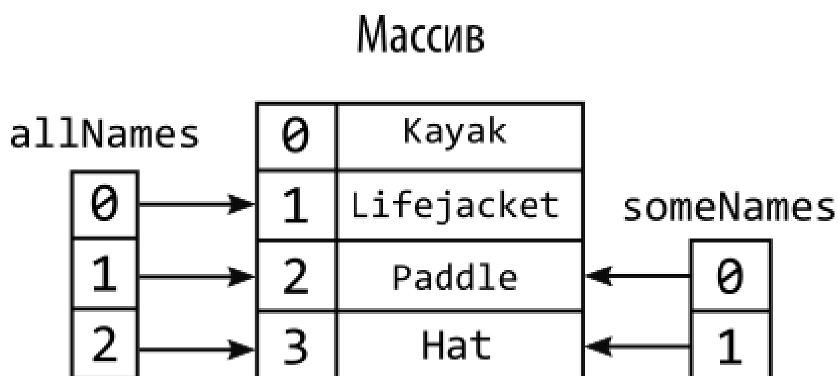

Рисунок 7-21 Фактическое расположение срезов

Срезы ведут себя так же, как и в других примерах в этой главе, и их размер будет изменен, если элементы будут добавлены, когда нет доступной емкости, и в этот момент они больше не будут использовать общий массив.

Использование функции копирования

Функция `copy` используется для копирования элементов между срезами. Эту функцию можно использовать для обеспечения того, чтобы срезы имели отдельные массивы, и для создания срезов, объединяющих элементы из разных источников.

Использование функции копирования для обеспечения разделения массива срезов

Функцию `copy` можно использовать для дублирования существующего среза, выбирая некоторые или все элементы, но гарантируя, что новый срез поддерживается собственным массивом, как показано в листинге 7-25.

```
package main

import "fmt"

func main() {
```

```
    products := [4]string { "Kayak", "Lifejacket", "Paddle",
"Hat"}

    allNames := products[1:]
    someNames := make([]string, 2)
    copy(someNames, allNames)

    fmt.Println("someNames:", someNames)
    fmt.Println("allNames", allNames)
}
```

Листинг 7-25 Дублирование среза в файле main.go в папке collections

Функция `copy` принимает два аргумента: срез назначения и срез источника, как показано на рисунке 7-22.

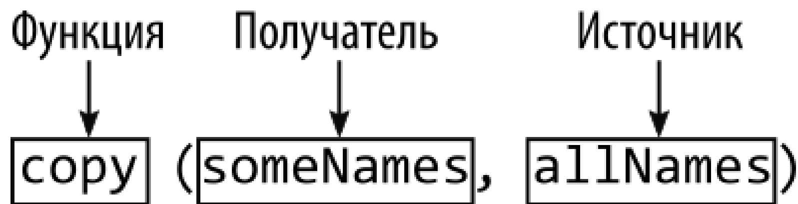

Рисунок 7-22 Использование встроенной функции копирования

Функция копирует элементы в целевой срез. Срезы не обязательно должны иметь одинаковую длину, потому что функция `copy` будет копировать элементы только до тех пор, пока не будет достигнут конец целевого или исходного среза. Размер целевого среза не изменяется, даже если в существующем резервном массиве есть доступная емкость, а это означает, что вы должны убедиться, что его длина достаточна для размещения количества элементов, которые вы хотите скопировать.

Эффект оператора `copy` в листинге 7-25 заключается в том, что элементы копируются из среза `allNames` до тех пор, пока не будет исчерпана длина среза `someNames`. Листинг производит следующий вывод при компиляции и выполнении:

```
someNames: [Lifejacket Paddle]
allNames [Lifejacket Paddle Hat]
```

Длина среза `someNames` равна `2`, что означает, что два элемента копируются из среза `allNames`. Даже если бы срез `someNames` имел дополнительную емкость, никакие другие элементы не были бы

скопированы, потому что это длина среза, на которую опирается функция copy.

Понимание ловушки неинициализированных срезов

Как я объяснял в предыдущем разделе, функция copy не изменяет размер целевого среза. Распространенной ошибкой является попытка скопировать элементы в срез, который не был инициализирован, как показано в листинге 7-26.

```
package main

import "fmt"

func main() {

     products := [4]string { "Kayak", "Lifejacket", "Paddle",
"Hat"}

    allNames := products[1:]
    var someNames []string
    copy(someNames, allNames)

    fmt.Println("someNames:", someNames)
    fmt.Println("allNames", allNames)
}
```

Листинг 7-26 Копирование элементов в неинициализированный срез в файле main.go в папке collections

Я заменил оператор, который инициализирует срез someNames, функцией make и заменил его оператором, который определяет переменную someNames без ее инициализации. Этот код компилируется и выполняется без ошибок, но дает следующие результаты:

```
someNames: []
allNames [Lifejacket Paddle Hat]
```

Никакие элементы не были скопированы в целевой срез. Это происходит потому, что неинициализированные срезы имеют нулевую длину и нулевую емкость. Функция copy останавливает копирование, когда достигается длина конечной длины, и, поскольку длина равна

нулю, копирование не происходит. Об ошибках не сообщается, потому что функция copy работала так, как предполагалось, но это редко является ожидаемым эффектом, и это вероятная причина, если вы неожиданно столкнулись с пустым срезом.

Указание диапазонов при копировании срезов

Детальный контроль над копируемыми элементами может быть достигнут с помощью диапазонов, как показано в листинге 7-27.

```
package main

import "fmt"

func main() {

     products := [4]string { "Kayak", "Lifejacket", "Paddle",
"Hat"}

    allNames := products[1:]
    someNames := []string { "Boots", "Canoe"}
    copy(someNames[1:], allNames[2:3])

    fmt.Println("someNames:", someNames)
    fmt.Println("allNames", allNames)
}
```

Листинг 7-27 Использование диапазонов при копировании элементов в файле main.go в папке collections

Диапазон, примененный к целевому срезу, означает, что копируемые элементы будут начинаться с позиции 1. Диапазон, примененный к исходному срезу, означает, что копирование начнется с элемента в позиции 2, и будет скопирован один элемент. Код в листинге 7-27 выдает следующий результат при компиляции и выполнении:

```
someNames: [Boots Hat]
allNames [Lifejacket Paddle Hat]
```

Копирование срезов разного размера

Поведение, которое приводит к проблеме, описанной в разделе «Понимание ловушки неинициализированных срезов», позволяет

копировать срезы разных размеров, если вы помните об их инициализации. Если целевой срез больше исходного, то копирование будет продолжаться до тех пор, пока не будет скопирован последний элемент в источнике, как показано в листинге 7-28.

```
package main

import "fmt"

func main() {

     products := []string { "Kayak", "Lifejacket", "Paddle",
"Hat"}
    replacementProducts := []string { "Canoe", "Boots"}

    copy(products, replacementProducts)

    fmt.Println("products:", products)
}
```

Листинг 7-28 Копирование меньшего исходного среза в файл main.go в папке collections

Исходный срез содержит только два элемента, и диапазон не используется. В результате функция `copy` начинает копирование элементов из среза `replaceProducts` в срез `products` и останавливается, когда достигается конец среза `replaceProducts`. Остальные элементы в срезе продуктов не затрагиваются операцией копирования, как показывают выходные данные примера:

```
products: [Canoe Boots Paddle Hat]
```

Если целевой срез меньше исходного, то копирование продолжается до тех пор, пока все элементы в целевом срезе не будут заменены, как показано в листинге 7-29.

```
package main

import "fmt"

func main() {
```

```
    products := []string { "Kayak", "Lifejacket", "Paddle",
"Hat"}
    replacementProducts := []string { "Canoe", "Boots"}

    copy(products[0:1], replacementProducts)

    fmt.Println("products:", products)
}
```

Листинг 7-29 Копирование исходного среза большего размера в файл main.go в папке collections

Диапазон, используемый для назначения, создает срез длиной один, что означает, что из исходного массива будет скопирован только один элемент, как показано в выводе примера:

```
products: [Canoe Lifejacket Paddle Hat]
```

Удаление элементов среза

Встроенной функции для удаления элементов среза нет, но эту операцию можно выполнить с помощью диапазонов и функции добавления, как показано в листинге 7-30.

```
package main

import "fmt"

func main() {

    products := [4]string { "Kayak", "Lifejacket", "Paddle",
"Hat"}

    deleted := append(products[:2], products[3:]...)
    fmt.Println("Deleted:", deleted)
}
```

Листинг 7-30 Удаление элементов среза в файле main.go в папке collections

Чтобы удалить значение, метод `append` используется для объединения двух диапазонов, содержащих все элементы среза, кроме того, который больше не требуется. Листинг 7-30 дает следующий результат при компиляции и выполнении:

```
Deleted: [Kayak Lifejacket Hat]
```

Перечисление срезов

Срезы нумеруются так же, как и массивы, с ключевыми словами `for` и `range`, как показано в листинге 7-31.

```
package main

import "fmt"

func main() {

    products := []string { "Kayak", "Lifejacket", "Paddle",
"Hat"}

    for index, value := range products[2:] {
        fmt.Println("Index:", index, "Value:", value)
    }
}
```

Листинг 7-31 Перечисление среза в файле main.go в папке collections

Я описываю различные способы использования цикла `for` в листинге 7-31, но в сочетании с ключевым словом `range` ключевое слово `for` может перечислять срез, создавая переменные индекса и значения для каждого элемента. Код в листинге 7-31 выдает следующий результат:

```
Index: 0 Value: Paddle
Index: 1 Value: Hat
```

Сортировка срезов

Встроенной поддержки сортировки срезов нет, но стандартная библиотека включает пакет `sort`, определяющий функции для сортировки различных типов срезов. Пакет `sort` подробно описан в главе 18, но в листинге 7-32 показан простой пример, обеспечивающий некоторый контекст в этой главе.

```
package main

import (
```

```
    "fmt"
    "sort"
)

func main() {

     products := []string { "Kayak", "Lifejacket", "Paddle",
"Hat"}

    sort.Strings(products)

    for index, value := range products {
        fmt.Println("Index:", index, "Value:", value)
    }
}
```

Листинг 7-32 Сортировка среза в файле main.go в папке collections

Функция `Strings` сортирует значения в `[]string` на месте, получая следующие результаты при компиляции и выполнении примера:

```
Index: 0 Value: Hat
Index: 1 Value: Kayak
Index: 2 Value: Lifejacket
Index: 3 Value: Paddle
```

Как объясняется в главе 18, пакет `sort` включает функции для сортировки срезов, содержащих целые числа и строки, а также поддержку сортировки пользовательских типов данных.

Сравнение срезов

Go ограничивает использование оператора сравнения, поэтому срезы можно сравнивать только с нулевым значением. Сравнение двух срезов приводит к ошибке, как показано в листинге 7-33.

```
package main

import (
    "fmt"
    //"sort"
)
```

```
func main() {

    p1 := []string { "Kayak", "Lifejacket", "Paddle", "Hat"}
    p2 := p1

    fmt.Println("Equal:", p1 == p2)
}
```

Листинг 7-33 Сравнение срезов в файле main.go в папке collections

При компиляции этого кода возникает следующая ошибка:

```
.\main.go:13:30: invalid operation: p1 == p2 (slice can only
be compared to nil)
```

Однако есть один способ сравнения срезов. Стандартная библиотека включает пакет с именем `reflect`, который включает в себя удобную функцию `DeepEqual`. Пакет `reflect` описан в главах 27–29 и содержит расширенные функции (именно поэтому для описания предоставляемых им функций требуется три главы). Функцию `DeepEqual` можно использовать для сравнения более широкого диапазона типов данных, чем оператор равенства, включая срезы, как показано в листинге 7-34.

```
package main

import (
    "fmt"
    "reflect"
)

func main() {

    p1 := []string { "Kayak", "Lifejacket", "Paddle", "Hat"}
    p2 := p1

    fmt.Println("Equal:", reflect.DeepEqual(p1, p2))
}
```

Листинг 7-34 Сравнение срезов удобной функцией в файле main.go в папке collections

Функция `DeepEqual` удобна, но вы должны прочитать главы, описывающие пакет `reflect`, чтобы понять, как он работает, прежде

чем использовать его в своих проектах. Листинг производит следующий вывод при компиляции и выполнении:

```
Equal: true
```

Получение массива, лежащего в основе среза

Если у вас есть срез, но вам нужен массив (обычно потому, что функция требует его в качестве аргумента), вы можете выполнить явное преобразование среза, как показано в листинге 7-35.

```
package main

import (
    "fmt"
    //"reflect"
)

func main() {

    p1 := []string { "Kayak", "Lifejacket", "Paddle", "Hat"}
    arrayPtr := (*[3]string)(p1)
    array := *arrayPtr

    fmt.Println(array)

}
```

Листинг 7-35 Получение массива в файле main.go в папке collections

Я выполнил эту задачу в два этапа. Первый шаг — выполнить явное преобразование типа среза `[]string` в `*[3]string`. Следует соблюдать осторожность при указании типа массива, поскольку произойдет ошибка, если количество элементов, требуемых массивом, превысит длину среза. Длина массива может быть меньше длины среза, и в этом случае массив не будет содержать все значения среза. В этом примере в срезе четыре значения, и я указал тип массива, который может хранить три значения, а это означает, что массив будет содержать только первые три значения среза.

На втором шаге я следую за указателем, чтобы получить значение массива, которое затем записывается. Код в листинге 7-35 выдает следующий результат при компиляции и выполнении:

```
[Kayak Lifejacket Paddle]
```

Работа с картами

Карты — это встроенная структура данных, которая связывает значения данных с ключами. В отличие от массивов, где значения связаны с последовательными целочисленными ячейками, карты могут использовать другие типы данных в качестве ключей, как показано в листинге 7-36.

```
package main

import "fmt"

func main() {

    products := make(map[string]float64, 10)

    products["Kayak"] = 279
    products["Lifejacket"] = 48.95

    fmt.Println("Map size:", len(products))
    fmt.Println("Price:", products["Kayak"])
    fmt.Println("Price:", products["Hat"])
}
```

Листинг 7-36 Использование карты в файле main.go в папке collections

Карты создаются с помощью встроенной функции `make`, как и срезы. Тип карты указывается с помощью ключевого слова `map`, за которым следует тип ключа в квадратных скобках, за которым следует тип значения, как показано на рисунке 7-23. Последний аргумент функции `make` указывает начальную емкость карты. Карты, как и срезы, изменяются автоматически, и аргумент размера может быть опущен.

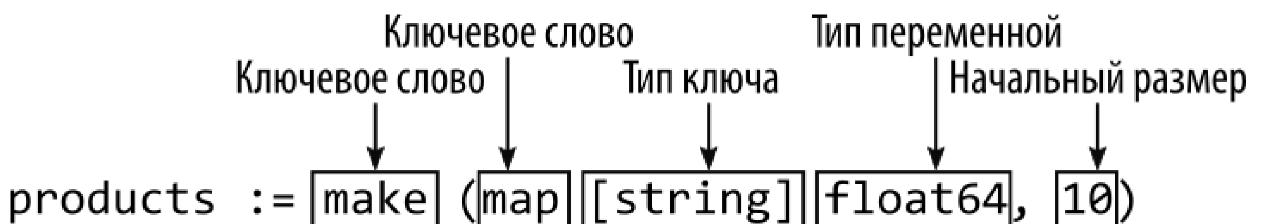

Рисунок 7-23 Определение карты

Оператор в листинге 7-36 будет хранить значения float64, которые индексируются string ключами. Значения хранятся на карте с использованием синтаксиса в стиле массива, с указанием ключа вместо местоположения, например:

```
...
products["Kayak"] = 279
...
```

Этот оператор сохраняет значение float64 с помощью ключа Kayak. Значения считываются с карты с использованием того же синтаксиса:

```
...
fmt.Println("Price:", products["Kayak"])
...
```

Если карта содержит указанный ключ, возвращается значение, связанное с ключом. Нулевое значение для типа значения карты возвращается, если карта не содержит ключ. Количество элементов, хранящихся на карте, получается с помощью встроенной функции len, например:

```
...
fmt.Println("Map size:", len(products))
...
```

Код в листинге 7-36 выдает следующий результат при компиляции и выполнении:

```
Map size: 2
Price: 279
Price: 0
```

Использование литерального синтаксиса карты

Срезы также могут быть определены с использованием литерального синтаксиса, как показано в листинге 7-37.

```
package main

import "fmt"
```

```
func main() {

    products := map[string]float64 {
        "Kayak" : 279,
        "Lifejacket": 48.95,
    }

    fmt.Println("Map size:", len(products))
    fmt.Println("Price:", products["Kayak"])
    fmt.Println("Price:", products["Hat"])
}
```

Листинг 7-37 Использование литерального синтаксиса карты в файле main.go в папке collections

Литеральный синтаксис указывает начальное содержимое карты между фигурными скобками. Каждая запись карты указывается с помощью ключа, двоеточия, значения и запятой, как показано на рисунке 7-24.

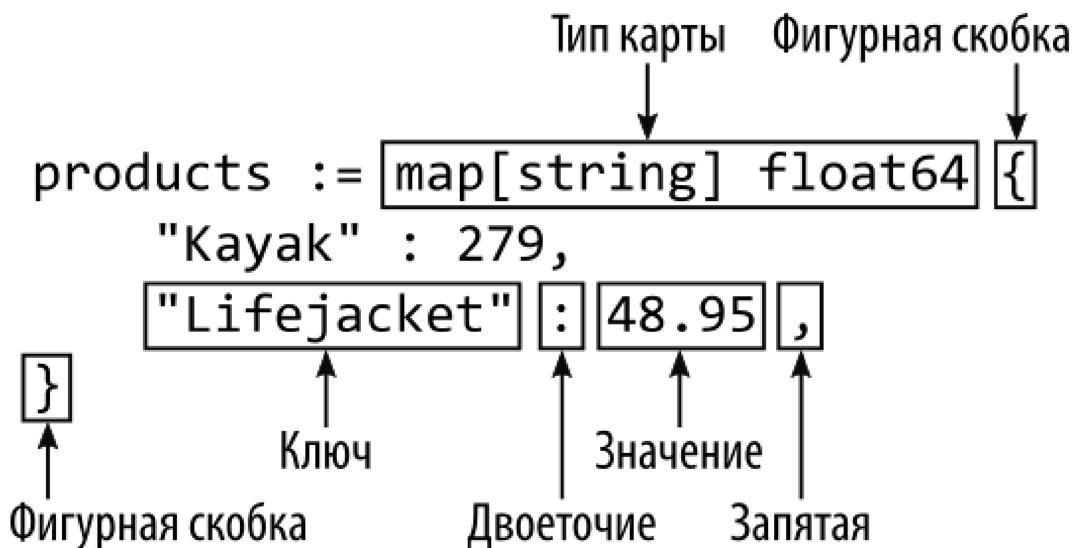

Рисунок 7-24 Литеральный синтаксис карты

Go очень требователен к синтаксису и выдаст ошибку, если за значением карты не следует ни запятая, ни закрывающая фигурная скобка. Я предпочитаю использовать завершающую запятую, которая позволяет поставить закрывающую фигурную скобку на следующую строку в файле кода.

Ключи, используемые в литеральном синтаксисе, должны быть уникальными, и компилятор сообщит об ошибке, если одно и то же имя используется для двух литеральных записей. Листинг 7-37 дает следующий результат при компиляции и выполнении:

```
Map size: 2
Price: 279
Price: 0
```

Проверка элементов в карте

Как отмечалось ранее, карты возвращают нулевое значение для типа значения, когда выполняются чтения, для которых нет ключа. Это может затруднить различение сохраненного значения, которое оказывается нулевым значением, и несуществующего ключа, как показано в листинге 7-38.

```
package main

import "fmt"

func main() {

    products := map[string]float64 {
        "Kayak" : 279,
        "Lifejacket": 48.95,
        "Hat": 0,
    }

    fmt.Println("Hat:", products["Hat"])
}
```

Листинг 7-38 Чтение значений карты в файле main.go в папке collections

Проблема с этим кодом заключается в том, что `products["Hat"]` возвращает ноль, но неизвестно, связано ли это с тем, что ноль является сохраненным значением, или с тем, что с ключом `Hat` не связано никакого значения. Чтобы решить эту проблему, карты создают два значения при чтении значения, как показано в листинге 7-39.

```
package main
```

```
import "fmt"

func main() {

    products := map[string]float64 {
        "Kayak" : 279,
        "Lifejacket": 48.95,
        "Hat": 0,
    }

    value, ok := products["Hat"]

    if (ok) {
        fmt.Println("Stored value:", value)
    } else {
        fmt.Println("No stored value")
    }
}
```

Листинг 7-39 Определение наличия значения на карте в файле main.go в папке collections

Это известно как метод «запятая ок», когда значения присваиваются двум переменным при чтении значения из карты:

```
...
value, ok := products["Hat"]
...
```

Первое значение — это либо значение, связанное с указанным ключом, либо нулевое значение, если ключ отсутствует. Второе значение — это логическое значение, которое равно `true`, если карта содержит указанный ключ, и `false` в противном случае. Второе значение обычно присваивается переменной с именем `ok`, откуда и возникает термин «запятая ok».

Этот метод можно упростить с помощью оператора инициализации, как показано в листинге 7-40.

```
package main

import "fmt"

func main() {
```

```
    products := map[string]float64 {
        "Kayak" : 279,
        "Lifejacket": 48.95,
        "Hat": 0,
    }

    if value, ok := products["Hat"]; ok {
        fmt.Println("Stored value:", value)
    } else {
        fmt.Println("No stored value")
    }
}
```

Листинг 7-40 Использование оператора инициализации в файле main.go в папке collections

Код в листингах 7-39 и 7-39 выдает после компиляции и выполнения следующий вывод, показывающий, что ключ `Hat` использовался для сохранения значения `0` в карте:

```
Stored value: 0
```

Удаление объектов с карты

Элементы удаляются с карты с помощью встроенной функции удаления, как показано в листинге 7-41.

```
package main

import "fmt"

func main() {

    products := map[string]float64 {
        "Kayak" : 279,
        "Lifejacket": 48.95,
        "Hat": 0,
    }

    delete(products, "Hat")

    if value, ok := products["Hat"]; ok {
        fmt.Println("Stored value:", value)
    } else {
```

```
        fmt.Println("No stored value")
    }
}
```

Листинг 7-41 Удаление с карты в файле main.go в папке collections

Аргументами функции `delete` являются карта и ключ для удаления. Об ошибке не будет сообщено, если указанный ключ не содержится в карте. Код в листинге 7-41 выдает следующий результат при компиляции и выполнении, подтверждая, что ключ `Hat` больше не находится в карте:

```
No stored value
```

Перечисление содержимого карты

Карты перечисляются с использованием ключевых слов `for` и `range`, как показано в листинге 7-42.

```
package main

import "fmt"

func main() {

    products := map[string]float64 {
        "Kayak" : 279,
        "Lifejacket": 48.95,
        "Hat": 0,
    }

    for key, value := range products {
        fmt.Println("Key:", key, "Value:", value)
    }
}
```

Листинг 7-42 Перечисление карты в файле main.go в папке collections

Когда ключевые слова `for` и `range` используются с картой, двум переменным присваиваются ключи и значения по мере перечисления содержимого карты. Код в листинге `7-42` выдает следующий результат при компиляции и выполнении (хотя они могут отображаться в другом порядке, как я объясню в следующем разделе):

```
Key: Kayak Value: 279
Key: Lifejacket Value: 48.95
Key: Hat Value: 0
```

Перечисление карты по порядку

Вы можете увидеть результаты из листинга 7-42 в другом порядке, потому что нет никаких гарантий, что содержимое карты будет пронумеровано в каком-либо конкретном порядке. Если вы хотите получить значения на карте по порядку, то лучший подход — перечислить карту и создать срез, содержащий ключи, отсортировать срез, а затем пронумеровать срез для чтения значений с карты, как показано на Листинг 7-43.

```
package main

import (
    "fmt"
    "sort"
)

func main() {

    products := map[string]float64 {
        "Kayak" : 279,
        "Lifejacket": 48.95,
        "Hat": 0,
    }

    keys := make([]string, 0, len(products))
    for key, _ := range products {
        keys = append(keys, key)
    }
    sort.Strings(keys)
    for _, key := range keys {
        fmt.Println("Key:", key, "Value:", products[key])
    }
}
```

Листинг 7-43 Перечисление карты в ключевом порядке в файле main.go в папке collections

Скомпилируйте и выполните проект, и вы увидите следующий вывод, который отображает значения, отсортированные в порядке их

ключа:

```
Key: Hat Value: 0
Key: Kayak Value: 279
Key: Lifejacket Value: 48.95
```

Понимание двойной природы строк

В главе 4 я описал строки как последовательности символов. Это правда, но есть сложности, потому что строки Go имеют раздвоение личности в зависимости от того, как вы их используете.

Go обрабатывает строки как массивы байтов и поддерживает нотацию индекса массива и диапазона среза, как показано в листинге 7-44.

```
package main

import (
    "fmt"
    "strconv"
)

func main() {

    var price string = "$48.95"

    var currency byte = price[0]
    var amountString string = price[1:]
    amount, parseErr  := strconv.ParseFloat(amountString, 64)

    fmt.Println("Currency:", currency)
    if (parseErr == nil) {
        fmt.Println("Amount:", amount)
    } else {
        fmt.Println("Parse Error:", parseErr)
    }
}
```

Листинг 7-44 Индексирование и создание среза строки в файле main.go в папке collections

Я использовал полный синтаксис объявления переменных, чтобы подчеркнуть тип каждой переменной. Когда используется нотация индекса, результатом является byte из указанного места в строке:

```
...
var currency byte = price[0]
...
```

Этот оператор выбирает byte в нулевой позиции и присваивает его переменной с именем currency. Когда строка нарезается, срез также описывается с использованием байтов, но результатом является string:

```
...
var amountString string = price[1:]
...
```

Диапазон выбирает все, кроме байта в нулевом местоположении, и присваивает укороченную строку переменной с именем amountString. Этот код выдает следующий результат при компиляции и выполнении с помощью команды, показанной в листинге 7-44:

```
Currency: 36
Amount: 48.95
```

Как я объяснял в главе 4, тип byte является псевдонимом для uint8, поэтому значение currency отображается в виде числа: Go понятия не имеет, что числовое значение 36 должно выражаться знаком доллара. На рисунке 7-25 строка представлена как массив байтов и показано, как они индексируются и нарезаются.

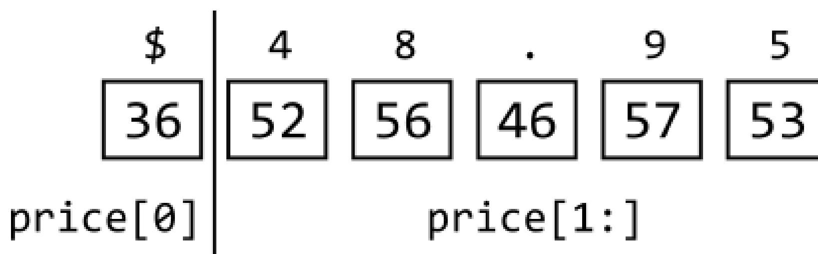

Рисунок 7-25 Строка как массив байтов

При разрезании строки получается другая строка, но для интерпретации byte как символа, который он представляет, требуется

явное преобразование, как показано в листинге 7-45.

```
package main

import (
    "fmt"
    "strconv"
)

func main() {

    var price string = "$48.95"

    var currency string = string(price[0])
    var amountString string = price[1:]
    amount, parseErr  := strconv.ParseFloat(amountString, 64)

    fmt.Println("Currency:", currency)
    if (parseErr == nil) {
        fmt.Println("Amount:", amount)
    } else {
        fmt.Println("Parse Error:", parseErr)
    }
}
```

Листинг 7-45 Преобразование результата в файл main.go в папку collections

Скомпилируйте и выполните код, и вы увидите следующие результаты:

```
Currency: $
Amount: 48.95
```

Похоже, что это работает, но в нем есть ловушка, которую можно увидеть, если изменить символ валюты, как показано в листинге 7-46. (Если вы не живете в той части мира, где на клавиатуре есть символ валюты евро, удерживайте нажатой клавишу **Alt** и нажмите 0128 на цифровой клавиатуре.)

```
package main

import (
    "fmt"
```

```
    "strconv"
)

func main() {

    var price string = "€48.95"

    var currency string = string(price[0])
    var amountString string = price[1:]
    amount, parseErr  := strconv.ParseFloat(amountString, 64)

    fmt.Println("Currency:", currency)
    if (parseErr == nil) {
        fmt.Println("Amount:", amount)
    } else {
        fmt.Println("Parse Error:", parseErr)
    }
}
```

Листинг 7-46 Изменение символа валюты в файле main.go в папке collections

Скомпилируйте и выполните код, и вы увидите вывод, подобный следующему:

```
Currency: â
Parse Error: strconv.ParseFloat: parsing "\x82\xac48.95": invalid syntax
```

Проблема в том, что нотации массива и диапазона выбирают байты, но не все символы выражаются одним байтом. Новый символ валюты хранится в трех байтах, как показано на рисунке 7-26.

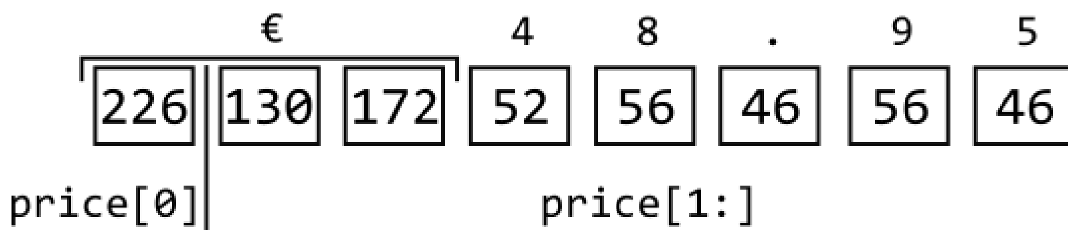

Рисунок 7-26 Изменение символа валюты

На рисунке показано, как при взятии одного байтового значения получается только часть символа валюты. Также видно, что срез включает в себя два из трех байтов символа, за которыми следует

остальная часть строки. Вы можете подтвердить, что изменение символа валюты увеличило размер массива, используя функцию `len`, как показано в листинге 7-47.

```
package main

import (
    "fmt"
    "strconv"
)

func main() {

    var price string = "€48.95"

    var currency string = string(price[0])
    var amountString string = price[1:]
    amount, parseErr  := strconv.ParseFloat(amountString, 64)

    fmt.Println("Length:", len(price))
    fmt.Println("Currency:", currency)
    if (parseErr == nil) {
        fmt.Println("Amount:", amount)
    } else {
        fmt.Println("Parse Error:", parseErr)
    }
}
```

Листинг 7-47 Получение длины строки в файле main.go в папке collections

Функция `len` обрабатывает строку как массив байтов, и код в листинге 7-47 выдает следующий результат при компиляции и выполнении:

```
Length: 8
Currency: â
Parse Error: strconv.ParseFloat: parsing "\x82\xac48.95": invalid syntax
```

Вывод подтверждает, что в строке восемь байтов, и это причина того, что индексация и нарезка дают странные результаты.

Преобразование строки в руны

Тип `rune` представляет собой кодовую точку Unicode, которая по сути является одним символом. Чтобы избежать нарезки строк в середине символов, можно выполнить явное преобразование в срез рун, как показано в листинге 7-48.

> **Подсказка**
> Юникод невероятно сложен, чего и следовало ожидать от любого стандарта, целью которого является описание нескольких систем письма, которые развивались на протяжении тысячелетий. В этой книге я не описываю Unicode и для простоты рассматриваю значения `rune` как одиночные символы, чего достаточно для большинства проектов разработки. Я достаточно описываю Unicode, чтобы объяснить, как работают функции Go.

```
package main

import (
    "fmt"
    "strconv"
)

func main() {

    var price []rune = []rune("€48.95")

    var currency string = string(price[0])
    var amountString string = string(price[1:])
    amount, parseErr  := strconv.ParseFloat(amountString, 64)

    fmt.Println("Length:", len(price))
    fmt.Println("Currency:", currency)
    if (parseErr == nil) {
        fmt.Println("Amount:", amount)
    } else {
        fmt.Println("Parse Error:", parseErr)
    }
}
```

Листинг 7-48 Преобразование в руны в файле main.go в папке collections

Я применяю явное преобразование к литеральной строке и присваиваю срез переменной price. При работе со срезом рун отдельные байты группируются в символы, которые они представляют, без ссылки на количество байтов, которое требуется для каждого символа, как показано на рисунке 7-27.

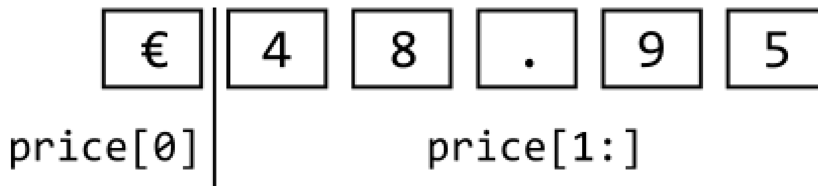

Рисунок 7-27 Срез руны

Как объяснялось в главе 4, тип rune является псевдонимом для int32, что означает, что при печати значения руны будет отображаться числовое значение, используемое для представления символа. Это означает, что, как и в предыдущем примере с байтами, я должен выполнить явное преобразование одной руны в строку, например:

```
...
var currency string = string(price[0])
...
```

Но, в отличие от предыдущих примеров, я также должен выполнить явное преобразование создаваемого среза, например::

```
...
var amountString string = string(price[1:])
...
```

Результатом среза является []rune; иными словами, разрезание среза руны дает другой срез руны. Код в листинге 7-48 выдает следующий результат при компиляции и выполнении:

```
Length: 6
Currency: €
Amount: 48.95
```

Функция len возвращает 6, поскольку массив содержит символы, а не байты. И, конечно же, остальная часть вывода соответствует

ожиданиям, потому что нет потерянных байтов, которые могли бы повлиять на результат.

ПОНИМАНИЕ ПОЧЕМУ И БАЙТЫ, И РУНЫ ПОЛЕЗНЫ

Подход, который Go использует для строк, может показаться странным, но у него есть свое применение. Байты важны, когда вы заботитесь о хранении строк, и вам нужно знать, сколько места нужно выделить. Символы важны, когда вы имеете дело с содержимым строк, например, при вставке нового символа в существующую строку.

Обе грани строк важны. Однако важно понимать, нужно ли вам иметь дело с байтами или символами для той или иной операции.

У вас может возникнуть соблазн работать только с байтами, что будет работать до тех пор, пока вы используете только те символы, которые представлены одним байтом, что обычно означает ASCII. Сначала это может сработать, но почти всегда заканчивается плохо, особенно когда ваш код обрабатывает символы, введенные пользователем с набором символов, отличным от ASCII, или обрабатывает файл, содержащий данные, отличные от ASCII. Для небольшого объема дополнительной работы проще и безопаснее признать, что Unicode действительно существует, и полагаться на Go для преобразования байтов в символы.

Перечисление строк

Цикл `for` можно использовать для перечисления содержимого строки. Эта функция показывает некоторые умные аспекты того, как Go работает с отображением байтов в руны. В листинге 7-49 перечисляется строка.

```
package main

import (
    "fmt"
    //"strconv"
)

func main() {
```

```
    var price = "€48.95"

    for index, char := range price {
        fmt.Println(index, char, string(char))
    }
}
```

Листинг 7-49 Перечисление строки в файле main.go в папке collections

В этом примере я использовал строку, содержащую символ валюты евро, что демонстрирует, что Go обрабатывает строки как последовательность рун при использовании с циклом `for`. Скомпилируйте и выполните код из листинга 7-49, и вы получите следующий вывод:

```
0 8364 €
3 52 4
4 56 8
5 46 .
6 57 9
7 53 5
```

Цикл `for` обрабатывает строку как массив элементов. Записанные значения представляют собой индекс текущего элемента, числовое значение этого элемента и числовой элемент, преобразованный в строку.

Обратите внимание, что значения индекса не являются последовательными. Цикл `for` обрабатывает строку как последовательность символов, полученную из базовой последовательности байтов. Значения индекса соответствуют первому байту, из которого состоит каждый символ, как показано на рисунке 7-2. Второе значение индекса равно `3`, например, потому что первый символ в строке состоит из байтов в позициях `0`, `1` и `2`.

Если вы хотите перечислить базовые байты без их преобразования в символы, вы можете выполнить явное преобразование в байтовый срез, как показано в листинге 7-50.

```
package main

import (
    "fmt"
```

```
    //"strconv"
)

func main() {

    var price = "€48.95"

    for index, char := range []byte(price) {
        fmt.Println(index, char)
    }
}
```

Листинг 7-50 Перечисление байтов в строке в файле main.go в папке collections

Скомпилируйте и выполните этот код с помощью команды, показанной в листинге 7-50, и вы увидите следующий вывод:

```
0 226
1 130
2 172
3 52
4 56
5 46
6 57
7 53
```

Значения индекса являются последовательными, а значения отдельных байтов отображаются без интерпретации как части символов, которые они представляют.

Резюме

В этой главе я описал типы коллекций Go. Я объяснил, что массивы — это последовательности значений фиксированной длины, срезы — это последовательности переменной длины, поддерживаемые массивом, а карты — это наборы пар ключ-значение. Я продемонстрировал использование диапазонов для выбора элементов, объяснил связи между срезами и лежащими в их основе массивами и показал, как выполнять распространенные задачи, такие как удаление элемента из среза, для которых нет встроенных функций. Я закончил эту главу, объяснив сложную природу строк, которая может вызвать проблемы у

программистов, которые предполагают, что все символы могут быть представлены с помощью одного байта данных. В следующей главе я объясню использование функций в Go.

8. Определение и использование функций

В этой главе я описываю функции Go, которые позволяют группировать операторы кода и выполнять их по мере необходимости. Функции Go обладают некоторыми необычными характеристиками, наиболее полезной из которых является возможность определения нескольких результатов. Как я объясняю, это элегантное решение общей проблемы, с которой сталкиваются функции. Таблица 8-1 помещает функции в контекст.

Таблица 8-1 Помещение функций в контекст

Вопрос	Ответ
Кто они такие?	Функции — это группы операторов кода, которые выполняются только тогда, когда функция вызывается во время выполнения.
Почему они полезны?	Функции позволяют определить свойства один раз и использовать их многократно.
Как они используются?	Функции вызываются по имени и могут быть снабжены значениями данных, с которыми можно работать, используя параметры. Результат выполнения операторов в функции может быть получен как результат функции.
Есть ли подводные камни или ограничения?	Функции Go ведут себя в основном так, как ожидалось, с добавлением полезных функций, таких как множественные результаты и именованные результаты.
Есть ли альтернативы?	Нет, функции — это основная особенность языка Go.

Таблица 8-2 суммирует главу.

Таблица 8-2 Краткое содержание главы

Проблема	Решение	Листинг
Групповые операторы, чтобы их можно было выполнять по мере необходимости	Определите функцию	4

Проблема	Решение	Листинг
Определите функцию, чтобы можно было изменить значения, используемые содержащимися в ней операторами.	Определить параметры функции	5–8
Разрешить функции принимать переменное количество аргументов	Определить переменный параметр	9–13
Использовать ссылки на значения, определенные вне функции	Определите параметры, которые принимают указатели	14, 15
Производить вывод из операторов, определенных в функции	Определите один или несколько результатов	16–22
Игнорировать результат, полученный функцией	Используйте пустой идентификатор	23
Запланировать вызов функции, когда текущая выполняемая функция будет завершена	Используйте ключевое слово `defer`	24

Подготовка к этой главе

Чтобы подготовиться к этой главе, откройте новую командную строку, перейдите в удобное место и создайте каталог с именем `functions`. Перейдите в папку `functions` и выполните команду, показанную в листинге 8-1, чтобы инициализировать проект.

> **Подсказка**
> Вы можете загрузить пример проекта для этой главы — и для всех остальных глав этой книги — с `https://github.com/apress/pro-go`. См. Главу 2 о том, как получить помощь, если у вас возникнут проблемы с запуском примеров.

```
go mod init functions
```

Листинг 8-1 Инициализация проекта

Добавьте файл с именем `main.go` в папку `functions` с содержимым, показанным в листинге 8-2.

```
package main

import "fmt"
```

```
func main() {

    fmt.Println("Hello, Functions")
}
```

Листинг 8-2 Содержимое файла main.go в папке functions.

Используйте командную строку для запуска команды, показанной в листинге 8-3, в папке `functions`.

```
go run .
```

Листинг 8-3 Запуск примера проекта

Код в файле `main.go` будет скомпилирован и выполнен, что приведет к следующему результату:

```
Hello, Functions
```

Определение простой функции

Функции — это группы операторов, которые можно использовать и повторно использовать как одно действие. Для начала в листинге 8-4 определяется простая функция.

```
package main

import "fmt"

func printPrice() {
    kayakPrice := 275.00
    kayakTax := kayakPrice * 0.2
    fmt.Println("Price:", kayakPrice, "Tax:", kayakTax)
}

func main() {
    fmt.Println("About to call function")
    printPrice()
    fmt.Println("Function complete")
}
```

Листинг 8-4 Определение функции в файле main.go в папке functions

Функции определяются ключевым словом `func`, за которым следует имя функции, круглые скобки и блок кода, заключенный в фигурные скобки, как показано на рисунке 8-1.

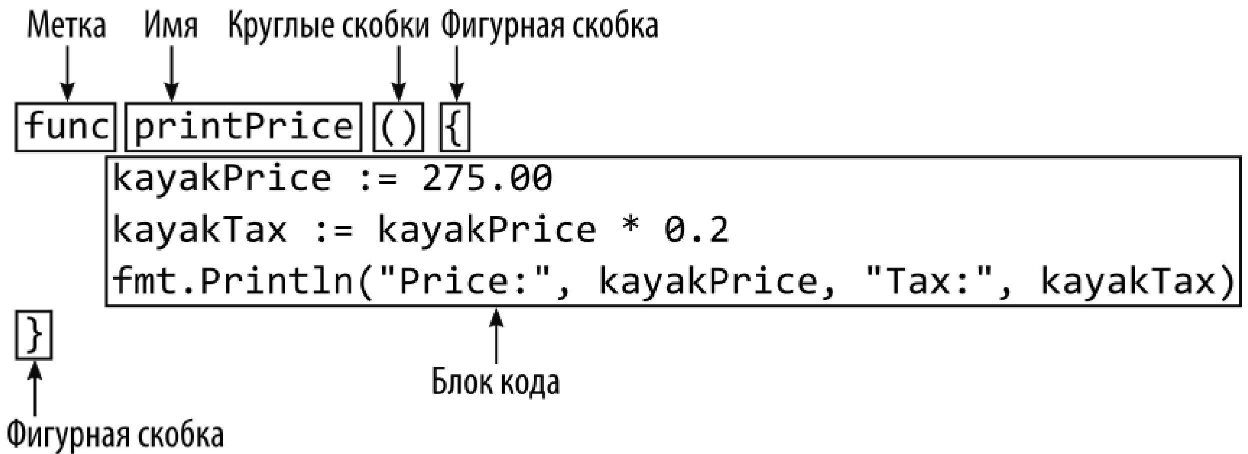

Рисунок 8-1 Анатомия функции

Теперь в файле кода `main.go` есть две функции. Новая функция называется `printPrice` и содержит операторы, определяющие две переменные и вызывающие функцию `Println` из пакета `fmt`. Основная функция — это точка входа в приложение, где начинается и заканчивается выполнение. Функции Go должны быть определены с помощью фигурных скобок, а открывающая фигурная скобка должна быть определена в той же строке, что и ключевое слово `func` и имя функции. Условные обозначения, принятые в других языках, такие как опускание фигурных скобок или размещение фигурной скобки на следующей строке, не допускаются.

> **Примечание**
> Обратите внимание, что функция `printPrice` определена вместе с существующей основной функцией в файле `main.go`. Go поддерживает определение функций внутри других функций, но требуется другой синтаксис, как описано в главе 9.

Основная функция вызывает функцию `printPrice`, что делается с помощью оператора, указывающего имя функции, за которым следуют круглые скобки, как показано на рисунке 8-2.

Рисунок 8-2 Вызов функции

При вызове функции выполняются операторы, содержащиеся в блоке кода функции. Когда все операторы были вызваны, выполнение продолжается с оператора, следующего за оператором, вызвавшим функцию. Это можно увидеть в выводе кода в листинге 8-4, когда он скомпилирован и выполнен:

```
About to call function
Price: 275 Tax: 55
Function complete
```

Определение и использование параметров функции

Параметры позволяют функции получать значения данных при ее вызове, что позволяет изменить ее поведение. Листинг 8-5 изменяет функцию `printPrice`, определенную в предыдущем разделе, так что она определяет параметры.

```
package main

import "fmt"

func printPrice(product string, price float64, taxRate
float64) {
    taxAmount := price * taxRate
    fmt.Println(product, "price:", price, "Tax:", taxAmount)
}

func main() {
    printPrice("Kayak", 275, 0.2)
    printPrice("Lifejacket", 48.95, 0.2)
    printPrice("Soccer Ball", 19.50, 0.15)
}
```

Листинг 8-5 Определение параметров функции в файле main.go в папке functions.

Параметры определяются именем, за которым следует тип. Несколько параметров разделяются запятыми, как показано на рисунке 8-3.


```
func printPrice(product string, price float64, taxRate float64) {
    taxAmount := price * taxRate
    fmt.Println(product, "price:", price, "Tax:", taxAmount)
}
```

Рисунок 8-3 Определение параметров функции

В листинге 8-5 к функции `printPrice` добавлены три параметра: строка с именем `product`, переменная `float64` именованная `price` и именованная переменная `float64` с именем `taxRate`. В блоке кода функции доступ к значению, присвоенному параметру, осуществляется по его имени, как показано на рисунке 8-4.


```
func printPrice(product string, price float64 , taxRate float64) {
    taxAmount := price * taxRate
    fmt.Println(product, "price:", price, "Tax:", taxAmount)
}
```


Рисунок 8-4 Доступ к параметру внутри блока кода

Значения параметров передаются в качестве аргументов при вызове функции, что означает, что каждый раз при вызове функции могут быть предоставлены разные значения. Аргументы указываются между круглыми скобками, которые следуют за именем функции, через запятую и в том же порядке, в котором были определены параметры, как показано на рисунке 8-5.


```
printPrice ("Lifejacket", 48.95, 0.2)
```

Рисунок 8-5 Вызов функции с аргументами

Значения, используемые в качестве аргументов, должны соответствовать типам параметров, определенных функцией. Код в листинге 8-5 выдает следующий результат при компиляции и выполнении:

```
Kayak price: 275 Tax: 55
Lifejacket price: 48.95 Tax: 9.790000000000001
Soccer Ball price: 19.5 Tax: 2.925
```

Значение, отображаемое для продукта `Lifejacket`, содержит значение длинной дроби, которое обычно округляется для сумм в валюте. Я объясню, как форматировать числовые значения в виде строк, в главе 17.

> **Примечание**
> Go не поддерживает необязательные параметры или значения по умолчанию для параметров.

Пропуск типов параметров

Тип можно не указывать, если смежные параметры имеют одинаковый тип, как показано в листинге 8-6.

```
package main

import "fmt"

func printPrice(product string, price, taxRate float64) {
    taxAmount := price * taxRate
    fmt.Println(product, "price:", price, "Tax:", taxAmount)
}

func main() {
    printPrice("Kayak", 275, 0.2)
    printPrice("Lifejacket", 48.95, 0.2)
    printPrice("Soccer Ball", 19.50, 0.15)
}
```

Листинг 8-6 Пропуск типа данных параметра в файле main.go в папке functions

Оба параметра `price` и `taxRate` имеют тип `float64`, и, поскольку они являются смежными, тип данных применяется только к последнему параметру этого типа. Пропуск типа данных параметра не меняет параметр или его тип. Код в листинге 8-6 выдает следующий результат:

```
Kayak price: 275 Tax: 55
Lifejacket price: 48.95 Tax: 9.790000000000001
Soccer Ball price: 19.5 Tax: 2.925
```

Пропуск имен параметров

Символ подчеркивания (символ _) может использоваться для параметров, определенных функцией, но не используемых в операторах кода функции, как показано в листинге 8-7.

```
package main

import "fmt"

func printPrice(product string, price, _ float64) {
    taxAmount := price * 0.25
    fmt.Println(product, "price:", price, "Tax:", taxAmount)
}

func main() {
    printPrice("Kayak", 275, 0.2)
    printPrice("Lifejacket", 48.95, 0.2)
    printPrice("Soccer Ball", 19.50, 0.15)
}
```

Листинг 8-7 Отсутствие имени параметра в файле main.go в папке functions.

Знак подчеркивания известен как *пустой идентификатор*, а результат — это параметр, значение которого должно быть предоставлено при вызове функции, но значение которого недоступно внутри блока кода функции. Это может показаться странным, но это может быть полезным способом указать, что параметр не используется внутри функции, что может возникнуть при реализации методов, требуемых интерфейсом. Код в листинге 8-7 выдает следующий результат при компиляции и выполнении:

```
Kayak price: 275 Tax: 68.75
```

```
Lifejacket price: 48.95 Tax: 12.2375
Soccer Ball price: 19.5 Tax: 4.875
```

Функции также могут опускать имена во всех своих параметрах, как показано в листинге 8-8.

```
package main

import "fmt"

func printPrice(string, float64, float64) {
    // taxAmount := price * 0.25
   fmt.Println("No parameters")
}

func main() {
    printPrice("Kayak", 275, 0.2)
    printPrice("Lifejacket", 48.95, 0.2)
    printPrice("Soccer Ball", 19.50, 0.15)
}
```

Листинг 8-8 Пропуск всех имен параметров в файле main.go в папке functions

Параметры без имен не могут быть доступны внутри функции, и эта функция в основном используется в сочетании с интерфейсами, описанными в главе 11, или при определении типов функций, описанных в главе 9. Листинг 8-8 дает следующий результат при компиляции и выполнении:

```
No parameters
No parameters
No parameters
```

Определение вариационных параметров

Вариативный параметр принимает переменное количество значений, что может упростить использование функций. Чтобы понять проблему, которую решают вариативные параметры, полезно рассмотреть альтернативу, показанную в листинге 8-9.

```
package main

import "fmt"
```

```
func printSuppliers(product string, suppliers []string ) {
    for _, supplier := range suppliers {
            fmt.Println("Product:", product, "Supplier:",
supplier)
    }
}

func main() {
     printSuppliers("Kayak", []string {"Acme Kayaks", "Bob's
Boats", "Crazy Canoes"})
    printSuppliers("Lifejacket", []string {"Sail Safe Co"})
}
```

Листинг 8-9 Определение функции в файле main.go в папке functions

Второй параметр, определенный функцией `printSuppliers`, принимает переменное количество поставщиков, используя `string` срез. Это работает, но может быть неудобным, поскольку требует построения срезов, даже если требуется только одна строка, например:

```
...
printSuppliers("Lifejacket", []string {"Sail Safe Co"})
...
```

Переменные параметры позволяют функции более элегантно получать переменное число аргументов, как показано в листинге 8-10.

```
package main

import "fmt"

func printSuppliers(product string, suppliers ...string ) {
    for _, supplier := range suppliers {
            fmt.Println("Product:", product, "Supplier:",
supplier)
    }
}

func main() {
      printSuppliers("Kayak", "Acme Kayaks", "Bob's Boats",
"Crazy Canoes")
    printSuppliers("Lifejacket", "Sail Safe Co")
}
```

Листинг 8-10 Определение вариативного параметра в файле main.go в папке functions

Вариативный параметр определяется многоточием (три точки), за которым следует тип, как показано на рисунке 8-6.

```
                                         Вариативный параметр
                                                 ↓
func printSuppliers(product string, suppliers ...string) {
```

Рисунок 8-6 Вариативный параметр

Вариативный параметр должен быть последним параметром, определенным функцией, и может использоваться только один тип, например строковый тип в этом примере. При вызове функции можно указать переменное количество строковых аргументов без необходимости создания среза:

```
...
printSuppliers("Kayak", "Acme Kayaks", "Bob's Boats", "Crazy Canoes")
...
```

Тип вариативного параметра не меняется, а предоставленные значения по-прежнему содержатся в срезе. Для листинга 8-10 это означает, что тип параметра `suppliers` остается `[]string`. Код в листингах 8-9 и 8-10 выдает следующий результат при компиляции и выполнении:

```
Product: Kayak Supplier: Acme Kayaks
Product: Kayak Supplier: Bob's Boats
Product: Kayak Supplier: Crazy Canoes
Product: Lifejacket Supplier: Sail Safe Co
```

Работа без аргументов для вариационного параметра

Go позволяет полностью опустить аргументы для переменных параметров, что может привести к неожиданным результатам, как показано в листинге 8-11.

```
package main

import "fmt"
```

```
func printSuppliers(product string, suppliers ...string ) {
    for _, supplier := range suppliers {
            fmt.Println("Product:", product, "Supplier:",
supplier)
    }
}

func main() {
      printSuppliers("Kayak", "Acme Kayaks", "Bob's Boats",
"Crazy Canoes")
    printSuppliers("Lifejacket", "Sail Safe Co")
    printSuppliers("Soccer Ball")
}
```

Листинг 8-11 Пропуск аргументов в файле main.go в папке functions

Новый вызов функции `printSuppliers` не предоставляет никаких аргументов для параметра `suppliers`. Когда это происходит, Go использует `nil` в качестве значения параметра, что может вызвать проблемы с кодом, предполагающим, что в срезе будет хотя бы одно значение. Скомпилируйте и запустите код из листинга 8-11; вы получите следующий вывод:

```
Product: Kayak Supplier: Acme Kayaks
Product: Kayak Supplier: Bob's Boats
Product: Kayak Supplier: Crazy Canoes
Product: Lifejacket Supplier: Sail Safe Co
```

Для продукта `Soccer Ball` нет выходных данных, поскольку срезы `nil` имеют нулевую длину, поэтому цикл `for` никогда не выполняется. Листинг 8-12 устраняет эту проблему, проверяя эту проблему.

```
package main

import "fmt"

func printSuppliers(product string, suppliers ...string ) {
    if (len(suppliers) == 0) {
        fmt.Println("Product:", product, "Supplier: (none)")
    } else {
        for _, supplier := range suppliers {
```

```
                fmt.Println("Product:", product, "Supplier:",
supplier)
        }
    }
}

func main() {
      printSuppliers("Kayak", "Acme Kayaks", "Bob's Boats",
"Crazy Canoes")
    printSuppliers("Lifejacket", "Sail Safe Co")
    printSuppliers("Soccer Ball")
}
```

Листинг 8-12 Проверка наличия пустых срезов в файле main.go в папке functions

Я использовал встроенную функцию `len`, описанную в главе 7, для идентификации пустых срезов, хотя мог бы также проверить значение `nil`. Скомпилируйте и выполните код; вы получите следующий вывод, который обслуживает функцию, вызываемую без значений для вариационного параметра:

```
Product: Kayak Supplier: Acme Kayaks
Product: Kayak Supplier: Bob's Boats
Product: Kayak Supplier: Crazy Canoes
Product: Lifejacket Supplier: Sail Safe Co
Product: Soccer Ball Supplier: (none)
```

Использование срезов в качестве значений переменных параметров

Вариативные параметры позволяют вызывать функцию без создания срезов, но это бесполезно, если у вас уже есть срез, который вы хотите использовать. В этих ситуациях после последнего аргумента, переданного в функцию с многоточием, можно будет использовать срез, как показано в листинге 8-13.

```
package main

import "fmt"

func printSuppliers(product string, suppliers ...string ) {
    if (len(suppliers) == 0) {
```

```
        fmt.Println("Product:", product, "Supplier: (none)")
    } else {
        for _, supplier := range suppliers {
                fmt.Println("Product:", product, "Supplier:",
supplier)
        }
    }
}

func main() {

     names := []string {"Acme Kayaks", "Bob's Boats", "Crazy
Canoes"}

    printSuppliers("Kayak", names...)
    printSuppliers("Lifejacket", "Sail Safe Co")
    printSuppliers("Soccer Ball")
}
```

Листинг 8-13 Использование среза в качестве аргумента в файле main.go в папке functions

Этот метод позволяет избежать необходимости распаковывать срез на отдельные значения, чтобы их можно было снова объединить в срез для вариативного параметра. Скомпилируйте и выполните код из листинга 8-13, и вы получите следующий вывод:

```
Product: Kayak Supplier: Acme Kayaks
Product: Kayak Supplier: Bob's Boats
Product: Kayak Supplier: Crazy Canoes
Product: Lifejacket Supplier: Sail Safe Co
Product: Soccer Ball Supplier: (none)
```

Использование указателей в качестве параметров функций

По умолчанию Go копирует значения, используемые в качестве аргументов, поэтому изменения ограничиваются функцией, как показано в листинге 8-14.

```
package main

import "fmt"

func swapValues(first, second int) {
```

```
    fmt.Println("Before swap:", first, second)
    temp := first
    first = second
    second = temp
    fmt.Println("After swap:", first, second)
}

func main() {

    val1, val2 := 10, 20
    fmt.Println("Before calling function", val1, val2)
    swapValues(val1, val2)
    fmt.Println("After calling function", val1, val2)
}
```

Листинг 8-14 Изменение значения параметра в файле main.go в папке functions

Функция `swapValues` получает два значения `int`, записывает их, меняет местами и снова записывает. Значения, переданные функции, записываются до и после вызова функции. Вывод из листинга 8-14 показывает, что изменения, внесенные в значения в функции `swpValues`, не влияют на переменные, определенные в функции `main`:

```
Before calling function 10 20
Before swap: 10 20
After swap: 20 10
After calling function 10 20
```

Go позволяет функциям получать указатели, что меняет это поведение, как показано в листинге 8-15.

```
package main

import "fmt"

func swapValues(first, second *int) {
    fmt.Println("Before swap:", *first, *second)
    temp := *first
    *first = *second
    *second = temp
    fmt.Println("After swap:", *first, *second)
}
```

```
func main() {

    val1, val2 := 10, 20
    fmt.Println("Before calling function", val1, val2)
    swapValues(&val1, &val2)
    fmt.Println("After calling function", val1, val2)
}
```

Листинг 8-15 Определение функции с указателями в файле main.go в папке functions

Функция `swapValues` по-прежнему меняет местами два значения, но делает это с помощью указателя, что означает, что изменения вносятся в области памяти, которые также используются функцией `main`, что можно увидеть в выводе кода:

```
Before calling function 10 20
Before swap: 10 20
After swap: 20 10
After calling function 20 10
```

Существуют лучшие способы выполнения таких задач, как замена значений, включая использование нескольких результатов функций, как описано в следующем разделе, но этот пример демонстрирует, что функции могут работать со значениями напрямую или косвенно через указатели.

Определение и использование результатов функции

Функции определяют результаты, которые позволяют функциям предоставлять своим вызывающим объектам выходные данные операций, как показано в листинге 8-16.

```
package main

import "fmt"

func calcTax(price float64) float64 {
    return price + (price * 0.2)
}
```

```
func main() {

    products := map[string]float64 {
        "Kayak" : 275,
        "Lifejacket": 48.95,
    }

    for product, price := range products {
        priceWithTax := calcTax(price)
                fmt.Println("Product:  ", product,  "Price:",
priceWithTax)
    }
}
```

Листинг 8-16 Создание результата функции в файле main.go в папке functions

Функция объявляет свой результат, используя тип данных, следующий за параметром, как показано на рисунке 8-7.

Тип результата

```
func calcTax(price float64) float64 {
    return price + (price * 0.2)
}
```

Рисунок 8-7 Определение результата функции

Функция `calcTax` выдает результат `float64`, который создается оператором `return`, как показано на рисунке 8-8.

```
func calcTax(price float64) float64 {
    return price + (price * 0.2)
}
```

Ключевое слово Значение результата

Рисунок 8-8 Возврат результата функцией

При вызове функции результат может быть присвоен переменной, как показано на рисинке 8-9.

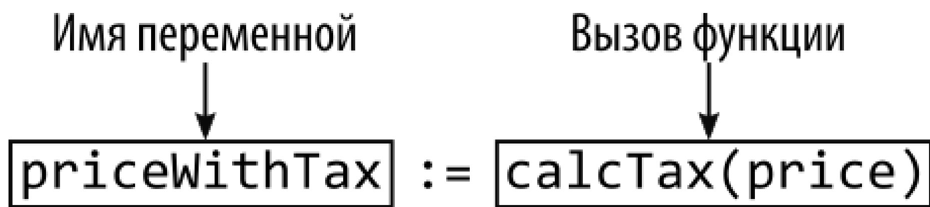

Рисунок 8-9 Использование результата функции

Результаты функции можно использовать непосредственно в выражениях. В листинге 8-17 переменная опущена, а функция `calcTax` вызывается напрямую для получения аргумента для функции `fmt.PrintLn`.

```
package main

import "fmt"

func calcTax(price float64) float64 {
    return price + (price * 0.2)
}

func main() {

    products := map[string]float64 {
        "Kayak" : 275,
        "Lifejacket": 48.95,
    }

    for product, price := range products {
            fmt.Println("Product:  ",  product,  "Price:",
calcTax(price))
    }
}
```

Листинг 8-17 Использование результата функции непосредственно в файле main.go в папке functions

Go использует результат, полученный функцией `calcTax`, без необходимости определять промежуточную переменную. Код в листингах 8-16 и 8-17 выдает следующий результат:

```
Product:  Kayak Price: 330
Product:  Lifejacket Price: 58.74
```

Возврат функцией нескольких результатов

Необычной особенностью функций Go является возможность получения более одного результата, как показано в листинге8-18.

```
package main

import "fmt"

func swapValues(first, second int) (int, int) {
    return second, first
}

func main() {

    val1, val2 := 10, 20
    fmt.Println("Before calling function", val1, val2)
    val1, val2 = swapValues(val1, val2)
    fmt.Println("After calling function", val1, val2)
}
```

Листинг 8-18 Создание нескольких результатов в файле main.go в папке functions

Типы результатов, выдаваемых функцией, сгруппированы с помощью круглых скобок, как показано на рисунке 8-10.

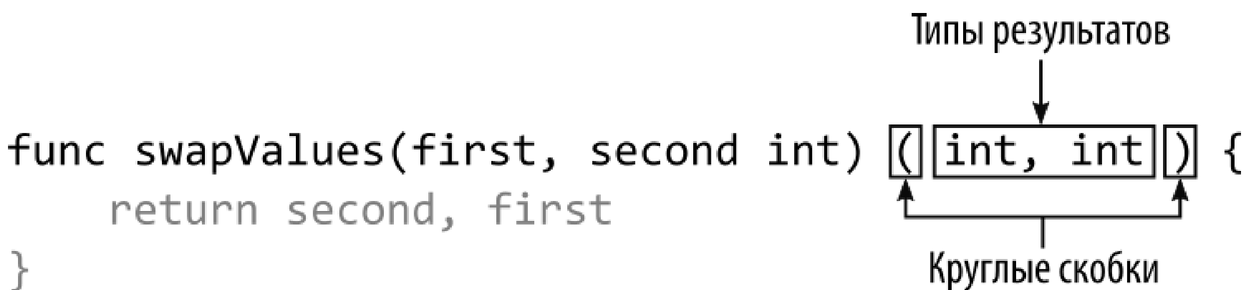

Рисунок 8-10 Определение нескольких результатов

Когда функция определяет несколько результатов, значения для каждого результата предоставляются с ключевым словом `return`, разделенным запятыми, как показано на рисунке 8-11.

```
func swapValues(first, second int) ( int, int ) {
    return second , first
}
```

Ключевое слово Результат Запятая Результат

Рисунок 8-11 Возврат нескольких результатов

Функция `swapValues` использует ключевое слово `return` для получения двух результатов типа `int`, которые она получает через свои параметры. Эти результаты могут быть присвоены переменным в операторе, вызывающем функцию, также через запятую, как показано на рисунке 8-12.

```
val1 , val2 = swapValues(val1, val2)
```

Переменная Запятая Переменная

Рисунок 8-12 Получение нескольких результатов

Код в листинге 8-18 выдает следующий результат при компиляции и выполнении:

```
Before calling function 10 20
After calling function 20 10
```

Использование нескольких результатов вместо нескольких значений

Поначалу результаты нескольких функций могут показаться странными, но их можно использовать, чтобы избежать источника ошибок, характерного для других языков, который заключается в придании разных значений одному результату на основе возвращаемого значения. В листинге 8-19 показана проблема, вызванная приданием дополнительного значения одному результату..

```
package main

import "fmt"
```

```
func calcTax(price float64) float64 {
    if (price > 100) {
        return price * 0.2
    }
    return -1
}

func main() {

    products := map[string]float64 {
        "Kayak" : 275,
        "Lifejacket": 48.95,
    }

    for product, price := range products {
        tax := calcTax(price)
        if (tax != -1) {
            fmt.Println("Product: ", product, "Tax:", tax)
        } else {
            fmt.Println("Product: ", product, "No tax due")

        }
    }
}
```

Листинг 8-19 Использование одного результата в файле main.go в папке functions

Функция `calcTax` использует результат `float64` для передачи двух результатов. Для значений больше 100 в результате будет указана сумма налога к уплате. Для значений менее 100 результат будет означать, что налог не взимается. Компиляция и выполнение кода из листинга 8-19 приводит к следующему результату:

```
Product:  Kayak Tax: 55
Product:  Lifejacket No tax due
```

Придание нескольких значений одному результату может стать проблемой по мере развития проектов. Налоговый орган может начать возвращать налог на определенные покупки, что делает значение `-1` двусмысленным, поскольку может указывать на то, что налог не уплачивается или что должен быть выдан возврат в размере 1 доллара.

Есть много способов разрешить этот тип неоднозначности, но использование результатов нескольких функций является элегантным решением, хотя и может занять некоторое время, чтобы привыкнуть к нему. В листинге 8-20 я изменил функцию calcTax, чтобы она выдавала несколько результатов.

```
package main

import "fmt"

func calcTax(price float64) (float64, bool) {
    if (price > 100) {
        return price * 0.2, true
    }
    return 0, false
}

func main() {

    products := map[string]float64 {
        "Kayak" : 275,
        "Lifejacket": 48.95,
    }

    for product, price := range products {
        taxAmount, taxDue := calcTax(price)
        if (taxDue) {
                  fmt.Println("Product:  ",  product,  "Tax:",
taxAmount)
        } else {
            fmt.Println("Product: ", product, "No tax due")

        }
    }
}
```

Листинг 8-20 Использование нескольких результатов в файле main.go в папке functions.

Дополнительный результат, возвращаемый методом calcTax, представляет собой логическое значение, указывающее, подлежит ли уплате налог, отделяя эту информацию от другого результата. В листинге 8-20 два результата получены в отдельном операторе, но

множественные результаты хорошо подходят для поддержки оператора if оператора инициализации, как показано в листинге 8-21. (Подробнее об этой функции см. в главе 12.)

```
package main

import "fmt"

func calcTax(price float64) (float64, bool) {
    if (price > 100) {
        return price * 0.2, true
    }
    return 0, false
}

func main() {

    products := map[string]float64 {
        "Kayak" : 275,
        "Lifejacket": 48.95,
    }

    for product, price := range products {
        if taxAmount, taxDue := calcTax(price); taxDue {
                  fmt.Println("Product:  ", product, "Tax:",
taxAmount)
        } else {
            fmt.Println("Product: ", product, "No tax due")

        }
    }
}
```

Листинг 8-21 Использование оператора инициализации в файле main.go в папке functions

Два результата получаются путем вызова функции calcTax в операторе инициализации, а результат bool затем используется в качестве выражения оператора. Код в листингах 8-20 и 8-21 выдает следующий результат:

```
Product:  Kayak Tax: 55
Product:  Lifejacket No tax due
```

Использование именованных результатов

Результатам функции можно давать имена, которым можно присваивать значения во время выполнения функции. Когда выполнение достигает ключевого слова `return`, возвращаются текущие значения, присвоенные результатам, как показано в листинге 8-22.

```
package main

import "fmt"

func calcTax(price float64) (float64, bool) {
    if (price > 100) {
        return price * 0.2, true
    }
    return 0, false
}

func calcTotalPrice(products map[string]float64,
        minSpend float64) (total, tax float64)  {
    total = minSpend
    for _, price := range products {
        if taxAmount, due := calcTax(price); due {
            total += taxAmount;
            tax += taxAmount
        } else {
            total += price
        }
    }
    return
}

func main() {

    products := map[string]float64 {
        "Kayak" : 275,
        "Lifejacket": 48.95,
    }

    total1, tax1 := calcTotalPrice(products, 10)
    fmt.Println("Total 1:", total1, "Tax 1:", tax1)
    total2, tax2 := calcTotalPrice(nil, 10)
    fmt.Println("Total 2:", total2, "Tax 2:", tax2)
```

```
}
```

Листинг 8-22 Использование именованных результатов в файле main.go в папке functions

Именованные результаты определяются как комбинация имени и типа результата, как показано на рисунке 8-13.

```
func calcTotalTax(products map[string]float64,
        minSpend float64) (total, tax float64) {
```


Рисунок 8-13 Именованные результаты

Функция `calcTotalPrice` определяет результаты с именами `total` и `tax`. Оба являются значениями `float64`, что означает, что я могу опустить тип данных в первом имени. Внутри функции результаты можно использовать как обычные переменные:

```
...
total = minSpend
for _, price := range products {
    if taxAmount, due := calcTax(price); due {
        total += taxAmount;
        tax += taxAmount
    } else {
        total += price
    }
}
...
```

Ключевое слово `return` используется само по себе, позволяя возвращать текущие значения, присвоенные именованным результатам. Код в листинге 8-22 выдает следующий результат:

```
Total 1: 113.95 Tax 1: 55
Total 2: 10 Tax 2: 0
```

Использование пустого идентификатора для сброса результатов

Go требует использования всех объявленных переменных, что может быть неудобно, когда функция возвращает значения, которые вам не нужны. Чтобы избежать ошибок компилятора, можно использовать пустой идентификатор (символ _) для обозначения результатов, которые не будут использоваться, как показано в листинге 8-23.

```
package main

import "fmt"

func calcTotalPrice(products map[string]float64) (count int,
total float64)  {
    count = len(products)
    for _, price := range products {
        total += price
    }
    return
}

func main() {

    products := map[string]float64 {
        "Kayak" : 275,
        "Lifejacket": 48.95,
    }

    _, total  := calcTotalPrice(products)
    fmt.Println("Total:", total)
}
```

Листинг 8-23 Сброс результатов функции в файле main.go в папке functions

Функция calcTotalPrice возвращает два результата, из которых используется только один. Пустой идентификатор используется для нежелательного значения, что позволяет избежать ошибки компилятора. Код в листинге 8-23 выдает следующий результат:

```
Total: 323.95
```

Использование ключевого слова defer

Ключевое слово defer используется для планирования вызова функции, который будет выполнен непосредственно перед возвратом из текущей функции, как показано в листинге 8-24.

```
package main

import "fmt"

func calcTotalPrice(products map[string]float64) (count int,
total float64)  {
    fmt.Println("Function started")
    defer fmt.Println("First defer call")
    count = len(products)
    for _, price := range products {
        total += price
    }
    defer fmt.Println("Second defer call")
    fmt.Println("Function about to return")
    return
}

func main() {

    products := map[string]float64 {
        "Kayak" : 275,
        "Lifejacket": 48.95,
    }

    _, total  := calcTotalPrice(products)
    fmt.Println("Total:", total)
}
```

Листинг 8-24 Использование ключевого слова defer в файле main.go в папке functions

Ключевое слово `defer` используется перед вызовом функции, как показано на рисунке 8-14.

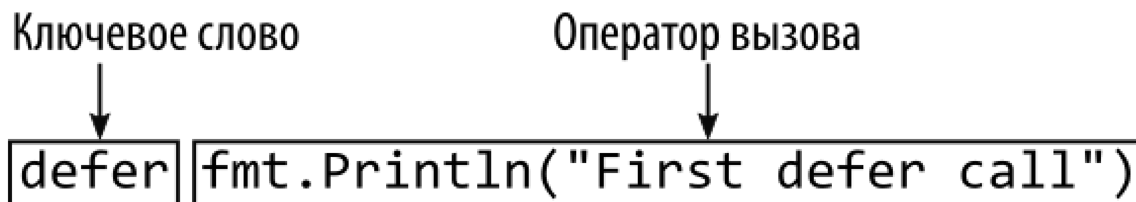

Рисунок 8-14 Ключевое слово defer

Ключевое слово `defer` в основном используется для вызова функций, освобождающих ресурсы, таких как закрытие открытых файлов (описано в главе 22) или соединений HTTP (главы 24 и 25). Без ключевого слова `defer` оператор, освобождающий ресурс, должен появиться в конце функции, а это может быть много операторов после создания и использования ресурса. Ключевое слово `defer` позволяет сгруппировать операторы, которые создают, используют и освобождают ресурс вместе.

Ключевое слово `defer` можно использовать с любым вызовом функции, как показано в листинге 8-24, и одна функция может использовать ключевое слово `defer` несколько раз. Непосредственно перед возвратом функции Go выполнит вызовы, запланированные с помощью ключевого слова `defer`, в том порядке, в котором они были определены. Код в листинге 8-24 планирует вызовы функции `fmt.Println` и при компиляции и выполнении выдает следующий вывод:

```
Function started
Function about to return
Second defer call
First defer call
Total: 323.95
```

Резюме

В этой главе я описал функции Go, объяснив, как они определяются и используются. Я продемонстрировал различные способы определения параметров и то, как функции Go могут давать результаты. В следующей главе я опишу, как функции могут использоваться в качестве типов.

9. Использование функциональных типов

В этой главе я описываю, как Go работает с функциональными типами, что является полезной — хотя иногда и запутанной — возможностью, позволяющей описывать функции согласованно и так же, как и другие значения. Таблица 9-1 помещает функциональные типы в контекст.

Таблица 9-1 Помещение функциональных типов в контекст

Вопрос	Ответ
Кто они такие?	Функции в Go имеют тип данных, описывающий комбинацию параметров, которые функция использует, и результатов, которые она производит. Этот тип может быть указан явно или выведен из функции, определенной с использованием литерального синтаксиса.
Почему они полезны?	Обращение к функциям как к типам данных означает, что они могут быть присвоены переменным и что одна функция может быть заменена другой при условии, что она имеет ту же комбинацию параметров и результатов.
Как они используются?	Типы функций определяются с помощью ключевого слова `func`, за которым следует подпись, описывающая параметры и результаты. Тело функции не указано.
Есть ли подводные камни или ограничения?	Расширенное использование типов функций может стать трудным для понимания и отладки, особенно если определены вложенные литеральные функции.
Есть ли альтернативы?	Вам не нужно использовать типы функций или определять функции, используя литеральный синтаксис, но это может уменьшить дублирование кода и повысить гибкость кода, который вы пишете.

Таблица 9-2 суммирует главу.

Таблица 9-2 Краткое содержание главы

Проблема	Решение	Листинг
Описать функции с определенной комбинацией параметров и результатов	Используйте функциональный тип	4–7
Упростить повторяющееся выражение функционального типа	Использовать псевдоним функционального типа	8

Проблема	Решение	Листинг
Определить функцию, относящуюся к области кода	Используйте литеральный синтаксис функции	9–12
Доступ к значениям, определенным вне функции	Используйте замыкание функции	13–18

Подготовка к этой главе

Чтобы подготовиться к этой главе, откройте новую командную строку, перейдите в удобное место и создайте каталог с именем `functionTypes`. Перейдите в папку `functionTypes` и выполните команду, показанную в листинге 9-1, чтобы инициализировать проект.

```
go mod init functionTypes
```

Листинг 9-1 Инициализация проекта

Добавьте файл с именем `main.go` в папку `functionTypes` с содержимым, показанным в листинге 9-2.

> **Подсказка**
> Вы можете загрузить пример проекта для этой главы — и для всех остальных глав этой книги — с `https://github.com/apress/pro-go`. См. Главу 2 о том, как получить помощь, если у вас возникнут проблемы с запуском примеров.

```
package main

import "fmt"

func main() {

    fmt.Println("Hello, Function Types")
}
```

Листинг 9-2 Содержимое файла main.go в папке functionTypes

Используйте командную строку для запуска команды, показанной в листинге 9-3, в папке `functionTypes`.

```
go run .
```

Листинг 9-3 Запуск примера проекта

Код в файле `main.go` будет скомпилирован и выполнен, что приведет к следующему результату:

```
Hello, Function Types
```

Понимание типов функций

Функции в Go имеют тип данных, что означает, что они могут быть назначены переменным и использоваться в качестве параметров функции, аргументов и результатов. В листинге 9-4 показано простое использование типа данных функции.

```
package main

import "fmt"

func calcWithTax(price float64) float64 {
    return price + (price * 0.2)
}

func calcWithoutTax(price float64) float64 {
    return price
}

func main() {

    products := map[string]float64 {
        "Kayak" : 275,
        "Lifejacket": 48.95,
    }

    for product, price := range products {
        var calcFunc func(float64) float64
        if (price > 100) {
            calcFunc = calcWithTax
        } else {
            calcFunc = calcWithoutTax
        }
```

```
        totalPrice := calcFunc(price)
                fmt.Println("Product:",  product,  "Price:",
totalPrice)
    }
}
```

Листинг 9-4 Использование типа данных функции в файле main.go в папке functionTypes

Этот пример содержит две функции, каждая из которых определяет параметр `float64` и возвращает результат `float64`. Цикл `for` в основной функции выбирает одну из этих функций и использует ее для расчета общей цены продукта. Первый оператор в цикле определяет переменную, как показано на рисунок 9-1.

Рисунок 9-1 Определение переменной функционального типа

Типы функций указываются с помощью ключевого слова `func`, за которым в скобках следуют типы параметров, а затем типы результатов. Это известно как *сигнатура функции*. Если результатов несколько, то типы результатов также заключаются в круглые скобки. Тип функции в листинге 9-4 описывает функцию, которая принимает аргумент `float64` и возвращает результат `float64`.

Переменной `calcFunc`, определенной в листинге 9-4, может быть присвоено любое значение, соответствующее ее типу, что означает любую функцию с правильным количеством и типом аргументов и результатов. Чтобы назначить определенную функцию переменной, используется имя функции, как показано на рисунке 9-2.

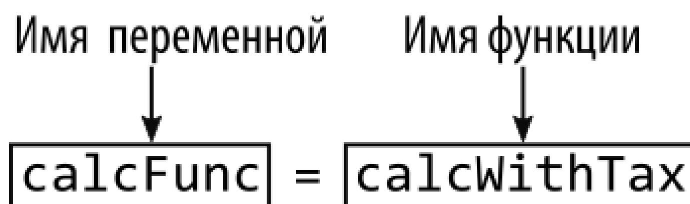

Рисунок 9-2 Назначение функции переменной

Как только функция была назначена переменной, ее можно вызвать, как если бы имя переменной было именем функции. В примере это означает, что функция, назначенная переменной `calcFunc`, может быть вызвана, как показано на рисунке 9-3.

Имя переменной

```
totalPrice := calcFunc(price)
```

Рисунок 9-3 Вызов функции через переменную

В результате будет вызвана любая функция, назначенная функции `totalPrice`. Если значение `price` больше 100, то переменной `totalPrice` присваивается функция `calcWithTax`, и именно эта функция будет выполняться. Если price меньше или равна 100, то переменной `totalPrice` присваивается функция `calcWithoutTax`, и вместо нее будет выполняться эта функция. Код в листинге 9-4 выдает следующий результат при компиляции и выполнении (хотя вы можете увидеть результаты в другом порядке, как описано в главе 7):

```
Product: Kayak Price: 330
Product: Lifejacket Price: 48.95
```

Понимание сравнения функций и нулевого типа

Операторы сравнения Go нельзя использовать для сравнения функций, но их можно использовать для определения того, была ли функция присвоена переменной, как показано в листинге 9-5.

```
package main

import "fmt"

func calcWithTax(price float64) float64 {
    return price + (price * 0.2)
}

func calcWithoutTax(price float64) float64 {
    return price
}
```

```
func main() {

    products := map[string]float64 {
        "Kayak" : 275,
        "Lifejacket": 48.95,
    }

    for product, price := range products {
        var calcFunc func(float64) float64
        fmt.Println("Function assigned:", calcFunc == nil)
        if (price > 100) {
            calcFunc = calcWithTax
        } else {
            calcFunc = calcWithoutTax
        }
        fmt.Println("Function assigned:", calcFunc == nil)
        totalPrice := calcFunc(price)
                 fmt.Println("Product:",  product,  "Price:",
totalPrice)
    }
}
```

Листинг 9-5 Проверка назначения в файле main.go в папке functionTypes

Нулевое значение для типов функций равно `nil`, а новые операторы в листинге 9-5 используют оператор равенства, чтобы определить, присвоена ли функция переменной `calcFunc`. Код в листинге 9-5 выдает следующий результат:

```
Function assigned: true
Function assigned: false
Product: Kayak Price: 330
Function assigned: true
Function assigned: false
Product: Lifejacket Price: 48.95
```

Использование функций в качестве аргументов

Типы функций можно использовать так же, как и любые другие типы, в том числе в качестве аргументов для других функций, как показано в листинге 9-6.

Примечание

> Некоторым описаниям в следующих разделах может быть трудно следовать, потому что слово *функция* требуется очень часто. Я предлагаю обратить пристальное внимание на примеры кода, которые помогут разобраться в тексте.

```
package main

import "fmt"

func calcWithTax(price float64) float64 {
    return price + (price * 0.2)
}

func calcWithoutTax(price float64) float64 {
    return price
}

func printPrice(product string, price float64, calculator
func(float64) float64 ) {
               fmt.Println("Product:", product, "Price:",
calculator(price))
}

func main() {

    products := map[string]float64 {
        "Kayak" : 275,
        "Lifejacket": 48.95,
    }

    for product, price := range products {
        if (price > 100) {
            printPrice(product, price, calcWithTax)
        } else {
            printPrice(product, price, calcWithoutTax)
        }
    }
}
```

Листинг 9-6 Использование функций в качестве аргументов в файле main.go в папке functionTypes

Функция `printPrice` определяет три параметра, первые два из которых получают значения типа `string` и `float64`. Третий параметр, названный `calculator`, получает функцию, которая получает значение `float64` и выдает результат `float64`, как показано на рисунке 9-4.

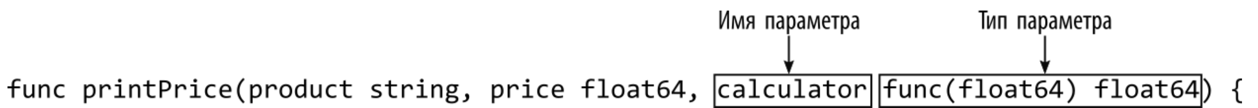

Рисунок 9-4 Параметр функции

В функции `printPrice` параметр `calculator` используется так же, как и любая другая функция:

```
...
fmt.Println("Product:", product, "Price:", calculator(price))
...
```

Важно то, что функция `printPrice` не знает — и не заботится о том, получает ли она функцию `calcWithTax` или `calcWithoutTax` через параметр `calculator`. Все, что знает функция `printPrice`, это то, что она сможет вызвать функцию `calculator` с аргументом `float64` и получить результат `float64`, потому что это функциональный тип параметра.

Выбор используемой функции осуществляется оператором `if` в `main` функции, а имя используется для передачи одной функции в качестве аргумента другой функции, например:

```
...
printPrice(product, price, calcWithTax)
...
```

Код в листинге 9-6 выдает следующий результат при компиляции и выполнении:

```
Product: Kayak Price: 330
Product: Lifejacket Price: 48.95
```

Использование функций в качестве результатов

Функции также могут быть результатами, что означает, что значение, возвращаемое функцией, является другой функцией, как показано в

листинге 9-7.

```
package main

import "fmt"

func calcWithTax(price float64) float64 {
    return price + (price * 0.2)
}

func calcWithoutTax(price float64) float64 {
    return price
}

func printPrice(product string, price float64, calculator
func(float64) float64 ) {
               fmt.Println("Product:", product, "Price:",
calculator(price))
}

func selectCalculator(price float64) func(float64) float64 {
    if (price > 100) {
        return calcWithTax
    }
    return calcWithoutTax
}

func main() {

    products := map[string]float64 {
        "Kayak" : 275,
        "Lifejacket": 48.95,
    }

    for product, price := range products {
        printPrice(product, price, selectCalculator(price))
    }
}
```

Листинг 9-7 Создание результата функции в файле main.go в папке functionTypes

Функция `selectCalculator` получает значение `float64` и возвращает функцию, как показано на рисунке 9-5.

Тип результата

```
func selectCalculator(price float64) func(float64) float64 {
```

Рисунок 9-5 Результат типа функции

Результатом `selectCalculator` является функция, которая принимает значение `float64` и выдает результат `float64`. Вызывающие `selectCalculator` не знают, получат ли они функцию `calcWithTax` или `calcWithoutTax`, только то, что они получат функцию с указанной сигнатурой. Код в листинге 9-7 выдает следующий результат при компиляции и выполнении:

```
Product: Kayak Price: 330
Product: Lifejacket Price: 48.95
```

Создание псевдонимов функциональных типов

Как показали предыдущие примеры, использование типов функций может быть многословным и повторяющимся, что приводит к созданию кода, который трудно читать и поддерживать. Go поддерживает псевдонимы типов, которые можно использовать для присвоения имени сигнатуре функции, чтобы типы параметров и результатов не определялись каждый раз при использовании типа функции, как показано в листинге 9-8.

```
package main

import "fmt"

type calcFunc func(float64) float64

func calcWithTax(price float64) float64 {
    return price + (price * 0.2)
}

func calcWithoutTax(price float64) float64 {
    return price
}
```

```
func printPrice(product string, price float64, calculator calcFunc) {
                fmt.Println("Product:", product, "Price:", calculator(price))
}

func selectCalculator(price float64) calcFunc {
    if (price > 100) {
        return calcWithTax
    }
    return calcWithoutTax
}

func main() {

    products := map[string]float64 {
        "Kayak" : 275,
        "Lifejacket": 48.95,
    }

    for product, price := range products {
        printPrice(product, price, selectCalculator(price))
    }
}
```

Листинг 9-8 Использование псевдонима типа в файле main.go в папке functionTypes

Псевдоним создается с помощью ключевого слова `type`, за которым следует имя псевдонима, а затем тип, как показано на рисунке 9-6.

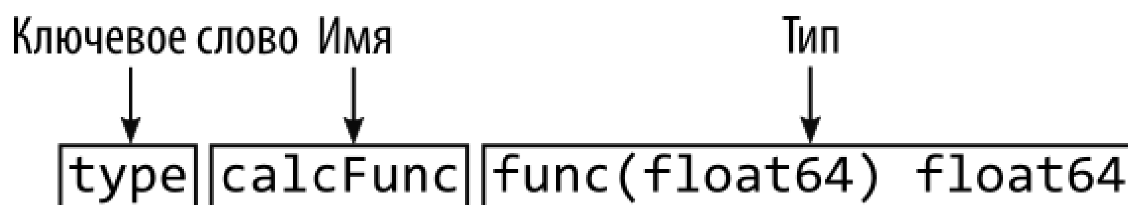

Рисунок 9-6 Псевдоним типа

Примечание
Ключевое слово `type` также используется для создания пользовательских типов, как описано в главе 10.

Псевдоним в листинге 9-8 присваивает имя `calcFunc` типу функции, которая принимает аргумент `float64` и выдает результат `float64`. Псевдоним можно использовать вместо типа функции, например:

```
...
func selectCalculator(price float64) calcFunc {
...
```

Вам не обязательно использовать псевдонимы для типов функций, но они могут упростить код и облегчить идентификацию использования конкретной сигнатуры функции. Код в листинге 9-8 выдает следующий результат:

```
Product: Kayak Price: 330
Product: Lifejacket Price: 48.95
```

Использование литерального синтаксиса функции

Синтаксис литерала функции позволяет определять функции так, чтобы они были специфичны для области кода, как показано в листинге 9-9.

```
package main

import "fmt"

type calcFunc func(float64) float64

// func calcWithTax(price float64) float64 {
//     return price + (price * 0.2)
// }

// func calcWithoutTax(price float64) float64 {
//     return price
// }

func printPrice(product string, price float64, calculator
calcFunc) {
        fmt.Println("Product:", product, "Price:",
calculator(price))
}
```

```
func selectCalculator(price float64) calcFunc {
    if (price > 100) {
        var withTax calcFunc = func (price float64) float64 {
            return price + (price * 0.2)
        }
        return withTax
    }
    withoutTax := func (price float64) float64 {
        return price
    }
    return withoutTax
}

func main() {

    products := map[string]float64 {
        "Kayak" : 275,
        "Lifejacket": 48.95,
    }

    for product, price := range products {
        printPrice(product, price, selectCalculator(price))
    }
}
```

Листинг 9-9 Использование литерального синтаксиса в файле main.go в папке functionTypes

Литеральный синтаксис не включает имя, поэтому за ключевым словом `func` следуют параметры, тип результата и блок кода, как показано на рисунке 9-7. Поскольку имя опущено, функции, определенные таким образом, называются *анонимными функциями*.

Рисунок 9-7 Синтаксис литерала функции

Примечание

> Go не поддерживает стрелочные функции, где функции более лаконично выражаются с помощью оператора `=>` без ключевого слова `func` и блока кода, заключенного в фигурные скобки. В Go функции всегда должны определяться ключевым словом и телом.

Литеральный синтаксис создает функцию, которую можно использовать как любое другое значение, включая назначение функции переменной, что я и сделал в листинге 9-9. Тип литерала функции определяется сигнатурой функции, что означает, что количество и типы параметров функции должны соответствовать типу переменной, например:

```
...
var withTax calcFunc = func (price float64) float64 {
    return price + (price * 0.2)
}
...
```

Эта литеральная функция имеет сигнатуру, соответствующую псевдониму типа `calcFunc`, с одним параметром `float64` и одним результатом `float64`. Литеральные функции также можно использовать с коротким синтаксисом объявления переменных:

```
...
withoutTax := func (price float64) float64 {
    return price
}
...
```

Компилятор Go определит тип переменной, используя сигнатуру функции, что означает, что тип переменной `WithoutTax` является `func(float64) float64`. Код в листинге 9-9 выдает следующий результат при компиляции и выполнении:

```
Product: Kayak Price: 330
Product: Lifejacket Price: 48.95
```

Понимание области действия функциональной переменной

Функции обрабатываются так же, как и любые другие значения, но доступ к функции, добавляющей налог, возможен только через переменную withTax, которая, в свою очередь, доступна только в кодовом блоке оператора if, как показано в листинге 9-10.

```
...
func selectCalculator(price float64) calcFunc {
    if (price > 100) {
         var withTax calcFunc = func (price float64) float64 {
             return price + (price * 0.2)
        }
        return withTax
    } else if (price < 10) {
        return withTax
    }
     withoutTax := func (price float64) float64 {
        return price
    }
    return withoutTax
}
...
```

Листинг 9-10 Использование функции вне ее области действия в файле main.go в папке functionTypes

Оператор в предложении else/if пытается получить доступ к функции, назначенной переменной withTax. Доступ к переменной недоступен, поскольку она находится в другом блоке кода, поэтому компилятор выдает следующую ошибку:

```
# command-line-arguments
.\main.go:18:16: undefined: withTax
```

Непосредственное использование значений функций

Я присвоил функции переменным в предыдущих примерах, потому что хотел продемонстрировать, что Go обрабатывает литеральные функции так же, как и любые другие значения. Но функции не обязательно присваивать переменным, и их можно использовать так же, как любое другое литеральное значение, как показано в листинге 9-11.

```
package main
```

```
import "fmt"

type calcFunc func(float64) float64

func printPrice(product string, price float64, calculator
calcFunc) {
               fmt.Println("Product:", product, "Price:",
calculator(price))
}

func selectCalculator(price float64) calcFunc {
    if (price > 100) {
        return func (price float64) float64 {
            return price + (price * 0.2)
        }
    }
     return func (price float64) float64 {
        return price
    }
}

func main() {

    products := map[string]float64 {
        "Kayak" : 275,
        "Lifejacket": 48.95,
    }

    for product, price := range products {
        printPrice(product, price, selectCalculator(price))
    }
}
```

Листинг 9-11 Использование функций непосредственно в файле main.go в папке functionTypes

Ключевое слово `return` применяется непосредственно к функции, не присваивая функцию переменной. Код в листинге 9-11 выдает следующий результат:

```
Product: Kayak Price: 330
Product: Lifejacket Price: 48.95
```

Литеральные функции также можно использовать в качестве аргументов для других функций, как показано в листинге 9-12.

```
package main

import "fmt"

type calcFunc func(float64) float64

func printPrice(product string, price float64, calculator
calcFunc) {
        fmt.Println("Product:", product, "Price:",
calculator(price))
}

func main() {

    products := map[string]float64 {
        "Kayak" : 275,
        "Lifejacket": 48.95,
    }

    for product, price := range products {
            printPrice(product, price, func (price float64)
float64 {
            return price + (price * 0.2)
        })
    }
}
```

Листинг 9-12 Использование литерального аргумента функции в файле main.go в папке functionTypes

Последний аргумент функции `printPrice` выражается с использованием литерального синтаксиса и без присвоения функции переменной. Код в листинге 9-12 выдает следующий результат:

```
Product: Kayak Price: 330
Product: Lifejacket Price: 58.74
```

Понимание замыкания функции

Функции, определенные с использованием литерального синтаксиса, могут ссылаться на переменные из окружающего кода — функция, известная как замыкание. Эту функцию может быть трудно понять, поэтому я начну с примера, который не зависит от замыкания, показанного в листинге 9-13, а затем объясню, как его можно улучшить.

```
package main

import "fmt"

type calcFunc func(float64) float64

func printPrice(product string, price float64, calculator calcFunc) {
               fmt.Println("Product:", product, "Price:", calculator(price))
}

func main() {

    watersportsProducts := map[string]float64 {
        "Kayak" : 275,
        "Lifejacket": 48.95,
    }

    soccerProducts := map[string] float64 {
        "Soccer Ball": 19.50,
        "Stadium": 79500,
    }

    calc := func(price float64) float64 {
        if (price > 100) {
            return price + (price * 0.2)
        }
        return price;
    }
    for product, price := range watersportsProducts {
        printPrice(product, price, calc)
    }

    calc = func(price float64) float64 {
        if (price > 50) {
```

```
            return price + (price * 0.1)
        }
        return price
    }
    for product, price := range soccerProducts {
        printPrice(product, price, calc)
    }
}
```

Листинг 9-13 Использование нескольких функций в файле main.go в папке functionTypes

Две карты содержат названия и цены товаров в категориях водного спорта и футбола. Карты перечисляются циклами `for`, которые вызывают функцию `printPrice` для каждого элемента карты. Одним из аргументов, требуемых функцией `printPrice`, является функция `calcFunc`, которая вычисляет общую цену продукта, включая налоги. Для каждой категории продуктов требуется свой порог необлагаемого налогом дохода и налоговая ставка, как описано в Таблице 9-3.

Таблица 9-3 Пороги категорий продуктов и налоговые ставки

Категория	Порог	Ставка
Водный спорт	100	20%
Футбол	50	10%

Примечание

Пожалуйста, не пишите мне с жалобами на то, что мои выдуманные налоговые ставки свидетельствуют о неприязни к футболу. Я одинаково не люблю все виды спорта, кроме бега на длинные дистанции, которым я занимаюсь в основном потому, что каждая миля уносит меня все дальше от людей, говорящих о спорте.

Я использую литеральный синтаксис для создания функций, применяющих пороги для каждой категории. Это работает, но существует высокая степень дублирования, и если есть изменения в способе расчета цен, я должен помнить об обновлении функции калькулятора для каждой категории.

Что мне нужно, так это возможность консолидировать общий код, необходимый для расчета цены, и разрешить настройку этого общего

кода с изменениями для каждой категории. Это легко сделать с помощью функции замыкания, как показано в листинге 9-14.

```
package main

import "fmt"

type calcFunc func(float64) float64

func printPrice(product string, price float64, calculator calcFunc) {
        fmt.Println("Product:", product, "Price:", calculator(price))
}

func priceCalcFactory(threshold, rate float64) calcFunc {
    return func(price float64) float64 {
        if (price > threshold) {
            return price + (price * rate)
        }
        return price
    }
}

func main() {

    watersportsProducts := map[string]float64 {
        "Kayak" : 275,
        "Lifejacket": 48.95,
    }

    soccerProducts := map[string] float64 {
        "Soccer Ball": 19.50,
        "Stadium": 79500,
    }

    waterCalc := priceCalcFactory(100, 0.2);
    soccerCalc := priceCalcFactory(50, 0.1)

    for product, price := range watersportsProducts {
        printPrice(product, price, waterCalc)
    }
```

```
    for product, price := range soccerProducts {
        printPrice(product, price, soccerCalc)
    }
}
```

Листинг 9-14 Использование замыкания функции в файле main.go в папке functionTypes

Ключевым дополнением является функция `priceCalcFactory`, которую я буду называть в этом разделе *фабричной функцией*, чтобы отличать ее от других частей кода. Работа фабричной функции заключается в создании функций калькулятора для определенной комбинации порога и налоговой ставки. Эта задача описывается сигнатурой функции, как показано на рисунке 9-8.


```
func priceCalcFactory(threshold, rate float64) calcFunc {
    return func(price float64) float64 {
        if (price > threshold) {
            return price + (price * rate)
        }
        return price
    }
}
```

Рисунок 9-8 Сигнатура заводской функции

Входными данными фабричной функции являются пороговое значение и ставка для категории, а выходными данными является функция, которая вычисляет цены, используемые для этой категории. Код фабричной функции использует литеральный синтаксис для определения функции калькулятора, которая содержит общий код для выполнения вычислений, как показано на рисунке 9-9.

```
func priceCalcFactory(threshold, rate float64) calcFunc {
    return func(price float64) float64 {
        if (price > threshold) {
            return price + (price * rate)
        }
        return price
    }
}
```


Рисунок 9-9 Общий код

Функция замыкания является связующим звеном между фабричной функцией и функцией калькулятора. Функция калькулятора использует две переменные для получения результата, например:

```
...
return func(price float64) float64 {
    if (price > threshold) {
        return price + (price * rate)
    }
    return price
}
...
```

Значения `threshold` и `rate` берутся из заводских параметров функции, например:

```
...
func priceCalcFactory(threshold, rate float64) calcFunc {
...
```

Функция замыкания позволяет функции получать доступ к переменным и параметрам в окружающем коде. В этом случае функция калькулятора опирается на параметры заводской функции. Когда вызывается функция калькулятора, значения параметров используются для получения результата, как показано на рисунке 9-10.

```
func priceCalcFactory(threshold, rate float64) calcFunc {
    return func(price float64) float64 {
        if (price > threshold) {
            return price + (price * rate)
        }
        return price
    }
}
```

Рисунок 9-10 Замыкание функции

Говорят, что функция *замыкается* на источниках требуемых значений, так что функция калькулятора закрывается на параметрах `threshold` и `rate` фабричной функции.

Результатом является фабричная функция, которая создает функции калькулятора, настроенные для налогового порога и ставки категории продукта. Код, необходимый для расчета цен, был объединен, поэтому изменения будут применяться ко всем категориям. Листинг 9-13 и Листинг 9-14 выдают следующий результат:

```
Product: Kayak Price: 330
Product: Lifejacket Price: 48.95
Product: Soccer Ball Price: 19.5
Product: Stadium Price: 87450
```

Понимание оценки замыкания

Переменные, по которым замыкается функция, оцениваются каждый раз, когда функция вызывается, а это означает, что изменения, сделанные вне функции, могут повлиять на результаты, которые она производит, как показано в листинге 9-15.

```
package main

import "fmt"

type calcFunc func(float64) float64

func printPrice(product string, price float64, calculator
calcFunc) {
```

```
            fmt.Println("Product:",   product,   "Price:",
calculator(price))
}

var prizeGiveaway = false

func priceCalcFactory(threshold, rate float64) calcFunc {
    return func(price float64) float64 {
        if (prizeGiveaway) {
            return 0
        } else if (price > threshold) {
            return price + (price * rate)
        }
        return price
    }
}

func main() {

    watersportsProducts := map[string]float64 {
        "Kayak" : 275,
        "Lifejacket": 48.95,
    }

    soccerProducts := map[string] float64 {
        "Soccer Ball": 19.50,
        "Stadium": 79500,
    }

    prizeGiveaway = false
    waterCalc := priceCalcFactory(100, 0.2);
    prizeGiveaway = true
    soccerCalc := priceCalcFactory(50, 0.1)

    for product, price := range watersportsProducts {
        printPrice(product, price, waterCalc)
    }

    for product, price := range soccerProducts {
        printPrice(product, price, soccerCalc)
    }
}
```

Листинг 9-15 Изменение замкнутого значения в файле main.go в папке functionTypes

Функция калькулятора замыкается на переменной PrizeGiveaway, в результате чего цены падают до нуля. Перед созданием функции для категории водных видов спорта переменной PrizeGiveaway присваивается значение false, а перед созданием функции для категории футбола — значение true.

Но, поскольку замыкания оцениваются при вызове функции, используется текущее значение переменной PrizeGiveaway, а не значение на момент создания функции. Как следствие, цены для обеих категорий сбрасываются до нуля, и код выдает следующий результат:

```
Product: Lifejacket Price: 0
Product: Kayak Price: 0
Product: Soccer Ball Price: 0
Product: Stadium Price: 0
```

Принудительная ранняя оценка

Оценка замыканий при вызове функции может быть полезна, но если вы хотите использовать значение, которое было текущим на момент создания функции, скопируйте это значение, как показано в листинге 9-16.

```
...
func priceCalcFactory(threshold, rate float64) calcFunc {
    fixedPrizeGiveway := prizeGiveaway
    return func(price float64) float64 {
        if (fixedPrizeGiveway) {
            return 0
        } else if (price > threshold) {
            return price + (price * rate)
        }
        return price
    }
}
...
```

Листинг 9-16 Принудительная оценка в файле main.go в папке functionTypes

Функция калькулятора замыкается на переменной fixedPrizeGiveway, значение которой устанавливается при вызове фабричной функции. Это гарантирует, что на функцию калькулятора не повлияет изменение значения PrizeGiveaway. Такого же эффекта

можно добиться, добавив параметр в фабричную функцию, поскольку по умолчанию параметры функции передаются по значению. Листинг 9-17 добавляет параметр к фабричной функции.

```
package main

import "fmt"

type calcFunc func(float64) float64

func printPrice(product string, price float64, calculator calcFunc) {
               fmt.Println("Product:", product, "Price:", calculator(price))
}

var prizeGiveaway = false

func priceCalcFactory(threshold, rate float64, zeroPrices bool) calcFunc {
    return func(price float64) float64 {
        if (zeroPrices) {
            return 0
        } else if (price > threshold) {
            return price + (price * rate)
        }
        return price
    }
}

func main() {

    watersportsProducts := map[string]float64 {
        "Kayak" : 275,
        "Lifejacket": 48.95,
    }

    soccerProducts := map[string] float64 {
        "Soccer Ball": 19.50,
        "Stadium": 79500,
    }

    prizeGiveaway = false
```

```
    waterCalc := priceCalcFactory(100, 0.2, prizeGiveaway);
    prizeGiveaway = true
    soccerCalc := priceCalcFactory(50, 0.1, prizeGiveaway)

    for product, price := range watersportsProducts {
        printPrice(product, price, waterCalc)
    }

    for product, price := range soccerProducts {
        printPrice(product, price, soccerCalc)
    }
}
```

Листинг 9-17 Добавление параметра в файл main.go в папку functionTypes

В листинге 9-16 и листинге 9-17 функции калькулятора не затрагиваются при изменении переменной `PrizeGiveaway` и получении следующего вывода:

```
Product: Kayak Price: 330
Product: Lifejacket Price: 48.95
Product: Stadium Price: 0
Product: Soccer Ball Price: 0
```

Замыкание по указателю для предотвращения ранней оценки

Большинство проблем с замыканием вызвано изменениями, внесенными в переменные после создания функции, которые можно решить с помощью методов, описанных в предыдущем разделе. Иногда вы можете столкнуться с противоположной проблемой, которая заключается в необходимости избегать ранней оценки, чтобы гарантировать, что текущее значение используется функцией. В этих ситуациях использование указателя предотвратит копирование значений, как показано в листинге 9-18.

```
package main

import "fmt"

type calcFunc func(float64) float64
```

```
func printPrice(product string, price float64, calculator
calcFunc) {
               fmt.Println("Product:",   product,   "Price:",
calculator(price))
}

var prizeGiveaway = false

func priceCalcFactory(threshold, rate float64, zeroPrices
*bool) calcFunc {
    return func(price float64) float64 {
        if (*zeroPrices) {
            return 0
        } else if (price > threshold) {
            return price + (price * rate)
        }
        return price
    }
}

func main() {

    watersportsProducts := map[string]float64 {
        "Kayak" : 275,
        "Lifejacket": 48.95,
    }

    soccerProducts := map[string] float64 {
        "Soccer Ball": 19.50,
        "Stadium": 79500,
    }

    prizeGiveaway = false
    waterCalc := priceCalcFactory(100, 0.2, &prizeGiveaway);
    prizeGiveaway = true
    soccerCalc := priceCalcFactory(50, 0.1, &prizeGiveaway)

    for product, price := range watersportsProducts {
        printPrice(product, price, waterCalc)
    }

    for product, price := range soccerProducts {
        printPrice(product, price, soccerCalc)
```

```
    }
}
```

Листинг 9-18 Замыкание указателя в файле main.go в папке functionTypes

В этом примере фабричная функция определяет параметр, который получает указатель на `bool` значение, на котором функция калькулятора закрывается. Указатель следует, когда вызывается функция калькулятора, что гарантирует использование текущего значения. Код в листинге 9-18 выводит следующий результат:

```
Product: Kayak Price: 0
Product: Lifejacket Price: 0
Product: Soccer Ball Price: 0
Product: Stadium Price: 0
```

Резюме

В этой главе я описал способ, которым Go обрабатывает типы функций, позволяя использовать их как любой другой тип данных и позволяя обрабатывать функции как любое другое значение. Я объяснил, как описываются типы функций, и показал, как их можно использовать для определения параметров и результатов других функций. Я продемонстрировал использование псевдонимов типов, чтобы избежать повторения сложных типов функций в коде, и объяснил использование синтаксиса литералов функций и принцип работы замыканий функций. В следующей главе я объясню, как можно определить пользовательские типы данных, создав типы структур.

10. Определение сруктур

В этой главе я описываю структуры — то, как в Go определяются пользовательские типы данных. Я покажу вам, как определять новые типы структур, опишу, как создавать значения из этих типов, и объясню, что происходит, когда значения копируются. Таблица 10-1 помещает структуры в контекст.

Таблица 10-1 Помещение структур в контекст

Вопрос	Ответ
Кто они такие?	Структуры — это типы данных, состоящие из полей.
Почему они полезны?	Структуры позволяют определять пользовательские типы данных.
Как они используются?	Ключевые слова `type` и `struct` используются для определения типа, позволяя указывать имена полей и типы.
Есть ли подводные камни или ограничения?	Необходимо соблюдать осторожность, чтобы избежать непреднамеренного дублирования значений структуры и убедиться, что поля, в которых хранятся указатели, инициализированы до их использования.
Есть ли альтернативы?	Простые приложения могут использовать только встроенные типы данных, но большинству приложений потребуется определить пользовательские типы, для которых структуры являются единственным вариантом.

Таблица 10-2 суммирует главу.

Таблица 10-2 Краткое содержание главы

Проблема	Решение	Листинг
Определить пользовательский тип данных	Определите тип структуры	4, 24
Создать структурное значение	Используйте литеральный синтаксис для создания нового значения и присвоения значений отдельным полям.	5–7, 15
Определить поле структуры, тип которого является другой структурой	Определите встроенное поле	8, 9

Проблема	Решение	Листинг
Сравнить значения структуры	Используйте оператор сравнения, гарантируя, что сравниваемые значения имеют один и тот же тип или типы с одинаковыми полями, и все они должны быть сопоставимы.	10, 11
Преобразовать типы структур	Выполните явное преобразование, убедившись, что типы имеют одинаковые поля.	12
Определить структуру без присвоения имени	Определите анонимную структуру	13–14
Предотвратить дублирование структуры, когда она назначается переменной или используется в качестве аргумента функции	Используйте указатель	16–21, 25–29
Согласованное создание структурных значений	Определить функцию-конструктор	22, 23

Подготовка к этой главе

Чтобы подготовиться к этой главе, откройте новую командную строку, перейдите в удобное место и создайте каталог с именем `structs`. Перейдите в папку `structs` и выполните команду, показанную в листинге 10-1, чтобы инициализировать проект.

> **Подсказка**
> Вы можете загрузить пример проекта для этой главы — и для всех остальных глав этой книги — с `https://github.com/apress/pro-go`. См. Главу 2 о том, как получить помощь, если у вас возникнут проблемы с запуском примеров.

```
go mod init structs
```

Листинг 10-1 Инициализация проекта

Добавьте файл с именем `main.go` в папку `structs` с содержимым, показанным в листинге 10-2.

```
package main

import "fmt"
```

```
func main() {

    fmt.Println("Hello, Structs")
}
```

Листинг 10-2 Содержимое файла main.go в папке structs

Используйте командную строку для запуска команды, показанной в листинге 10-3, в папке `structs`.

```
go run .
```

Листинг 10-3 Запуск примера проекта

Код в файле `main.go` будет скомпилирован и выполнен, что приведет к следующему результату:

```
Hello, Structs
```

Определение и использование структуры

Пользовательские типы данных определяются с помощью функции структур Go, которая демонстрируется в листинге 10-4.

```
package main

import "fmt"

func main() {

    type Product struct {
        name, category string
        price float64
    }

    kayak := Product {
        name: "Kayak",
        category: "Watersports",
        price: 275,
    }

     fmt.Println(kayak.name, kayak.category, kayak.price)
    kayak.price = 300
```

```
    fmt.Println("Changed price:", kayak.price)
}
```

Листинг 10-4 Создание пользовательского типа данных в файле main.go в папке structs

Пользовательские типы данных известны в Go как *структурные типы* и определяются с помощью ключевого слова `type`, имени и ключевого слова `struct`. Скобки окружают ряд полей, каждое из которых определяется именем и типом. Поля одного типа могут быть объявлены вместе, как показано на рисунке 10-1, и все поля должны иметь разные имена.

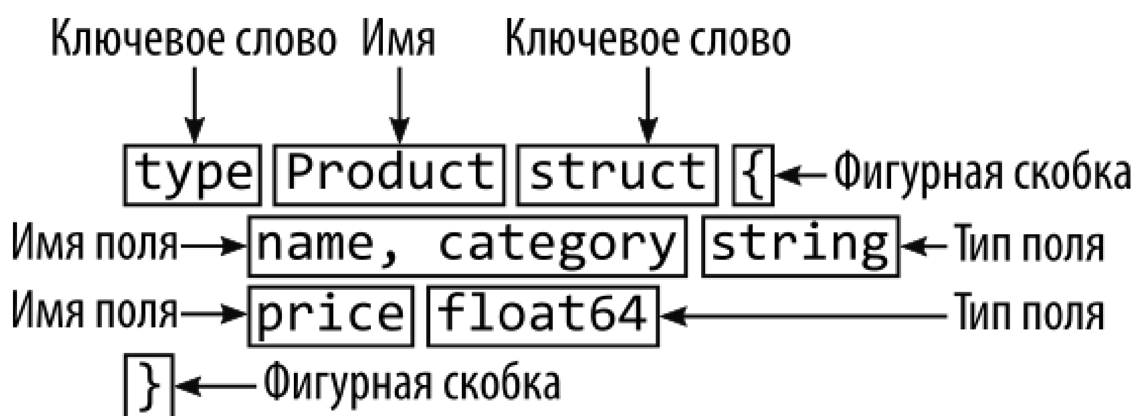

Рисунок 10-1 Определение типа структуры

Этот тип структуры называется `Product` и имеет три поля: поля `name` и `category` содержат строковые значения, а поле `price` содержит значение `float64`. Поля `name` и `category` имеют одинаковый тип и могут быть определены вместе.

ГДЕ GO КЛАССЫ?

Go не делает различий между структурами и классами, как это делают другие языки. Все пользовательские типы данных определяются как структуры, и решение о передаче их по ссылке или по значению принимается в зависимости от того, используется ли указатель. Как я объяснял в главе 4, это дает тот же эффект, что и наличие отдельных категорий типов, но с дополнительной гибкостью, позволяя делать выбор каждый раз, когда используется значение. Однако это требует большего усердия от программиста, который должен продумать последствия своего выбора во время

кодирования. Ни один из подходов не лучше, и результаты по существу одинаковы.

Создание структурных значений

Следующим шагом является создание значения с использованием пользовательского типа, что делается с использованием имени типа структуры, за которым следуют фигурные скобки, содержащие значения для полей структуры, как показано на рисунке 10-2.

Рисунок 10-2 Создание значения структуры

Значение, созданное в листинге 10-4, представляет собой `Product`, поле `name` которого имеет значение `Kayak`, поле `category` — `Watersports`, а поле `price` — `275`. Значение структуры присваивается переменной с именем `kayak`.

Go привередлив к синтаксису и выдаст ошибку, если за конечным значением поля не следует ни запятая, ни закрывающая фигурная скобка. Обычно я предпочитаю конечные запятые, которые позволяют поставить закрывающую фигурную скобку на следующую строку в файле кода, как я сделал с синтаксисом литерала карты в главе 7.

Примечание

Go не позволяет использовать структуры с ключевым словом `const`, и компилятор сообщит об ошибке, если вы попытаетесь определить

константную структуру. Для создания констант можно использовать только типы данных, описанные в главе 9.

Использование значения структуры

Доступ к полям значения структуры осуществляется через имя, присвоенное переменной, так что доступ к значению поля `name` значения структуры, присвоенного переменной `kayak`, осуществляется с помощью `kayak.name`, как показано на рисунок 10-3.

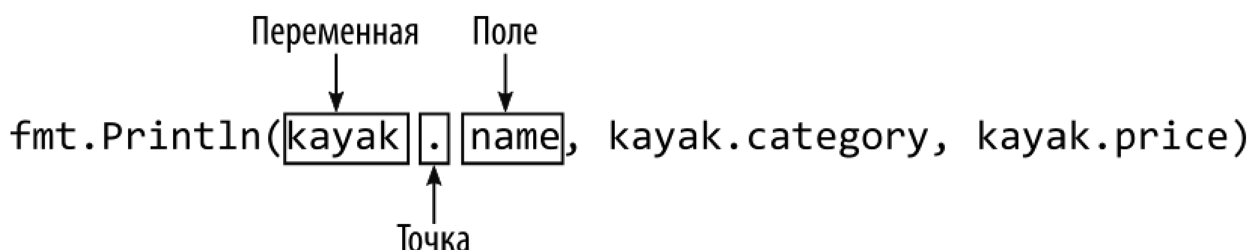

Рисунок 10-3 Доступ к полям структуры

Новые значения могут быть присвоены полю структуры с использованием того же синтаксиса, как показано на рисунке 10-4.

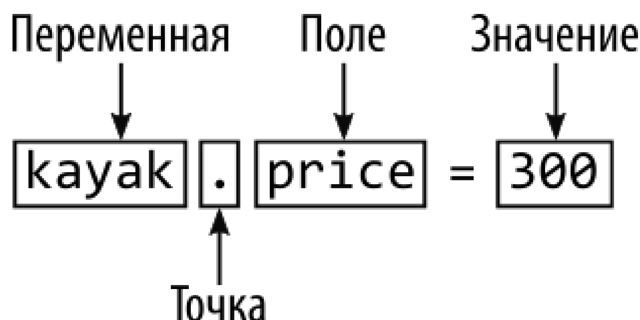

Рисунок 10-4 Изменение поля структуры

Этот оператор присваивает значение `300` полю `price` значения структуры `Product`, присвоенного переменной `kayak`. Код в листинге 10-4 выдает следующий результат при компиляции и выполнении:

```
Kayak Watersports 275
Changed price: 300
```

ПОНИМАНИЕ ТЕГОВ СТРУКТУРЫ

> Тип структуры можно определить с помощью тегов, которые предоставляют дополнительную информацию о том, как следует обрабатывать поле. Теги структуры — это просто строки, которые интерпретируются кодом, обрабатывающим значения структуры, с использованием функций, предоставляемых пакетом `reflect`. См. в главе 21 пример того, как можно использовать теги структур для изменения способа кодирования структур в данных JSON, и см. в главе 28 сведения о том, как самостоятельно получить доступ к тегам структур.

Частичное присвоение значений структуры

При создании значения структуры не обязательно указывать значения для всех полей, как показано в листинге 10-5.

```
package main

import "fmt"

func main() {

    type Product struct {
        name, category string
        price float64
    }

    kayak := Product {
        name: "Kayak",
        category: "Watersports",
    }

    fmt.Println(kayak.name, kayak.category, kayak.price)
    kayak.price = 300
    fmt.Println("Changed price:", kayak.price)
}
```

Листинг 10-5 Назначение некоторых полей в файле main.go в папке structs

Для поля `price` структуры, назначенной переменной `kayak`, начальное значение не указано. Если поле не указано, используется нулевое значение для типа поля. В листинге 10-5 тип нуля для поля

price равен 0, потому что тип поля — float64; код выдает следующий результат при компиляции и выполнении:

```
Kayak Watersports 0
Changed price: 300
```

Как видно из выходных данных, пропуск начального значения не препятствует тому, чтобы значение впоследствии было присвоено полю.

Нулевые типы назначаются всем полям, если вы определяете переменную структурного типа, но не присваиваете ей значение, как показано в листинге 10-6.

```
package main

import "fmt"

func main() {

    type Product struct {
        name, category string
        price float64
    }

    kayak := Product {
        name: "Kayak",
        category: "Watersports",
    }

    fmt.Println(kayak.name, kayak.category, kayak.price)
    kayak.price = 300
    fmt.Println("Changed price:", kayak.price)

    var lifejacket Product
    fmt.Println("Name is zero value:", lifejacket.name == "")
            fmt.Println("Category is zero value:",
lifejacket.category == "")
     fmt.Println("Price is zero value:", lifejacket.price ==
0)
}
```

Листинг 10-6 Неназначенная переменная в файле main.go в папке structs

Тип переменной `lifejacket` — `Product`, но ее полям не присваиваются значения. Значение всех полей `lifejacket` равно нулю для их типа, что подтверждается выходными данными из листинга 10-6:

```
Kayak Watersports 0
Changed price: 300
Name is zero value: true
Category is zero value: true
Price is zero value: true
```

ИСПОЛЬЗОВАНИЕ ФУНКЦИИ NEW ДЛЯ СОЗДАНИЯ СТРУКТУРНЫХ ЗНАЧЕНИЙ

Вы можете увидеть код, который использует встроенную функцию `new` для создания значений структуры, например:

```
...
var lifejacket = new(Product)
...
```

Результатом является указатель на значение структуры, поля которого инициализируются нулевым значением их типа. Это эквивалентно этому утверждению:

```
...
var lifejacket = &Product{}
...
```

Эти подходы взаимозаменяемы, и выбор между ними является вопросом предпочтения.

Использование позиций полей для создания значений структуры

Значения структуры могут быть определены без использования имен, если типы значений соответствуют порядку, в котором поля определяются типом структуры, как показано в листинге 10-7.

```
package main
```

```
import "fmt"

func main() {

    type Product struct {
        name, category string
        price float64
    }

    var kayak = Product { "Kayak", "Watersports", 275.00 }

    fmt.Println("Name:", kayak.name)
    fmt.Println("Category:", kayak.category)
    fmt.Println("Price:", kayak.price)
}
```

Листинг 10-7 Пропуск имен полей в файле main.go в папке structs

Литеральный синтаксис, используемый для определения значения структуры, содержит только значения, которые присваиваются полям структуры в том порядке, в котором они указаны. Код в листинге 10-7 выводит следующий результат:

```
Name: Kayak
Category: Watersports
Price: 275
```

Определение встроенных полей

Если поле определено без имени, оно известно как *встроенное* поле, и доступ к нему осуществляется с использованием имени его типа, как показано в листинге 10-8.

```
package main

import "fmt"

func main() {

    type Product struct {
        name, category string
        price float64
    }
```

```
    type StockLevel struct {
        Product
        count int
    }

    stockItem := StockLevel {
        Product: Product { "Kayak", "Watersports", 275.00 },
        count: 100,
    }

    fmt.Println("Name:", stockItem.Product.name)
    fmt.Println("Count:", stockItem.count)
}
```

Листинг 10-8 Определение встроенных полей в файле main.go в папке structs

Тип структуры `StockLevel` имеет два поля. Первое поле встроено и определяется только с использованием типа, который является типом структуры `Product`, как показано на рисунке 10-5

```
type StockLevel struct {
    Product     ← Встроенное поле
    count int   ← Регулярное поле
}
```

Рисунок 10-5 Определение встроенного поля

Доступ к встроенным полям осуществляется с использованием имени типа поля, поэтому эта функция наиболее полезна для полей, тип которых является структурой. В этом случае встроенное поле определяется с типом `Product`, что означает, что оно назначается и читается с использованием `Product` в качестве имени поля, например:

```
...
stockItem := StockLevel {
    Product: Product { "Kayak", "Watersports", 275.00 },
    count: 100,
}
...
fmt.Println(fmt.Sprint("Name: ", stockItem.Product.name))
...
```

Код в листинге 10-8 выдает следующий результат при компиляции и выполнении:

```
Name: Kayak
Count: 100
```

Как отмечалось ранее, имена полей должны быть уникальными для типа структуры, что означает, что вы можете определить только одно встроенное поле для определенного типа. Если вам нужно определить два поля одного типа, вам нужно будет присвоить имя одному из них, как показано в листинге 10-9.

```
package main

import "fmt"

func main() {

    type Product struct {
        name, category string
        price float64
    }

    type StockLevel struct {
        Product
        Alternate Product
        count int
    }

    stockItem := StockLevel {
        Product: Product { "Kayak", "Watersports", 275.00 },
         Alternate: Product{"Lifejacket", "Watersports", 48.95
},
        count: 100,
    }

    fmt.Println("Name:", stockItem.Product.name)
    fmt.Println("Alt Name:", stockItem.Alternate.name)
}
```

Листинг 10-9 Определение дополнительного поля в файле main.go в папке structs

Тип StockLevel имеет два поля типа Product, но только одно из них может быть встроенным полем. Для второго поля я присвоил имя, через которое осуществляется доступ к полю. Код в листинге 10-9 выдает следующий результат при компиляции и выполнении:

```
Name: Kayak
Alt Name: Lifejacket
```

Сравнение значений структуры

Значения структур сопоставимы, если можно сравнить все их поля. В листинге 10-10 создается несколько значений структуры и применяется оператор сравнения, чтобы определить, равны ли они.

```
package main

import "fmt"

func main() {

    type Product struct {
        name, category string
        price float64
    }

     p1 := Product { name: "Kayak", category: "Watersports",
price: 275.00 }
     p2 := Product { name: "Kayak", category: "Watersports",
price: 275.00 }
    p3 := Product { name: "Kayak", category: "Boats", price:
275.00 }

    fmt.Println("p1 == p2:", p1 == p2)
    fmt.Println("p1 == p3:", p1 == p3)
}
```

Листинг 10-10 Сравнение значений структуры в файле main.go в папке structs

Значения структуры p1 и p2 равны, потому что все их поля равны. Значения структуры p1 и p3 не равны, потому что значения, присвоенные их полям category, различны. Скомпилируйте и запустите проект, и вы увидите следующие результаты:

```
p1 == p2: true
p1 == p3: false
```

Структуры нельзя сравнивать, если тип структуры определяет поля с несравнимыми типами, например срезы, как показано в листинге 10-11.

```
package main

import "fmt"

func main() {

    type Product struct {
        name, category string
        price float64
        otherNames []string
    }

     p1 := Product { name: "Kayak", category: "Watersports",
price: 275.00 }
     p2 := Product { name: "Kayak", category: "Watersports",
price: 275.00 }
    p3 := Product { name: "Kayak", category: "Boats", price:
275.00 }

    fmt.Println("p1 == p2:", p1 == p2)
    fmt.Println("p1 == p3:", p1 == p3)
}
```

Листинг 10-11 Добавление несравнимого поля в файл main.go в папку structs

Как объяснялось в главе 7, оператор сравнения Go нельзя применять к срезам, что означает невозможность сравнения значений `Product`. При компиляции этот код выдает следующие ошибки:

```
.\main.go:17:33: invalid operation: p1 == p2 (struct
containing []string cannot be compared)
.\main.go:18:33: invalid operation: p1 == p3 (struct
containing []string cannot be compared)
```

Преобразование между типами структур

Тип структуры можно преобразовать в любой другой тип структуры с теми же полями, что означает, что все поля имеют одинаковые имена и типы и определяются в одном и том же порядке, как показано в листинге 10-12.

```
package main

import "fmt"

func main() {

    type Product struct {
        name, category string
        price float64
        //otherNames []string
    }

    type Item struct {
        name string
        category string
        price float64
    }

    prod := Product { name: "Kayak", category: "Watersports",
price: 275.00 }
    item := Item { name: "Kayak", category: "Watersports",
price: 275.00 }

    fmt.Println("prod == item:", prod == Product(item))
}
```

Листинг 10-12 Преобразование типа структуры в файле main.go в папке structs

Значения, созданные из типов структур `Product` и `Item`, можно сравнивать, поскольку они определяют одни и те же поля в одном и том же порядке. Скомпилировать и выполнить проект; вы увидите следующий вывод:

```
prod == item: true
```

Определение анонимных типов структур

Анонимные типы структур определяются без использования имени, как показано в листинге 10-13.

```
package main

import "fmt"

func writeName(val struct {
        name, category string
        price float64}) {
    fmt.Println("Name:", val.name)
}

func main() {

    type Product struct {
        name, category string
        price float64
        //otherNames []string
    }

    type Item struct {
        name string
        category string
        price float64
    }

    prod := Product { name: "Kayak", category: "Watersports",
price: 275.00 }
      item := Item { name: "Stadium", category: "Soccer",
price: 75000 }

    writeName(prod)
    writeName(item)
}
```

Листинг 10-13 Определение анонимного типа структуры в файле main.go в папке structs

Функция `writeName` использует в качестве параметра анонимный тип структуры, что означает, что она может принимать любой тип структуры, определяющий указанный набор полей. Скомпилировать и выполнить проект; вы увидите следующий вывод:

```
Name: Kayak
Name: Stadium
```

Я не нахожу эту функцию особенно полезной, поскольку она показана в листинге 10-13, но есть вариант, который я использую: определение анонимной структуры и присвоение ей значения за один шаг. Это полезно при вызове кода, который проверяет типы, которые он получает во время выполнения, используя возможности, предоставляемые пакетом `reflect`, который я описываю в главах 27–29. Пакет `reflect` содержит расширенные функции, но он используется другими частями стандартной библиотеки, такими как встроенная поддержка кодирования данных JSON. Я подробно объясню функции JSON в главе 21, но в этой главе листинг 10-14 демонстрирует использование анонимной структуры для выбора полей, которые должны быть включены в строку JSON.

```
package main

import (
    "fmt"
    "encoding/json"
    "strings"
)

func main() {

    type Product struct {
        name, category string
        price float64
    }

    prod := Product { name: "Kayak", category: "Watersports",
price: 275.00 }

    var builder strings.Builder
    json.NewEncoder(&builder).Encode(struct {
        ProductName string
        ProductPrice float64
    }{
        ProductName: prod.name,
        ProductPrice: prod.price,
```

```
    })
    fmt.Println(builder.String())
}
```

Листинг 10-14 Присвоение значения анонимной структуре в файле main.go в папке structs

Не беспокойтесь о пакетах `encoding/json` и `strings`, которые описаны в последующих главах. В этом примере показано, как можно определить анонимную структуру и присвоить ей значение за один шаг, который я использую в листинге `10-14` для создания структуры с полями `ProductName` и `ProductPrice`, которым я затем присваиваю значения из полей `Product`. Скомпилировать и выполнить проект; вы увидите следующий вывод:

```
{"ProductName":"Kayak","ProductPrice":275}
```

Создание массивов, срезов и карт, содержащих структурные значения

Тип структуры можно не указывать при заполнении массивов, срезов и карт значениями структуры, как показано в листинге 10-15.

```
package main

import "fmt"

func main() {

    type Product struct {
        name, category string
        price float64
        //otherNames []string
    }

    type StockLevel struct {
        Product
        Alternate Product
        count int
    }
```

```
    array := [1]StockLevel {
        {
            Product: Product { "Kayak", "Watersports", 275.00 },
              Alternate: Product{"Lifejacket", "Watersports", 48.95 },
            count: 100,
        },
    }
    fmt.Println("Array:", array[0].Product.name)

    slice := []StockLevel {
        {
            Product: Product { "Kayak", "Watersports", 275.00 },
              Alternate: Product{"Lifejacket", "Watersports", 48.95 },
            count: 100,
        },
    }
    fmt.Println("Slice:", slice[0].Product.name)

    kvp := map[string]StockLevel {
        "kayak": {
            Product: Product { "Kayak", "Watersports", 275.00 },
              Alternate: Product{"Lifejacket", "Watersports", 48.95 },
            count: 100,
        },
    }
    fmt.Println("Map:", kvp["kayak"].Product.name)
}
```

Листинг 10-15 Пропуск типа структуры в файле main.go в папке structs

Код в листинге 10-15 создает массив, срез и карту, все из которых заполняются значением `StockLevel`. Компилятор может вывести тип значения структуры из содержащейся структуры данных, что позволяет выразить код более лаконично. В листинге 10-15 выводится следующий результат:

```
Array: Kayak
```

```
Slice: Kayak
Map: Kayak
```

Понимание структур и указателей

Присвоение структуры новой переменной или использование структуры в качестве параметра функции создает новое значение, которое копирует значения поля, как показано в листинге 10-16.

```
package main

import "fmt"

func main() {

    type Product struct {
        name, category string
        price float64
    }

    p1 := Product {
        name: "Kayak",
        category: "Watersports",
        price: 275,
    }

    p2 := p1

    p1.name = "Original Kayak"

    fmt.Println("P1:", p1.name)
    fmt.Println("P2:", p2.name)
}
```

Листинг 10-16 Копирование значения структуры в файле main.go в папке structs

Значение структуры создается и присваивается переменной `p1` и копируется в переменную `p2`. Поле `name` первого значения структуры изменяется, а затем записываются оба значения `name`. Вывод из листинга 10-16 подтверждает, что при присвоении значения структуры создается копия:

```
P1: Original Kayak
P2: Kayak
```

Как и другие типы данных, ссылки на значения структур можно создавать с помощью указателей, как показано в листинге 10-17.

```
package main

import "fmt"

func main() {

    type Product struct {
        name, category string
        price float64
    }

    p1 := Product {
        name: "Kayak",
        category: "Watersports",
        price: 275,
    }

    p2 := &p1

    p1.name = "Original Kayak"

    fmt.Println("P1:", p1.name)
    fmt.Println("P2:", (*p2).name)
}
```

Листинг 10-17 Использование указателя на структуру в файле main.go в папке structs

Я использовал амперсанд для создания указателя на переменную `p1` и присвоил адрес `p2`, тип которого становится `*Product`, что означает указатель на значение `Product`. Обратите внимание, что я должен использовать круглые скобки, чтобы следовать указателю на значение структуры, а затем читать значение поля `name`, как показано на рисунке 10-6.

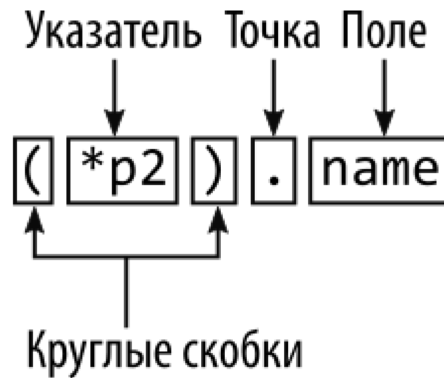

Рисунок 10-6 Чтение поля структуры через указатель

Эффект заключается в том, что изменение, внесенное в поле name, считывается как через p1, так и через p2, создавая следующий вывод, когда код компилируется и выполняется:

```
P1: Original Kayak
P2: Original Kayak
```

Понимание удобного синтаксиса указателя структуры

Доступ к полям структуры с помощью указателя неудобен, что является проблемой, поскольку структуры обычно используются в качестве аргументов и результатов функций, а указатели необходимы для того, чтобы структуры не дублировались без необходимости и чтобы изменения, сделанные функциями, влияли на значения, полученные в качестве параметров, т.к. показано в листинге 10-18.

```
package main

import "fmt"

type Product struct {
    name, category string
    price float64
}

func calcTax(product *Product) {
    if ((*product).price > 100) {
        (*product).price += (*product).price * 0.2
    }
}
```

```
func main() {

    kayak := Product {
        name: "Kayak",
        category: "Watersports",
        price: 275,
    }

    calcTax(&kayak)

    fmt.Println("Name:", kayak.name, "Category:",
        kayak.category, "Price", kayak.price)
}
```

Листинг 10-18 Использование указателей структуры в файле main.go в папке structs

Этот код работает, но его трудно читать, особенно когда в одном и том же блоке кода, например в теле метода `calcTax`, есть несколько ссылок.

Чтобы упростить этот тип кода, Go будет следовать указателям на поля структуры без символа звездочки, как показано в листинге 10-19.

```
package main

import "fmt"

type Product struct {
    name, category string
    price float64
}

func calcTax(product *Product) {
    if (product.price > 100) {
        product.price += product.price * 0.2
    }
}

func main() {

    kayak := Product {
        name: "Kayak",
        category: "Watersports",
        price: 275,
```

```
    }

    calcTax(&kayak)

    fmt.Println("Name:", kayak.name, "Category:",
        kayak.category, "Price", kayak.price)
}
```

Листинг 10-19 Использование удобного синтаксиса указателя структуры в файле main.go в папке structs

Звездочка и круглые скобки не требуются, что позволяет рассматривать указатель на структуру так, как если бы он был значением структуры, как показано на рисунке 10-7.

Рисунок 10-7 Использование структуры или указателя на структуру

Эта функция не меняет тип данных параметра функции, который по-прежнему имеет значение `*Product`, и применяется только при доступе к полям. Оба листинга 10-18 и `10-19` выдают следующий результат:

```
Name: Kayak Category: Watersports Price 330
```

Понимание указателей на значения

В более ранних примерах указатели использовались в два этапа. Первый шаг — создать значение и присвоить его переменной, например:

```
...
kayak := Product {
    name: "Kayak",
    category: "Watersports",
    price: 275,
}
...
```

Второй шаг — использовать оператор адреса для создания указателя, например:

```
...
calcTax(&kayak)
...
```

Нет необходимости присваивать переменной значение структуры перед созданием указателя, а оператор адреса можно использовать непосредственно с литеральным синтаксисом структуры, как показано в листинге 10-20.

```
package main

import "fmt"

type Product struct {
    name, category string
    price float64
}

func calcTax(product *Product) {
    if (product.price > 100) {
        product.price += product.price * 0.2
    }
}

func main() {

    kayak := &Product {
        name: "Kayak",
        category: "Watersports",
        price: 275,
    }

    calcTax(kayak)

    fmt.Println("Name:", kayak.name, "Category:",
        kayak.category, "Price", kayak.price)
}
```

Листинг 10-20 Создание указателя непосредственно в файле main.go в папке structs

Оператор адреса используется перед типом структуры, как показано на рисунке 10-8.

```
         Оператор   Тип
              |       |
              v       v
kayak := [&][Product] {
```

Рисунок 10-8 Создание указателя на значение структуры

Код в листинге 10-20 использует только указатель на значение `Product`, а это означает, что нет смысла создавать обычную переменную и затем использовать ее для создания указателя. Возможность создавать указатели непосредственно из значений может помочь сделать код более кратким, как показано в листинге 10-21.

```
package main

import "fmt"

type Product struct {
    name, category string
    price float64
}

func calcTax(product *Product) *Product {
    if (product.price > 100) {
        product.price += product.price * 0.2
    }
    return product
}

func main() {

    kayak := calcTax(&Product {
        name: "Kayak",
        category: "Watersports",
        price: 275,
    })

    fmt.Println("Name:", kayak.name, "Category:",
        kayak.category, "Price", kayak.price)
```

```
}
```

Листинг 10-21 Использование указателей непосредственно в файле main.go в папке structs

Я изменил функцию `calcTax`, чтобы она выдавала результат, который позволяет функции преобразовывать значение `Product` с помощью указателя. В основной функции я использовал оператор адреса с литеральным синтаксисом для создания значения `Product` и передал указатель на него в функцию `calcTax`, присваивая преобразованный результат переменной типа `*Pointer`. Оба листинга 10-20 и 10-21 выдают следующий результат:

```
Name: Kayak Category: Watersports Price 330
```

Понимание функций конструктора структуры

Функция-конструктор отвечает за создание значений структуры с использованием значений, полученных через параметры, как показано в листинге 10-22.

```
package main

import "fmt"

type Product struct {
    name, category string
    price float64
}

func newProduct(name, category string, price float64)
*Product {
    return &Product{name, category, price}
}

func main() {

    products := [2]*Product {
        newProduct("Kayak", "Watersports", 275),
        newProduct("Hat", "Skiing", 42.50),
    }

    for _, p := range products {
```

```
                            fmt.Println("Name:",    p.name,
"Category:",  p.category, "Price", p.price)
    }
}
```

Листинг 10-22 Определение функции конструктора в файле main.go в папке structs

Функции-конструкторы используются для согласованного создания структурных значений. Функции-конструкторы обычно называются `new` или `New`, за которыми следует тип структуры, так что функция-конструктор для создания значений `Product` называется `newProduct`. (Я объясняю, почему имена функций-конструкторов часто начинаются с заглавной буквы в главе 12.)

Функции-конструкторы возвращают указатели на структуры, а оператор адреса используется непосредственно с литеральным синтаксисом структуры, как показано на рисунке 10-9.


```
func newProduct(name, category string, price float64) *Product {
    return &Product {name, category, price}
}
```

Рисунок 10-9 Использование указателей в функции-конструкторе

Мне нравится создавать значения в функциях-конструкторах, полагаясь на позиции полей, как показано в листинге 10-22, хотя это только мое предпочтение. Важно, чтобы вы не забывали возвращать указатель, чтобы избежать дублирования значения структуры при выходе из функции. В листинге 10-22 для хранения данных о товарах используется массив, и вы можете увидеть использование указателей в типе массива:

```
...
products := [2]*Product {
...
```

Этот тип задает массив, который будет содержать два указателя на значения структуры `Product`. Код в листинге 10-22 при компиляции и выполнении выдает следующий результат:

```
Name: Kayak Category: Watersports Price 275
Name: Hat Category: Skiing Price 42.5
```

Преимуществом использования функций-конструкторов является согласованность, гарантирующая, что изменения в процессе построения отражаются во всех значениях структуры, созданных функцией. Например, в листинге 10-23 конструктор изменяется для применения скидки ко всем продуктам.

```
...
func newProduct(name, category string, price float64)
*Product {
    return &Product{name, category, price - 10}
}
...
```

Листинг 10-23 Изменение конструктора в файле main.go в папке structs

Это простое изменение, но оно будет применено ко всем значениям `Product`, созданным функцией `newProduct`, а это означает, что мне не нужно находить все точки в коде, где создаются значения `Product`, и изменять их по отдельности. К сожалению, Go не препятствует использованию литерального синтаксиса, когда определена функция-конструктор, что означает необходимость тщательного использования функций-конструкторов. Код в листинге 10-23 выдает следующий результат:

```
Name: Kayak Category: Watersports Price 265
Name: Hat Category: Skiing Price 32.5
```

Использование типов указателей для полей структуры

Указатели также можно использовать для полей структур, включая указатели на другие типы структур, как показано в листинге 10-24.

```
package main

import "fmt"

type Product struct {
    name, category string
    price float64
```

```
    *Supplier
}

type Supplier struct {
    name, city string
}

func newProduct(name, category string, price float64,
supplier *Supplier) *Product {
    return &Product{name, category, price -10, supplier}
}

func main() {

    acme := &Supplier { "Acme Co", "New York"}

    products := [2]*Product {
        newProduct("Kayak", "Watersports", 275, acme),
        newProduct("Hat", "Skiing", 42.50, acme),
    }

    for _, p := range products {
        fmt.Println("Name:", p.name, "Supplier:",
            p.Supplier.name, p.Supplier.city)
    }
}
```

Листинг 10-24 Использование указателей для полей структуры в файле main.go в папке structs

Я добавил к типу Product встроенное поле, которое использует тип Supplier, и обновил функцию newProduct, чтобы она принимала указатель на Supplier. Доступ к полям, определенным структурой Supplier, осуществляется с использованием поля, определенного структурой Product, как показано на рисунке 10-10.

Рисунок 10-10 Доступ к вложенному полю структуры

Обратите внимание, как Go обрабатывает использование типа указателя для встроенного поля структуры, что позволяет мне обращаться к полю по имени типа структуры, которым в данном примере является Supplier. Код в листинге 10-24 выдает следующий результат:

```
Name: Kayak Supplier: Acme Co New York
Name: Hat Supplier: Acme Co New York
```

Общие сведения о копировании поля указателя

При копировании структур необходимо соблюдать осторожность, чтобы учесть влияние на поля указателя, как показано в листинге 10-25.

```
package main

import "fmt"

type Product struct {
    name, category string
    price float64
    *Supplier
}

type Supplier struct {
    name, city string
}

func newProduct(name, category string, price float64,
supplier *Supplier) *Product {
    return &Product{name, category, price -10, supplier}
}

func main() {

    acme := &Supplier { "Acme Co", "New York"}

    p1 := newProduct("Kayak", "Watersports", 275, acme)
    p2 := *p1

    p1.name = "Original Kayak"
    p1.Supplier.name = "BoatCo"
```

```
    for _, p := range []Product { *p1, p2 } {
        fmt.Println("Name:", p.name, "Supplier:",
            p.Supplier.name, p.Supplier.city)
    }
}
```

Листинг 10-25 Копирование структуры в файле main.go в папке structs

Функция newProduct используется для создания указателя на значение Product, которое присваивается переменной с именем p1. Указатель следует и присваивается переменной с именем p2, что приводит к копированию значения Product. Поля p1.name и p1.Supplier.name изменяются, а затем используется цикл for для записи сведений об обоих значениях Product, что приводит к следующему результату:

```
Name: Original Kayak Supplier: BoatCo New York
Name: Kayak Supplier: BoatCo New York
```

Выходные данные показывают, что изменение поля name затронуло только одно из значений Product, в то время как изменение поля Supplier.name затронуло оба. Это происходит потому, что при копировании структуры Product был скопирован указатель, присвоенный полю Supplier, а не значение, на которое он указывает, создавая эффект, показанный на рисунке 10-11.

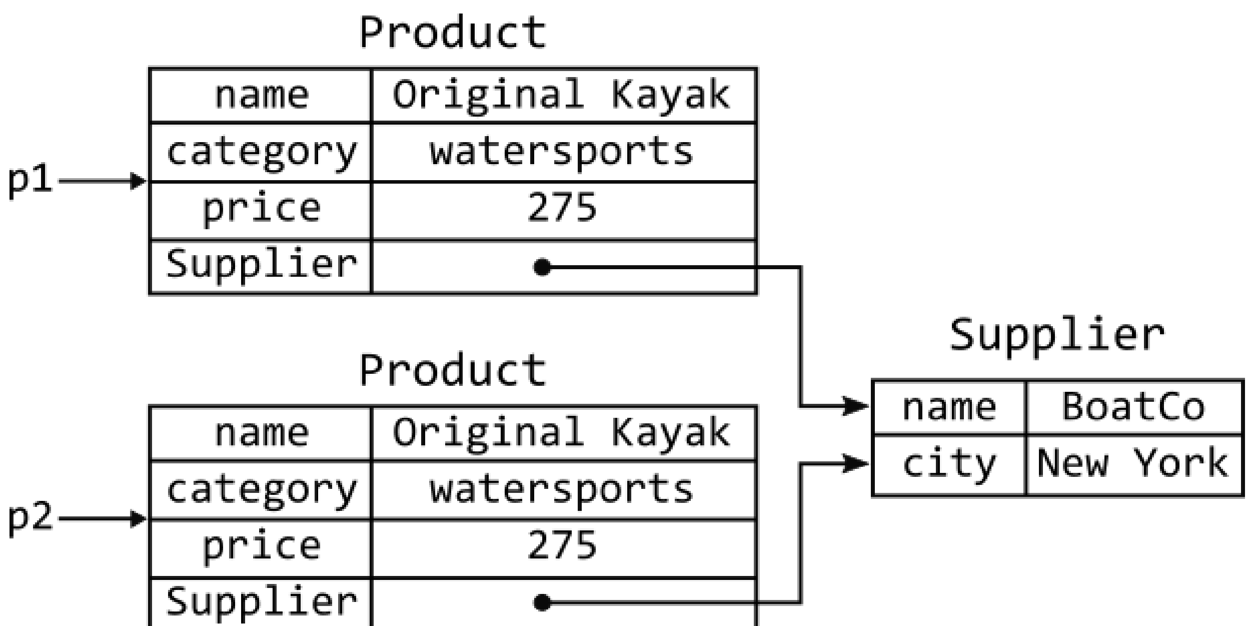

Рисунок 10-11 Эффект копирования структуры с полем указателя

Это часто называют *поверхностной копией*, когда копируются указатели, но не значения, на которые они указывают. В Go нет встроенной поддержки выполнения *глубокого копирования*, когда указатели отслеживаются, а их значения дублируются. Вместо этого необходимо выполнить копирование вручную, как показано в листинге 10-26.

```
package main

import "fmt"

type Product struct {
    name, category string
    price float64
    *Supplier
}

type Supplier struct {
    name, city string
}

func newProduct(name, category string, price float64, supplier *Supplier) *Product {
    return &Product{name, category, price -10, supplier}
}

func copyProduct(product *Product) Product {
    p := *product
    s := *product.Supplier
    p.Supplier = &s
    return p
}

func main() {

    acme := &Supplier { "Acme Co", "New York"}

    p1 := newProduct("Kayak", "Watersports", 275, acme)
    p2 := copyProduct(p1)
```

```
    p1.name = "Original Kayak"
    p1.Supplier.name = "BoatCo"

    for _, p := range []Product { *p1, p2 } {
        fmt.Println("Name:", p.name, "Supplier:",
            p.Supplier.name, p.Supplier.city)
    }
}
```

Листинг 10-26 Копирование значения структуры в файле main.go в папке structs

Чтобы обеспечить дублирование `Supplier`, функция `copyProduct` присваивает его отдельной переменной, а затем создает указатель на эту переменную. Это неудобно, но эффект заключается в принудительном копировании структуры, хотя этот метод специфичен для одного типа структуры и должен повторяться для каждого поля вложенной структуры. Вывод из листинга 10-26 показывает эффект глубокого копирования:

```
Name: Original Kayak Supplier: BoatCo New York
Name: Kayak Supplier: Acme Co New York
```

Понимание нулевого значения для структур и указателей на структуры

Нулевое значение для типа структуры — это значение структуры, полям которой присвоен нулевой тип. Нулевое значение указателя на структуру равно `nil`, как показано в листинге 10-27.

```
package main

import "fmt"

type Product struct {
    name, category string
    price float64
}

func main() {

    var prod Product
    var prodPtr *Product
```

```
            fmt.Println("Value:",   prod.name,   prod.category,
prod.price)
    fmt.Println("Pointer:", prodPtr)
}
```

Листинг 10-27 Изучение нулевых типов в файле main.go в папке structs

Скомпилируйте и выполните проект, и вы увидите нулевые значения, представленные в выводе, с пустыми строками для полей `name` и `category`, поскольку пустая строка является нулевым значением для строкового типа:

```
Value: 0
Pointer: <nil>
```

Существует ловушка, с которой я часто сталкиваюсь, когда структура определяет поле с указателем на другой тип структуры, как показано в листинге 10-28.

```
package main

import "fmt"

type Product struct {
    name, category string
    price float64
    *Supplier
}

type Supplier struct {
    name, city string
}

func main() {

    var prod Product
    var prodPtr *Product

            fmt.Println("Value:",   prod.name,   prod.category,
prod.price, prod.Supplier.name)
    fmt.Println("Pointer:", prodPtr)
}
```

Листинг 10-28 Добавление поля указателя в файл main.go в папку structs

Проблема здесь заключается в попытке доступа к полю `name` встроенной структуры. Нулевое значение встроенного поля равно `nil`, что вызывает следующую ошибку времени выполнения:

```
panic: runtime error: invalid memory address or nil pointer
dereference
[signal 0xc0000005 code=0x0 addr=0x0 pc=0x5bc592]
goroutine 1 [running]:
main.main()
        C:/structs/main.go:20 +0x92
exit status 2
```

Я сталкиваюсь с этой ошибкой так часто, что обычно инициализирую поля указателя структуры, как показано в листинге 10-29 и часто повторяется в последующих главах.

```
...
func main() {

    var prod Product = Product{ Supplier: &Supplier{}}
    var prodPtr *Product

          fmt.Println("Value:",  prod.name,  prod.category,
prod.price, prod.Supplier.name)
    fmt.Println("Pointer:", prodPtr)
}
...
```

Листинг 10-29 Инициализация поля указателя структуры в файле main.go в папке structs

Это позволяет избежать ошибки времени выполнения, которую вы можете увидеть в выводе, полученном при компиляции и выполнении проекта:

```
Value:   0
Pointer: <nil>
```

Резюме

В этой главе я описываю функцию структур Go, которая используется для создания пользовательских типов данных. Я объяснил, как определять поля структур, как создавать значения из типов структур и как использовать типы структур в коллекциях. Я также показал вам, как создавать анонимные структуры и как использовать указатели для управления обработкой значений при их копировании. В следующей главе я опишу поддержку Go для методов и интерфейсов.

11. Использование методов и интерфейсов

В этой главе я описываю поддержку Go для методов, которые можно использовать для предоставления функций структурам и для создания абстракций через интерфейсы. В Таблице 11-1 эти функции представлены в контексте.

Таблица 11-1 Помещение методов и интерфейсов в контекст

Вопрос	Ответ
Кто они такие?	Методы — это функции, которые вызываются в структуре и имеют доступ ко всем полям, определенным типом значения. Интерфейсы определяют наборы методов, которые могут быть реализованы типами структур.
Почему они полезны?	Эти функции позволяют смешивать и использовать типы благодаря их общим характеристикам.
Как они используются?	Методы определяются с помощью ключевого слова `func`, но с добавлением получателя. Интерфейсы определяются с использованием ключевых слов `type` и `interface`.
Есть ли подводные камни или ограничения?	Аккуратное использование указателей важно при создании методов, а при использовании интерфейсов необходимо соблюдать осторожность, чтобы избежать проблем с базовыми динамическими типами.
Есть ли альтернативы?	Это необязательные функции, но они позволяют создавать сложные типы данных и использовать их с помощью общих функций, которые они предоставляют.

Таблица 11-2 суммирует главу.

Таблица 11-2 Краткое содержание главы

Проблема	Решение	Листинг
Определить метод	Используйте синтаксис функции, но добавьте приемник, через который будет вызываться метод.	4–8, 13–15
Вызывать методы для ссылок на значения структуры	Используйте указатель на полученный метод	9, 10

Проблема	**Решение**	**Листинг**
Определить методы для неструктурных типов	Используйте псевдоним типа	11, 12
Описать общие характеристики, которые будут общими для нескольких типов	Определите интерфейс	16
Реализовать интерфейс	Определите все методы, указанные интерфейсом, используя выбранный тип структуры в качестве получателя.	17, 18
Использовать интерфейс	Вызовите методы для значения интерфейса	19–21
Решить, будут ли создаваться копии значений структуры при назначении переменным интерфейса.	Используйте указатель или значение при назначении или используйте тип указателя в качестве получателя при реализации методов интерфейса.	22–25
Сравнить значения интерфейса	Используйте операторы сравнения и убедитесь, что динамические типы сопоставимы	26, 27
Доступ к динамическому типу значения интерфейса	Используйте утверждение типа	28–31
Определить переменную, которой можно присвоить любое значение	Используйте пустой интерфейс	32–34

Подготовка к этой главе

Чтобы подготовиться к этой главе, откройте новую командную строку, перейдите в удобное место и создайте каталог с именем `methodAndInterfaces`. Перейдите в папку `methodAndInterfaces` и выполните команду, показанную в листинге 11-1, для инициализации проекта.

```
go mod init methodsandinterfaces
```

Листинг 11-1 Инициализация проекта

Добавьте файл с именем `main.go` в папку `methodAndInterfaces` с содержимым, показанным в листинге 11-2.

Подсказка
Вы можете загрузить пример проекта для этой главы — и для всех остальных глав этой книги — с `https://github.com/apress/pro-`

go. См. Главу 2 о том, как получить помощь, если у вас возникнут проблемы с запуском примеров.

```
package main

import "fmt"

type Product struct {
    name, category string
    price float64
}

func main() {

    products := []*Product {
        {"Kayak", "Watersports", 275 },
        {"Lifejacket", "Watersports", 48.95 },
        {"Soccer Ball", "Soccer", 19.50},
    }

    for _, p := range products {
         fmt.Println("Name:", p.name, "Category:", p.category,
"Price", p.price)
    }
}
```

Листинг 11-2 Содержимое файла main.go в папке methodAndInterfaces

Используйте командную строку для запуска команды, показанной в листинге 11-3, в папке `methodAndInterfaces`.

```
go run .
```

Листинг 11-3 Запуск примера проекта

Код в файле `main.go` будет скомпилирован и выполнен, что приведет к следующему результату:

```
Name: Kayak Category: Watersports Price 275
Name: Lifejacket Category: Watersports Price 48.95
Name: Soccer Ball Category: Soccer Price 19.5
```

Определение и использование методов

Методы — это функции, которые можно вызывать через значение, и они представляют собой удобный способ выражения функций, которые работают с определенным типом. Лучший способ понять, как работают методы, — начать с обычной функции, как показано в листинге 11-4.

```
package main

import "fmt"

type Product struct {
    name, category string
    price float64
}

func printDetails(product *Product) {
           fmt.Println("Name:",   product.name,   "Category:",
product.category,
        "Price", product.price)
}

func main() {

    products := []*Product {
        {"Kayak", "Watersports", 275 },
        {"Lifejacket", "Watersports", 48.95 },
        {"Soccer Ball", "Soccer", 19.50},
    }

    for _, p := range products {
        printDetails(p)
    }
}
```

Листинг 11-4 Определение функции в файле main.go в папке methodAndInterfaces

Функция `printDetails` получает указатель на `Product`, который используется для записи значения полей `name`, `category` и `price`. Ключевым моментом в этом разделе является способ вызова функции `printDetails`:

```
...
printDetails(p)
...
```

За именем функции следуют аргументы, заключенные в круглые скобки. В листинге 11-5 реализована та же функциональность, что и в методе.

```
package main

import "fmt"

type Product struct {
    name, category string
    price float64
}

func newProduct(name, category string, price float64)
*Product {
    return &Product{ name, category, price }
}

func (product *Product) printDetails() {
        fmt.Println("Name:", product.name, "Category:",
product.category,
        "Price", product.price)
}

func main() {

    products := []*Product {
        newProduct("Kayak", "Watersports", 275),
        newProduct("Lifejacket", "Watersports", 48.95),
        newProduct("Soccer Ball", "Soccer", 19.50),
    }

    for _, p := range products {
        p.printDetails()
    }
}
```

Листинг 11-5 Определение метода в файле main.go в папке methodAndInterfaces

Методы определяются как функции с использованием того же ключевого слова func, но с добавлением *приемника*, обозначающего специальный параметр, являющийся типом, с которым работает метод, как показано на рисунке 11-1.

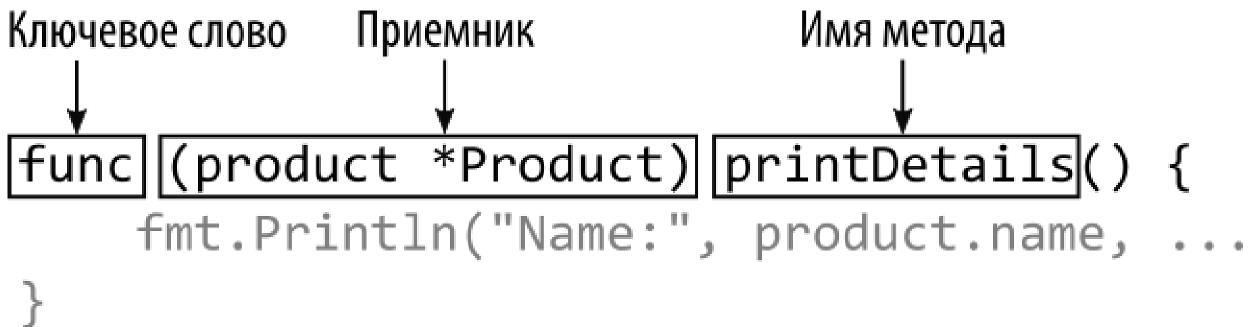

Рисунок 11-1 Метод

Тип получателя для этого метода — *Product, и ему дается имя product, которое можно использовать в методе так же, как и любой нормальный параметр функции. Не требуется никаких изменений в кодовом блоке, который может обрабатывать приемник как обычный параметр функции:

```
...
func (product *Product) printDetails() {
           fmt.Println("Name:",   product.name,   "Category:",
product.category,
        "Price", product.price)
}
...
```

Что отличает методы от обычных функций, так это способ вызова метода:

```
...
p.printDetails()
...
```

Методы вызываются через значение, тип которого соответствует получателю. В этом случае я использую значение *Product, сгенерированное циклом for, чтобы вызвать метод printDetails для каждого значения в срезе и получить следующий результат:

```
Name: Kayak Category: Watersports Price 275
Name: Lifejacket Category: Watersports Price 48.95
Name: Soccer Ball Category: Soccer Price 19.5
```

Определение параметров метода и результатов

Методы могут определять параметры и результаты точно так же, как обычные функции, как показано в листинге 11-6, но с добавлением получателя.

```
package main

import "fmt"

type Product struct {
    name, category string
    price float64
}

func  newProduct(name,  category  string,  price  float64)
*Product {
    return &Product{ name, category, price }
}

func (product *Product) printDetails() {
           fmt.Println("Name:",   product.name,   "Category:",
product.category,
        "Price",  product.calcTax(0.2, 100))
}

func  (product  *Product)  calcTax(rate,  threshold  float64)
float64 {
    if (product.price > threshold) {
        return product.price + (product.price * rate)
    }
    return product.price;
}

func main() {

    products := []*Product {
        newProduct("Kayak", "Watersports", 275),
        newProduct("Lifejacket", "Watersports", 48.95),
```

```
        newProduct("Soccer Ball", "Soccer", 19.50),
    }

    for _, p := range products {
        p.printDetails()
    }
}
```

Листинг 11-6 Параметр и результат в файле main.go в папке methodAndInterfaces

Параметры метода определяются между круглыми скобками, которые следуют за именем, за которым следует тип результата, как показано на рисунке 11-2.

Рисунок 11-2 Метод с параметрами и результатом

Метод `calcTax` определяет параметры `rate` и `threshold` и возвращает результат `float64`. В блоке кода метода не требуется никакой специальной обработки, чтобы отличить получатель от обычных параметров.

При вызове метода аргументы предоставляются так же, как и для обычной функции, например:

```
...
product.calcTax(0.2, 100)
...
```

В этом примере метод `printDetails` вызывает метод `calcTax`, что приводит к следующему результату:

```
Name: Kayak Category: Watersports Price 330
Name: Lifejacket Category: Watersports Price 48.95
Name: Soccer Ball Category: Soccer Price 19.5
```

Понимание перегрузки метода

Go не поддерживает перегрузку методов, когда можно определить несколько методов с одним и тем же именем, но с разными параметрами. Вместо этого каждая комбинация имени метода и типа

приемника должна быть уникальной, независимо от других определенных параметров. В листинге 11-7 я определил методы с одинаковыми именами, но разными типами получателей.

```
package main

import "fmt"

type Product struct {
    name, category string
    price float64
}

type Supplier struct {
    name, city string
}

func newProduct(name, category string, price float64)
*Product {
    return &Product{ name, category, price }
}

func (product *Product) printDetails() {
        fmt.Println("Name:", product.name, "Category:",
product.category,
        "Price", product.calcTax(0.2, 100))
}

func (product *Product) calcTax(rate, threshold float64)
float64 {
    if (product.price > threshold) {
        return product.price + (product.price * rate)
    }
    return product.price;
}

func (supplier *Supplier) printDetails() {
        fmt.Println("Supplier:", supplier.name, "City:",
supplier.city)
}

func main() {
```

```
    products := []*Product {
        newProduct("Kayak", "Watersports", 275),
        newProduct("Lifejacket", "Watersports", 48.95),
        newProduct("Soccer Ball", "Soccer", 19.50),
    }

    for _, p := range products {
        p.printDetails()
    }

    suppliers := []*Supplier {
        { "Acme Co", "New York City"},
        { "BoatCo", "Chicago"},
    }
    for _,s := range suppliers {
        s.printDetails()
    }
}
```

Листинг 11-7 Методы с одинаковыми именами в файле main.go в папке methodAndInterfaces

Существуют методы `printDetails` как для типов `*Product`, так и для `*Supplier`, что разрешено, поскольку каждый из них представляет уникальную комбинацию имени и типа получателя. Код в листинге `11-7` выводит следующий результат:

```
Name: Kayak Category: Watersports Price 330
Name: Lifejacket Category: Watersports Price 48.95
Name: Soccer Ball Category: Soccer Price 19.5
Supplier: Acme Co City: New York City
Supplier: BoatCo City: Chicago
```

Компилятор сообщит об ошибке, если я попытаюсь определить метод, который дублирует существующую комбинацию имя/получатель, независимо от того, отличаются ли остальные параметры метода, как показано в листинге 11-8.

```
package main

import "fmt"

type Product struct {
    name, category string
```

```
    price float64
}

type Supplier struct {
    name, city string
}

// ...other methods omitted for brevity...

func (supplier *Supplier) printDetails() {
          fmt.Println("Supplier:",  supplier.name,  "City:",
supplier.city)
}

func (supplier *Supplier) printDetails(showName bool) {
    if (showName) {
            fmt.Println("Supplier:", supplier.name, "City:",
supplier.city)
    } else {
        fmt.Println("Supplier:", supplier.name)
    }
}

func main() {

    products := []*Product {
        newProduct("Kayak", "Watersports", 275),
        newProduct("Lifejacket", "Watersports", 48.95),
        newProduct("Soccer Ball", "Soccer", 19.50),
    }

    for _, p := range products {
        p.printDetails()
    }

    suppliers := []*Supplier {
        { "Acme Co", "New York City"},
        { "BoatCo", "Chicago"},
    }
    for _,s := range suppliers {
        s.printDetails()
    }
}
```

Листинг 11-8 Определение другого метода в файле main.go в папке методовAndInterfaces

Новый метод выдает следующую ошибку компилятора:

```
# command-line-arguments
.\main.go:34:6: method redeclared: Supplier.printDetails
        method(*Supplier) func()
        method(*Supplier) func(bool)
.\main.go:34:27: (*Supplier).printDetails redeclared in this
block
        previous declaration at .\main.go:30:6
```

Понимание получателей указателей и значений

Метод, получатель которого является типом указателя, также может быть вызван через обычное значение базового типа, а это означает, что метод типа `*Product`, например, может использоваться со значением `Product`, как показано в листинге 11-9.

```
package main

import "fmt"

type Product struct {
    name, category string
    price float64
}

// type Supplier struct {
//     name, city string
// }

// func newProduct(name, category string, price float64)
*Product {
//     return &Product{ name, category, price }
// }

func (product *Product) printDetails() {
            fmt.Println("Name:", product.name, "Category:",
product.category,
        "Price", product.calcTax(0.2, 100))
}
```

```
func  (product  *Product)  calcTax(rate,  threshold  float64)
float64 {
    if (product.price > threshold) {
        return product.price + (product.price * rate)
    }
    return product.price;
}

// func (supplier *Supplier) printDetails() {
//          fmt.Println("Supplier:",  supplier.name,  "City:",
supplier.city)
// }

func main() {
    kayak := Product { "Kayak", "Watersports", 275 }
    kayak.printDetails()
}
```

Листинг 11-9 Вызов метода в файле main.go в папке methodAndInterfaces

Переменной kayak присваивается значение Product, но она используется с методом printDetails, получателем которого является *Product. Go позаботится о несоответствии и без проблем вызовет метод. Верен и противоположный процесс: метод, который получает значение, может быть вызван с помощью указателя, как показано в листинге 11-10.

```
package main

import "fmt"

type Product struct {
    name, category string
    price float64
}

func (product Product) printDetails() {
           fmt.Println("Name:",   product.name,   "Category:",
product.category,
        "Price",  product.calcTax(0.2, 100))
}
```

```
func (product *Product) calcTax(rate, threshold float64)
float64 {
    if (product.price > threshold) {
        return product.price + (product.price * rate)
    }
    return product.price;
}

func main() {
    kayak := &Product { "Kayak", "Watersports", 275 }
    kayak.printDetails()
}
```

Листинг 11-10 Вызов метода в файле main.go в папке methodAndInterfaces

Эта функция означает, что вы можете писать методы в зависимости от того, как вы хотите, чтобы они вели себя, используя указатели, чтобы избежать копирования значений или позволить получателю быть измененным методом.

> **Примечание**
> Одним из результатов этой функции является то, что типы значений и указателей считаются одинаковыми, когда речь идет о перегрузке методов, а это означает, что метод с именем `printDetails`, тип получателя которого — `Product`, будет конфликтовать с методом `printDetails`, тип получателя которого — `*Product`.

Оба листинга 11-9 и 11-10 выдают следующий результат:

```
Name: Kayak Category: Watersports Price 330
```

ВЫЗОВ МЕТОДОВ ЧЕРЕЗ ТИП ПОЛУЧАТЕЛЯ

Необычным аспектом методов Go является то, что они могут вызываться с использованием типа получателя, поэтому метод с такой сигнатурой:

```
...
func (product Product) printDetails() {
...
```

можно вызывать так:

```
...
Product.printDetails(Product{ "Kayak", "Watersports", 275
})
...
```

За именем типа получателя метода, `Product`, в данном случае следует точка и имя метода. Аргумент — это значение `Product`, которое будет использоваться в качестве значения получателя. Функция автоматического сопоставления указателя и значения, показанная в листингах 11-9 и 11-10, не применяется при вызове метода через его тип получателя, что означает, что метод с сигнатурой указателя, например:

```
...
func (product *Product) printDetails() {
...
```

должен вызываться через тип указателя и передавать аргумент указателя, например:

```
...
(*Product).printDetails(&Product{ "Kayak", "Watersports",
275 })
...
```

Не путайте эту функцию со статическими методами, предоставляемыми такими языками, как C# или Java. В Go нет статических методов, и вызов метода через его тип имеет тот же эффект, что и вызов метода через значение или указатель.

Определение методов для псевдонимов типов

Методы могут быть определены для любого типа, определенного в текущем пакете. Я объясню, как добавлять пакеты в проект в главе 12, но для этой главы есть один файл кода, содержащий один пакет, а это означает, что методы могут быть определены только для типов, определенных в файле `main.go`.

Но это не ограничивает методы только структурами, потому что ключевое слово type может использоваться для создания псевдонимов для любого типа, а методы могут быть определены для псевдонима. (Я ввел ключевое слово type в главе 9, чтобы упростить работу с типами функций.) В листинге 11-11 создается псевдоним и метод.

```
package main

import "fmt"

type Product struct {
    name, category string
    price float64
}

type ProductList []Product

func     (products     *ProductList)     calcCategoryTotals()
map[string]float64 {
    totals := make(map[string]float64)
    for _, p := range *products {
        totals[p.category] = totals[p.category] + p.price
    }
    return totals
}

func main() {

    products := ProductList {
        { "Kayak", "Watersports", 275 },
        { "Lifejacket", "Watersports", 48.95 },
        {"Soccer Ball", "Soccer", 19.50 },
    }

                   for     category,     total     :=     range
products.calcCategoryTotals() {
        fmt.Println("Category: ", category, "Total:", total)
    }
}
```

Листинг 11-11 Определение метода для псевдонима типа в файле main.go в папке methodAndInterfaces

Ключевое слово type используется для создания псевдонима для типа []Product с именем ProductList. Этот тип можно использовать для определения методов либо непосредственно для приемников типа значения, либо с помощью указателя, как в этом примере.

Вы не всегда сможете получить данные с типом, необходимым для вызова метода, определенного для псевдонима, например, при обработке результатов функции. В этих ситуациях вы можете выполнить преобразование типа, как показано в листинге 11-12.

```
package main

import "fmt"

type Product struct {
    name, category string
    price float64
}

type ProductList []Product

func (products *ProductList) calcCategoryTotals()
map[string]float64 {
    totals := make(map[string]float64)
    for _, p := range *products {
        totals[p.category] = totals[p.category] + p.price
    }
    return totals
}

func getProducts() []Product {
    return []Product {
        { "Kayak", "Watersports", 275 },
        { "Lifejacket", "Watersports", 48.95 },
        {"Soccer Ball", "Soccer", 19.50 },
    }
}

func main() {

    products := ProductList(getProducts())
```

```
                  for     category,     total     :=     range
products.calcCategoryTotals() {
        fmt.Println("Category: ", category, "Total:", total)
    }
}
```

Листинг 11-12 Выполнение преобразования типов в файле main.go в папке methodAndInterfaces

Результатом функции `getProducts` является `[]Product`, который преобразуется в `ProductList` с явным преобразованием, позволяющим использовать метод, определенный для псевдонима. Код в листингах 11-11 и 11-11 выдает следующий результат:

```
Category:  Watersports Total: 323.95
Category:  Soccer Total: 19.5
```

Размещение типов и методов в отдельных файлах

По мере усложнения проекта количество кода, необходимого для определения пользовательских типов и их методов, быстро становится слишком большим для управления в одном файле кода. Проекты Go могут быть структурированы в несколько файлов, которые компилятор объединяет при сборке проекта.

Примеры в следующем разделе слишком длинные, чтобы их можно было выразить в одном листинге кода, не заполняя оставшуюся часть главы длинными разделами кода, которые не меняются, поэтому я собираюсь представить несколько файлов кода.

Эта функция является частью поддержки Go для *пакетов*, которая предоставляет различные способы структурирования файлов кода в проекте и которые я описываю в главе 12. В этой главе я собираюсь использовать самый простой аспект пакетов, который заключается в использовании несколько файлов кода в папке проекта.

Добавьте файл с именем `product.go` в папку `methodAndInterfaces` с содержимым, показанным в листинге 11-13.

```
package main

type Product struct {
```

```
    name, category string
    price float64
}
```

Листинг 11-13 Содержимое файла product.go в папке methodAndInterfaces

Добавьте файл с именем `service.go` в папку `methodAndInterfaces` и используйте его для определения типа, показанного в листинге 11-14.

```
package main

type Service struct {
    description string
    durationMonths int
    monthlyFee float64
}
```

Листинг 11-14 Содержимое файла service.go в папке методовAndInterfaces

Наконец, замените содержимое файла `main.go` тем, что показано в листинге 11-15.

```
package main

import "fmt"

func main() {

    kayak := Product { "Kayak", "Watersports", 275 }
    insurance := Service {"Boat Cover", 12, 89.50 }

            fmt.Println("Product:",   kayak.name,   "Price:",
kayak.price)
    fmt.Println("Service:", insurance.description, "Price:",
     insurance.monthlyFee * float64(insurance.durationMonths))
}
```

Листинг 11-15 Замена содержимого файла main.go в папке methodAndInterfaces

Этот код создает значения, используя типы структур, определенные в других файлах. Скомпилируйте и выполните проект, который выдаст следующий результат:

```
Product: Kayak Price: 275
```

```
Service: Boat Cover Price: 1074
```

Определение и использование интерфейсов

Легко представить сценарий, в котором типы `Product` и `Service`, определенные в предыдущем разделе, используются вместе. Например, в пакете личных счетов может потребоваться предоставить пользователю список расходов, некоторые из которых представлены значениями `Product`, а другие — значениями `Service`. Несмотря на то, что эти типы имеют общее назначение, правила типов Go запрещают их совместное использование, например создание среза, содержащего оба типа значений.

Определение интерфейса

Эта проблема решается с помощью *интерфейсов*, которые описывают набор методов без указания реализации этих методов. Если тип реализует все методы, определенные интерфейсом, то значение этого типа можно использовать везде, где разрешен интерфейс. Первым шагом является определение интерфейса, как показано в листинге 11-16.

```
package main

import "fmt"

type Expense interface {
    getName() string
    getCost(annual bool) float64
}

func main() {

    kayak := Product { "Kayak", "Watersports", 275 }
    insurance := Service {"Boat Cover", 12, 89.50 }

            fmt.Println("Product:",   kayak.name,   "Price:",
kayak.price)
    fmt.Println("Service:", insurance.description, "Price:",
                                insurance.monthlyFee     *
float64(insurance.durationMonths))
```

```
}
```

Листинг 11-16 Определение интерфейса в файле main.go в папке methodAndInterfaces

Интерфейс определяется с помощью ключевого слова `type`, имени, ключевого слова `interface` и тела, состоящего из сигнатур методов, заключенных в фигурные скобки, как показано на рисунке 11-3.

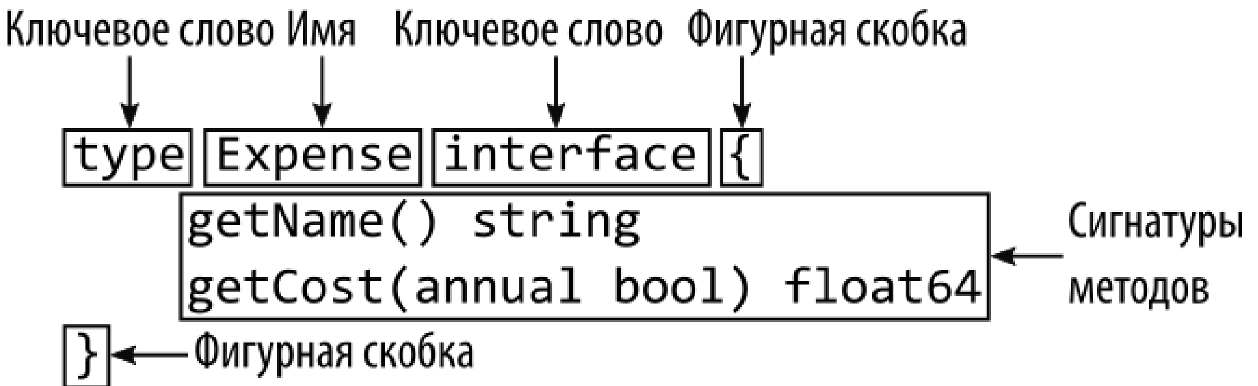

Рисунок 11-3 Определение интерфейса

Этому интерфейсу было присвоено имя `Expense`, а тело интерфейса содержит единственную сигнатуру метода. Сигнатуры методов состоят из имени, параметров и типов результатов, как показано на рисунке 11-4.

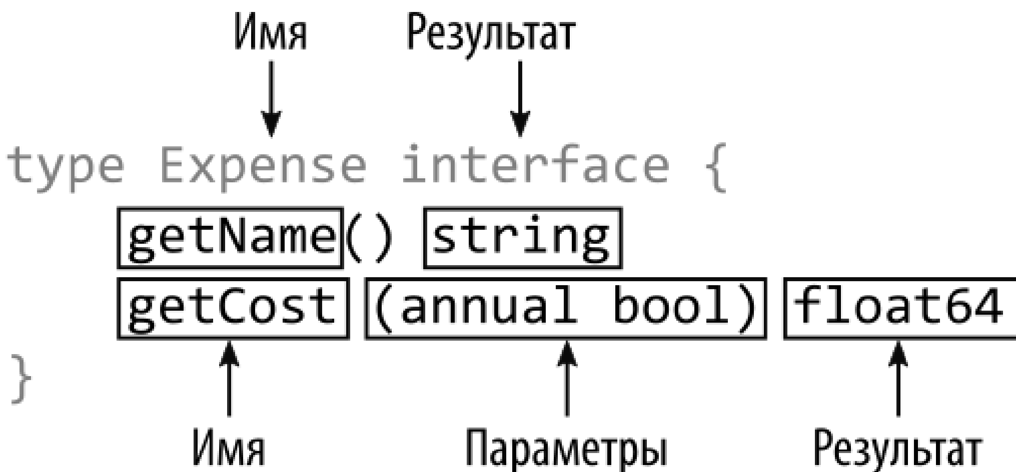

Рисунок 11-4 Сигнатура метода

Интерфейс `Expense` описывает два метода. Первый метод — это `getName`, который не принимает аргументов и возвращает строку.

Второй метод называется getCost, он принимает логический аргумент и возвращает результат типа float64.

Реализация интерфейса

Чтобы реализовать интерфейс, все методы, указанные интерфейсом, должны быть определены для типа структуры, как показано в листинге 11-17.

```
package main

type Product struct {
    name, category string
    price float64
}

func (p Product) getName() string {
    return p.name
}

func (p Product) getCost(_ bool) float64 {
    return p.price
}
```

Листинг 11-17 Реализация интерфейса в файле product.go в папке methodAndInterfaces

В большинстве языков требуется использование ключевого слова, чтобы указать, когда тип реализует интерфейс, но Go просто требует, чтобы все методы, указанные интерфейсом, были определены. Go позволяет использовать разные имена параметров и результатов, но методы должны иметь одинаковые имена, типы параметров и типы результатов. В листинге 11-18 определены методы, необходимые для реализации интерфейса для типа Service.

```
package main

type Service struct {
    description string
    durationMonths int
    monthlyFee float64
}

func (s Service) getName() string {
```

```
    return s.description
}

func (s Service) getCost(recur bool) float64 {
    if (recur) {
        return s.monthlyFee * float64(s.durationMonths)
    }
    return s.monthlyFee
}
```

Листинг 11-18 Реализация интерфейса в файле service.go в папке methodAndInterfaces

Интерфейсы описывают только методы, а не поля. По этой причине в интерфейсах часто указываются методы, которые возвращают значения, хранящиеся в полях структуры, например метод `getName` в листингах 11-17 и 11-18.

Использование интерфейса

После того как вы реализовали интерфейс, вы можете ссылаться на значения через тип интерфейса, как показано в листинге 11-19.

```
package main

import "fmt"

type Expense interface {
    getName() string
    getCost(annual bool) float64
}

func main() {

    expenses := []Expense {
        Product { "Kayak", "Watersports", 275 },
        Service {"Boat Cover", 12, 89.50 },
     }

    for _, expense := range expenses {
          fmt.Println("Expense:", expense.getName(), "Cost:",
expense.getCost(true))
    }
}
```

Листинг 11-19 Использование интерфейса в файле main.go в папке methodAndInterfaces

В этом примере я определил срез `Expense` и заполнил его значениями `Product` и `Service`, созданными с использованием литерального синтаксиса. Срез используется в цикле `for`, который вызывает методы `getName` и `getCost` для каждого значения.

Переменные, тип которых является интерфейсом, имеют два типа: *статический* тип и *динамический* тип. Статический тип — это интерфейсный тип. Динамический тип — это тип значения, присвоенного переменной, которая реализует интерфейс, например `Product` или `Service` в данном случае. Статический тип никогда не меняется — например, статический тип переменной `Expense` — это всегда `Expense`, — но динамический тип может измениться путем присвоения нового значения другого типа, реализующего интерфейс.

Цикл `for` имеет дело только со статическим типом — `Expense` — и не знает (и не должен знать) динамический тип этих значений. Использование интерфейса позволило мне сгруппировать разрозненные динамические типы вместе и использовать общие методы, указанные для статического типа интерфейса. Скомпилируйте и выполните проект; вы получите следующий вывод:

```
Expense: Kayak Cost: 275
Expense: Boat Cover Cost: 1074
```

Использование интерфейса в функции

Типы интерфейсов могут использоваться для переменных, параметров функций и результатов функций, как показано в листинге 11-20.

> **Примечание**
> Методы не могут быть определены с использованием интерфейсов в качестве приемников. С интерфейсом связаны только те методы, которые он указывает.

```
package main

import "fmt"

type Expense interface {
```

```
    getName() string
    getCost(annual bool) float64
}

func calcTotal(expenses []Expense) (total float64) {
    for _, item := range expenses {
        total += item.getCost(true)
    }
    return
}

func main() {

    expenses := []Expense {
        Product { "Kayak", "Watersports", 275 },
        Service {"Boat Cover", 12, 89.50 },
     }

    for _, expense := range expenses {
          fmt.Println("Expense:", expense.getName(), "Cost:",
expense.getCost(true))
    }
    fmt.Println("Total:", calcTotal(expenses))
}
```

Листинг 11-20 Использование интерфейса в файле main.go в папке methodAndInterfaces

Функция `calcTotal` получает срез, содержащий значения `Expense`, которые обрабатываются с помощью цикла `for` для получения итогового значения `float64`. Скомпилируйте и выполните проект, который выдаст следующий результат:

```
Expense: Kayak Cost: 275
Expense: Boat Cover Cost: 1074
Total: 1349
```

Использование интерфейса для полей структуры

Типы интерфейса могут использоваться для полей структуры, что означает, что полям могут быть присвоены значения любого типа, реализующего методы, определенные интерфейсом, как показано в листинге 11-21.

```
package main

import "fmt"

type Expense interface {
    getName() string
    getCost(annual bool) float64
}

func calcTotal(expenses []Expense) (total float64) {
    for _, item := range expenses {
        total += item.getCost(true)
    }
    return
}

type Account struct {
    accountNumber int
    expenses []Expense
}

func main() {

    account := Account {
        accountNumber: 12345,
        expenses: []Expense {
            Product { "Kayak", "Watersports", 275 },
            Service {"Boat Cover", 12, 89.50 },
         },
    }

    for _, expense := range account.expenses {
          fmt.Println("Expense:", expense.getName(), "Cost:",
expense.getCost(true))
    }
    fmt.Println("Total:", calcTotal(account.expenses))
}
```

Листинг 11-21 Использование интерфейса в поле структуры в файле main.go в папке methodAndInterfaces

Структура `Account` имеет поле расходов, тип которого представляет собой срез значений `Expense`, который можно использовать так же, как

и любое другое поле. Скомпилируйте и выполните проект, который выдаст следующий результат:

```
Expense: Kayak Cost: 275
Expense: Boat Cover Cost: 1074
Total: 1349
```

Понимание эффекта приемников метода указателя

Методы, определенные типами Product и Service, имеют приемники значений, что означает, что методы будут вызываться с копиями значения Product или Service. Это может сбивать с толку, поэтому в листинге 11-22 приведен простой пример.

```
package main

import "fmt"

type Expense interface {
    getName() string
    getCost(annual bool) float64
}

func main() {

    product := Product { "Kayak", "Watersports", 275 }

    var expense Expense = product

    product.price = 100

    fmt.Println("Product field value:", product.price)
                fmt.Println("Expense method result:",
expense.getCost(false))
}
```

Листинг 11-22 Использование значения в файле main.go в папке методовAndInterfaces

В этом примере создается значение структуры Product, оно присваивается переменной Expense, изменяется значение поля price значения структуры и выводится значение поля напрямую и через метод интерфейса. Скомпилируйте и выполните код; вы получите следующий результат:

```
Product field value: 100
Expense method result: 275
```

Значение Product было скопировано, когда оно было присвоено переменной Expense, что означает, что изменение поля price не влияет на результат метода getCost.

Указатель на значение структуры можно использовать при назначении переменной интерфейса, как показано в листинге 11-23.

```
package main

import "fmt"

type Expense interface {
    getName() string
    getCost(annual bool) float64
}

func main() {

    product := Product { "Kayak", "Watersports", 275 }

    var expense Expense = &product

    product.price = 100

    fmt.Println("Product field value:", product.price)
                  fmt.Println("Expense    method    result:",
expense.getCost(false))
}
```

Листинг 11-23 Использование указателя в файле main.go в папке методовAndInterfaces

Использование указателя означает, что ссылка на значение Product присваивается переменной Expense, но это не меняет тип переменной интерфейса, который по-прежнему является Expense. Скомпилируйте и выполните проект, и вы увидите эффект ссылки в выводе, который показывает, что изменение в поле price отражается в результате метода getCost:

```
Product field value: 100
Expense method result: 100
```

Это полезно, потому что это означает, что вы можете выбрать, как будет использоваться значение, присвоенное переменной интерфейса. Но это также может противоречить здравому смыслу, потому что переменная всегда имеет тип Expense, независимо от того, присвоено ли ей значение Product или *Product.

Вы можете принудительно использовать ссылки, указав получатели указателей при реализации методов интерфейса, как показано в листинге 11-24.

```
package main

type Product struct {
    name, category string
    price float64
}

func (p *Product) getName() string {
    return p.name
}

func (p *Product) getCost(_ bool) float64 {
    return p.price
}
```

Листинг 11-24 Использование приемников указателей в файле product.go в папке methodAndInterfaces

Это небольшое изменение, но оно означает, что тип Product больше не реализует интерфейс Expense, поскольку необходимые методы больше не определены. Вместо этого интерфейс реализуется типом *Product, что означает, что указатели на значения Product можно рассматривать как значения Expense, но не как обычные значения. Скомпилируйте и выполните проект, и вы получите тот же результат, что и в листинге 11-23:

```
Product field value: 100
Expense method result: 100
```

В листинге 11-25 значение Product присваивается переменной Expense.

```
package main

import "fmt"

type Expense interface {
    getName() string
    getCost(annual bool) float64
}

func main() {

    product := Product { "Kayak", "Watersports", 275 }

    var expense Expense = product

    product.price = 100

    fmt.Println("Product field value:", product.price)
              fmt.Println("Expense    method    result:",
expense.getCost(false))
}
```

Листинг 11-25 Присвоение значения в файле main.go в папке methodAndInterfaces

Скомпилируйте проект; вы получите следующую ошибку, сообщающую вам, что требуется получение указателя:

```
.\main.go:14:9: cannot use product (type Product) as type
Expense in assignment:
          Product does not implement Expense (getCost method
has pointer receiver)
```

Сравнение значений интерфейса

Значения интерфейса можно сравнивать с помощью операторов сравнения Go, как показано в листинге 11-26. Два значения интерфейса равны, если они имеют одинаковый динамический тип и все их поля равны.

```
package main

import "fmt"
```

```
type Expense interface {
    getName() string
    getCost(annual bool) float64
}

func main() {

    var e1 Expense = &Product { name: "Kayak" }
    var e2 Expense = &Product { name: "Kayak" }
    var e3 Expense = Service { description: "Boat Cover" }
    var e4 Expense = Service { description: "Boat Cover" }

    fmt.Println("e1 == e2", e1 == e2)
    fmt.Println("e3 == e4", e3 == e4)
}
```

Листинг 11-26 Сравнение значений интерфейса в файле main.go в папке methodAndInterfaces

Следует соблюдать осторожность при сравнении значений интерфейса, и неизбежно требуются некоторые знания о динамических типах.

Первые два значения `Expense` не равны. Это связано с тем, что динамический тип этих значений является типом указателя, а указатели равны, только если они указывают на одно и то же место в памяти. Вторые два значения `Expense` равны, потому что это простые значения структуры с одинаковыми значениями поля. Скомпилируйте и выполните проект, чтобы подтвердить равенство этих значений:

```
e1 == e2 false
e3 == e4 true
```

Проверки на равенство интерфейсов также могут вызывать ошибки времени выполнения, если динамический тип не сопоставим. Листинг 11-27 добавляет поле в структуру `Service`.

```
package main

type Service struct {
    description string
    durationMonths int
    monthlyFee float64
    features []string
```

```
}

func (s Service) getName() string {
    return s.description
}

func (s Service) getCost(recur bool) float64 {
    if (recur) {
        return s.monthlyFee * float64(s.durationMonths)
    }
    return s.monthlyFee
}
```

Листинг 11-27 Добавление поля в файл service.go в папку methodAndServices

Как объяснялось в главе 7, срезы несопоставимы. Скомпилируйте и выполните проект, и вы увидите эффект нового поля:

```
panic: runtime error: comparing uncomparable type
main.Service
goroutine 1 [running]:
main.main()
        C:/main.go:20 +0x1c5
exit status 2
```

Выполнение утверждений типа

Интерфейсы могут быть полезны, но они могут создавать проблемы, и часто полезно иметь возможность прямого доступа к динамическому типу, что известно как *сужение типа*, процесс перехода от менее точного типа к более точному типу.

Утверждение типа используется для доступа к динамическому типу значения интерфейса, как показано в листинге 11-28.

```
package main

import "fmt"

type Expense interface {
    getName() string
    getCost(annual bool) float64
}
```

```
func main() {

    expenses := []Expense {
        Service {"Boat Cover", 12, 89.50, []string{} },
        Service {"Paddle Protect", 12, 8, []string{} },
    }

    for _, expense := range expenses {
        s := expense.(Service)
        fmt.Println("Service:", s.description, "Price:",
            s.monthlyFee * float64(s.durationMonths))
    }
}
```

Листинг 11-28 Использование утверждения типа в файле main.go в папке methodAndInterfaces

Утверждение типа выполняется путем применения точки после значения, за которым следует целевой тип в круглых скобках, как показано на рисунке 11-5.

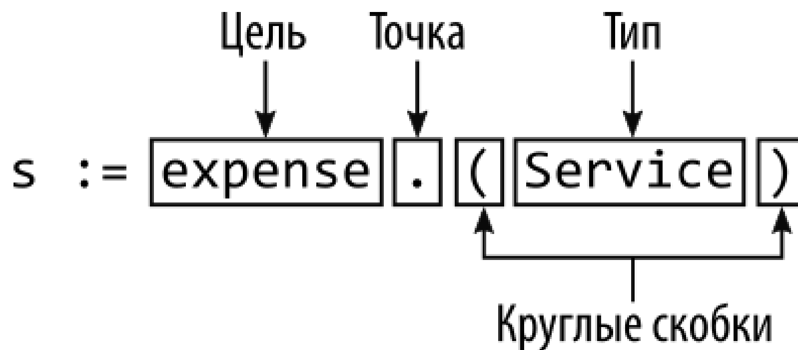

Рисунок 11-5 Утверждение типа

В листинге 11-28 я использовал утверждение типа для доступа к динамическому значению `Service` из среза типов интерфейса `Expense`. Когда у меня есть значение службы для работы, я могу использовать все поля и методы, определенные для типа `Service`, а не только методы, определенные интерфейсом `Expense`.

УТВЕРЖДЕНИЕ ТИПА ПРОТИВ ПРЕОБРАЗОВАНИЯ ТИПА

Не путайте утверждения типа, как показано на рисунке 11-6, с синтаксисом преобразования типов, описанным в главе 5. Утверждения типа можно применять только к интерфейсам, и они

используются для того, чтобы сообщить компилятору, что значение интерфейса имеет определенный динамический тип. Преобразования типов могут применяться только к определенным типам, а не к интерфейсам, и только в том случае, если структура этих типов совместима, например, преобразование между типами структур с одинаковыми полями.

Код в листинге 11-28 выдает следующий результат при компиляции и выполнении:

```
Service: Boat Cover Price: 1074
Service: Paddle Protect Price: 96
```

Тестирование перед выполнением утверждения типа

Когда используется утверждение типа, компилятор полагает, что у программиста больше знаний и знаний о динамических типах в коде, чем он может сделать вывод, например, что срез `Expense` содержит только значения `Supplier`. Чтобы увидеть, что происходит, когда это не так, в листинге 11-29 к срезу `Expense` добавляется значение `*Product`.

```
package main

import "fmt"

type Expense interface {
    getName() string
    getCost(annual bool) float64
}

func main() {

    expenses := []Expense {
        Service {"Boat Cover", 12, 89.50, []string{} },
        Service {"Paddle Protect", 12, 8, []string{} },
        &Product { "Kayak", "Watersports", 275 },
    }

    for _, expense := range expenses {
        s := expense.(Service)
        fmt.Println("Service:", s.description, "Price:",
```

```
            s.monthlyFee * float64(s.durationMonths))
    }
}
```

Листинг 11-29 Смешивание динамических типов в файле main.go в папке methodAndInterfaces

Скомпилируйте и выполните проект; вы увидите следующую ошибку при выполнении кода:

```
panic: interface conversion: main.Expense is *main.Product,
not main.Service
```

Среда выполнения Go попыталась выполнить утверждение и потерпела неудачу. Чтобы избежать этой проблемы, существует специальная форма утверждения типа, которая указывает, может ли быть выполнено утверждение, как показано в листинге 11-30.

```
package main

import "fmt"

type Expense interface {
    getName() string
    getCost(annual bool) float64
}

func main() {

    expenses := []Expense {
        Service {"Boat Cover", 12, 89.50, []string{} },
        Service {"Paddle Protect", 12, 8, []string{} },
        &Product { "Kayak", "Watersports", 275 },
    }

    for _, expense := range expenses {
        if s, ok := expense.(Service); ok {
            fmt.Println("Service:", s.description, "Price:",
                s.monthlyFee * float64(s.durationMonths))
        } else {
            fmt.Println("Expense:", expense.getName(),
                "Cost:", expense.getCost(true))
        }
```

```
    }
}
```

Листинг 11-30 Тестирование утверждения в файле main.go в папке methodAndInterfaces

Утверждения типа могут давать два результата, как показано на рисунке 11-6. Первому результату присваивается динамический тип, а второму результату присваивается `bool` значение, указывающее, можно ли выполнить утверждение.

Результат Результат Утверждение типа

```
if s , ok := expense.(Service); ok {
```

Рисунок 11-6 Два результата утверждения типа

Значение `bool` можно использовать с оператором `if` для выполнения операторов для определенного динамического типа. Скомпилируйте и выполните проект; вы увидите следующий вывод:

```
Service: Boat Cover Price: 1074
Service: Paddle Protect Price: 96
Expense: Kayak Cost: 275
```

Включение динамических типов

Операторы Go `switch` могут использоваться для доступа к динамическим типам, как показано в листинге 11-31, что может быть более кратким способом выполнения утверждений типа с операторами `if`.

```
package main

import "fmt"

type Expense interface {
    getName() string
    getCost(annual bool) float64
}

func main() {
```

```
    expenses := []Expense {
        Service {"Boat Cover", 12, 89.50, []string{} },
        Service {"Paddle Protect", 12, 8, []string{} },
        &Product { "Kayak", "Watersports", 275 },
    }

    for _, expense := range expenses {
        switch value := expense.(type) {
            case Service:
                fmt.Println("Service:", value.description,
"Price:",
                                        value.monthlyFee  *
float64(value.durationMonths))
            case *Product:
                fmt.Println("Product:", value.name, "Price:",
value.price)
            default:
                fmt.Println("Expense:", expense.getName(),
                    "Cost:", expense.getCost(true))
        }
    }
}
```

Листинг 11-31 Включение типов в файле main.go в папке methodAndInterfaces

Оператор `switch` использует специальное утверждение типа, в котором используется ключевое слово `type`, как показано на рисунке 11-7.

Рисунок 11-7 Переключатель типа

Каждый оператор `case` указывает тип и блок кода, который будет выполняться, когда значение, оцениваемое оператором `switch`, имеет указанный тип. Компилятор Go достаточно умен, чтобы понять взаимосвязь между значениями, оцениваемыми оператором `switch`, и не будет разрешать операторы `case` для типов, которые не совпадают. Например, компилятор будет жаловаться, если имеется оператор `case`

для типа Product, потому что оператор `switch` оценивает значения `Expense`, а тип Product не имеет методов, необходимых для реализации интерфейса (поскольку методы в файле `product.go` использовать приемники указателей, показанные в листинге 11-24).

В операторе `case` результат может рассматриваться как указанный тип, а это означает, что в операторе `case`, указывающем, например, тип `Supplier`, могут использоваться все поля и методы, определенные типом `Supplier`.

Оператор `default` можно использовать для указания блока кода, который будет выполняться, когда ни один из операторов `case` не совпадает. Скомпилируйте и запустите проект, и вы увидите следующий вывод:

```
Service: Boat Cover Price: 1074
Service: Paddle Protect Price: 96
Product: Kayak Price: 275
```

Использование пустого интерфейса

Go позволяет пользователю пустого интерфейса — то есть интерфейса, не определяющего методы — представлять любой тип, что может быть полезным способом группировки разрозненных типов, не имеющих общих характеристик, как показано в листинге 11-32.

```
package main

import "fmt"

type Expense interface {
    getName() string
    getCost(annual bool) float64
}

type Person struct {
    name, city string
}

func main() {
```

```
    var expense Expense = &Product { "Kayak", "Watersports",
275 }

    data := []interface{} {
        expense,
        Product { "Lifejacket", "Watersports", 48.95 },
        Service {"Boat Cover", 12, 89.50, []string{} },
        Person { "Alice", "London"},
        &Person { "Bob", "New York"},
        "This is a string",
        100,
        true,
    }

    for _, item := range data {
        switch value := item.(type) {
            case Product:
                fmt.Println("Product:", value.name, "Price:",
value.price)
            case *Product:
                 fmt.Println("Product Pointer:", value.name,
"Price:", value.price)
            case Service:
                 fmt.Println("Service:", value.description,
"Price:",
                                       value.monthlyFee  *
float64(value.durationMonths))
            case Person:
                 fmt.Println("Person:", value.name, "City:",
value.city)
            case *Person:
                 fmt.Println("Person Pointer:", value.name,
"City:", value.city)
            case string, bool, int:
                fmt.Println("Built-in type:", value)
            default:
                fmt.Println("Default:", value)

        }
    }
}
```

Листинг 11-32 Использование пустого интерфейса в файле main.go в папке methodAndInterfaces

Пустой интерфейс используется в литеральном синтаксисе, определяемом ключевым словом `interface` и пустыми фигурными скобками, как показано на рисунке 11-8.

Ключевое слово Фигурные скобки

```
data := []interface{} {
```

Рисунок 11-8 Пустой интерфейс

Пустой интерфейс представляет все типы, включая встроенные типы и любые определенные структуры и интерфейсы. В листинге я определяю пустой срез массива со смесью значений `Product`, `*Product`, `Service`, `Person`, `*Person`, `string`, `int` и `bool`. Срез обрабатывается циклом `for` с операторами `switch`, которые сужают каждое значение до определенного типа. Скомпилируйте и выполните проект, который выдаст следующий результат:

```
Product Pointer: Kayak Price: 275
Product: Lifejacket Price: 48.95
Service: Boat Cover Price: 1074
Person: Alice City: London
Person Pointer: Bob City: New York
Built-in type: This is a string
Built-in type: 100
Built-in type: true
```

Использование пустого интерфейса для параметров функций

Пустой интерфейс можно использовать в качестве типа параметра функции, что позволяет вызывать функцию с любым значением, как показано в листинге 11-33.

```
package main

import "fmt"

type Expense interface {
    getName() string
```

```
    getCost(annual bool) float64
}

type Person struct {
    name, city string
}

func processItem(item interface{}) {
    switch value := item.(type) {
        case Product:
                fmt.Println("Product:", value.name, "Price:",
value.price)
        case *Product:
                 fmt.Println("Product Pointer:", value.name,
"Price:", value.price)
        case Service:
                  fmt.Println("Service:", value.description,
"Price:",
                                                value.monthlyFee  *
float64(value.durationMonths))
        case Person:
                 fmt.Println("Person:", value.name, "City:",
value.city)
        case *Person:
                  fmt.Println("Person Pointer:", value.name,
"City:", value.city)
        case string, bool, int:
            fmt.Println("Built-in type:", value)
        default:
            fmt.Println("Default:", value)
    }
}

func main() {

     var expense Expense = &Product { "Kayak", "Watersports",
275 }

    data := []interface{} {
        expense,
        Product { "Lifejacket", "Watersports", 48.95 },
        Service {"Boat Cover", 12, 89.50, []string{} },
        Person { "Alice", "London"},
```

```
        &Person { "Bob", "New York"},
        "This is a string",
        100,
        true,
    }

    for _, item := range data {
        processItem(item)
    }
}
```

Листинг 11-33 Использование пустого параметра интерфейса в файле main.go в папке methodAndInterfaces

Пустой интерфейс также можно использовать для переменных параметров, что позволяет вызывать функцию с любым количеством аргументов, каждый из которых может быть любого типа, как показано в листинге 11-34.

```
package main

import "fmt"

type Expense interface {
    getName() string
    getCost(annual bool) float64
}

type Person struct {
    name, city string
}

func processItems(items ...interface{}) {
    for _, item := range items {
        switch value := item.(type) {
            case Product:
                fmt.Println("Product:", value.name, "Price:",
value.price)
            case *Product:
                fmt.Println("Product Pointer:", value.name,
"Price:", value.price)
            case Service:
```

```
                    fmt.Println("Service:", value.description,
"Price:",
                                              value.monthlyFee  *
float64(value.durationMonths))
            case Person:
                   fmt.Println("Person:", value.name, "City:",
value.city)
            case *Person:
                    fmt.Println("Person Pointer:", value.name,
"City:", value.city)
            case string, bool, int:
                fmt.Println("Built-in type:", value)
            default:
                fmt.Println("Default:", value)
        }
    }
}

func main() {

     var expense Expense = &Product { "Kayak", "Watersports",
275 }

    data := []interface{} {
        expense,
        Product { "Lifejacket", "Watersports", 48.95 },
        Service {"Boat Cover", 12, 89.50, []string{} },
        Person { "Alice", "London"},
        &Person { "Bob", "New York"},
        "This is a string",
        100,
        true,
    }

    processItems(data...)
}
```

Листинг 11-34 Использование переменных параметров в файле main.go в папке methodAndInterfaces

Листинг 11-33 и Листинг 11-33 выдают следующий результат, когда проект компилируется и выполняется:

```
Product Pointer: Kayak Price: 275
```

```
Product: Lifejacket Price: 48.95
Service: Boat Cover Price: 1074
Person: Alice City: London
Person Pointer: Bob City: New York
Built-in type: This is a string
Built-in type: 100
Built-in type: true
```

Резюме

В этой главе я описал поддержку, которую Go предоставляет для методов, как с точки зрения их определения для типов структур, так и с точки зрения определения наборов интерфейсов методов. Я продемонстрировал, как структура может реализовать интерфейс, и это позволяет использовать смешанные типы вместе. В следующей главе я объясню, как Go поддерживает структуру в проектах с использованием пакетов и модулей.

12. Создание и использование пакетов

Пакеты — это функция Go, которая позволяет структурировать проекты таким образом, чтобы соответствующие функции можно было сгруппировать вместе без необходимости помещать весь код в один файл или папку. В этой главе я описываю, как создавать и использовать пакеты, а также как использовать пакеты, разработанные третьими сторонами. Таблица 12-1 помещает пакеты в контекст.

Таблица 12-1 Помещение пакетов в контекст

Вопрос	**Ответ**
Кто они такие?	Пакеты позволяют структурировать проекты, чтобы связанные функции можно было разрабатывать вместе.
Почему они полезны?	Пакеты — это то, как Go реализует управление доступом, чтобы реализация функции могла быть скрыта от кода, который ее использует.
Как они используются?	Пакеты определяются путем создания файлов кода в папках и использования ключевого слова `package` для обозначения того, к какому пакету они принадлежат.
Есть ли подводные камни или ограничения?	Значимых имен не так уж много, и часто возникают конфликты между именами пакетов, требующие использования псевдонимов, чтобы избежать ошибок.
Есть ли альтернативы?	Простые приложения могут быть написаны без использования пакетов.

Таблица 12-2 суммирует главу.

Таблица 12-2 Краткое содержание главы

Проблема	**Решение**	**Листинг**
Определить пакет	Создайте папку и добавьте файлы кода с операторами `package`.	4, 9, 10, 15, 16
Использовать пакет	Добавьте оператор `import`, указывающий путь к пакету и включающему его модулю.	5
Управление доступом к функциям в пакете	Экспортируйте объекты, используя начальную заглавную букву в их именах. Начальные буквы нижнего регистра являются неожиданными и не могут использоваться вне пакета.	6–8

Проблема	Решение	Листинг
Устранение конфликтов пакетов	Используйте псевдоним или точечный импорт.	11–14
Выполнение задач при загрузке пакета	Определите функцию инициализации.	17, 18
Выполнение функции инициализации пакета без импорта содержащихся в нем функций	Используйте пустой идентификатор в операторе `import`.	19, 20
Использовать внешний пакет	Используйте команду `go get`.	21, 22
Удалить неиспользуемые зависимости пакетов	Используйте команду `go mod tidy`.	23

Подготовка к этой главе

Чтобы подготовиться к этой главе, откройте новую командную строку, перейдите в удобное место и создайте каталог с именем `packages`. Перейдите в папку `packages` и выполните команду, показанную в листинге 12-1, чтобы инициализировать проект.

> **Подсказка**
> Вы можете загрузить пример проекта для этой главы — и для всех остальных глав этой книги — с `https://github.com/apress/pro-go`. См. Главу 2 о том, как получить помощь, если у вас возникнут проблемы с запуском примеров.

```
go mod init packages
```

Листинг 12-1 Инициализация проекта

Добавьте файл с именем `main.go` в папку `packages` с содержимым, показанным в листинге 12-2.

```
package main

import "fmt"

func main() {
    fmt.Println("Hello, Packages and Modules")
```

```
}
```

Листинг 12-2 Содержимое файла main.go в папке packages

Используйте командную строку для запуска команды, показанной в листинге 12-3, в папке packages.

```
go run .
```

Листинг 12-3 Запуск примера проекта

Код в файле main.go будет скомпилирован и выполнен, что приведет к следующему результату:

```
Hello, Packages and Modules
```

Понимание файла модуля

Первым шагом для всех примеров проектов в этой книге было создание файла модуля, что было сделано с помощью команды в листинге 12-1.

Первоначальная цель файла модуля заключалась в том, чтобы разрешить публикацию кода, чтобы его можно было использовать в других проектах и, возможно, другими разработчиками. Файлы модулей все еще используются для этой цели, но Go начал получать широкое развитие, и, как это произошло, процент опубликованных проектов упал. В наши дни наиболее распространенной причиной создания файла модуля является то, что он упрощает установку опубликованных пакетов и имеет дополнительный эффект, позволяя использовать команду, показанную в листинге 12-3, вместо того, чтобы предоставлять Go build tools со списком отдельных файлов для компиляции.

Команда в листинге 12-1 создала файл с именем go.mod в папке packages со следующим содержимым:

```
module packages
go 1.17
```

Оператор module указывает имя модуля, которое было указано командой в листинге 12-1. Это имя важно, поскольку оно используется для импорта функций из других пакетов, созданных в рамках того же

проекта, и сторонних пакетов, как будет показано в следующих примерах. Оператор `go` указывает используемую версию Go, для этой книги это 1.17.

Создание пользовательского пакета

Пакеты позволяют добавить структуру в проект, чтобы связанные функции были сгруппированы вместе. Создайте папку `packages/store` и добавьте в нее файл с именем `product.go`, содержимое которого показано в листинге 12-4.

```
package store

type Product struct {
    Name, Category string
    price float64
}
```

Листинг 12-4 Содержимое файла product.go в папке packages/store

Пользовательский пакет определяется с помощью ключевого слова `package`, а указанный мной пакет называется `store`:

```
...
package store
...
```

Имя, указанное в операторе `package`, должно совпадать с именем папки, в которой создаются файлы кода, которая в данном случае `store`.

Тип `Product` имеет несколько важных отличий от аналогичных типов, определенных в предыдущих главах, как я объясню в следующих разделах.

КОММЕНТАРИИ ЭКСПОРТИРУЕМЫХ ФУНКЦИЙ

Линтер Go сообщит об ошибке для любой функции, экспортированной из пакета и не описанной в комментарии. Комментарии должны быть простыми и описательными, и принято начинать комментарий с названия функции, например:

```
...
```

```
// Product describes an item for sale
type Product struct {
    Name, Category string // Name and type of the product
    price float64
}
...
```

При комментировании пользовательских типов также можно описать экспортированные поля. Go также поддерживает комментарий, описывающий весь пакет, который появляется перед ключевым словом `package`, например:

```
...
// Package store provides types and methods
// commonly required for online sales
package store
...
```

Эти комментарии обрабатываются инструментом `go doc`, который создает документацию по коду. Я не добавлял комментарии к примерам в этой книге для краткости, но комментирование кода особенно важно при написании пакетов, которые используются другими разработчиками.

Использование пользовательского пакета

Зависимости от пользовательских пакетов объявляются с помощью оператора `import`, как показано в листинге 12-5.

```
package main

import (
    "fmt"
    "packages/store"
)

func main() {

    product := store.Product {
        Name: "Kayak",
        Category: "Watersports",
    }
```

```
    fmt.Println("Name:", product.Name)
    fmt.Println("Category:", product.Category)
}
```

Листинг 12-5 Использование пользовательского пакета в файле main.go в папке packages

Оператор `import` задает пакет в виде пути, состоящего из имени модуля, созданного командой в листинге 12-1, и имени пакета, разделенных косой чертой, как показано на рисунке 12-1.

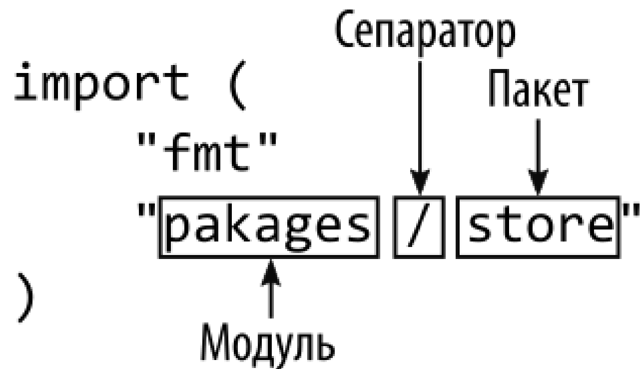

Рисунок 12-1 Импорт пользовательского пакета

Доступ к экспортируемым функциям, предоставляемым пакетом, осуществляется с использованием имени пакета в качестве префикса, например:

```
...
var product *store.Product = &store.Product {
...
```

Чтобы указать тип `Product`, я должен указать префикс типа с именем пакета, как показано на рисунке 12-2.

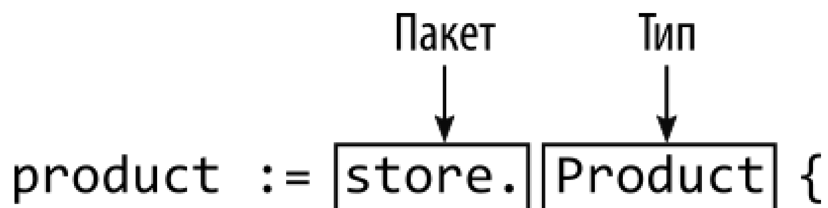

Рисунок 12-2 Использование имени пакета

Создайте и выполните проект, который выдаст следующий результат:

```
Name: Kayak
Category: Watersports
```

Понимание управления доступом к пакетам

Тип `Product`, определенный в листинге 12-4, имеет важное отличие от аналогичных типов, определенных в предыдущих главах: свойства `Name` и `Category` начинаются с заглавной буквы.

В Go необычный подход к управлению доступом. Вместо того, чтобы полагаться на специальные ключевые слова, такие как `public` и `private`, Go проверяет первую букву имен, присвоенных функциям в файле кода, таким как типы, функции и методы. Если первая буква строчная, то функция может использоваться только в пакете, который ее определяет. Функции экспортируются для использования вне пакета, если им присваивается первая буква в верхнем регистре.

Имя типа структуры в листинге 12-4 — `Product`, что означает, что этот тип можно использовать вне пакета `store`. Имена полей `Name` и `Category` также начинаются с заглавной буквы, что означает, что они также экспортируются. Поле `price` имеет первую строчную букву, что означает, что доступ к нему возможен только внутри пакета `store`. Рисунок 12-3 иллюстрирует эти различия.

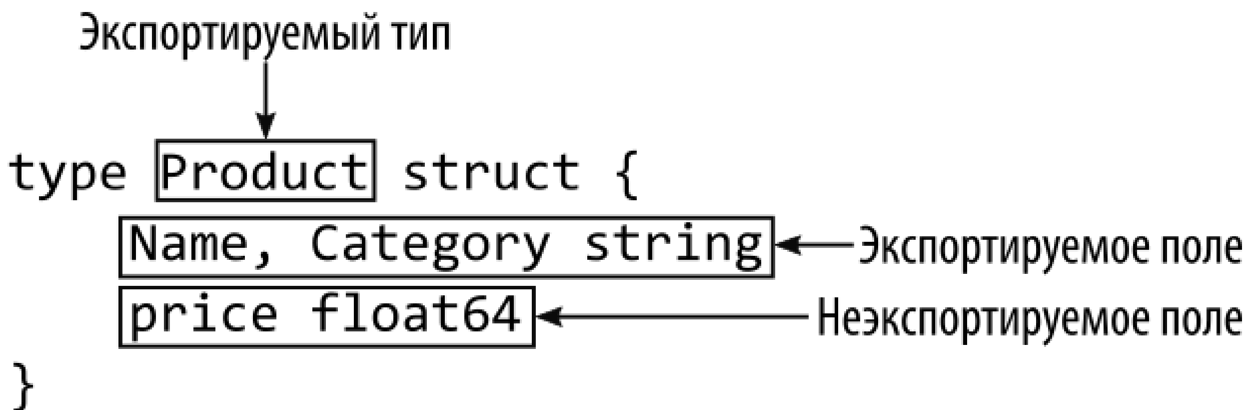

Рисунок 12-3 Экспортированные и частные функции

Компилятор применяет правила экспорта пакета, а это означает, что при доступе к полю `price` за пределами пакета `store` будет сгенерирована ошибка, как показано в листинге 12-6.

```
package main
```

```
import (
    "fmt"
    "packages/store"
)

func main() {

    product := store.Product {
        Name: "Kayak",
        Category: "Watersports",
        price: 279,
    }

    fmt.Println("Name:", product.Name)
    fmt.Println("Category:", product.Category)
    fmt.Println("Price:", product.price)
}
```

Листинг 12-6 Доступ к неэкспортированному полю в файле main.go в папке packages

Первое изменение пытается установить значение для поля `price` при использовании литерального синтаксиса для создания значения `Product`. Второе изменение пытается прочитать значение поля `price`.

Правила контроля доступа применяются компилятором, который сообщает о следующих ошибках при компиляции кода:

```
.\main.go:13:9: cannot refer to unexported field 'price' in struct literal of type store.Product
.\main.go:18:34: product.price undefined (cannot refer to unexported field or method price)
```

Чтобы устранить эти ошибки, я могу либо экспортировать поле `price`, либо экспортировать методы или функции, обеспечивающие доступ к значению поля. В листинге 12-7 определяется функция-конструктор для создания значений `Product` и методов для получения и установки поля `price`.

```
package store

type Product struct {
    Name, Category string
    price float64
```

```
}

func   NewProduct(name,   category   string,   price   float64)
*Product {
    return &Product{ name, category, price }
}

func (p *Product) Price() float64 {
    return p.price
}

func (p *Product) SetPrice(newPrice float64)  {
    p.price = newPrice
}
```

Листинг 12-7 Определение методов в файле product.go в папке store

Правила управления доступом не применяются к отдельным параметрам функции или метода, а это означает, что функция `NewProduct` должна иметь первый экспортируемый символ в верхнем регистре, но имена параметров могут быть строчными.

Методы следуют типичному соглашению об именах для экспортированных методов, обращающихся к полю, так что метод `Price` возвращает значение поля, а метод `SetPrice` присваивает новое значение. Листинг 12-8 обновляет код в файле `main.go` для использования новых функций.

```
package main

import (
    "fmt"
    "packages/store"
)

func main() {

    product := store.NewProduct("Kayak", "Watersports", 279)

    fmt.Println("Name:", product.Name)
    fmt.Println("Category:", product.Category)
    fmt.Println("Price:", product.Price())
}
```

Листинг 12-8 Использование функций пакета в файле main.go в папке packages

Скомпилируйте и выполните проект с помощью команды из листинга 12-8, и вы получите следующий вывод, демонстрирующий, что код в `main` пакете может считывать поле `price` с помощью метода `Price`:

```
Name: Kayak
Category: Watersports
Price: 279
```

Добавление файлов кода в пакеты

Пакеты могут содержать несколько файлов кода, и для упрощения разработки правила управления доступом и префиксы пакетов не применяются при доступе к функциям, определенным в одном пакете. Добавьте файл с именем `tax.go` в папку `store` с содержимым, показанным в листинге 12-9.

```
package store

const defaultTaxRate float64 = 0.2
const minThreshold = 10

type taxRate struct {
    rate, threshold float64
}

func newTaxRate(rate, threshold float64) *taxRate {
    if (rate == 0) {
        rate = defaultTaxRate
    }
    if (threshold < minThreshold) {
        threshold = minThreshold
    }
    return &taxRate { rate, threshold }
}

func (taxRate *taxRate) calcTax(product *Product) float64 {
    if (product.price > taxRate.threshold) {
        return product.price + (product.price * taxRate.rate)
    }
```

```
    return product.price
}
```

Листинг 12-9 Содержимое файла tax.go в папке store

Все функции, определенные в файле `tax.go`, не экспортируются, что означает, что их можно использовать только в пакете `store`. Обратите внимание, что метод `calcTax` может получить доступ к полю `price` типа `Product` и делает это без обращения к типу как к `store.Product`, поскольку он находится в том же пакете:

```
...
func (taxRate *taxRate) calcTax(product *Product) float64 {
    if (product.price > taxRate.threshold) {
        return product.price + (product.price * taxRate.rate)
    }
    return product.price
}
...
```

В листинге 12-10 я изменил метод `Product.Price`, чтобы он возвращал значение поля `price` плюс налог.

```
package store

var standardTax = newTaxRate(0.25, 20)

type Product struct {
    Name, Category string
    price float64
}

func  NewProduct(name,  category  string,  price  float64)
*Product {
    return &Product{ name, category, price }
}

func (p *Product) Price() float64 {
    return standardTax.calcTax(p)
}

func (p *Product) SetPrice(newPrice float64)  {
    p.price = newPrice
```

```
}
```

Листинг 12-10 Расчет налога в файле product.go в папке store

Метод Price может получить доступ к неэкспортированному методу calcTax, но этот метод и тип, к которому он применяется, доступны для использования только в пакете store. Скомпилируйте и выполните код с помощью команды, показанной в листинге 12-10, и вы получите следующий вывод:

```
Name: Kayak
Category: Watersports
Price: 348.75
```

КАК ИЗБЕЖАТЬ ЛОВУШКИ ПЕРЕОПРЕДЕЛЕНИЯ

Распространенной ошибкой является повторное использование имен в разных файлах в одном пакете. Это то, что я делаю часто, в том числе при написании примера, показанного в листинге 12-10. Моя первоначальная версия кода в файле product.go содержала следующее утверждение:

```
...
var taxRate = newTaxRate(0.25, 20)
...
```

Это вызывает ошибку компилятора, поскольку файл tax.go определяет тип структуры с именем taxRate. Компилятор не делает различий между именами, присвоенными переменным, и именами, присвоенными типам, и сообщает об ошибке, например:

```
store\tax.go:6:6: taxRate redeclared in this block
        previous declaration at store\product.go:3:5
```

Вы также можете увидеть ошибки в редакторе кода, говорящие о том, что taxRate является недопустимым типом. Это одна и та же проблема, выраженная по-разному. Чтобы избежать этих ошибок, вы должны убедиться, что функции верхнего уровня, определенные в пакете, имеют уникальные имена. Имена не обязательно должны быть уникальными в пакетах или внутри функций и методов.

Разрешение конфликтов имен пакетов

Когда пакет импортируется, комбинация имени модуля и имени пакета обеспечивает уникальную идентификацию пакета. Но при доступе к функциям, предоставляемым пакетом, используется только имя пакета, что может привести к конфликтам. Чтобы увидеть, как возникает эта проблема, создайте папку `packages/fmt` и добавьте в нее файл с именем `formats.go` с кодом, показанным в листинге 12-11.

```
package fmt

import "strconv"

func ToCurrency(amount float64) string {
    return "$" + strconv.FormatFloat(amount, 'f', 2, 64)
}
```

Листинг 12-11 Содержимое файла formats.go в папке fmt

Этот файл экспортирует функцию с именем `ToCurrency`, которая получает значение `float64` и создает отформатированную сумму в долларах с помощью функции `strconv.FormatFloat`, описанной в главе 17.

Пакет `fmt`, определенный в листинге 12-11, имеет то же имя, что и один из наиболее широко используемых пакетов стандартных библиотек. Это вызывает проблему при использовании обоих пакетов, как показано в листинге 12-12.

```
package main

import (
    "fmt"
    "packages/store"
    "packages/fmt"
)

func main() {

    product := store.NewProduct("Kayak", "Watersports", 279)

    fmt.Println("Name:", product.Name)
    fmt.Println("Category:", product.Category)
```

```
    fmt.Println("Price:", fmt.ToCurrency(product.Price()))
}
```

Листинг 12-12 Использование пакетов с тем же именем в файле main.go в папке packages

Скомпилируйте проект, и вы получите следующие ошибки:

```
.\main.go:6:5: fmt redeclared as imported package name
        previous declaration at .\main.go:4:5
.\main.go:13:5: undefined: "packages/fmt".Println
.\main.go:14:5: undefined: "packages/fmt".Println
.\main.go:15:5: undefined: "packages/fmt".Println
```

Использование псевдонима пакета

Одним из способов решения конфликтов имен пакетов является использование псевдонима, который позволяет получить доступ к пакету с использованием другого имени, как показано в листинге 12-13.

```
package main

import (
    "fmt"
    "packages/store"
    currencyFmt "packages/fmt"
)

func main() {

    product := store.NewProduct("Kayak", "Watersports", 279)

    fmt.Println("Name:", product.Name)
    fmt.Println("Category:", product.Category)
                                        fmt.Println("Price:",
currencyFmt.ToCurrency(product.Price()))
}
```

Листинг 12-13 Использование псевдонима пакета в файле main.go в папке packages

Псевдоним, под которым будет известен пакет, объявляется перед путем импорта, как показано на рисунке 12-4.

```
import (
    "fmt"
    "pakages/store"
    currencyFmt "pakages/fmt"
)
```

Псевдоним

Путь импорта пакета

Рисунок 12-4 Псевдоним пакета

Псевдоним в этом примере разрешает конфликт имен, поэтому к функциям, определенным пакетом, импортированным с помощью пути `packages/fmt`, можно получить доступ с использованием `currencyFmt` в качестве префикса, например:

```
...
fmt.Println("Price:",
currencyFmt.ToCurrency(product.Price()))
...
```

Скомпилируйте и выполните проект, и вы получите следующий вывод, основанный на функциях, определенных пакетом `fmt` в стандартной библиотеке, и пользовательским пакетом `fmt`, которому присвоен псевдоним:

```
Name: Kayak
Category: Watersports
Price: $348.75
```

Использование точечного импорта

Существует специальный псевдоним, известный как *точечный импорт*, который позволяет использовать функции пакета без использования префикса, как показано в листинге 12-14.

```
package main

import (
    "fmt"
    "packages/store"
```

```
    . "packages/fmt"
)

func main() {

    product := store.NewProduct("Kayak", "Watersports", 279)

    fmt.Println("Name:", product.Name)
    fmt.Println("Category:", product.Category)
    fmt.Println("Price:", ToCurrency(product.Price()))
}
```

Листинг 12-14 Использование точечного импорта в файле main.go в папке packages

Точечный импорт использует точку в качестве псевдонима пакета, как показано на рисунке 12-5.

```
import (
    "fmt"
    "pakages/store"
    . "pakages/fmt"
)
```


Рисунок 12-5 Использование точечного импорта

Точечный импорт позволяет мне получить доступ к функции `ToCurrency` без использования префикса, например:

```
...
fmt.Println("Price:", ToCurrency(product.Price()))
...
```

При использовании точечного импорта вы должны убедиться, что имена функций, импортированных из пакета, не определены в импортирующем пакете. Например, это означает, что я должен убедиться, что имя `ToCurrency` не используется какой-либо функцией, определенной в `main` пакете. По этой причине точечный импорт следует использовать с осторожностью.

Создание вложенных пакетов

Пакеты могут быть определены внутри других пакетов, что упрощает разбиение сложных функций на максимально возможное количество модулей. Создайте папку packages/store/cart и добавьте в нее файл с именем cart.go с содержимым, показанным в листинге 12-15.

```
package cart

import "packages/store"

type Cart struct {
    CustomerName string
    Products []store.Product
}

func (cart *Cart) GetTotal() (total float64) {
    for _, p := range cart.Products {
        total += p.Price()
    }
    return
}
```

Листинг 12-15 Содержимое файла cart.go в папке store/cart

Оператор package используется так же, как и любой другой пакет, без необходимости включать имя родительского или включающего пакета. И зависимость от пользовательских пакетов должна включать полный путь к пакету, как показано в листинге. Код в листинге 12-15 определяет тип структуры с именем Cart, который экспортирует поля CustomerName и Products, а также метод GetTotal.

При импорте вложенного пакета путь к пакету начинается с имени модуля и перечисляет последовательность пакетов, как показано в листинге 12-16.

```
package main

import (
    "fmt"
    "packages/store"
    . "packages/fmt"
    "packages/store/cart"
```

```
)

func main() {

    product := store.NewProduct("Kayak", "Watersports", 279)

    cart := cart.Cart {
        CustomerName: "Alice",
        Products: []store.Product{ *product },
    }

    fmt.Println("Name:", cart.CustomerName)
    fmt.Println("Total:",  ToCurrency(cart.GetTotal()))
}
```

Листинг 12-16 Использование вложенного пакета в файле main.go в папке packages

Доступ к функциям, определенным вложенным пакетом, осуществляется по имени пакета, как и к любому другому пакету. В листинге 12-16 это означает, что доступ к типу и функции, экспортируемым пакетом `store/cart`, осуществляется с использованием `cart` в качестве префикса. Скомпилируйте и запустите проект, и вы получите следующий вывод:

```
Name: Alice
Total: $348.75
```

Использование функций инициализации пакета

Каждый файл кода может содержать функцию инициализации, которая выполняется только после загрузки всех пакетов и выполнения всех остальных инициализаций, таких как определение констант и переменных. Чаще всего функции инициализации используются для выполнения вычислений, которые сложно выполнить или для выполнения которых требуется дублирование, как показано в листинге 12-17.

```
package store

const defaultTaxRate float64 = 0.2
const minThreshold = 10
```

```
var categoryMaxPrices = map[string]float64 {
    "Watersports": 250 + (250 * defaultTaxRate),
    "Soccer": 150 + (150 * defaultTaxRate),
    "Chess": 50 + (50 * defaultTaxRate),
}

type taxRate struct {
    rate, threshold float64
}

func newTaxRate(rate, threshold float64) *taxRate {
    if (rate == 0) {
        rate = defaultTaxRate
    }
    if (threshold < minThreshold) {
        threshold = minThreshold
    }
    return &taxRate { rate, threshold }
}

func (taxRate *taxRate) calcTax(product *Product) (price
float64) {
    if (product.price > taxRate.threshold) {
                price = product.price + (product.price *
taxRate.rate)
    } else {
        price = product.price
    }
     if max, ok := categoryMaxPrices[product.Category]; ok &&
price > max {
        price = max
    }
    return
}
```

Листинг 12-17 Расчет максимальных цен в файле tax.go в папке store

Эти изменения вводят максимальные цены для конкретных категорий, которые хранятся на карте. Максимальная цена для каждой категории рассчитывается одинаково, что приводит к дублированию и приводит к тому, что код может быть трудным для чтения и обслуживания.

Эту проблему можно легко решить с помощью цикла for, но Go допускает циклы только внутри функций, и мне нужно выполнять эти вычисления на верхнем уровне файла кода.

Решение состоит в использовании функции инициализации, которая вызывается автоматически при загрузке пакета и в которой можно использовать языковые функции, такие как циклы for, как показано в листинге 12-18.

```
package store

const defaultTaxRate float64 = 0.2
const minThreshold = 10

var categoryMaxPrices = map[string]float64 {
    "Watersports": 250,
    "Soccer": 150,
    "Chess": 50,
}

func init() {
    for category, price := range categoryMaxPrices {
            categoryMaxPrices[category] = price + (price *
defaultTaxRate)
    }
}

type taxRate struct {
    rate, threshold float64
}

func newTaxRate(rate, threshold float64) *taxRate {
    // ...statements omitted for brevity...
}

func (taxRate *taxRate) calcTax(product *Product) (price
float64) {
    // ...statements omitted for brevity...
}
```

Листинг 12-18 Использование функции инициализации в файле tax.go в папке store

Функция инициализации называется `init` и определяется без параметров и результата. Функция `init` вызывается автоматически и предоставляет возможность подготовить пакет к использованию. Оба листинга 12-17 и 12-18 при компиляции и выполнении выдают следующий результат:

```
Name: Kayak
Price: $300.00
```

Функция `init` не является обычной функцией Go и не может быть вызвана напрямую. И, в отличие от обычных функций, в одном файле может быть определено несколько функций `init`, и все они будут выполняться.

> **ИЗБЕЖАНИЕ ЛОВУШКИ ФУНКЦИЙ МНОЖЕСТВЕННОЙ ИНИЦИАЛИЗАЦИИ**
> Каждый файл кода может иметь свою собственную функцию инициализации. При использовании стандартного компилятора Go функции инициализации выполняются на основе алфавитного порядка имен файлов, поэтому функция в файле `a.go` будет выполняться перед функцией в файле `b.go` и так далее.
>
> Но этот порядок не является частью спецификации языка Go, и на него не следует полагаться. Ваши функции инициализации должны быть автономными и не зависеть от других функций `init`, которые были вызваны ранее.

Импорт пакета только для эффектов инициализации

Go предотвращает импорт пакетов, но не их использование, что может быть проблемой, если вы полагаетесь на эффект функции инициализации, но вам не нужно использовать какие-либо функции, экспортируемые пакетом. Создайте папку `packages/data` и добавьте в нее файл с именем `data.go` с содержимым, показанным в листинге 12-19.

```
package data

import "fmt"
```

```
func init() {
    fmt.Println(("data.go init function invoked"))
}

func GetData() []string {
    return []string {"Kayak", "Lifejacket", "Paddle", "Soccer
Ball"}
}
```

Листинг 12-19 Содержимое файла data.go в папке data

Функция инициализации записывает сообщение, когда она вызывается для целей этого примера. Если мне нужен эффект функции инициализации, но мне не нужно использовать функцию `GetData`, которую экспортирует пакет, я могу импортировать пакет, используя пустой идентификатор в качестве псевдонима для имени пакета, как показано в листинге 12-20.

```
package main

import (
    "fmt"
    "packages/store"
    . "packages/fmt"
    "packages/store/cart"
    _ "packages/data"
)

func main() {

    product := store.NewProduct("Kayak", "Watersports", 279)

    cart := cart.Cart {
        CustomerName: "Alice",
        Products: []store.Product{ *product },
    }

    fmt.Println("Name:", cart.CustomerName)
    fmt.Println("Total:",  ToCurrency(cart.GetTotal()))

}
```

Листинг 12-20 Импорт для инициализации в файл main.go в папке packages

Пустой идентификатор — символ подчеркивания — позволяет импортировать пакет, не требуя использования его экспортированных функций. Скомпилируйте и запустите проект, и вы увидите сообщение, написанное функцией инициализации, определенной в листинге 12-19:

```
data.go init function invoked
Name: Alice
Total: $300.00
```

Использование внешних пакетов

Проекты могут быть расширены с использованием пакетов, разработанных третьими сторонами. Пакеты загружаются и устанавливаются с помощью команды `go get`. Запустите команду, показанную в листинге 12-21, в папке `packages`, чтобы добавить пакет в пример проекта.

```
go get github.com/fatih/color@v1.10.0
```

Листинг 12-21 Установка пакета

Аргументом команды `go get` является путь к модулю, содержащему пакет, который вы хотите использовать. За именем следует символ `@`, а затем номер версии пакета, перед которым стоит буква `v`, как показано на рисунке 12-6.

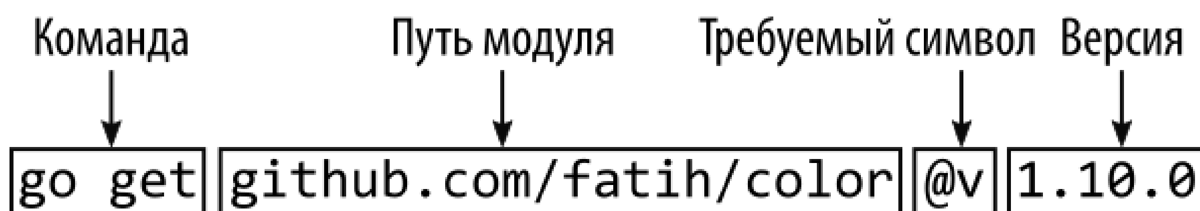

Рисунок 12-6 Выбор пакета

Команда `go get` является сложной и знает, что путь, указанный в листинге 12-21, является URL-адресом GitHub. Загружается указанная версия модуля, а содержащиеся в нем пакеты компилируются и устанавливаются, чтобы их можно было использовать в проекте. (Пакеты распространяются в виде исходного кода, что позволяет компилировать их для платформы, на которой вы работаете.)

ПОИСК ПАКЕТОВ GO

Есть два полезных ресурса для поиска пакетов Go. Первый — это `https://pkg.go.dev`, который предоставляет поисковую систему. К сожалению, может потребоваться некоторое время, чтобы выяснить, какие ключевые слова необходимы для поиска определенного типа пакета.

Второй ресурс — `https://github.com/golang/go/wiki/Projects`, который предоставляет кураторский список проектов Go, сгруппированных по категориям. Не все проекты, перечисленные на `pkg.go.dev`, есть в списке, и я предпочитаю использовать оба ресурса для поиска пакетов.

При выборе модулей следует соблюдать осторожность. Многие модули Go пишутся отдельными разработчиками для решения проблемы, а затем публикуются для использования кем-либо еще. Это создает богатую модульную экосистему, но это означает, что обслуживание и поддержка могут быть непоследовательными. Например, модуль `github.com/fatih/color`, который я использую в этом разделе, устарел и больше не получает обновлений. Я рад продолжать использовать его, так как мое применение в этой главе простое, а код работает хорошо. Вы должны выполнить такую же оценку для модулей, на которые вы полагаетесь в своих проектах.

Изучите файл `go.mod` после завершения команды `go get`, и вы увидите новые операторы конфигурации:

```
module packages
go 1.17
require (
    github.com/fatih/color v1.10.0 // indirect
    github.com/mattn/go-colorable v0.1.8 // indirect
    github.com/mattn/go-isatty v0.0.12 // indirect
      golang.org/x/sys  v0.0.0-20200223170610-d5e6a3e2c0ae  //
indirect
)
```

Оператор `require` отмечает зависимость от модуля `github.com/fatih/color` и других необходимых ему модулей.

Комментарий indirect в конце операторов добавляется автоматически, поскольку пакеты не используются кодом в проекте. Файл с именем go.sum создается при получении модуля и содержит контрольные суммы, используемые для проверки пакетов.

> **Примечание**
> Вы также можете использовать файл go.mod для создания зависимостей от проектов, которые вы создали локально, и именно этот подход я использую в третьей части для примера SportsStore. Подробности см. в главе 35.

После установки модуля содержащиеся в нем пакеты можно использовать в проекте, как показано в листинге 12-22.

```
package main

import (
    //"fmt"
    "packages/store"
    . "packages/fmt"
    "packages/store/cart"
    _ "packages/data"
    "github.com/fatih/color"
)

func main() {

    product := store.NewProduct("Kayak", "Watersports", 279)

    cart := cart.Cart {
        CustomerName: "Alice",
        Products: []store.Product{ *product },
    }

    color.Green("Name: " + cart.CustomerName)
    color.Cyan("Total: " + ToCurrency(cart.GetTotal()))
}
```

Листинг 12-22 Использование стороннего пакета в файле main.go в папке packages

Внешние пакеты импортируются и используются как пользовательские пакеты. Оператор `import` указывает путь к модулю, и последняя часть этого пути используется для доступа к функциям, экспортируемым пакетом. В этом случае пакет называется `color`, и это префикс, используемый для доступа к функциям пакета.

Функции `Green` и `Cyan`, используемые в листинге 12-22, записывают цветной вывод, и если вы скомпилируете и запустите проект, вы увидите вывод, показанный на рисунке 12-7.

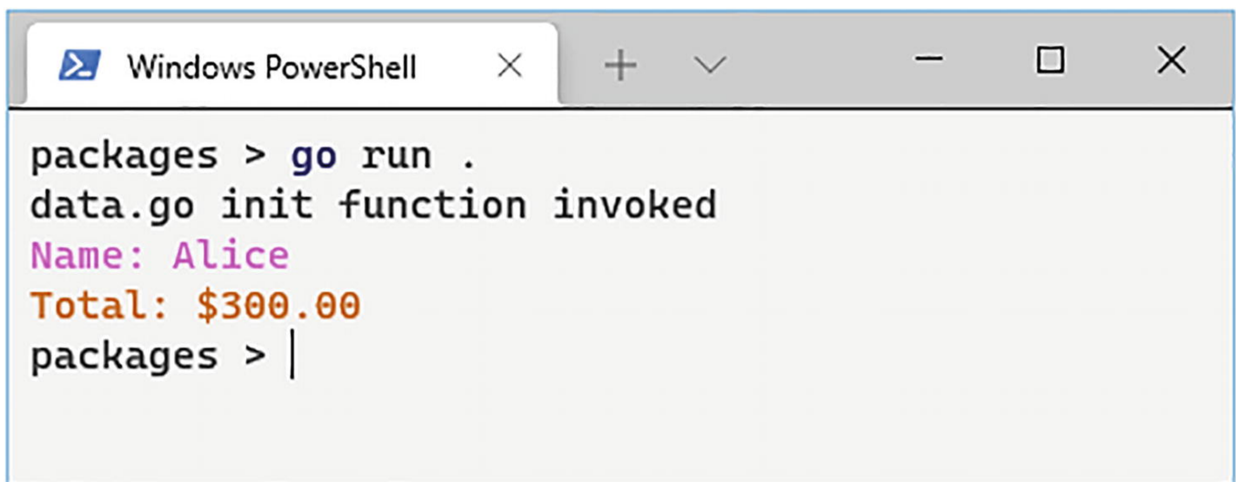

Рисунок 12-7 Запуск примера приложения

ПОНИМАНИЕ ВЫБОРА МИНИМАЛЬНОЙ ВЕРСИИ

При первом запуске команды `go get` в листинге 12-22 вы увидите список загруженных модулей, который иллюстрирует, что модули имеют свои собственные зависимости и что они разрешаются автоматически:

```
go: downloading github.com/fatih/color v1.10.0
go: downloading github.com/mattn/go-isatty v0.0.12
go: downloading github.com/mattn/go-colorable v0.1.8
go: downloading golang.org/x/sys v0.0.0-20200223170610-d5e6a3e2c0ae
```

Загрузки кэшируются, поэтому вы не увидите сообщения при следующем использовании команды `go get` для того же модуля.

Вы можете обнаружить, что ваш проект зависит от разных версий модуля, особенно в сложных проектах с большим

количеством зависимостей. В таких ситуациях Go разрешает эту зависимость, используя самую последнюю версию, указанную в этих зависимостях. Так, например, если есть зависимости от версии 1.1 и 1.5 модуля, Go будет использовать версию 1.5 при сборке проекта. Go будет использовать только самую последнюю версию, указанную в зависимости, даже если доступна более новая версия. Например, если в самой последней зависимости для модуля указана версия 1.5, Go не будет использовать версию 1.6, даже если она доступна.

Результатом этого подхода является то, что ваш проект не может быть скомпилирован с использованием версии модуля, которую вы выбрали с помощью команды `go get`, если модуль зависит от более поздней версии. Точно так же модуль не может быть скомпилирован с версиями, которые он ожидает для своих зависимостей, если другой модуль — или файл `go.mod` — указывает более позднюю версию.

Управление внешними пакетами

Команда `go get` добавляет зависимости в файл `go.mod`, но они не удаляются автоматически, если внешний пакет больше не требуется. В листинге 12-23 изменено содержимое файла `main.go`, чтобы исключить использование пакета `github.com/fatih/color`.

```
package main

import (
    "fmt"
    "packages/store"
    . "packages/fmt"
    "packages/store/cart"
    _ "packages/data"
    //"github.com/fatih/color"
)

func main() {

    product := store.NewProduct("Kayak", "Watersports", 279)

    cart := cart.Cart {
        CustomerName: "Alice",
```

```
        Products: []store.Product{ *product },
    }

    // color.Green("Name: " + cart.CustomerName)
    // color.Cyan("Total: " + ToCurrency(cart.GetTotal()))
    fmt.Println("Name:", cart.CustomerName)
    fmt.Println("Total:",  ToCurrency(cart.GetTotal()))
}
```

Листинг 12-23 Удаление пакета в файле main.go в папке packages

Чтобы обновить файл `go.mod`, чтобы отразить изменения, запустите команду, показанную в листинге 12-24, в папке `packages`.

```
go mod tidy
```

Листинг 12-24 Обновление зависимостей пакетов

Команда проверяет код проекта, определяет, что больше нет зависимости ни от одного из пакетов от модуля `github.com/fatih/color`, и удаляет оператор `require` из файла `go.mod`:

```
module packages
go 1.17
```

Резюме

В этой главе я объяснил роль пакетов в разработке Go. Я показал вам, как использовать пакеты для добавления структуры в проект и как они могут предоставить доступ к функциям, разработанным третьими сторонами. В следующей главе я опишу возможности Go для составления типов, которые позволяют создавать сложные типы.

13. Тип и композиция интерфейса

В этой главе я объясню, как типы комбинируются для создания новых функций. Go не использует наследование, с которым вы, возможно, знакомы в других языках, а вместо этого полагается на подход, известный как *композиция*. Это может быть трудно понять, поэтому в этой главе описываются некоторые функции, описанные в предыдущих главах, чтобы заложить прочную основу для объяснения процесса композиции. В таблице 13-1 тип и состав интерфейса представлены в контексте.

Таблица 13-1 Помещение типа и композиции интерфейса в контекст

Вопрос	Ответ
Что это?	Композиция — это процесс создания новых типов путем объединения структур и интерфейсов.
Почему это полезно?	Композиция позволяет определять типы на основе существующих типов.
Как это используется?	Существующие типы встраиваются в новые типы.
Есть ли подводные камни или ограничения?	Композиция работает не так, как наследование, и для достижения желаемого результата необходимо соблюдать осторожность.
Есть ли альтернативы?	Композиция необязательна, и вы можете создавать полностью независимые типы.

Таблица 13-2 суммирует главу.

Таблица 13-2 Краткое содержание главы

Проблема	Решение	Листинг
Составление типа структуры	Добавить встроенное поле	7-9, 14–17
Построить на уже составленном типе	Создайте цепочку встроенных типов	10–13
Составьте тип интерфейса	Добавьте имя существующего интерфейса в определение нового интерфейса.	25–26

Подготовка к этой главе

Чтобы подготовиться к этой главе, откройте новую командную строку, перейдите в удобное место и создайте каталог с именем `composition`. Запустите команду, показанную в листинге 13-1, в папке `composition`, чтобы создать файл модуля.

> **Подсказка**
> Вы можете загрузить пример проекта для этой главы — и для всех остальных глав этой книги — с `https://github.com/apress/pro-go`. См. Главу 2 о том, как получить помощь, если у вас возникнут проблемы с запуском примеров.

```
go mod init composition
```

Листинг 13-1 Инициализация модуля

Добавьте файл с именем `main.go` в папку `composition` с содержимым, показанным в листинге 13-2.

```
package main

import "fmt"

func main() {

    fmt.Println("Hello, Composition")
}
```

Листинг 13-2 Содержимое файла main.go в папке composition

Используйте командную строку для запуска команды, показанной в листинге 13-3, в папке `composition`.

```
go run .
```

Листинг 13-3 Запуск примера проекта

Код в файле `main.go` будет скомпилирован и выполнен, что приведет к следующему результату:

```
Hello, Composition
```

Понимание композиции типов

Если вы привыкли к таким языкам, как C# или Java, то вы создали базовый класс и создали подклассы для добавления более специфических функций. Подклассы наследуют функциональные возможности базового класса, что предотвращает дублирование кода. Результатом является набор классов, где базовый класс определяет общую функциональность, которая дополняется более специфическими функциями в отдельных подклассах, как показано на рисунке 13-1.

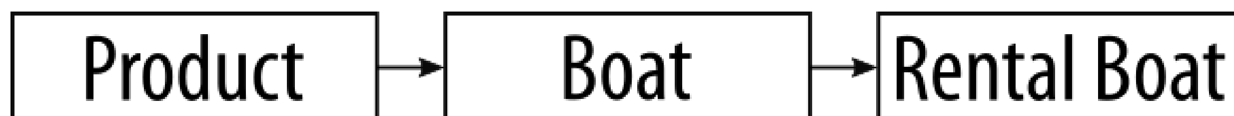

Рисунок 13-1 Набор классов

Go не поддерживает классы или наследование и вместо этого фокусируется на *композиции*. Но, несмотря на различия, композицию можно использовать для создания иерархий типов, только по-другому.

Определение базового типа

Отправной точкой является определение типа структуры и метода, которые я буду использовать для создания более конкретных типов в последующих примерах. Создайте папку `composition/store` и добавьте в нее файл с именем `product.go` с содержимым, показанным в листинге 13-4.

```
package store

type Product struct {
    Name, Category string
    price float64
}

func (p *Product) Price(taxRate float64) float64 {
    return p.price + (p.price * taxRate)
}
```

Листинг 13-4 Содержимое файла product.go в папке store

Структура `Product` определяет поля `Name` и `Category`, которые экспортируются, и поле `price`, которое не экспортируется. Существует также метод `Price`, который принимает параметр `float64` и использует его с полем цены для расчета `price` с учетом налогов.

Определение конструктора

Поскольку Go не поддерживает классы, он также не поддерживает конструкторы классов. Как я объяснил, общепринятым соглашением является определение функции-конструктора с именем `New<Type>`, такой как `NewProduct`, как показано в листинге 13-5, и которая позволяет предоставлять значения для всех полей, даже для тех, которые не были экспортируется. Как и в случае с другими функциями кода, использование заглавной буквы в имени функции-конструктора определяет, экспортируется ли оно за пределы пакета.

```
package store

type Product struct {
    Name, Category string
    price float64
}

func NewProduct(name, category string, price float64)
*Product {
    return &Product{ name, category, price }
}

func (p *Product) Price(taxRate float64) float64 {
    return p.price + (p.price * taxRate)
}
```

Листинг 13-5 Определение конструктора в файле product.go в папке store

Функции-конструкторы являются лишь соглашением, и их использование не является принудительным, что означает, что экспортированные типы могут быть созданы с использованием литерального синтаксиса, если неэкспортируемым полям не присвоены значения. В листинге 13-6 показано использование функции-конструктора и литерального синтаксиса.

```
package main
```

```
import (
    "fmt"
    "composition/store"
)

func main() {

    kayak := store.NewProduct("Kayak", "Watersports", 275)
        lifejacket  :=  &store.Product{  Name:  "Lifejacket",
Category:  "Watersports"}

    for _, p := range []*store.Product { kayak, lifejacket} {
        fmt.Println("Name:", p.Name, "Category:", p.Category,
"Price:", p.Price(0.2))
    }
}
```

Листинг 13-6 Создание структурных значений в файле main.go в папке composition

Конструкторы следует использовать всякий раз, когда они определены, поскольку они облегчают управление изменениями в способе создания значений и обеспечивают правильную инициализацию полей. В листинге 13-6 использование литерального синтаксиса означает, что полю price не присваивается значение, что влияет на выходные данные метода Price. Но поскольку Go не поддерживает принудительное использование конструкторов, их использование требует дисциплины.

Скомпилируйте и запустите проект, и вы получите следующий вывод:

```
Name: Kayak Category: Watersports Price: 330
Name: Lifejacket Category: Watersports Price: 0
```

Типы композиций

Go поддерживает композицию, а не наследование, которое достигается путем объединения типов структур. Добавьте файл с именем boat.go в папку store с содержимым, показанным в листинге 13-7.

```
package store
```

```
type Boat struct {
    *Product
    Capacity int
    Motorized bool
}

func  NewBoat(name  string,  price  float64,  capacity  int,
motorized bool) *Boat {
    return &Boat {
        NewProduct(name, "Watersports", price), capacity,
motorized,
    }
}
```

Листинг 13-7 Содержимое файла boat.go в папке store

Тип структуры `Boat` определяет встроенное поле `*Product`, как показано на рисунке 13-2.

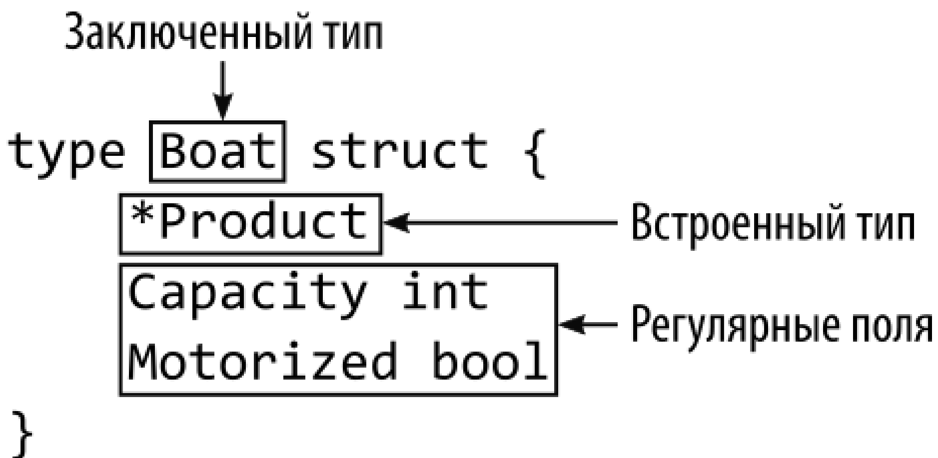

Рисунок 13-2 Встраивание типа

Структура может смешивать обычные и встроенные типы полей, но встроенные поля являются важной частью функции композиции, как вы скоро увидите.

Функция `NewBoat` — это конструктор, который использует свои параметры для создания `Boat` со встроенным значением `Product`. В листинге 13-8 показано использование новой структуры.

```
package main

import (
```

```
    "fmt"
    "composition/store"
)

func main() {

    boats := []*store.Boat {
        store.NewBoat("Kayak", 275, 1, false),
        store.NewBoat("Canoe", 400, 3, false),
        store.NewBoat("Tender", 650.25, 2, true),
    }

    for _, b := range boats {
            fmt.Println("Conventional:",  b.Product.Name,
"Direct:", b.Name)
    }
}
```

Листинг 13-8 Использование структуры лодки в файле main.go в папке composition

Новые операторы создают срез Boat *Boat, который заполняется с помощью функции-конструктора NewBoat.

Go уделяет особое внимание типам структур, которые имеют поля, тип которых является другим типом структуры, таким же образом, как тип Boat имеет поле *Product в примере проекта. Вы можете увидеть эту специальную обработку в операторе цикла for, который отвечает за запись сведений о каждой Boat.

Go позволяет получить доступ к полям вложенного типа двумя способами. Первый — это традиционный подход к навигации по иерархии типов для достижения требуемого значения. Поле *Product является встроенным, что означает, что его имя соответствует его типу. Чтобы добраться до поля Name, я могу перемещаться по вложенному типу, например так:

```
...
fmt.Println("Conventional:",    b.Product.Name,    "Direct:",
b.Name)
...
```

Go также позволяет напрямую использовать вложенные типы полей, например:

```
...
fmt.Println("Conventional:",    b.Product.Name,    "Direct:",
b.Name)
...
```

Тип Boat не определяет поле Name, но его можно рассматривать так, как если бы оно было определено, благодаря функции прямого доступа. Это известно как *продвижение полей*, и Go по существу выравнивает типы, так что тип Boat ведет себя так, как будто он определяет поля, предоставляемые вложенным типом Product, как показано на рисунке 13-3.

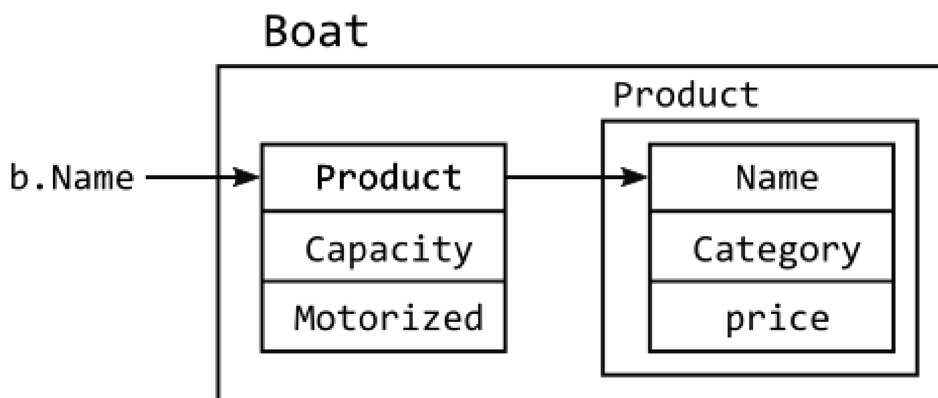

Рисунок 13-3 Продвигаемые поля

Скомпилируйте и выполните проект, и вы увидите, что значения, получаемые обоими подходами, одинаковы:

```
Conventional: Kayak Direct: Kayak
Conventional: Canoe Direct: Canoe
Conventional: Tender Direct: Tender
```

Также продвигаются методы, так что методы, определенные для вложенного типа, могут быть вызваны из включающего типа, как показано в листинге 13-9.

```
package main

import (
    "fmt"
    "composition/store"
)
```

```
func main() {

    boats := []*store.Boat {
        store.NewBoat("Kayak", 275, 1, false),
        store.NewBoat("Canoe", 400, 3, false),
        store.NewBoat("Tender", 650.25, 2, true),
    }

    for _, b := range boats {
        fmt.Println("Boat:", b.Name, "Price:", b.Price(0.2))
    }
}
```

Листинг 13-9 Вызов метода в файле main.go в папке composition

Если тип поля является значением, например `Product`, то будут продвинуты любые методы, определенные с приемниками `Product` или `*Product`. Если тип поля является указателем, например `*Product`, то будут запрашиваться только методы с приемниками `*Product`.

Для типа `*Boat` не определен метод `Price`, но Go продвигает метод, определенный с помощью приемника `*Product`. Скомпилируйте и запустите проект, и вы получите следующий вывод:

```
Boat: Kayak Price: 330
Boat: Canoe Price: 480
Boat: Tender Price: 780.3
```

ПОНИМАНИЕ ПРОДВИГАЕМЫХ ПОЛЕЙ И ЛИТЕРАЛЬНОГО СИНТАКСИСА

Go применяет специальную обработку к продвинутым полям после создания значения структуры. Так, например, если я использую функцию `NewBoat` для создания такого значения:

```
...
boat := store.NewBoat("Kayak", 275, 1, false)
...
```

затем я могу читать и назначать значения продвигаемым полям, например:

```
...
```

```
boat.Name = "Green Kayak"
...
```

Но эта функция недоступна при использовании литерального синтаксиса для создания значений в первую очередь, а это означает, что я не могу заменить функцию NewBoat, например:

```
...
boat := store.Boat { Name: "Kayak", Category:
"Watersports",
    Capacity: 1, Motorized: false }
...
```

Компилятор не позволяет присваивать значения напрямую и сообщает об ошибке «неизвестное поле» при компиляции кода. Если вы используете литеральный синтаксис, вы должны присвоить значение вложенному полю, например:

```
...
boat := store.Boat { Product: &store.Product{ Name:
"Kayak",
    Category: "Watersports"}, Capacity: 1, Motorized: false
}
...
```

Как я объяснял в разделе «Создание цепочки вложенных типов», Go упрощает использование функции композиции для создания сложных типов, что делает литеральный синтаксис все более сложным в использовании и создает код, подверженный ошибкам и сложный в обслуживании. Я советую использовать функции-конструкторы и вызывать один конструктор из другого, как функция NewBoat вызывает функцию NewProduct в листинге 13-7.

Создание цепочки вложенных типов

Функцию композиции можно использовать для создания сложных цепочек вложенных типов, поля и методы которых повышаются до включающего типа верхнего уровня. Добавьте файл с именем rentboats.go в папку store с содержимым, показанным в листинге 13-10.

```
package store

type RentalBoat struct {
    *Boat
    IncludeCrew bool
}

func NewRentalBoat(name string, price float64, capacity int,
        motorized, crewed bool) *RentalBoat {
         return  &RentalBoat{NewBoat(name,  price,  capacity,
motorized), crewed}
}
```

Листинг 13-10 Содержимое файла Rentalboats.go в папке store

Тип RentalBoat составлен из типа *Boat, который, в свою очередь, составлен из типа *Product, образуя цепочку. Go выполняет продвижение, так что к полям, определенным всеми тремя типами в цепочке, можно получить прямой доступ, как показано в листинге 13-11.

```
package main

import (
    "fmt"
    "composition/store"
)

func main() {

    rentals := []*store.RentalBoat {
            store.NewRentalBoat("Rubber  Ring",  10,  1,  false,
false),
        store.NewRentalBoat("Yacht", 50000, 5, true, true),
         store.NewRentalBoat("Super Yacht", 100000, 15, true,
true),
    }

    for _, r := range rentals {
         fmt.Println("Rental Boat:", r.Name, "Rental Price:",
r.Price(0.2))
    }
}
```

Листинг 13-11 Доступ к вложенным полям непосредственно в файле main.go в папке composition

Go продвигает поля из вложенных типов Boat и Product, чтобы к ним можно было получить доступ через тип RentalBoat верхнего уровня, который позволяет читать поле Name в листинге 13-11. Методы также повышаются до типа верхнего уровня, поэтому я могу использовать метод Price, даже если он определен для типа *Product, который находится в конце цепочки. Код в листинге 13-11 выдает следующий результат при компиляции и выполнении:

```
Rental Boat: Rubber Ring Rental Price: 12
Rental Boat: Yacht Rental Price: 60000
Rental Boat: Super Yacht Rental Price: 120000
```

Использование нескольких вложенных типов в одной и той же структуре

Типы могут определять несколько полей структуры, и Go будет продвигать поля для всех из них. В листинге 13-12 определяется новый тип, описывающий экипаж лодки и использующий его в качестве типа для поля в другой структуре.

```
package store

type Crew struct {
    Captain, FirstOfficer string
}

type RentalBoat struct {
    *Boat
    IncludeCrew bool
    *Crew
}

func NewRentalBoat(name string, price float64, capacity int,
        motorized, crewed bool, captain, firstOfficer string)
*RentalBoat {
        return  &RentalBoat{NewBoat(name,  price,  capacity,
motorized), crewed,
        &Crew{captain, firstOfficer}}
}
```

Листинг 13-12 Определение нового типа в файле Rentalboats.go в папке store

Тип `RentalBoat` имеет поля `*Boat` и `*Crew`, а Go продвигает поля и методы из обоих вложенных типов, как показано в листинге 13-13.

```
package main

import (
    "fmt"
    "composition/store"
)

func main() {

    rentals := []*store.RentalBoat {
            store.NewRentalBoat("Rubber Ring", 10, 1, false,
false, "N/A", "N/A"),
           store.NewRentalBoat("Yacht", 50000, 5, true, true,
"Bob", "Alice"),
          store.NewRentalBoat("Super Yacht", 100000, 15, true,
true,
            "Dora", "Charlie"),
    }

    for _, r := range rentals {
         fmt.Println("Rental Boat:", r.Name, "Rental Price:",
r.Price(0.2),
            "Captain:", r.Captain)
    }
}
```

Листинг 13-13 Использование продвигаемых полей в файле main.go в папке composition

Скомпилируйте и выполните проект, и вы получите следующий вывод, показывающий добавление сведений о команде:

```
Rental Boat: Rubber Ring Rental Price: 12 Captain: N/A
Rental Boat: Yacht Rental Price: 60000 Captain: Bob
Rental Boat: Super Yacht Rental Price: 120000 Captain: Dora
```

Понимание, когда продвижение не может быть выполнено

Go может выполнять продвижение только в том случае, если в охватывающем типе нет поля или метода, определенного с тем же

именем, что может привести к неожиданным результатам. Добавьте файл с именем specialdeal.go в папку store с кодом, показанным в листинге 13-14.

```
package store

type SpecialDeal struct {
    Name string
    *Product
    price float64
}

func  NewSpecialDeal(name  string,  p  *Product,  discount
float64) *SpecialDeal {
    return &SpecialDeal{ name, p, p.price - discount }
}

func  (deal  *SpecialDeal  )  GetDetails()  (string,  float64,
float64) {
    return deal.Name, deal.price, deal.Price(0)
}
```

Листинг 13-14 Содержимое файла specialdeal.go в папке store

Тип SpecialDeal определяет встроенное поле *Product. Эта комбинация приводит к дублированию полей, поскольку оба типа определяют поля Name и price. Существует также функция-конструктор и метод GetDetails, который возвращает значения полей Name и price, а также результат метода Price, который вызывается с нулем в качестве аргумента, чтобы упростить следование примеру. В листинге 13-15 новый тип используется для демонстрации того, как обрабатывается продвижение.

```
package main

import (
    "fmt"
    "composition/store"
)

func main() {
```

```
    product := store.NewProduct("Kayak", "Watersports", 279)

     deal := store.NewSpecialDeal("Weekend Special", product,
50)

    Name, price, Price := deal.GetDetails()

    fmt.Println("Name:", Name)
    fmt.Println("Price field:", price)
    fmt.Println("Price method:", Price)
}
```

Листинг 13-15 Использование нового типа в файле main.go в папке composition

Этот листинг создает `*Product`, который затем используется для создания `*SpecialDeal`. Вызывается метод `GetDetails`, и записываются три возвращаемых им результата. Скомпилируйте и запустите код, и вы увидите следующий вывод:

```
Name: Weekend Special
Price field: 229
Price method: 279
```

Первые два результата вполне ожидаемы: поля `Name` и `price` из типа `Product` не продвигаются, поскольку в типе `SpecialDeal` есть поля с одинаковыми именами.

Третий результат может вызвать проблемы. Go может продвигать метод `Price`, но когда он вызывается, он использует поле `price` из `Product`, а не из `SpecialDeal`.

Легко забыть, что продвижение полей и методов — это просто функция удобства. Этот оператор в листинге 13-14:

```
...
return deal.Name, deal.price, deal.Price(0)
...
```

это более краткий способ выразить это утверждение:

```
...
return deal.Name, deal.price, deal.Product.Price(0)
...
```

Когда метод вызывается через его поле структуры, становится ясно, что результат вызова метода Price не будет использовать поле price, определенное типом SpecialDeal.

Если я хочу иметь возможность вызвать метод Price и получить результат, основанный на поле SpecialDeal.price, я должен определить новый метод, как показано в листинге 13-16.

```
package store

type SpecialDeal struct {
    Name string
    *Product
    price float64
}

func NewSpecialDeal(name string, p *Product, discount
float64) *SpecialDeal {
    return &SpecialDeal{ name, p, p.price - discount }
}

func (deal *SpecialDeal ) GetDetails() (string, float64,
float64) {
    return deal.Name, deal.price, deal.Price(0)
}

func (deal *SpecialDeal) Price(taxRate float64) float64 {
    return deal.price
}
```

Листинг 13-16 Определение метода в файле specialdeal.go в папке store

Новый метод Price не позволяет Go продвигать метод Product и выдает следующий результат при компиляции и выполнении проекта:

```
Name: Weekend Special
Price field: 229
Price method: 229
```

Понимание неоднозначности продвижения

Связанная проблема возникает, когда два встроенных поля используют одни и те же имена полей или методов, как показано в листинге 13-17.

```
package main

import (
    "fmt"
    "composition/store"
)

func main() {

    kayak := store.NewProduct("Kayak", "Watersports", 279)

    type OfferBundle struct {
        *store.SpecialDeal
        *store.Product
    }

    bundle := OfferBundle {
        store.NewSpecialDeal("Weekend Special", kayak, 50),
               store.NewProduct("Lifrejacket",  "Watersports",
48.95),
    }

    fmt.Println("Price:", bundle.Price(0))
}
```

Листинг 13-17 Неоднозначный метод в файле main.go в папке composition

Тип `OfferBundle` имеет два встроенных поля, каждое из которых имеет метод `Price`. Go не может различать методы, и код в листинге 13-17 выдает следующую ошибку при компиляции:

```
.\main.go:22:33: ambiguous selector bundle.Price
```

Понимание композиции и интерфейсов

Составление типов упрощает создание специализированных функций без дублирования кода, необходимого для более общего типа, так что, например, тип `Boat` в проекте может опираться на функциональные возможности, предоставляемые типом `Product`.

Это может показаться похожим на написание классов на других языках, но есть важное отличие, заключающееся в том, что каждый составленный тип отличается и не может использоваться там, где

требуются типы, из которых он составлен, как показано в листинге 13-18.

```
package main

import (
    "fmt"
    "composition/store"
)

func main() {

    products := map[string]*store.Product {
        "Kayak": store.NewBoat("Kayak", 279, 1, false),
           "Ball": store.NewProduct("Soccer Ball", "Soccer",
19.50),
    }

    for _, p := range products {
         fmt.Println("Name:", p.Name, "Category:", p.Category,
"Price:", p.Price(0.2))
    }
}
```

Листинг 13-18 Смешивание типов в файле main.go в папке composition

Компилятор Go не позволит использовать Boat в качестве значения в срезе, где требуются значения Product. В таких языках, как C# или Java, это было бы разрешено, потому что Boat был бы подклассом Product, но Go не так работает с типами. Если вы скомпилируете проект, вы получите следующую ошибку:

```
.\main.go:11:9: cannot use store.NewBoat("Kayak", 279, 1,
false) (type *store.Boat) as type *store.Product in map value
```

Использование композиции для реализации интерфейсов

Как я объяснял в главе 11, Go использует интерфейсы для описания методов, которые могут быть реализованы несколькими типами.

Go учитывает продвигаемые методы при определении того, соответствует ли тип интерфейсу, что позволяет избежать необходимости дублировать методы, которые уже присутствуют во

встроенном поле. Чтобы увидеть, как это работает, добавьте файл с именем forsale.go в папку store с содержимым, показанным в листинге 13-19.

```
package store

type ItemForSale interface {
    Price(taxRate float64) float64
}
```

Листинг 13-19 Содержимое файла forsale.go в папке store

Тип ItemForSale — это интерфейс, определяющий единственный метод с именем Price, с одним параметром float64 и одним результатом float64. В листинге 13-20 тип интерфейса используется для создания карты, которая заполняется элементами, соответствующими интерфейсу.

```
package main

import (
    "fmt"
    "composition/store"
)

func main() {

    products := map[string]store.ItemForSale {
        "Kayak": store.NewBoat("Kayak", 279, 1, false),
           "Ball": store.NewProduct("Soccer Ball", "Soccer",
19.50),
    }

    for key, p := range products {
        fmt.Println("Key:", key, "Price:", p.Price(0.2))
    }
}
```

Листинг 13-20 Использование интерфейса в файле main.go в папке composition

Изменение карты таким образом, чтобы она использовала интерфейс, позволяет мне сохранять значения Product и Boat. Тип Product напрямую соответствует интерфейсу ItemForSale, поскольку

существует метод Price, который соответствует сигнатуре, указанной интерфейсом, и имеет приемник *Product.

Не существует метода Price, принимающего приемник *Boat, но Go учитывает метод Price, продвигаемый из встроенного поля типа Boat, который он использует для удовлетворения требований интерфейса. Скомпилируйте и запустите проект, и вы получите следующий вывод:

```
Key: Kayak Price: 334.8
Key: Ball Price: 23.4
```

Понимание ограничения переключения типа

Интерфейсы могут указывать только методы, поэтому при записи вывода я использовал ключ, используемый для хранения значений в карте в листинге 13-20. В главе 11 я объяснил, что операторы switch могут использоваться для получения доступа к базовым типам, но это не работает так, как можно было бы ожидать, как показано в листинге 13-21.

```
package main

import (
    "fmt"
    "composition/store"
)

func main() {

    products := map[string]store.ItemForSale {
        "Kayak": store.NewBoat("Kayak", 279, 1, false),
           "Ball": store.NewProduct("Soccer Ball", "Soccer",
19.50),
    }

    for key, p := range products {
        switch item := p.(type) {
            case *store.Product, *store.Boat:
                 fmt.Println("Name:", item.Name, "Category:",
item.Category,
                     "Price:", item.Price(0.2))
            default:
```

```
                    fmt.Println("Key:", key, "Price:",
p.Price(0.2))
        }

    }
}
```

Листинг 13-21 Доступ к базовому типу в файле main.go в папке composition

Оператор `case` в листинге 13-21 указывает `*Product` и `*Boat`, что приводит к сбою компилятора со следующей ошибкой:

```
.\main.go:21:42: item.Name undefined (type store.ItemForSale
has no field or method Name)
.\main.go:21:66: item.Category undefined (type
store.ItemForSale has no field or method Category)
```

Эта проблема заключается в том, что операторы `case`, которые определяют несколько типов, будут соответствовать значениям всех этих типов, но не будут выполнять утверждение типа. Для листинга 13-21 это означает, что значения `*Product` и `*Boat` будут соответствовать оператору `case`, но тип переменной `item` будет `ItemForSale`, поэтому компилятор выдает ошибку. Вместо этого должны использоваться дополнительные утверждения типа или однотипные операторы `case`, как показано в листинге 13-22.

```
package main

import (
    "fmt"
    "composition/store"
)

func main() {

    products := map[string]store.ItemForSale {
        "Kayak": store.NewBoat("Kayak", 279, 1, false),
           "Ball": store.NewProduct("Soccer Ball", "Soccer",
19.50),
    }

    for key, p := range products {
```

```
        switch item := p.(type) {
            case *store.Product:
                 fmt.Println("Name:", item.Name, "Category:",
item.Category,
                   "Price:", item.Price(0.2))
            case *store.Boat:
                 fmt.Println("Name:", item.Name, "Category:",
item.Category,
                   "Price:", item.Price(0.2))
            default:
                          fmt.Println("Key:", key, "Price:",
p.Price(0.2))
        }
    }
}
```

Листинг 13-22 Использование отдельных операторов case в файле main.go в папке composition

Утверждение типа выполняется оператором `case`, когда указан один тип, хотя это может привести к дублированию при обработке каждого типа. Код в листинге 13-22 выдает следующий результат, когда проект компилируется и выполняется:

```
Name: Kayak Category: Watersports Price: 334.8
Name: Soccer Ball Category: Soccer Price: 23.4
```

Альтернативным решением является определение методов интерфейса, обеспечивающих доступ к значениям свойств. Это можно сделать, добавив методы к существующему интерфейсу или определив отдельный интерфейс, как показано в листинге 13-23.

```
package store

type Product struct {
    Name, Category string
    price float64
}

func   NewProduct(name,   category   string,   price   float64)
*Product {
    return &Product{ name, category, price }
}
```

```
func (p *Product) Price(taxRate float64) float64 {
    return p.price + (p.price * taxRate)
}

type Describable interface  {
    GetName() string
    GetCategory() string
}

func (p *Product) GetName() string {
    return p.Name
}

func (p *Product) GetCategory() string {
    return p.Category
}
```

Листинг 13-23 Определение интерфейса в файле product.go в папке store

Интерфейс `Describable` определяет методы `GetName` и `GetCategory`, которые реализованы для типа `*Product`. В листинге 13-24 оператор `switch` изменен так, что вместо полей используются интерфейсы.

```
package main

import (
    "fmt"
    "composition/store"
)

func main() {

    products := map[string]store.ItemForSale {
        "Kayak": store.NewBoat("Kayak", 279, 1, false),
           "Ball": store.NewProduct("Soccer Ball", "Soccer",
19.50),
    }

    for key, p := range products {

        switch item := p.(type) {
            case store.Describable:
```

```
                    fmt.Println("Name:", item.GetName(),
"Category:", item.GetCategory(),
                                        "Price:", item.
(store.ItemForSale).Price(0.2))
            default:
                    fmt.Println("Key:", key, "Price:",
p.Price(0.2))
        }
    }
}
```

Листинг 13-24 Использование интерфейсов в файле main.go в папке composition

Это работает, но для доступа к методу `Price` требуется утверждение типа интерфейса `ItemForSale`. Это проблематично, поскольку тип может реализовать интерфейс `Describable`, но не интерфейс `ItemForSale`, что может вызвать ошибку времени выполнения. Я мог бы справиться с утверждением типа, добавив метод `Price` в интерфейс `Describable`, но есть альтернатива, которую я опишу в следующем разделе. Скомпилируйте и запустите проект, и вы увидите следующий вывод:

```
Name: Kayak Category: Watersports Price: 334.8
Name: Soccer Ball Category: Soccer Price: 23.4
```

Составление интерфейсов

Go позволяет составлять интерфейсы из других интерфейсов, как показано в листинге 13-25.

```
package store

type Product struct {
    Name, Category string
    price float64
}

func NewProduct(name, category string, price float64)
*Product {
    return &Product{ name, category, price }
}

func (p *Product) Price(taxRate float64) float64 {
```

```
    return p.price + (p.price * taxRate)
}

type Describable interface  {
    GetName() string
    GetCategory() string
    ItemForSale
}

func (p *Product) GetName() string {
    return p.Name
}

func (p *Product) GetCategory() string {
    return p.Category
}
```

Листинг 13-25 Составление интерфейса в файле product.go в папке store

Один интерфейс может заключать в себе другой, в результате чего типы должны реализовывать все методы, определенные включающим и вложенным интерфейсами. Интерфейсы проще, чем структуры, и нет полей или методов для продвижения. Результатом составления интерфейсов является объединение методов, определенных включающим и вложенным типами. В этом примере объединение означает, что для реализации интерфейса `Describable` требуются методы `GetName`, `GetCategory` и `Price`. Методы `GetName` и `GetCategory`, определенные непосредственно интерфейсом `Describable`, объединяются с методом `Price`, определенным интерфейсом `ItemForSale`.

Изменение интерфейса `Describable` означает, что утверждение типа, которое я использовал в предыдущем разделе, больше не требуется, как показано в листинге 13-26.

```
package main

import (
    "fmt"
    "composition/store"
)

func main() {
```

```
    products := map[string]store.ItemForSale {
        "Kayak": store.NewBoat("Kayak", 279, 1, false),
           "Ball": store.NewProduct("Soccer Ball", "Soccer",
19.50),
    }

    for key, p := range products {

        switch item := p.(type) {
            case store.Describable:
                         fmt.Println("Name:", item.GetName(),
"Category:", item.GetCategory(),
                     "Price:", item.Price(0.2))
            default:
                          fmt.Println("Key:", key, "Price:",
p.Price(0.2))
        }
    }
}
```

Листинг 13-26 Удаление утверждения в файле main.go в папке composition

Значение любого типа, реализующего интерфейс `Describable`, должно иметь метод `Price` из-за композиции, выполненной в листинге 13-25, что означает, что метод может быть вызван без потенциально рискованного утверждения типа. Скомпилируйте и запустите проект, и вы получите следующий вывод:

```
Name: Kayak Category: Watersports Price: 334.8
Name: Soccer Ball Category: Soccer Price: 23.4
```

Резюме

В этой главе я описываю способ компоновки типов Go для создания более сложной функциональности, предоставляя альтернативу основанному на наследовании подходу, принятому в других языках. В следующей главе я опишу горутины и каналы, которые являются функциями Go для управления параллелизмом.

14. Использование горутин и каналов

Go имеет отличную поддержку для написания параллельных приложений, используя функции, которые проще и интуитивно понятнее, чем любой другой язык, который я использовал. В этой главе я описываю использование *горутин*, позволяющих выполнять функции одновременно, и *каналов*, по которым горутины могут асинхронно выдавать результаты. Таблица 14-1 помещает горутины и каналы в контекст.

Таблица 14-1 Горутины и каналы в контексте

Вопрос	**Ответ**
Кто они такие?	Горутины — это легкие потоки, созданные и управляемые средой выполнения Go. Каналы — это конвейеры, передающие значения определенного типа.
Почему они полезны?	Горутины позволяют выполнять функции одновременно, без необходимости иметь дело со сложностями потоков операционной системы. Каналы позволяют горутинам асинхронно выдавать результаты.
Как они используются?	Горутины создаются с использованием ключевого слова `go`. Каналы определяются как типы данных.
Есть ли подводные камни или ограничения?	Необходимо соблюдать осторожность, чтобы управлять направлением каналов. Горутины, которые совместно используют данные, требуют дополнительных функций, которые описаны в главе 14.
Есть ли альтернативы?	Горутины и каналы — это встроенные функции параллелизма Go, но некоторые приложения могут полагаться на один поток выполнения, который создается по умолчанию для выполнения основной функции.

Таблица 14-2 суммирует содержание главы.

Таблица 14-2 Краткое содержание главы

Проблема	**Решение**	**Листинг**
Выполненить функции асинхронно	Создайте горутину	7
Получить результат из функции, выполняемой асинхронно	Использовать канал	10, 15, 16, 22–26
Отправка и получение значений с помощью канала	Используйте выражения со стрелками	11–13
Индицировать, что дальнейшие значения не будут передаваться по каналу	Используйте функцию закрытия	17–20

Проблема	Решение	Листинг
Перечислить значения, полученные из канала	Используйте цикл `for` с ключевым словом `range`	21
Отправка или получение значений с использованием нескольких каналов	Используйте оператор `select`	27–32

Подготовка к этой главе

Чтобы подготовиться к этой главе, откройте новую командную строку, перейдите в удобное место и создайте каталог с именем `concurrency`. Запустите команду, показанную в листинге 14-1, чтобы создать файл модуля.

> **Подсказка**
> Вы можете загрузить пример проекта для этой главы — и для всех остальных глав этой книги — с `https://github.com/apress/pro-go`. См. Главу 2 о том, как получить помощь, если у вас возникнут проблемы с запуском примеров.

```
go mod init concurrency
```

Листинг 14-1 Инициализация модуля

Добавьте файл с именем `product.go` в папку параллелизма с содержимым, показанным в листинге 14-2.

```
package main

import "strconv"

type Product struct {
    Name, Category string
    Price float64
}

var ProductList = []*Product {
    { "Kayak", "Watersports", 279 },
    { "Lifejacket", "Watersports", 49.95 },
    { "Soccer Ball", "Soccer", 19.50 },
    { "Corner Flags", "Soccer", 34.95 },
    { "Stadium", "Soccer", 79500 },
    { "Thinking Cap", "Chess", 16 },
```

```
    { "Unsteady Chair", "Chess", 75 },
    { "Bling-Bling King", "Chess", 1200 },
}

type ProductGroup []*Product

type ProductData = map[string]ProductGroup

var Products =  make(ProductData)

func ToCurrency(val float64) string {
    return "$" + strconv.FormatFloat(val, 'f', 2, 64)
}

func init() {
    for _, p := range ProductList {
        if _, ok := Products[p.Category]; ok {
            Products[p.Category] = append(Products[p.Category],
p)
        } else {
            Products[p.Category] = ProductGroup{ p }
        }
    }
}
```

Листинг 14-2 Содержимое файла product.go в папке concurrency

Этот файл определяет настраиваемый тип с именем `Product`, а также псевдонимы типов, которые я использую для создания карты, которая упорядочивает продукты по категориям. Я использую тип `Product` в срезе и карте и полагаюсь на функцию инициализации, описанную в главе 12, для заполнения карты содержимым среза, который сам заполняется с использованием литерального синтаксиса. Этот файл также содержит функцию `ToCurrency`, которая форматирует значения `float64` в строки долларовой валюты, которые я буду использовать для форматирования результатов в этой главе.

Добавьте файл с именем `operations.go` в папку `concurrency` с содержимым, показанным в листинге 14-3.

```
package main

import "fmt"

func CalcStoreTotal(data ProductData) {
```

```
    var storeTotal float64
    for category, group := range data {
        storeTotal += group.TotalPrice(category)
    }
    fmt.Println("Total:", ToCurrency(storeTotal))
}

func (group ProductGroup) TotalPrice(category string, ) (total
float64) {
    for _, p := range group {
        total += p.Price
    }
    fmt.Println(category, "subtotal:", ToCurrency(total))
    return
}
```

Листинг 14-3 Содержимое файла operations.go в папке concurrency

В этом файле определяются методы, работающие с псевдонимами типов, созданными в файле `product.go`. Как я объяснял в главе 11, методы могут быть определены только для типов, созданных в том же пакете, что означает, что я не могу определить метод, например, для типа `[]*Product`, но я могу создать псевдоним для этого типа и используйте псевдоним в качестве приемника метода.

Добавьте файл с именем `main.go` в папку параллелизма с содержимым, показанным в листинге 14-4.

```
package main

import "fmt"

func main() {

    fmt.Println("main function started")
    CalcStoreTotal(Products)
    fmt.Println("main function complete")
}
```

Листинг 14-4 Содержимое файла main.go в папке concurrency

Используйте командную строку для запуска команды, показанной в листинге 14-5, в папке `concurrency`.

```
go run .
```

Листинг 14-5 Запуск примера проекта

Код будет скомпилирован и выполнен, что приведет к следующему выводу:

```
main function started
Watersports subtotal: $328.95
Soccer subtotal: $79554.45
Chess subtotal: $1291.00
Total: $81174.40
main function complete
```

Понимание того, как Go выполняет код

Ключевым строительным блоком для выполнения программы Go является *горутина*, представляющая собой облегченный поток, созданный средой выполнения Go. Все программы Go используют по крайней мере одну горутину, потому что именно так Go выполняет код в `main` функции. Когда скомпилированный код Go выполняется, среда выполнения создает горутину, которая начинает выполнять операторы в точке входа, которая является `main` функцией в основном пакете. Каждый оператор в `main` функции выполняется в том порядке, в котором они определены. Горутина продолжает выполнять операторы, пока не достигнет конца основной функции, после чего приложение завершает работу.

Горутина выполняет каждый оператор в `main` функции *синхронно*, что означает, что она ожидает завершения оператора, прежде чем перейти к следующему оператору. Операторы в функции `main` могут вызывать другие функции, использовать циклы `for`, создавать значения и использовать все другие возможности, описанные в этой книге. Основная горутина будет проходить через код, следуя своему пути, выполняя по одному оператору за раз.

Для примера приложения это означает, что карта продуктов обрабатывается последовательно, так что каждая категория продуктов обрабатывается по очереди, а внутри каждой категории обрабатывается каждый продукт, как показано на рисунке 14-1.

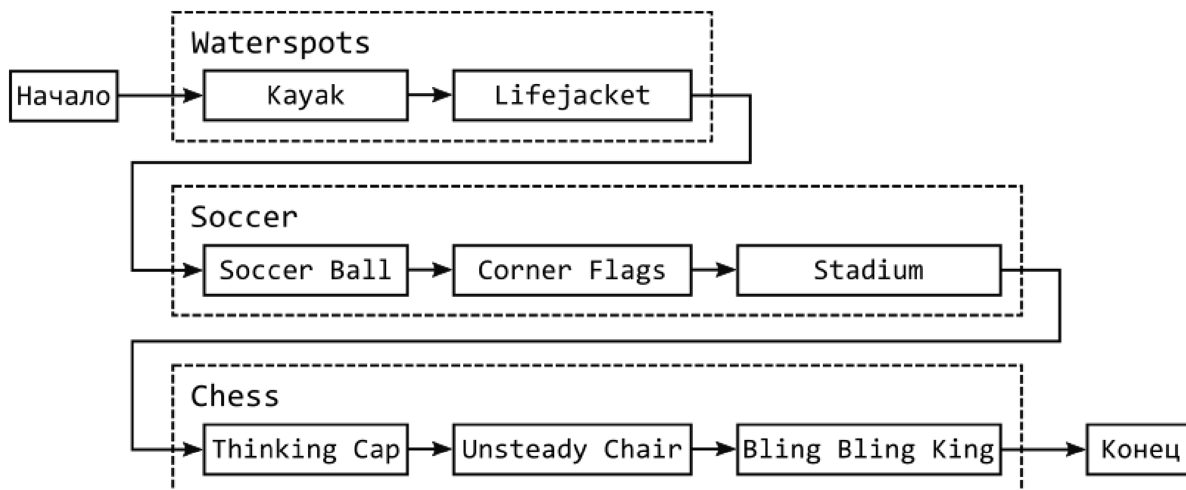

Рисунок 14-1 Последовательное исполнение

В листинге 14-6 добавлен оператор, который записывает сведения о каждом продукте по мере его обработки, что демонстрирует поток, показанный на рисунке.

```
package main

import "fmt"

func CalcStoreTotal(data ProductData) {
    var storeTotal float64
    for category, group := range data {
        storeTotal += group.TotalPrice(category)
    }
    fmt.Println("Total:", ToCurrency(storeTotal))
}

func (group ProductGroup) TotalPrice(category string) (total
float64) {
    for _, p := range group {
        fmt.Println(category, "product:", p.Name)
        total += p.Price
    }
    fmt.Println(category, "subtotal:", ToCurrency(total))
    return
}
```

Листинг 14-6 Добавление оператора в файл operations.go в папке concurrency

Скомпилируйте и выполните код, и вы увидите вывод, подобный следующему:

```
main function started
Soccer product: Soccer Ball
Soccer product: Corner Flags
Soccer product: Stadium
Soccer subtotal: $79554.45
Chess product: Thinking Cap
Chess product: Unsteady Chair
Chess product: Bling-Bling King
Chess subtotal: $1291.00
Watersports product: Kayak
Watersports product: Lifejacket
Watersports subtotal: $328.95
Total: $81174.40
main function complete
```

Вы можете увидеть разные результаты в зависимости от порядка, в котором ключи извлекаются из карты, но важно то, что все продукты в категории обрабатываются до того, как выполнение перейдет к следующей категории.

Преимущества синхронного выполнения заключаются в простоте и согласованности — поведение синхронного кода легко понять и предсказать. Недостатком является то, что он может быть неэффективным. Последовательная работа с девятью элементами данных, как в примере, не представляет никаких проблем, но большинство реальных проектов имеют большие объемы данных или требуют выполнения других задач, а это означает, что последовательное выполнение занимает слишком много времени и не дает результатов достаточно быстро.

Создание дополнительных горутин

Go позволяет разработчику создавать дополнительные горутины, которые выполняют код одновременно с `main` горутиной. Go упрощает создание новых горутин, как показано в листинге 14-7.

```
package main

import "fmt"

func CalcStoreTotal(data ProductData) {
    var storeTotal float64
    for category, group := range data {
        go group.TotalPrice(category)
```

```
    }
    fmt.Println("Total:", ToCurrency(storeTotal))
}

func (group ProductGroup) TotalPrice(category string) (total float64) {
    for _, p := range group {
        fmt.Println(category, "product:", p.Name)
        total += p.Price
    }
    fmt.Println(category, "subtotal:", ToCurrency(total))
    return
}
```

Листинг 14-7 Создание подпрограмм Go в файле operations.go в папке concurrency

Горутина создается с использованием ключевого слова `go`, за которым следует функция или метод, которые должны выполняться асинхронно, как показано на рисунке 14-2.

Рисунок 14-2 Горутина

Когда среда выполнения Go встречает ключевое слово `go`, она создает новую горутину и использует ее для выполнения указанной функции или метода.

Это изменяет выполнение программы, потому что в любой момент существует несколько горутин, каждая из которых выполняет свой собственный набор операторов. Эти операторы выполняются *конкурентно*, что означает, что они выполняются одновременно.

В примере горутина создается для каждого вызова метода `TotalPrice`, а это означает, что категории обрабатываются одновременно, как показано на рисунке 14-3.

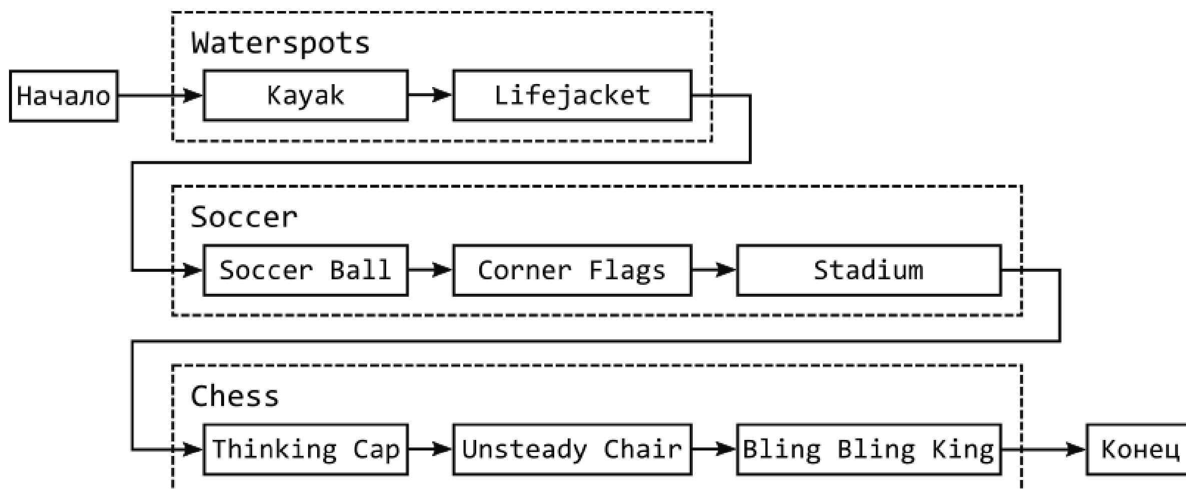

Рисунок 14-3 Параллельные вызовы функций

Подпрограммы Go упрощают вызов функций и методов, но изменение в листинге 14-7 привело к общей проблеме. Скомпилируйте и выполните проект, и вы получите следующие результаты:

```
main function started
Total: $0.00
main function complete
```

Вы можете увидеть немного другие результаты, которые могут включать промежуточные итоги по одной или нескольким категориям. Но, в большинстве случаев, вы увидите эти сообщения. Перед введением в код горутин метод `TotalPrice` вызывался так:

```
...
storeTotal += group.TotalPrice(category)
...
```

Это синхронный вызов функции. Он указывает среде выполнения выполнять операторы в методе `TotalPrice` один за другим и присваивать результат переменной с именем `storeTotal`. Выполнение не будет продолжаться до тех пор, пока не будут обработаны все операторы `TotalPrice`. Но в листинге 14-7 представлена горутина для выполнения функции, например:

```
...
go group.TotalPrice(category)
...
```

Этот оператор указывает среде выполнения выполнять операторы в методе TotalPrice с использованием новой горутины. Среда выполнения не ждет, пока горутина выполнит метод, и немедленно переходит к следующему оператору. В этом весь смысл горутин, потому что метод TotalPrice будет вызываться асинхронно, а это означает, что его операторы оцениваются одной горутиной в то же время, когда исходная горутина выполняет операторы в основной функции. Но, как я объяснял ранее, программа завершается, когда main горутина выполняет все операторы в main функции.

В результате программа завершается до того, как будут созданы горутины для завершения выполнения метода TotalPrice, поэтому промежуточные итоги отсутствуют.

Я объясню, как решить эту проблему, когда буду вводить дополнительные функции, но на данный момент все, что мне нужно сделать, это предотвратить завершение программы на время, достаточное для завершения горутин, как показано в листинге 14-8.

```
package main

import (
    "fmt"
    "time"
)

func main() {

    fmt.Println("main function started")
    CalcStoreTotal(Products)
    time.Sleep(time.Second * 5)
    fmt.Println("main function complete")
}
```

Листинг 14-8 Отложенный выход программы в файле main.go в папке concurrency

Пакет time является частью стандартной библиотеки и описан в главе 19. Пакет time предоставляет функцию Sleep, которая приостанавливает горутину, выполняющую инструкцию. Период ожидания указывается с помощью набора числовых констант, представляющих интервалы, так что time.Second представляет одну секунду и умножается на 5, чтобы создать пятисекундный период.

В этом случае он приостановит выполнение main горутины, что даст созданным горутинам время для выполнения метода TotalPrice. По

истечении периода ожидания `main` горутина возобновит выполнение операторов, достигнет конца функции и заставит программу завершиться.

Скомпилируйте и запустите проект, и вы получите следующий вывод:

```
main function started
Watersports product: Kayak
Watersports product: Lifejacket
Watersports subtotal: $328.95
Soccer product: Soccer Ball
Soccer product: Corner Flags
Soccer product: Stadium
Soccer subtotal: $79554.45
Chess product: Thinking Cap
Chess product: Unsteady Chair
Chess product: Bling-Bling King
Chess subtotal: $1291.00
Total: $0.00
main function complete
```

Программа больше не существует раньше, но трудно быть уверенным, что горутины работают одновременно. Это потому, что пример настолько прост, что одна горутина может завершиться за небольшое количество времени, которое требуется Go для создания и запуска следующего. В листинге 14-9 я добавил еще одну паузу, которая замедлит выполнение метода `TotalPrice`, чтобы показать, как выполняется код. (Это то, чего вы не должны делать в реальном проекте, но это полезно для понимания того, как работают эти функции.)

```
package main

import (
    "fmt"
    "time"
)

func CalcStoreTotal(data ProductData) {
    var storeTotal float64
    for category, group := range data {
        go group.TotalPrice(category)
    }
    fmt.Println("Total:", ToCurrency(storeTotal))
}
```

```
func (group ProductGroup) TotalPrice(category string) (total
float64) {
    for _, p := range group {
        fmt.Println(category, "product:", p.Name)
        total += p.Price
        time.Sleep(time.Millisecond * 100)
    }
    fmt.Println(category, "subtotal:", ToCurrency(total))
    return
}
```

Листинг 14-9 Добавление оператора Sleep в файл operations.go в папке concurrency

Новый оператор добавляет 100 миллисекунд к каждой итерации цикла `for` в методе `TotalPrice`. Скомпилируйте и выполните код, и вы увидите вывод, подобный следующему:

```
main function started
Total: $0.00
Soccer product: Soccer Ball
Watersports product: Kayak
Chess product: Thinking Cap
Chess product: Unsteady Chair
Watersports product: Lifejacket
Soccer product: Corner Flags
Chess product: Bling-Bling King
Soccer product: Stadium
Watersports subtotal: $328.95
Soccer subtotal: $79554.45
Chess subtotal: $1291.00
main function complete
```

Вы можете увидеть другой порядок результатов, но ключевым моментом является то, что сообщения для разных категорий чередуются, показывая, что данные обрабатываются параллельно. (Если изменение в листинге 14-9 не дает ожидаемых результатов, возможно, вам придется увеличить паузу, введенную функцией `time.Sleep`.)

Возврат результатов из горутин

Когда я создавал горутины в листинге 14-7, я изменил способ вызова метода `TotalPrice`. Изначально код выглядел так:

```
...
```

```
storeTotal += group.TotalPrice(category)
...
```

Но когда я представил подпрограмму Go, я изменил утверждение на следующее:

```
...
go group.TotalPrice(category)
...
```

Я получил асинхронное выполнение, но потерял результат метода, поэтому вывод из листинга 14-9 включает в себя нулевой результат для общего итога:

```
...
Total: $0.00
...
```

Получение результата от функции, которая выполняется асинхронно, может быть сложным, поскольку требует координации между горутиной, которая создает результат, и горутиной, которая использует результат.

Чтобы решить эту проблему, Go предоставляет *каналы*, которые являются магистралями, по которым данные могут быть отправлены и получены. Я собираюсь ввести канал в пример поэтапно, начиная с листинга 14-10, что означает, что пример не будет компилироваться, пока процесс не будет завершен.

```
package main

import (
    "fmt"
    "time"
)

func CalcStoreTotal(data ProductData) {
    var storeTotal float64
    var channel chan float64 = make(chan float64)
    for category, group := range data {
        go group.TotalPrice(category)
    }
    fmt.Println("Total:", ToCurrency(storeTotal))
}
```

```
func  (group  ProductGroup)  TotalPrice(category  string)  (total
float64) {
    for _, p := range group {
        fmt.Println(category, "product:", p.Name)
        total += p.Price
        time.Sleep(time.Millisecond * 100)
    }
    fmt.Println(category, "subtotal:", ToCurrency(total))
    return
}
```

Листинг 14-10 Определение канала в файле operations.go в папке concurrency

Каналы строго типизированы, что означает, что они будут нести значения указанного типа или интерфейса. Тип канала — это ключевое слово `chan`, за которым следует тип, который будет передавать канал, как показано на рисунке 14-4. Каналы создаются с помощью встроенной функции `make` с указанием типа канала.

Рисунок 14-4 Определение канала

Я использовал полный синтаксис объявления переменных в этом листинге, чтобы подчеркнуть тип, которым является `chan float64`, что означает канал, который будет передавать значения `float64`.

> Примечание Пакет `sync` предоставляет функции для управления горутинами, которые совместно используют данные, как описано в главе 30.

Отправка результата с использованием канала

Следующим шагом является обновление метода `TotalPrice`, чтобы он отправлял свой результат по каналу, как показано в листинге 14-11.

```
package main

import (
    "fmt"
    "time"
```

```
)

func CalcStoreTotal(data ProductData) {
    var storeTotal float64
    var channel chan float64 = make(chan float64)
    for category, group := range data {
        go group.TotalPrice(category)
    }
    fmt.Println("Total:", ToCurrency(storeTotal))
}

func (group ProductGroup) TotalPrice(category string,
resultChannel chan float64) {
    var total float64
    for _, p := range group {
        fmt.Println(category, "product:", p.Name)
        total += p.Price
        time.Sleep(time.Millisecond * 100)
    }
    fmt.Println(category, "subtotal:", ToCurrency(total))
    resultChannel <- total
}
```

Листинг 14-11 Использование канала для отправки результата в файл operations.go в папке concurrency

Первое изменение заключается в удалении обычного результата и добавлении параметра `chan float64`, тип которого соответствует каналу, созданному в листинге 14-10. Я также определил переменную с именем `total`, которая ранее не требовалась, потому что функция имела именованный результат.

Другое изменение демонстрирует, как результат отправляется с использованием канала. Задается канал, за которым следует стрелка направления, выраженная символами `<` и `-`, а затем значение, как показано на рисунке 14-5.

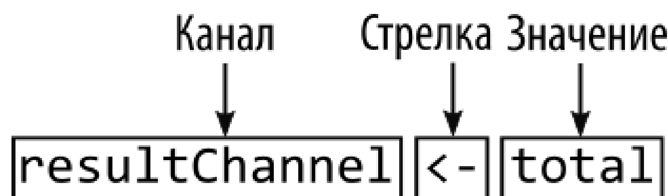

Рисунок 14-5 Отправка результата

Этот оператор отправляет значение total через канал resultChannel, что делает его доступным для получения в другом месте приложения. Обратите внимание, что когда значение отправляется через канал, отправителю не нужно знать, как это значение будет получено и использовано, точно так же, как обычная синхронная функция не знает, как будет использоваться ее результат.

Получение результата с использованием канала

Синтаксис стрелки используется для получения значения из канала, что позволит функции CalcStoreTotal получать данные, отправленные методом TotalPrice, как показано в листинге 14-12.

```
package main

import (
    "fmt"
    "time"
)

func CalcStoreTotal(data ProductData) {
    var storeTotal float64
    var channel chan float64 = make(chan float64)
    for category, group := range data {
        go group.TotalPrice(category, channel)
    }
    for i := 0; i < len(data); i++ {
        storeTotal += <- channel
    }
    fmt.Println("Total:", ToCurrency(storeTotal))
}

func    (group    ProductGroup)    TotalPrice(category    string,
resultChannel chan float64)  {
    var total float64
    for _, p := range group {
        fmt.Println(category, "product:", p.Name)
        total += p.Price
        time.Sleep(time.Millisecond * 100)
    }
    fmt.Println(category, "subtotal:", ToCurrency(total))
    resultChannel <- total
}
```

Листинг 14-12 Получение результата в файлеoperations.go в папке concurrency

Стрелка помещается перед каналом для получения от него значения, как показано на рисунке 14-6, и полученное значение может использоваться как часть любого стандартного выражения Go, такого как операция +=, использованная в примере.

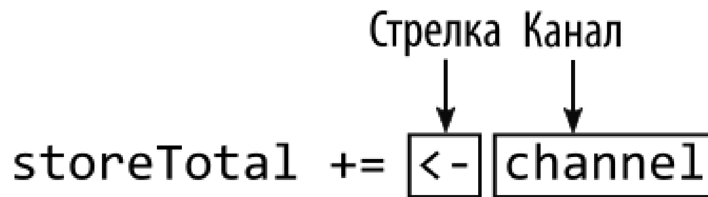

Рисунок 14-6 Получение результата

В этом примере я знаю, что количество результатов, которые можно получить от канала, точно соответствует количеству созданных мной горутин. И, поскольку я создал горутину для каждого ключа на карте, я могу использовать функцию len в цикле for для чтения всех результатов.

Каналы могут быть безопасно разделены между несколькими горутинами, и эффект изменений, сделанных в этом разделе, заключается в том, что подпрограммы Go, созданные для вызова метода TotalPrice, отправляют свои результаты через канал, созданный функцией CalcStoreTotal, где они принимаются и обрабатываются.

Получение из канала является блокирующей операцией, означающей, что выполнение не будет продолжаться до тех пор, пока не будет получено значение, что означает, что мне больше не нужно предотвращать завершение программы, как показано в листинге 14-13.

```
package main

import (
    "fmt"
    //"time"
)

func main() {

    fmt.Println("main function started")
    CalcStoreTotal(Products)
    //time.Sleep(time.Second * 5)
    fmt.Println("main function complete")
}
```

Листинг 14-13 Удаление инструкции Sleep в файле main.go в папке concurrency

Общий эффект от этих изменений заключается в том, что программа запускается и начинает выполнять операторы в основной функции. Это приводит к вызову функции `CalcStoreTotal`, которая создает канал и запускает несколько горутин. Горутины выполняют операторы в методе `TotalPrice`, который отправляет результат по каналу.

Горутина `main` продолжает выполнять инструкции в функции `CalcStoreTotal`, которая получает результаты по каналу. Эти результаты используются для создания общей суммы, которая записывается. Остальные операторы в `main` функции выполняются, и программа завершается.

Скомпилируйте и запустите проект, и вы увидите следующий вывод:

```
main function started
Watersports product: Kayak
Chess product: Thinking Cap
Soccer product: Soccer Ball
Soccer product: Corner Flags
Watersports product: Lifejacket
Chess product: Unsteady Chair
Chess product: Bling-Bling King
Soccer product: Stadium
Watersports subtotal: $328.95
Chess subtotal: $1291.00
Soccer subtotal: $79554.45
Total: $81174.40
main function complete
```

Вы можете увидеть сообщения, отображаемые в другом порядке, но важно отметить, что общая сумма рассчитывается правильно, например:

```
...
Total: $81174.40
...
```

Канал используется для координации горутин, позволяя `main` горутине ожидать отдельных результатов, полученных горутинами, созданными в функции `CalcStoreTotal`. На рисунке 14-7 показаны отношения между подпрограммами и каналом.

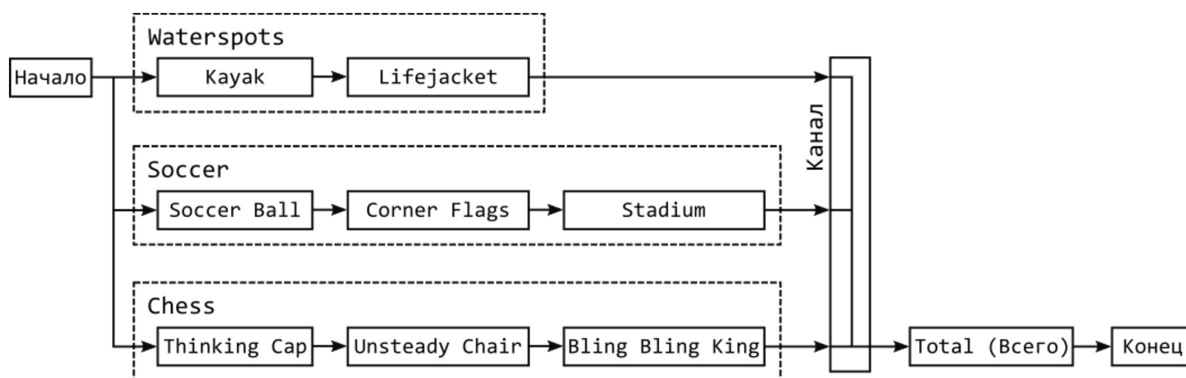

Рисунок 14-7 Координация с помощью канала

ИСПОЛЬЗОВАНИЕ АДАПТЕРОВ ДЛЯ АСИНХРОННОГО ВЫПОЛНЕНИЯ ФУНКЦИЙ

Не всегда возможно переписать существующие функции или методы для использования каналов, но асинхронно выполнять синхронные функции в оболочке несложно, например:

```
...
calcTax := func(price float64) float64 {
    return price + (price * 0.2)
}

wrapper := func (price float64, c chan float64)  {
    c <- calcTax(price)
}

resultChannel := make(chan float64)
go wrapper(275, resultChannel)
result := <- resultChannel
fmt.Println("Result:", result)
...
```

Функция `wrapper` получает канал, который используется для синхронной отправки значения, полученного при выполнении функции `calcTax`. Это можно выразить более кратко, определив функцию, не присваивая ее переменной, например:

```
...
go func (price float64, c chan float64) {
    c <- calcTax(price)
}(275, resultChannel)
...
```

Синтаксис немного неудобен, потому что аргументы, используемые для вызова функции, выражаются сразу после определения функции. Но результат тот же: синхронная функция может быть выполнена горутиной, а результат будет отправлен через канал.

Работа с каналами

В предыдущем разделе продемонстрировано основное использование каналов и их использование в координации горутин. В следующих разделах я опишу различные способы использования каналов для изменения того, как происходит координация, что позволяет адаптировать горутины к различным ситуациям.

Координация каналов

По умолчанию отправка и получение через канал являются блокирующими операциями. Это означает, что горутина, которая отправляет значение, не будет выполнять никаких дальнейших инструкций, пока другая горутина не получит значение из канала. Если вторая горутина отправляет значение, она будет заблокирована до тех пор, пока канал не будет очищен, что приведет к созданию очереди горутин, ожидающих получения значений. Это происходит и в другом направлении, поэтому горутины, которые получают значения, блокируются до тех пор, пока другая горутина не отправит их. В листинге 14-14 изменен способ отправки и получения значений в примере проекта, чтобы подчеркнуть это поведение.

```
package main

import (
    "fmt"
    "time"
)

func CalcStoreTotal(data ProductData) {
    var storeTotal float64
    var channel chan float64 = make(chan float64)
    for category, group := range data {
        go group.TotalPrice(category, channel)
    }
    time.Sleep(time.Second * 5)
```

```
    fmt.Println("-- Starting to receive from channel")
    for i := 0; i < len(data); i++ {
        fmt.Println("-- channel read pending")
        value :=  <- channel
        fmt.Println("-- channel read complete", value)
        storeTotal += value
        time.Sleep(time.Second)
    }
    fmt.Println("Total:", ToCurrency(storeTotal))
}

func    (group    ProductGroup)    TotalPrice(category    string,
resultChannel chan float64)  {
    var total float64
    for _, p := range group {
        //fmt.Println(category, "product:", p.Name)
        total += p.Price
        time.Sleep(time.Millisecond * 100)
    }
    fmt.Println(category, "channel sending", ToCurrency(total))
    resultChannel <- total
    fmt.Println(category, "channel send complete")
}
```

Листинг 14-14 Отправка и получение значений в файлеoperations.go в папке concurrency

Изменения вводят задержку после того, как `CalcStoreTotal` создаст горутины и получит первое значение из канала. Существует также задержка до и после получения каждого значения.

Эти задержки позволяют программам завершить свою работу и отправить значения по каналу до того, как какие-либо значения будут получены. Скомпилируйте и запустите проект, и вы увидите следующий вывод:

```
main function started
Watersports channel sending $328.95
Chess channel sending $1291.00
Soccer channel sending $79554.45
-- Starting to receive from channel
-- channel read pending
Watersports channel send complete
-- channel read complete 328.95
-- channel read pending
-- channel read complete 1291
Chess channel send complete
```

```
-- channel read pending
-- channel read complete 79554.45
Soccer channel send complete
Total: $81174.40
main function complete
```

Я склонен понимать параллельные приложения, визуализируя взаимодействие между людьми. Если у Боба есть сообщение для Алисы, поведение канала по умолчанию требует, чтобы Алиса и Боб договорились о месте встречи, и тот, кто доберется туда первым, будет ждать прибытия другого. Боб передаст сообщение Алисе только тогда, когда они оба будут присутствовать. Когда у Чарли также будет сообщение для Алисы, он выстроится в очередь за Бобом. Все терпеливо ждут, сообщения передаются только тогда, когда доступны и отправитель, и получатель, а сообщения обрабатываются последовательно.

Вы можете увидеть этот шаблон в выходных данных листинга 14-14. Горутины запускаются, обрабатывают свои данные и отправляют результаты по каналу:

```
...
Watersports channel sending $328.95
Chess channel sending $1291.00
Soccer channel sending $79554.45
...
```

Доступного получателя нет, поэтому горутины вынуждены ждать, формируя очередь терпеливых отправителей, пока получатель не начнет свою работу. При получении каждого значения отправляющая горутина разблокируется и может продолжать выполнять операторы в методе `TotalPrice`.

Использование буферизованного канала

Поведение канала по умолчанию может привести к вспышкам активности, поскольку горутины выполняют свою работу, за которыми следует длительный период простоя в ожидании получения сообщений. Это не влияет на пример приложения, потому что горутины завершают работу после получения их сообщений, но в реальном проекте горутины часто выполняют повторяющиеся задачи, и ожидание получателя может стать узким местом в производительности.

Альтернативным подходом является создание канала с буфером, который используется для приема значений от отправителя и их

сохранения до тех пор, пока получатель не станет доступным. Это делает отправку сообщения неблокирующей операцией, позволяя отправителю передать свое значение каналу и продолжить работу, не дожидаясь получателя. Это похоже на то, как Алиса имеет почтовый ящик на своем столе. Отправители приходят в офис Алисы и помещают свое сообщение в папку «Входящие», оставляя Алисе прочитать его, когда она будет готова. Но если почтовый ящик переполнен, им придется подождать, пока она не обработает часть своей очереди, прежде чем отправлять новое сообщение. В листинге 14-15 создается канал с буфером.

```
...
func CalcStoreTotal(data ProductData) {
    var storeTotal float64
    var channel chan float64 = make(chan float64, 2)
    for category, group := range data {
        go group.TotalPrice(category, channel)
    }
    time.Sleep(time.Second * 5)
    fmt.Println("-- Starting to receive from channel")
    for i := 0; i < len(data); i++ {
        fmt.Println("-- channel read pending")
        value :=  <- channel
        fmt.Println("-- channel read complete", value)
        storeTotal += value
        time.Sleep(time.Second)
    }
    fmt.Println("Total:", ToCurrency(storeTotal))
}
...
```

Листинг 14-15 Создание буферизованного канала в файлеoperations.go в папке concurrency

Размер буфера указывается в качестве аргумента функции `make`, как показано на рисунке 14-8.

Рисунок 14-8 Буферизованный канал

Для этого примера я установил размер буфера равным 2, что означает, что два отправителя смогут отправлять значения через канал, не

дожидаясь их получения. Любым последующим отправителям придется ждать, пока не будет получено одно из буферизованных сообщений. Вы можете увидеть это поведение, скомпилировав и выполнив проект, который выдает следующий результат:

```
main function started
Watersports channel sending $328.95
Watersports channel send complete
Chess channel sending $1291.00
Chess channel send complete
Soccer channel sending $79554.45
-- Starting to receive from channel
-- channel read pending
Soccer channel send complete
-- channel read complete 328.95
-- channel read pending
-- channel read complete 1291
-- channel read pending
-- channel read complete 79554.45
Total: $81174.40
main function complete
```

Вы можете видеть, что значения, отправленные для категорий `Watersports` и `Chess`, принимаются каналом, даже если приемник не готов. Отправитель для `Soccer` канала вынужден ждать, пока не истечет время вызова `time.Sleep` для получателя, и значения не будут получены из канала.

В реальных проектах используется буфер большего размера, выбираемый таким образом, чтобы у горутин было достаточно места для отправки сообщений без ожидания. (Обычно я указываю размер буфера 100, что обычно достаточно для большинства проектов, но не настолько велико, чтобы требовался значительный объем памяти.)

Проверка буфера канала

Вы можете определить размер буфера канала с помощью встроенной функции `cap` и определить количество значений в буфере с помощью функции `len`, как показано в листинге 14-16.

```
...
func CalcStoreTotal(data ProductData) {
    var storeTotal float64
    var channel chan float64 = make(chan float64, 2)
```

```
    for category, group := range data {
        go group.TotalPrice(category, channel)
    }
    time.Sleep(time.Second * 5)

    fmt.Println("-- Starting to receive from channel")
    for i := 0; i < len(data); i++ {
        fmt.Println(len(channel), cap(channel))
        fmt.Println("-- channel read pending",
            len(channel), "items in buffer, size", cap(channel))
        value :=  <- channel
        fmt.Println("-- channel read complete", value)
        storeTotal += value
        time.Sleep(time.Second)
    }
    fmt.Println("Total:", ToCurrency(storeTotal))
}
...
```

Листинг 14-16 Проверка буфера канала в файле operations.go в папке concurrency

Модифицированный оператор использует функции `len` и `cap`, чтобы сообщить количество значений в буфере канала и общий размер буфера. Скомпилируйте и выполните код, и вы увидите детали буфера по мере получения значений:

```
main function started
Watersports channel sending $328.95
Watersports channel send complete
Chess channel sending $1291.00
Chess channel send complete
Soccer channel sending $79554.45
-- Starting to receive from channel
-- channel read pending 2 items in buffer, size 2
Soccer channel send complete
-- channel read complete 328.95
-- channel read pending 2 items in buffer, size 2
-- channel read complete 1291
-- channel read pending 1 items in buffer, size 2
-- channel read complete 79554.45
Total: $81174.40
main function complete
```

Использование функций `len` и `cap` может дать представление о буфере канала, но результаты не следует использовать, чтобы попытаться

избежать блокировки при отправке сообщения. Горутины выполняются параллельно, что означает, что значения могут быть отправлены в канал после того, как вы проверите емкость буфера, но до того, как вы отправите значение. См. раздел «Использование операторов Select» для получения подробной информации о том, как надежно отправлять и получать без блокировки.

Отправка и получение неизвестного количества значений

Функция `CalcStoreTotal` использует свои знания об обрабатываемых данных, чтобы определить, сколько раз она должна получать значения из канала. Такая аналитика не всегда доступна, и количество значений, которые будут отправлены в канал, часто неизвестно заранее. В качестве демонстрации добавьте файл с именем `orderdispatch.go` в папку `concurrency` с содержимым, показанным в листинге 14-17.

```
package main

import (
    "fmt"
    "math/rand"
    "time"
)

type DispatchNotification struct {
    Customer string
    *Product
    Quantity int
}

var Customers = []string{"Alice", "Bob", "Charlie", "Dora"}

func DispatchOrders(channel chan DispatchNotification) {
    rand.Seed(time.Now().UTC().UnixNano())
    orderCount := rand.Intn(3) + 2
    fmt.Println("Order count:", orderCount)
    for i := 0; i < orderCount; i++ {
        channel <- DispatchNotification{
            Customer: Customers[rand.Intn(len(Customers)-1)],
            Quantity: rand.Intn(10),
            Product:  ProductList[rand.Intn(len(ProductList)-1)],
        }
    }
}
```

Листинг 14-17 Содержимое файла orderdispatch.go в папке concurrency

Функция `DispatchOrders` создает случайное количество значений `DispatchNotification` и отправляет их по каналу, полученному через параметр `channel`. Я описываю, как пакет `math/rand` используется для создания случайных чисел, в главе 18, но для этой главы достаточно знать, что детали каждого уведомления об отправке также являются случайными, так что имя клиента, продукт и изменится количество, а также общее количество значений, отправленных по каналу (хотя будет отправлено как минимум два, просто чтобы был какой-то результат).

Невозможно заранее узнать, сколько значений `DispatchNotification` создаст функция `DispatchOrders`, что представляет собой проблему при написании кода, который получает данные из канала. В листинге 14-18 используется самый простой подход, заключающийся в использовании цикла `for`, что означает, что код будет постоянно пытаться получить значения.

```
package main

import (
    "fmt"
    //"time"
)

func main() {

    dispatchChannel := make(chan DispatchNotification, 100)
    go DispatchOrders(dispatchChannel)
    for {
        details := <- dispatchChannel
             fmt.Println("Dispatch  to",  details.Customer,  ":",
details.Quantity,
            "x", details.Product.Name)
    }
}
```

Листинг 14-18 Получение значений в цикле for в файле main.go в папке concurrency

Цикл `for` не работает, потому что принимающий код попытается получить значения из канала после того, как отправитель перестанет их создавать. Среда выполнения Go завершит программу, если все горутины заблокированы, что вы можете увидеть, скомпилировав и выполнив проект, который выдаст следующий вывод:

```
Order count: 4
Dispatch to Charlie : 3 x Lifejacket
Dispatch to Bob : 6 x Soccer Ball
Dispatch to Bob : 7 x Thinking Cap
Dispatch to Charlie : 5 x Stadium
fatal error: all goroutines are asleep - deadlock!
goroutine 1 [chan receive]:
main.main()
        C:/concurrency/main.go:12 +0xa6
exit status 2
```

Вы увидите другой вывод, отражающий случайный характер данных `DispatchNotification`. Что важно, так это то, что горутина завершает работу после того, как она отправила свои значения, оставляя `main` горутину бездействующей, поскольку она продолжает ждать получения значения. Среда выполнения Go обнаруживает, что активных горутин нет, и завершает работу приложения.

Закрытие канала

Решение этой проблемы заключается в том, что отправитель указывает, когда через канал больше не поступают значения, что делается путем закрытия канала, как показано в листинге 14-19.

```
package main

import (
    "fmt"
    "math/rand"
    "time"
)

type DispatchNotification struct {
    Customer string
    *Product
    Quantity int
}

var Customers = []string{"Alice", "Bob", "Charlie", "Dora"}

func DispatchOrders(channel chan DispatchNotification) {
    rand.Seed(time.Now().UTC().UnixNano())
    orderCount := rand.Intn(3) + 2
    fmt.Println("Order count:", orderCount)
```

```
    for i := 0; i < orderCount; i++ {
        channel <- DispatchNotification{
            Customer: Customers[rand.Intn(len(Customers)-1)],
            Quantity: rand.Intn(10),
            Product:  ProductList[rand.Intn(len(ProductList)-1)],
        }
    }
    close(channel)
}
```

Листинг 14-19 Закрытие канала в файле orderdispatch.go в папке concurrency

Встроенная функция `close` принимает канал в качестве аргумента и используется для указания того, что через канал больше не будут отправляться значения. Получатели могут проверять, закрыт ли канал при запросе значения, как показано в листинге >14-20.

> **Подсказка**
> Вам нужно закрывать каналы только тогда, когда это полезно для координации ваших горутин. Go не требует закрытия каналов для высвобождения ресурсов или выполнения каких-либо хозяйственных задач.

```
package main

import (
    "fmt"
    //"time"
)

func main() {

    dispatchChannel := make(chan DispatchNotification, 100)
    go DispatchOrders(dispatchChannel)
    for {
        if details, open := <- dispatchChannel; open {
              fmt.Println("Dispatch to", details.Customer, ":",
details.Quantity,
                "x", details.Product.Name)
        } else {
            fmt.Println("Channel has been closed")
            break
        }
```

```
    }
}
```

Листинг 14-20 Проверка закрытых каналов в файле main.go в папке concurrency

Оператор получения может использоваться для получения двух значений. Первому значению присваивается значение, полученное от канала, а второе значение указывает, закрыт ли канал, как показано на рисунке 14-9.

Полученное значение Закрытый индикатор

```
if details, open := <- dispatchChanel; open {
```

Рисунок 14-9 Проверка на закрытый канал

Если канал открыт, то закрытый индикатор будет `false`, а значение, полученное из канала, будет присвоено другой переменной. Если канал закрыт, индикатор закрыт будет `true`, а другой переменной будет присвоено нулевое значение для типа канала.

Описание более сложное, чем код, с которым легко работать, потому что операция чтения канала может использоваться как оператор инициализации для выражения `if`, при этом индикатор закрытия используется для определения того, когда канал был закрыт. Код в листинге 14-20 определяет предложение `else`, которое выполняется при закрытии канала, что предотвращает дальнейшие попытки получения данных из канала и позволяет программе завершить работу корректно.

Осторожно
Незаконно отправлять значения в канал после его закрытия.

Скомпилируйте и запустите проект, и вы увидите вывод, подобный следующему:

```
Order count: 3
Dispatch to Bob : 2 x Soccer Ball
Dispatch to Alice : 9 x Thinking Cap
Dispatch to Bob : 3 x Soccer Ball
Channel has been closed
```

Перечисление значений канала

Цикл for можно использовать с ключевым словом range для перечисления значений, отправляемых через канал, что упрощает получение значений и завершает цикл при закрытии канала, как показано в листинге 14-21.

```
package main

import (
    "fmt"
    //"time"
)

func main() {

    dispatchChannel := make(chan DispatchNotification, 100)

    go DispatchOrders(dispatchChannel)
    for details := range dispatchChannel {
              fmt.Println("Dispatch  to",  details.Customer,  ":",
details.Quantity,
             "x", details.Product.Name)
    }
    fmt.Println("Channel has been closed")
}
```

Листинг 14-21 ЕПеречисление значений канала в файле main.go в папке concurrency

Выражение range производит одно значение за итерацию, которое является значением, полученным из канала. Цикл for будет продолжать получать значения, пока канал не будет закрыт. (Вы можете использовать цикл for...range на незакрытом канале, и в этом случае цикл никогда не завершится.) Скомпилируйте и выполните проект, и вы увидите вывод, подобный следующему:

```
Order count: 2
Dispatch to Alice : 9 x Kayak
Dispatch to Charlie : 8 x Corner Flags
Channel has been closed
```

Ограничение направления канала

По умолчанию каналы могут использоваться для отправки и получения данных, но это может быть ограничено при использовании каналов в качестве аргументов, так что могут выполняться только операции отправки или получения. Я считаю эту функцию полезной, чтобы

избежать ошибок, когда я намеревался отправить сообщение, но вместо этого выполнил получение, потому что синтаксис для этих операций похож, как показано в листинге 14-22.

```
package main

import (
    "fmt"
    "math/rand"
    "time"
)

type DispatchNotification struct {
    Customer string
    *Product
    Quantity int
}

var Customers = []string{"Alice", "Bob", "Charlie", "Dora"}

func DispatchOrders(channel chan DispatchNotification) {
    rand.Seed(time.Now().UTC().UnixNano())
    orderCount := rand.Intn(3) + 2
    fmt.Println("Order count:", orderCount)
    for i := 0; i < orderCount; i++ {
        channel <- DispatchNotification{
            Customer: Customers[rand.Intn(len(Customers)-1)],
            Quantity: rand.Intn(10),
            Product:  ProductList[rand.Intn(len(ProductList)-1)],
        }
        if (i == 1) {
            notification := <- channel
            fmt.Println("Read:", notification.Customer)
        }
    }
    close(channel)
}
```

Листинг 14-22 Ошибочные операции в файле orderdispatch.go в папке concurrency

Эту проблему легко заметить в примере кода, но я обычно допускаю эту ошибку, когда оператор `if` используется для условной отправки дополнительных значений через канал. В результате, однако, функция

получает сообщение, которое она только что отправила, удаляя его из канала.

Иногда отсутствующие сообщения вызывают блокировку горутины предполагаемого получателя, вызывая обнаружение взаимоблокировки, описанное ранее, и завершая программу, но часто программа запускается, но дает неожиданные результаты. Скомпилируйте и выполните код, и вы получите вывод, подобный следующему:

```
Order count: 4
Read: Alice
Dispatch to Alice : 4 x Unsteady Chair
Dispatch to Alice : 7 x Unsteady Chair
Dispatch to Bob : 0 x Thinking Cap
Channel has been closed
```

Выход сообщает, что по каналу будут отправлены четыре значения, но получены только три. Эту проблему можно решить, ограничив направление канала, как показано в листинге 14-23.

```
package main

import (
    "fmt"
    "math/rand"
    "time"
)

type DispatchNotification struct {
    Customer string
    *Product
    Quantity int
}

var Customers = []string{"Alice", "Bob", "Charlie", "Dora"}

func DispatchOrders(channel chan<- DispatchNotification) {
    rand.Seed(time.Now().UTC().UnixNano())
    orderCount := rand.Intn(3) + 2
    fmt.Println("Order count:", orderCount)
    for i := 0; i < orderCount; i++ {
        channel <- DispatchNotification{
            Customer: Customers[rand.Intn(len(Customers)-1)],
            Quantity: rand.Intn(10),
```

```
            Product:  ProductList[rand.Intn(len(ProductList)-1)],
        }
        if (i == 1) {
            notification := <- channel
            fmt.Println("Read:", notification.Customer)
        }
    }
    close(channel)
}
```

Листинг 14-23 Ограничение направления канала в файле orderdispatch.go в папке concurrency

Направление канала указывается вместе с ключевым словом `chan`, как показано на рисунке 14-10.

Ключевое слово Направление

```
func DispatchOrders(channel chan <- DispatchNotification) {
```

Рисунок 14-10 Указание направления канала

Расположение стрелки указывает направление канала. Когда стрелка следует за ключевым словом `chan`, как в листинге 14-23, тогда канал можно использовать только для отправки. Канал можно использовать для приема, только если стрелка предшествует ключевому слову chan (например, `<-chan`). Попытка получения из канала только для отправки (и наоборот) является ошибкой времени компиляции, которую вы можете увидеть, если скомпилируете проект:

```
# concurrency
.\orderdispatch.go:29:29: invalid operation: <-channel (receive
from send-only type chan<- DispatchNotification)
```

Это позволяет легко увидеть ошибку в функции `DispatchOrders`, и я могу удалить оператор, который получает данные из канала, как показано в листинге 14-24.

```
package main

import (
    "fmt"
    "math/rand"
    "time"
)
```

```
type DispatchNotification struct {
    Customer string
    *Product
    Quantity int
}

var Customers = []string{"Alice", "Bob", "Charlie", "Dora"}

func DispatchOrders(channel chan<- DispatchNotification) {
    rand.Seed(time.Now().UTC().UnixNano())
    orderCount := rand.Intn(3) + 2
    fmt.Println("Order count:", orderCount)
    for i := 0; i < orderCount; i++ {
        channel <- DispatchNotification{
            Customer: Customers[rand.Intn(len(Customers)-1)],
            Quantity: rand.Intn(10),
            Product:  ProductList[rand.Intn(len(ProductList)-1)],
        }
        // if (i == 1) {
        //     notification := <- channel
        //     fmt.Println("Read:", notification.Customer)
        // }
    }
    close(channel)
}
```

Листинг 14-24 Исправление ошибки в файле orderdispatch.go в папке concurrency

Код скомпилируется без ошибок и выдаст результат, аналогичный листингу 14-22.

Ограничение направления аргумента канала

Изменения в предыдущем разделе позволяют функции `DispatchOrders` объявить, что ей нужно только отправлять сообщения через канал, но не получать их. Это полезная функция, но она не подходит для ситуации, когда вы хотите предоставить только однонаправленный канал, а не позволить функции решать, что она получает.

Направленные каналы являются типами, поэтому тип параметра функции в листинге 14-24 — `chan<-DispatchNotification`, что означает канал только для отправки, который будет нести значения `DispatchNotification`. Go позволяет назначать двунаправленные каналы переменным однонаправленного канала, позволяя применять ограничения, как показано в листинге 14-25.

```
package main

import (
    "fmt"
    //"time"
)

func receiveDispatches(channel <-chan DispatchNotification) {
    for details := range channel {
             fmt.Println("Dispatch to", details.Customer, ":",
details.Quantity,
            "x", details.Product.Name)
    }
    fmt.Println("Channel has been closed")
}

func main() {

    dispatchChannel := make(chan DispatchNotification, 100)

        var sendOnlyChannel chan<- DispatchNotification =
dispatchChannel
        var receiveOnlyChannel <-chan DispatchNotification =
dispatchChannel

    go DispatchOrders(sendOnlyChannel)
    receiveDispatches(receiveOnlyChannel)
}
```

Листинг 14-25 Создание ограниченного канала в файле main.go в папке concurrency

Я использую полный синтаксис переменных для определения переменных канала только для отправки и только для приема, которые затем используются в качестве аргументов функции. Это гарантирует, что получатель канала только для отправки может только отправлять значения или закрывать канал, а получатель канала только для приема может только получать значения. Эти ограничения применяются к одному и тому же базовому каналу, поэтому сообщения, отправленные через `sendOnlyChannel`, будут получены через `ReceiveOnlyChannel`.

Ограничения на направление канала также могут быть созданы посредством явного преобразования, как показано в листинге 14-26.

```
package main
```

```
import (
    "fmt"
    //"time"
)

func receiveDispatches(channel <-chan DispatchNotification) {
    for details := range channel {
             fmt.Println("Dispatch  to",  details.Customer,  ":",
details.Quantity,
            "x", details.Product.Name)
    }
    fmt.Println("Channel has been closed")
}

func main() {
    dispatchChannel := make(chan DispatchNotification, 100)

        //  var  sendOnlyChannel  chan<-  DispatchNotification  =
dispatchChannel
       //  var  receiveOnlyChannel  <-chan  DispatchNotification  =
dispatchChannel

                                   go          DispatchOrders(chan<-
DispatchNotification(dispatchChannel))
              receiveDispatches((<-chan    DispatchNotification)
(dispatchChannel))
}
```

Листинг 14-26 Использование явных преобразований для каналов в файле main.go в папке concurrency

Явное преобразование для канала только для приема требует круглых скобок вокруг типа канала, чтобы предотвратить интерпретацию компилятором преобразования в тип `DispatchNotification`. Код в листингах 14-25 и 14-26 выдает один и тот же результат, который будет похож на следующий:

```
Order count: 4
Dispatch to Bob : 0 x Kayak
Dispatch to Alice : 2 x Stadium
Dispatch to Bob : 6 x Stadium
Dispatch to Alice : 3 x Thinking Cap
Channel has been closed
```

Использование операторов select

Ключевое слово select используется для группировки операций, которые будут отправлять или получать данные из каналов, что позволяет создавать сложные схемы горутин и каналов. Операторы select можно использовать по-разному, поэтому я начну с основ и перейду к более сложным параметрам. Чтобы подготовиться к примерам в этом разделе, в листинге 14-27 увеличивается количество значений DispatchNotification, отправляемых функцией DispatchOrders, и вводится задержка, поэтому они отправляются в течение более длительного периода.

```
package main

import (
    "fmt"
    "math/rand"
    "time"
)

type DispatchNotification struct {
    Customer string
    *Product
    Quantity int
}

var Customers = []string{"Alice", "Bob", "Charlie", "Dora"}

func DispatchOrders(channel chan<- DispatchNotification) {
    rand.Seed(time.Now().UTC().UnixNano())
    orderCount := rand.Intn(5) + 5
    fmt.Println("Order count:", orderCount)
    for i := 0; i < orderCount; i++ {
        channel <- DispatchNotification{
            Customer: Customers[rand.Intn(len(Customers)-1)],
            Quantity: rand.Intn(10),
            Product:  ProductList[rand.Intn(len(ProductList)-1)],
        }
        // if (i == 1) {
        //     notification := <- channel
        //     fmt.Println("Read:", notification.Customer)
        // }
        time.Sleep(time.Millisecond * 750)
    }
```

```
    close(channel)
}
```

Листинг 14-27 Пример Подготовка в файле orderdispatch.go в папке concurrency

Получение без блокировки

Простейшее использование операторов select — получение из канала без блокировки, что гарантирует, что горутине не придется ждать, когда канал станет пустым. В листинге 14-28 показан простой оператор select, используемый таким образом.

```
package main

import (
    "fmt"
    "time"
)

// func receiveDispatches(channel <-chan DispatchNotification) {
//     for details := range channel {
//         fmt.Println("Dispatch to", details.Customer, ":",
details.Quantity,
//             "x", details.Product.Name)
//     }
//     fmt.Println("Channel has been closed")
// }

func main() {
    dispatchChannel := make(chan DispatchNotification, 100)
                                        go      DispatchOrders(chan<-
DispatchNotification(dispatchChannel))
           //   receiveDispatches((<-chan   DispatchNotification)
(dispatchChannel))

    for {
        select {
            case details, ok := <- dispatchChannel:
                if ok {
                    fmt.Println("Dispatch to", details.Customer,
":",
                                            details.Quantity,  "x",
details.Product.Name)
                } else {
                    fmt.Println("Channel has been closed")
                    goto alldone
```

```
                }
            default:
                    fmt.Println("-- No message ready to be received")
                time.Sleep(time.Millisecond * 500)
        }
    }
    alldone: fmt.Println("All values received")
}
```

Листинг 14-28 Использование инструкции select в файле main.go в папке concurrency

Оператор `select` имеет структуру, аналогичную оператору `switch`, за исключением того, что операторы `case` являются операциями канала. Когда выполняется оператор `select`, каждая операция канала оценивается до тех пор, пока не будет достигнута операция, которую можно выполнить без блокировки. Выполняется операция канала и выполняются операторы, заключенные в оператор `case`. Если ни одна из операций канала не может быть выполнена, выполняются операторы в предложении `default`. На рисунке 14-11 показана структура оператора `select`.

Ключевое слово

Работа канала

Ключевое слово

```
select {
    case details, ok := <- dispatchChannel:
        if ok {
            fmt.Println("Dispatch to", details.Customer, ":",
                details.Quantity, "x", details.Product.Name)
        } else {
            fmt.Println("Channel has been closed")
            goto alldone
        }
    default:
        fmt.Println("-- No message ready to be received")
        time.Sleep(time.Millisecond * 500)
}
```

Ключевое слово

Рисунок 14-11 Оператор select

Оператор `select` оценивает свои операторы `case` один раз, поэтому я также использовал цикл `for` в листинге 14-28. Цикл продолжает выполнять оператор `select`, который будет получать значения из канала,

когда они станут доступны. Если значение недоступно, выполняется предложение по умолчанию, которое вводит период ожидания.

Операция канала оператора case в листинге 14-28 проверяет, был ли канал закрыт, и, если да, использует ключевое слово goto для перехода к оператору с меткой, находящемуся вне цикла for.

Скомпилируйте и выполните проект, и вы увидите вывод, аналогичный следующему, с некоторыми отличиями, поскольку данные генерируются случайным образом:

```
-- No message ready to be received
Order count: 5
Dispatch to Bob : 5 x Soccer Ball
-- No message ready to be received
Dispatch to Bob : 0 x Thinking Cap
-- No message ready to be received
Dispatch to Alice : 2 x Corner Flags
-- No message ready to be received
-- No message ready to be received
Dispatch to Bob : 6 x Corner Flags
-- No message ready to be received
Dispatch to Alice : 2 x Corner Flags
-- No message ready to be received
-- No message ready to be received
Channel has been closed
All values received
```

Задержки, вносимые методом time.Sleep, создают небольшое несоответствие между скоростью, с которой значения передаются по каналу, и скоростью, с которой они принимаются. В результате оператор select иногда выполняется, когда канал пуст. Вместо блокировки, которая произошла бы при обычной операции с каналом, оператор select выполняет операторы в предложении default. Как только канал закрывается, цикл завершается.

Прием с нескольких каналов

Оператор select можно использовать для приема без блокировки, как показано в предыдущем примере, но эта функция становится более полезной при наличии нескольких каналов, по которым значения отправляются с разной скоростью. Оператор select позволит получателю получать значения из любого канала, в котором они есть, без блокировки какого-либо отдельного канала, как показано в листинге 14-29.

```
package main

import (
    "fmt"
    "time"
)

func enumerateProducts(channel chan<- *Product) {
    for _, p := range ProductList[:3] {
        channel <- p
        time.Sleep(time.Millisecond * 800)
    }
    close(channel)
}

func main() {
    dispatchChannel := make(chan DispatchNotification, 100)
                                 go       DispatchOrders(chan<-
DispatchNotification(dispatchChannel))

    productChannel := make(chan *Product)
    go enumerateProducts(productChannel)

    openChannels := 2

    for  {
        select {
            case details, ok := <- dispatchChannel:
                if ok {
                    fmt.Println("Dispatch to", details.Customer,
":",
                                            details.Quantity,  "x",
details.Product.Name)
                } else {
                         fmt.Println("Dispatch channel has been
closed")
                    dispatchChannel = nil
                    openChannels--
                }
            case product, ok := <- productChannel:
                if ok {
                    fmt.Println("Product:", product.Name)
                } else {
                          fmt.Println("Product channel has been
closed")
```

```
                    productChannel = nil
                    openChannels--
                }
            default:
                if (openChannels == 0) {
                    goto alldone
                }
                        fmt.Println("-- No message ready to be
received")
                time.Sleep(time.Millisecond * 500)
        }
    }
    alldone: fmt.Println("All values received")
}
```

Листинг 14-29 Получение с нескольких каналов в файле main.go в папке concurrency

В этом примере оператор `select` используется для получения значений из двух каналов, один из которых содержит значения `DispatchNofitication`, а другой — значения `Product`. Каждый раз, когда выполняется оператор `select`, он проходит через операторы `case`, создавая список тех, из которых значение может быть прочитано без блокировки. Один из `case`-операторов выбирается из списка случайным образом и выполняется. Если ни один из операторов `case` не может быть выполнен, выполняется предложение `default`.

Необходимо соблюдать осторожность при управлении закрытыми каналами, поскольку они будут предоставлять `nil` значение для каждой операции приема, которая происходит после закрытия канала, полагаясь на закрытый индикатор, показывающий, что канал закрыт. К сожалению, это означает, что операторы `case` для закрытых каналов всегда будут выбираться операторами `select`, потому что они всегда готовы предоставить значение без блокировки, даже если это значение бесполезно.

> **Подсказка**
> Если предложение по умолчанию опущено, то оператор `select` будет блокироваться до тех пор, пока один из каналов не получит значение, которое нужно получить. Это может быть полезно, но не касается каналов, которые можно закрыть.

Управление закрытыми каналами требует двух мер. Во-первых, запретить оператору `select` выбирать канал после его закрытия. Это можно сделать, присвоив `nil` переменной канала, например:

```
...
dispatchChannel = nil
...
```

Канал `nil` никогда не будет готов и не будет выбран, что позволяет оператору `select` перейти к другим операторам case, каналы которых могут быть еще открыты.

Вторая мера — выйти из цикла `for`, когда все каналы закрыты, без чего оператор `select` бесконечно выполнял бы предложение `default`. В листинге 14-29 используется переменная типа `int`, значение которой уменьшается при закрытии канала. Когда количество открытых каналов достигает нуля, оператор `goto` выходит из цикла. Скомпилируйте и выполните проект, и вы увидите вывод, аналогичный следующему, показывающий, как один приемник получает значения из двух каналов:

```
Order count: 5
Product: Kayak
Dispatch to Alice : 9 x Unsteady Chair
-- No message ready to be received
Dispatch to Bob : 6 x Kayak
-- No message ready to be received
Product: Lifejacket
Dispatch to Charlie : 5 x Thinking Cap
-- No message ready to be received
-- No message ready to be received
Dispatch to Alice : 1 x Stadium
Product: Soccer Ball
-- No message ready to be received
Dispatch to Charlie : 8 x Lifejacket
-- No message ready to be received
Product channel has been closed
-- No message ready to be received
Dispatch channel has been closed
All values received
```

Отправка без блокировки

Оператор `select` также может использоваться для отправки в канал без блокировки, как показано в листинге 14-30.

```
package main

import (
    "fmt"
    "time"
)

func enumerateProducts(channel chan<- *Product) {
    for _, p := range ProductList {
        select {
            case channel <- p:
                fmt.Println("Sent product:", p.Name)
            default:
                fmt.Println("Discarding product:", p.Name)
                time.Sleep(time.Second)
        }
    }
    close(channel)
}

func main() {
    productChannel := make(chan *Product, 5)
    go enumerateProducts(productChannel)

    time.Sleep(time.Second)

    for p := range productChannel {
        fmt.Println("Received product:", p.Name)
    }
}
```

Листинг 14-30 Отправка с использованием инструкции select в файле main.go в папке concurrency

Канал в листинге 14-30 создан с небольшим буфером, и значения не принимаются из канала до небольшой задержки. Это означает, что функция `enumerateProducts` может отправлять значения по каналу без блокировки, пока буфер не заполнится. Предложение `default` оператора `select` отбрасывает значения, которые не могут быть отправлены. Скомпилируйте и выполните код, и вы увидите вывод, подобный следующему:

```
Sent product: Kayak
Sent product: Lifejacket
Sent product: Soccer Ball
Sent product: Corner Flags
```

```
Sent product: Stadium
Discarding product: Thinking Cap
Discarding product: Unsteady Chair
Received product: Kayak
Received product: Lifejacket
Received product: Soccer Ball
Received product: Corner Flags
Received product: Stadium
Sent product: Bling-Bling King
Received product: Bling-Bling King
```

Вывод показывает, где оператор `select` определил, что операция отправки будет заблокирована, и вместо этого вызвала предложение default. В листинге 14-30 оператор `case` содержит оператор, который выводит сообщение, но это не требуется, а оператор `case` может указывать операции отправки без дополнительных операторов, как показано в листинге 14-31.

```
package main

import (
    "fmt"
    "time"
)

func enumerateProducts(channel chan<- *Product) {
    for _, p := range ProductList {
        select {
            case channel <- p:
                //fmt.Println("Sent product:", p.Name)
            default:
                fmt.Println("Discarding product:", p.Name)
                time.Sleep(time.Second)
        }
    }
    close(channel)
}

func main() {
    productChannel := make(chan *Product, 5)
    go enumerateProducts(productChannel)

    time.Sleep(time.Second)
```

```
    for p := range productChannel {
        fmt.Println("Received product:", p.Name)
    }
}
```

Листинг 14-31 Пропуск операторов в файле main.go в папке concurrency

Отправка на несколько каналов

Если доступно несколько каналов, можно использовать оператор `select`, чтобы найти канал, для которого отправка не будет блокироваться, как показано в листинге 14-32.

> **Подсказка**
> Вы можете комбинировать операторы `case` с операциями отправки и получения в одном операторе `select`. Когда выполняется оператор `select`, среда выполнения Go создает комбинированный список операторов `case`, которые могут выполняться без блокировки, и выбирает один из них случайным образом, который может быть либо оператором отправки, либо оператором получения.

```
package main

import (
    "fmt"
    "time"
)

func enumerateProducts(channel1, channel2 chan<- *Product) {
    for _, p := range ProductList {
        select {
            case channel1 <- p:
                fmt.Println("Send via channel 1")
            case channel2 <- p:
                fmt.Println("Send via channel 2")
        }
    }
    close(channel1)
    close(channel2)
}

func main() {
    c1 := make(chan *Product, 2)
    c2 := make(chan *Product, 2)
```

```
    go enumerateProducts(c1, c2)

    time.Sleep(time.Second)

    for p := range c1 {
        fmt.Println("Channel 1 received product:", p.Name)
    }
    for p := range c2 {
        fmt.Println("Channel 2 received product:", p.Name)
    }
}
```

Листинг 14-32 Отправка по нескольким каналам в файле main.go в папке concurrency

В этом примере есть два канала с небольшими буферами. Как и в случае с получением, оператор `select` создает список каналов, по которым значение может быть отправлено без блокировки, а затем случайным образом выбирает один из этих каналов. Если ни один из каналов не может быть использован, то выполняется предложение `default`. В этом примере нет предложения `default`, что означает, что оператор `select` будет блокироваться до тех пор, пока один из каналов не сможет получить значение.

Значения из канала не принимаются до тех пор, пока через секунду после создания горутины, выполняющей функцию `enumerateProducts`, это означает, что только буферы определяют, будет ли блокироваться отправка в канал. Скомпилируйте и запустите проект, и вы получите следующий вывод:

```
Send via channel 1
Send via channel 1
Send via channel 2
Send via channel 2
Channel 1 received product: Kayak
Channel 1 received product: Lifejacket
Channel 1 received product: Stadium
Send via channel 1
Send via channel 1
Send via channel 1
Send via channel 1
Channel 1 received product: Thinking Cap
Channel 1 received product: Unsteady Chair
Channel 1 received product: Bling-Bling King
Channel 2 received product: Soccer Ball
Channel 2 received product: Corner Flags
```

Распространенной ошибкой является предположение, что оператор `select` будет равномерно распределять значения по нескольким каналам. Как уже отмечалось, оператор `select` выбирает случайный оператор `case`, который можно использовать без блокировки, а это означает, что распределение значений непредсказуемо и может быть неравномерным. Вы можете увидеть этот эффект, повторно запустив пример, который покажет, что значения отправляются в каналы в другом порядке.

Резюме

В этой главе я описал использование горутин, которые позволяют выполнять функции Go одновременно. Горутины производят результаты асинхронно, используя каналы, которые также были представлены в этой главе. Горутины и каналы упрощают написание параллельных приложений без необходимости управлять отдельными потоками выполнения. В следующей главе я опишу поддержку Go для обработки ошибок.

15. Обработка ошибок

В этой главе описывается, как Go обрабатывает ошибки. Я описываю интерфейс, представляющий ошибки, показываю, как создавать ошибки, и объясняю различные способы их обработки. Я также описываю панику, то есть то, как обрабатываются неисправимые ошибки. Таблица 15-1 рассматривает обработку ошибок в контексте.

Таблица 15-1 Помещение обработки ошибок в контекст

Вопрос	Ответ
Что это?	Обработка ошибок в Go позволяет отображать и обрабатывать исключительные условия и сбои.
Почему это полезно?	Приложения часто сталкиваются с непредвиденными ситуациями, и функции обработки ошибок позволяют реагировать на такие ситуации, когда они возникают.
Как это используется?	Интерфейс `error` используется для определения условий ошибки, которые обычно возвращаются в виде результатов функции. Функция `panic` вызывается при возникновении неисправимой ошибки.
Есть ли подводные камни или ограничения?	Необходимо позаботиться о том, чтобы об ошибках сообщалось той части приложения, которая лучше всего может определить, насколько серьезна ситуация.
Есть ли альтернативы?	Вам не нужно использовать интерфейс `error` в своем коде, но он используется во всей стандартной библиотеке Go, и его трудно избежать.

Таблица 15-2 суммирует главу.

Таблица 15-2 Краткое содержание главы

Проблема	Решение	Листинг
Укажите, что произошла ошибка	Создайте структуру, реализующую интерфейс `error`, и верните ее как результат функции.	7–8, 11, 12
Сообщить об ошибке в канале	Добавьте поле error к типу структуры, используемому для сообщений канала.	9–10
Указать, что произошла неисправимая ошибка	Вызовите функцию `panic`	13, 16

Проблема	Решение	Листинг
Восстановиться от паники	Используйте ключевое слово defer для регистрации функции, которая вызывает функцию recover.	14, 15, 17–19

Подготовка к этой главе

Чтобы подготовиться к этой главе, откройте новую командную строку, перейдите в удобное место и создайте каталог с именем errorHandling. Запустите команду, показанную в листинге 15-1, в папке errorHandling, чтобы создать файл модуля.

> **Подсказка**
> Вы можете загрузить пример проекта для этой главы — и для всех остальных глав этой книги — с https://github.com/apress/pro-go. См. Главу 2 о том, как получить помощь, если у вас возникнут проблемы с запуском примеров.

```
go mod init errorHandling
```

Листинг 15-1 Инициализация модуля

Добавьте файл с именем product.go в папку errorHandling с содержимым, показанным в листинге 15-2.

```
package main

import "strconv"

type Product struct {
    Name, Category string
    Price float64
}

type ProductSlice []*Product

var Products = ProductSlice {
    { "Kayak", "Watersports", 279 },
    { "Lifejacket", "Watersports", 49.95 },
    { "Soccer Ball", "Soccer", 19.50 },
    { "Corner Flags", "Soccer", 34.95 },
```

```
    { "Stadium", "Soccer", 79500 },
    { "Thinking Cap", "Chess", 16 },
    { "Unsteady Chair", "Chess", 75 },
    { "Bling-Bling King", "Chess", 1200 },
}

func ToCurrency(val float64) string {
    return "$" + strconv.FormatFloat(val, 'f', 2, 64)
}
```

Листинг 15-2 Содержимое файла product.go в папке errorHandling

Этот файл определяет настраиваемый тип с именем `Product`, псевдоним для среза значений `*Product` и срез, заполненный с использованием литерального синтаксиса. Я также определил функцию для форматирования значений `float64` в суммы в долларах.

Добавьте файл с именем `operations.go` в папку `errorHandling` с содержимым, показанным в листинге 15-3.

```
package main

func (slice ProductSlice) TotalPrice(category string) (total
float64) {
    for _, p := range slice {
        if (p.Category == category) {
            total += p.Price
        }
    }
    return
}
```

Листинг 15-3 Содержимое файла operations.go в папке errorHandling

Этот файл определяет метод, который получает `ProductSlice` и суммирует поле `Price` для этих значений `Product` с указанным значением `Category`.

Добавьте файл с именем `main.go` в папку `errorHandling` с содержимым, показанным в листинге 15-4.

```
package main

import "fmt"
```

```
func main() {

    categories := []string { "Watersports", "Chess" }

    for _, cat := range categories {
        total := Products.TotalPrice(cat)
        fmt.Println(cat, "Total:", ToCurrency(total))
    }
}
```

Листинг 15-4 Содержимое файла main.go в папке errorHandling

Используйте командную строку для запуска команды, показанной в листинге 15-5, в папке `errorHandling`.

```
go run .
```

Листинг 15-5 Запуск примера проекта

Код будет скомпилирован и выполнен, что приведет к следующему выводу:

```
Watersports Total: $328.95
Chess Total: $1291.00
```

Работа с исправимыми ошибками

Go упрощает выражение исключительных условий, что позволяет функции или методу указать вызывающему коду, что что-то пошло не так. Например, в листинге 15-6 добавлены операторы, которые вызывают проблемный ответ от метода `TotalPrice`.

```
package main

import "fmt"

func main() {

        categories := []string { "Watersports", "Chess",
"Running" }

    for _, cat := range categories {
```

```
        total := Products.TotalPrice(cat)
        fmt.Println(cat, "Total:", ToCurrency(total))
    }
}
```

Листинг 15-6 Вызов метода в файле main.go в папке errorHandling

Скомпилируйте и запустите проект, и вы получите следующий вывод:

```
Watersports Total: $328.95
Chess Total: $1291.00
Running Total: $0.00
```

Ответ метода `TotalPrice` для категории `Running` неоднозначен. Нулевой результат может означать, что в указанной категории нет продуктов, или это может означать, что продукты есть, но их суммарное значение равно нулю. Код, который вызывает метод `TotalPrice`, не может знать, что представляет собой нулевое значение.

На простом примере легко понять результат из его контекста: в категории `Running` нет товаров. В реальных проектах такой результат может быть труднее понять и отреагировать на него.

Go предоставляет предопределенный интерфейс с именем `error`, который обеспечивает один из способов решения этой проблемы. Вот определение интерфейса:

```
type error interface {
    Error() string
}
```

Интерфейс требует ошибок, чтобы определить метод с именем `Error`, который возвращает строку.

Генерация ошибок

Функции и методы могут выражать исключительные или неожиданные результаты, выдавая `error` ответы, как показано в листинге 15-7.

```
package main

type CategoryError struct {
```

```
    requestedCategory string
}

func (e *CategoryError) Error() string {
     return "Category " + e.requestedCategory + " does not
exist"
}

func (slice ProductSlice) TotalPrice(category string) (total
float64,
        err *CategoryError) {
    productCount := 0
    for _, p := range slice {
        if (p.Category == category) {
            total += p.Price
            productCount++
        }
    }
    if (productCount == 0) {
        err = &CategoryError{ requestedCategory: category}
    }
    return
}
```

Листинг 15-7 Определение ошибки в файле operations.go в папке errorHandling

Тип `CategoryError` определяет неэкспортированное поле `requestedCategory`, и существует метод, соответствующий интерфейсу `error`. Сигнатура метода `TotalPrice` была обновлена, и теперь он возвращает два результата: исходное значение `float64` и `error`. Если цикл `for` не находит продуктов с указанной категорией, результату `err` присваивается значение `CategoryError`, указывающее на то, что была запрошена несуществующая категория. Листинг 15-8 обновляет вызывающий код, чтобы использовать результат ошибки.

```
package main

import "fmt"

func main() {
```

```
        categories  :=  []string  {  "Watersports",  "Chess",
"Running" }

    for _, cat := range categories {
        total, err := Products.TotalPrice(cat)
        if (err == nil) {
            fmt.Println(cat, "Total:", ToCurrency(total))
        } else {
            fmt.Println(cat, "(no such category)")
        }
    }
}
```

Листинг 15-8 Обработка ошибки в файле main.go в папке errorHandling

Результат вызова метода TotalPrice определяется путем проверки комбинации двух результатов.

Если результат ошибки равен nil, то запрошенная категория существует, а результат float64 представляет собой сумму их цен, даже если эта сумма равна нулю. Если результат error не nil, то запрошенная категория не существует, и значение float64 следует игнорировать. Скомпилируйте и выполните проект, и вы увидите, что ошибка позволяет коду в листинге 15-8 идентифицировать несуществующую категорию продукта:

```
Watersports Total: $328.95
Chess Total: $1291.00
Running (no such category)
```

ИГНОРИРОВАНИЕ РЕЗУЛЬТАТОВ ОШИБКИ

Я не рекомендую игнорировать результаты ошибок, потому что это означает, что вы потеряете важную информацию, но если вам не нужно знать, когда что-то пойдет не так, вы можете использовать пустой идентификатор вместо имени для результата ошибки, например:

```
package main

import "fmt"
```

```
func main() {

        categories  :=  []string  {  "Watersports",  "Chess",
"Running" }

    for _, cat := range categories {
        total, _ := Products.TotalPrice(cat)
        fmt.Println(cat, "Total:", ToCurrency(total))
    }
}
```

Этот метод не позволяет вызывающему коду понять полный ответ от метода `TotalPrice`, и его следует использовать с осторожностью.

Сообщение об ошибках через каналы

Если функция выполняется с использованием горутины, то связь осуществляется только через канал, а это означает, что сведения о любых проблемах должны передаваться вместе с успешными операциями. Важно, чтобы обработка ошибок была как можно более простой, и я рекомендую избегать попыток использовать дополнительные каналы или создавать сложные механизмы для попыток сигнализировать об ошибках за пределами канала. Я предпочитаю создавать собственный тип, объединяющий оба результата, как показано в листинге 15-9.

```
package main

type CategoryError struct {
    requestedCategory string
}

func (e *CategoryError) Error() string {
     return "Category " + e.requestedCategory + " does not
exist"
}

type ChannelMessage struct {
    Category string
    Total float64
```

```
    *CategoryError
}

func (slice ProductSlice) TotalPrice(category string) (total
float64,
        err *CategoryError) {
    productCount := 0
    for _, p := range slice {
        if (p.Category == category) {
            total += p.Price
            productCount++
        }
    }
    if (productCount == 0) {
        err = &CategoryError{ requestedCategory: category}
    }
    return
}

func   (slice   ProductSlice)   TotalPriceAsync   (categories
[]string,
        channel chan<- ChannelMessage) {
    for _, c := range categories {
        total, err := slice.TotalPrice(c)
        channel <- ChannelMessage{
            Category: c,
            Total: total,
            CategoryError: err,
        }
    }
    close(channel)
}
```

Листинг 15-9 Определение типов и функций в файле operations.go в папке errorHandling

Тип `ChannelMessage` позволяет мне передать пару результатов, необходимых для точного отражения результата метода `TotalPrice`, который выполняется асинхронно с помощью нового метода `TotalPriceAsync`. Результат аналогичен тому, как результаты синхронного метода могут выражать ошибки.

Если для канала имеется только один отправитель, вы можете закрыть канал после возникновения ошибки. Но необходимо

соблюдать осторожность, чтобы не закрыть канал, если есть несколько отправителей, потому что они все еще могут генерировать действительные результаты и попытаются отправить их по закрытому каналу, что приведет к панике завершения программы.

Листинг 15-10 обновляет функцию `main` для использования новой асинхронной версии метода `TotalPrice`.

```
package main

import "fmt"

func main() {

        categories := []string { "Watersports", "Chess",
"Running" }

    channel := make(chan ChannelMessage, 10)

    go Products.TotalPriceAsync(categories, channel)
    for message := range channel {
        if message.CategoryError == nil {
                     fmt.Println(message.Category, "Total:",
ToCurrency(message.Total))
        } else {
                     fmt.Println(message.Category, "(no such
category)")
        }
    }
}
```

Листинг 15-10 Использование нового метода в файле main.go в папке errorHandling

Скомпилируйте и выполните проект, и вы получите вывод, подобный следующему:

```
Watersports Total: $328.95
Chess Total: $1291.00
Running (no such category)
```

Использование удобных функций обработки ошибок

Может быть неудобно определять типы данных для каждого типа ошибки, с которой может столкнуться приложение. Пакет errors, являющийся частью стандартной библиотеки, предоставляет функцию New, которая возвращает ошибку, содержимое которой представляет собой строку. Недостатком этого подхода является то, что он создает простые ошибки, но его преимущество заключается в простоте, как показано в листинге 15-11.

```
package main

import "errors"

// type CategoryError struct {
//     requestedCategory string
// }

// func (e *CategoryError) Error() string {
//     return "Category " + e.requestedCategory + " does not
exist"
//}

type ChannelMessage struct {
    Category string
    Total float64
    CategoryError error
}

func (slice ProductSlice) TotalPrice(category string) (total
float64,
        err error) {
    productCount := 0
    for _, p := range slice {
        if (p.Category == category) {
            total += p.Price
            productCount++
        }
    }
    if (productCount == 0) {
        err = errors.New("Cannot find category")
    }
    return
}
```

```
func   (slice   ProductSlice)   TotalPriceAsync   (categories
[]string,
        channel chan<- ChannelMessage) {
    for _, c := range categories {
        total, err := slice.TotalPrice(c)
        channel <- ChannelMessage{
            Category: c,
            Total: total,
            CategoryError: err,
        }
    }
    close(channel)
}
```

Листинг 15-11 Использование функции удобства работы с ошибками в файле operations.go в папке errorHandling

Хотя в этом примере мне удалось удалить пользовательский тип ошибки, возникающая `error` больше не содержит сведений о запрошенной категории. Это не является серьезной проблемой, поскольку вполне разумно ожидать, что вызывающий код будет иметь эту информацию, но в ситуациях, когда это неприемлемо, можно использовать пакет `fmt` для легкого создания ошибок с более сложным строковым содержимым.

Пакет `fmt` отвечает за форматирование строк, что он и делает с глаголами форматирования. Эти команды подробно описаны в главе 17, а одной из функций, предоставляемых пакетом `fmt`, является `Errorf`, которая создает значения `error` с использованием форматированной строки, как показано в листинге 15-12.

```
package main

import "fmt"

type ChannelMessage struct {
    Category string
    Total float64
    CategoryError error
}

func (slice ProductSlice) TotalPrice(category string) (total
float64,
```

```
        err error) {
    productCount := 0
    for _, p := range slice {
        if (p.Category == category) {
            total += p.Price
            productCount++
        }
    }
    if (productCount == 0) {
              err = fmt.Errorf("Cannot find category: %v",
category)
    }
    return
}

func (slice ProductSlice) TotalPriceAsync (categories
[]string,
        channel chan<- ChannelMessage) {
    for _, c := range categories {
        total, err := slice.TotalPrice(c)
        channel <- ChannelMessage{
            Category: c,
            Total: total,
            CategoryError: err,
        }
    }
    close(channel)
}
```

Листинг 15-12 Использование функции форматирования ошибок в файле operations.go в папке errorHandling

%v в первом аргументе функции Errorf является примером глагола форматирования, и он заменяется следующим аргументом, как описано в главе 17. Листинг 15-11 и Листинг 15-12 дают следующий вывод, который создается независимо от сообщения в ответе на ошибку:

```
Watersports Total: $328.95
Chess Total: $1291.00
Running (no such category)
```

Работа с неисправимыми ошибками

Некоторые ошибки настолько серьезны, что должны привести к немедленному завершению работы приложения, процессу, известному как *паника*, как показано в листинге 15-13.

```
package main

import "fmt"

func main() {

        categories  :=  []string  {  "Watersports",  "Chess",
"Running" }

    channel := make(chan ChannelMessage, 10)

    go Products.TotalPriceAsync(categories, channel)
    for message := range channel {
        if message.CategoryError == nil {
                     fmt.Println(message.Category,  "Total:",
ToCurrency(message.Total))
        } else {
            panic(message.CategoryError)
                  //fmt.Println(message.Category,  "(no  such
category)")
        }
    }
}
```

Листинг 15-13 Инициирование паники в файле main.go в папке errorHandling

Вместо того, чтобы распечатывать сообщение, когда категория не может быть найдена, функция `main` вызывает панику, которая выполняется с помощью встроенной функции `panic`, как показано на рисунке 15-1.

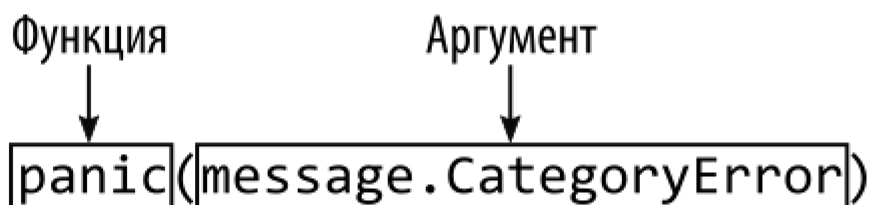

Рисунок 15-1 Функция panic

Функция `panic` вызывается с аргументом, который может быть любым значением, помогающим объяснить панику. В листинге 15-13 функция `panic` вызывается с `error`, что является полезным способом объединения функций обработки ошибок Go.

Когда вызывается функция `panic`, выполнение объемлющей функции останавливается, и выполняются все `defer` функции. (Функция `defer` описана в главе 8.) Паника поднимается вверх по стеку вызовов, прерывая выполнение вызывающих функций и вызывая их функции `defer`. В примере паникует функция `GetProducts`, что приводит к завершению функции `CountProducts` и, наконец, `main` функции, после чего приложение завершается. Скомпилируйте и выполните код, и вы увидите следующий вывод, показывающий трассировку стека для паники:

```
Watersports Total: $328.95
Chess Total: $1291.00
panic: Cannot find category: Running

goroutine 1 [running]:
main.main()
        C:/errorHandling/main.go:16 +0x309
exit status 2
```

Вывод показывает, что произошла паника, и что это произошло внутри функции `main` в `main` пакете, вызванной оператором в строке 13 файла `main.go`. В более сложных приложениях трассировка стека, отображаемая паникой, может помочь выяснить, почему возникла паника.

ОПРЕДЕЛЕНИЕ ПАНИЧЕСКИХ И НЕПАНИКОВЫХ ФУНКЦИЙ

Не существует четких правил, определяющих, когда ошибка уместна, а когда паника будет более полезной. Проблема в том, что серьезность проблемы часто лучше всего определяется вызывающей функцией, а не тем, где обычно принимается решение о панике. Как я объяснял ранее, использование несуществующей категории продукта может быть серьезной и неустранимой

проблемой в одних ситуациях и ожидаемым результатом в других, и эти два условия вполне могут существовать в одном и том же проекте.

Обычное соглашение состоит в том, чтобы предлагать две версии функции, одна из которых возвращает ошибку, а другая вызывает панику. Вы можете увидеть пример такого расположения в главе 16, где пакет `regexp` определяет функцию `Compile`, которая возвращает ошибку, и функцию `MustCompile`, которая вызывает панику.

Восстановление после паники

Go предоставляет встроенную функцию `recover`, которую можно вызвать, чтобы не дать панике подняться вверх по стеку вызовов и завершить программу. Функция `recover` должна вызываться в коде, который выполняется с использованием ключевого слова `defer`, как показано в листинге 15-14.

```
package main

import "fmt"

func main() {

    recoveryFunc := func() {
        if arg := recover(); arg != nil {
            if err, ok := arg.(error); ok {
                fmt.Println("Error:", err.Error())
            } else if str, ok := arg.(string); ok {
                fmt.Println("Message:", str)
            } else {
                fmt.Println("Panic recovered")
            }
        }
    }
    defer recoveryFunc()

        categories := []string { "Watersports", "Chess",
"Running" }

    channel := make(chan ChannelMessage, 10)
```

```
    go Products.TotalPriceAsync(categories, channel)
    for message := range channel {
        if message.CategoryError == nil {
                    fmt.Println(message.Category,  "Total:",
ToCurrency(message.Total))
        } else {
            panic(message.CategoryError)
                  //fmt.Println(message.Category, "(no  such
category)")
        }
    }
}
```

Листинг 15-14 Восстановление после паники в файле main.go в папке errorHandling

В этом примере используется ключевое слово `defer` для регистрации функции, которая будет выполняться после завершения `main` функции, даже если паники не было. Вызов `recover` восстановления возвращает значение, если была паника, останавливая развитие паники и предоставляя доступ к аргументу, используемому для вызова функции `panic`, как показано на рисунке 15-2.

Функция восстановления

```
if arg := recover(); arg != nil {
```

Рисунок 15-2 Выход из паники

Поскольку любое значение может быть передано в функцию `panic`, тип значения, возвращаемого функцией восстановления, является пустым интерфейсом (`interface{}`), который требует утверждения типа, прежде чем его можно будет использовать. Функция восстановления в листинге 15-14 работает с типами `error` и `string`, которые являются двумя наиболее распространенными типами аргумента паники.

Может быть неудобно определять функцию и сразу же использовать ее с ключевым словом `defer`, поэтому аварийное восстановление обычно выполняется с помощью анонимной функции, как показано в листинге 15-15.

```
package main

import "fmt"

func main() {

    defer func() {
        if arg := recover(); arg != nil {
            if err, ok := arg.(error); ok {
                fmt.Println("Error:", err.Error())
            } else if str, ok := arg.(string); ok {
                fmt.Println("Message:", str)
            } else {
                fmt.Println("Panic recovered")
            }
        }
    }()

        categories  :=  []string  {  "Watersports",  "Chess",
"Running" }

    channel := make(chan ChannelMessage, 10)

    go Products.TotalPriceAsync(categories, channel)
    for message := range channel {
        if message.CategoryError == nil {
                    fmt.Println(message.Category,  "Total:",
ToCurrency(message.Total))
        } else {
            panic(message.CategoryError)
                 //fmt.Println(message.Category,  "(no  such
category)")
        }
    }
}
```

Листинг 15-15 Использование анонимной функции в файле main.go в папке errorHandling

Обратите внимание на использование круглых скобок после закрывающей фигурной скобки анонимной функции, которые необходимы для вызова, а не просто для определения анонимной

функции. Листинги 15-14 и 15-15 дают один и тот же результат при компиляции и выполнении:

```
Watersports Total: $328.95
Chess Total: $1291.00
Error: Cannot find category: Running
```

Паника после восстановления

Вы можете оправиться от паники только для того, чтобы понять, что ситуация уже не исправима. Когда это происходит, вы можете запустить новую панику, либо предоставив новый аргумент, либо повторно используя значение, полученное при вызове функции `recover`, как показано в листинге 15-16.

```
package main

import "fmt"

func main() {

    defer func() {
        if arg := recover(); arg != nil {
            if err, ok := arg.(error); ok {
                fmt.Println("Error:", err.Error())
                panic(err)
            } else if str, ok := arg.(string); ok {
                fmt.Println("Message:", str)
            } else {
                fmt.Println("Panic recovered")
            }
        }
    }()

        categories := []string { "Watersports", "Chess",
"Running" }

    channel := make(chan ChannelMessage, 10)

    go Products.TotalPriceAsync(categories, channel)
    for message := range channel {
        if message.CategoryError == nil {
```

```
                     fmt.Println(message.Category,  "Total:",
ToCurrency(message.Total))
        } else {
            panic(message.CategoryError)
                   //fmt.Println(message.Category,  "(no  such
category)")
        }
    }
}
```

Листинг 15-16 Выборочная паника после восстановления в файле main.go в папке errorHandling

Отложенная функция восстанавливает панику, проверяет детали ошибки, а затем снова вызывает панику. Скомпилируйте и запустите проект, и вы увидите эффект изменения:

```
Watersports Total: $328.95
Chess Total: $1291.00
Error: Cannot find category: Running
panic: Cannot find category: Running [recovered]
        panic: Cannot find category: Running
goroutine 1 [running]:
main.main.func1()
        C:/errorHandling/main.go:11 +0x1c8
panic({0xad91a0, 0xc000088230})
        C:/Program Files/Go/src/runtime/panic.go:1038 +0x215
main.main()
        C:/errorHandling/main.go:29 +0x333
exit status 2
```

Восстановление после паники в горутинах

Паника поднимается вверх по стеку только до вершины текущей горутины, после чего вызывает завершение приложения. Это ограничение означает, что паники должны восстанавливаться внутри кода, выполняемого горутиной, как показано в листинге 15-17.

```
package main

import "fmt"

type CategoryCountMessage struct {
```

```
    Category string
    Count int
}

func processCategories(categories [] string, outChan chan <-
CategoryCountMessage) {
    defer func() {
        if arg := recover(); arg != nil {
            fmt.Println(arg)
        }
    }()
    channel := make(chan ChannelMessage, 10)
    go Products.TotalPriceAsync(categories, channel)
    for message := range channel {
        if message.CategoryError == nil {
            outChan <- CategoryCountMessage {
                Category: message.Category,
                Count: int(message.Total),
            }
        } else {
            panic(message.CategoryError)
        }
    }
    close(outChan)
}

func main() {

        categories  :=  []string  {  "Watersports",  "Chess",
"Running" }

    channel := make(chan CategoryCountMessage)
    go processCategories(categories, channel)

    for message := range channel {
                    fmt.Println(message.Category,   "Total:",
message.Count)
    }
}
```

Листинг 15-17 Восстановление после паники в файле main.go в папке errorHandling

Функция main использует горутину для вызова функции processCategories, которая вызывает панику, если функция TotalPriceAsync отправляет error. ProcessCategories восстанавливается после паники, но это имеет неожиданные последствия, которые вы можете увидеть в выводе, полученном при компиляции и выполнении проекта:

```
Watersports Total: 328
Chess Total: 1291
Cannot find category: Running
fatal error: all goroutines are asleep - deadlock!
goroutine 1 [chan receive]:
main.main()
        C:/errorHandling/main.go:39 +0x1c5
exit status 2
```

Проблема в том, что восстановление после паники не возобновляет выполнение функции processCategories, а это означает, что функция close никогда не вызывается на канале, из которого основная функция получает сообщения. Функция main пытается получить сообщение, которое никогда не будет отправлено, и блокирует канал, вызывая обнаружение взаимоблокировки среды выполнения Go.

Самый простой подход — вызвать функцию close закрытия канала во время восстановления, как показано в листинге 15-18.

```
...
defer func() {
    if arg := recover(); arg != nil {
        fmt.Println(arg)
        close(outChan)
    }
}()
...
```

Листинг 15-18 Обеспечение закрытия канала в файле main.go в папке errorHandling

Это предотвращает взаимоблокировку, но делает это без указания функции main, что функция processCategories не смогла завершить свою работу, что может иметь последствия. Лучшим подходом является указание этого результата через канал перед его закрытием, как показано в листинге 15-19.

```
package main

import "fmt"

type CategoryCountMessage struct {
    Category string
    Count int
    TerminalError interface{}
}

func processCategories(categories [] string, outChan chan <-
CategoryCountMessage) {
    defer func() {
        if arg := recover(); arg != nil {
            fmt.Println(arg)
            outChan <- CategoryCountMessage{
                TerminalError: arg,
            }
            close(outChan)
        }
    }()
    channel := make(chan ChannelMessage, 10)
    go Products.TotalPriceAsync(categories, channel)
    for message := range channel {
        if message.CategoryError == nil {
            outChan <- CategoryCountMessage {
                Category: message.Category,
                Count: int(message.Total),
            }
        } else {
            panic(message.CategoryError)
        }
    }
    close(outChan)
}

func main() {

        categories  :=  []string  {  "Watersports",  "Chess",
"Running" }

    channel := make(chan CategoryCountMessage)
    go processCategories(categories, channel)
```

```
    for message := range channel {
        if (message.TerminalError == nil) {
                    fmt.Println(message.Category,  "Total:",
message.Count)
        } else {
            fmt.Println("A terminal error occured")
        }
    }
}
```

Листинг 15-19 Указание на сбой в файле main.go в папке errorHandling

В результате решение о том, как справиться с паникой, передается от горутины вызывающему коду, который может выбрать продолжение выполнения или инициировать новую панику в зависимости от проблемы. Скомпилируйте и запустите проект, и вы получите следующий вывод:

```
Watersports Total: 328
Chess Total: 1291
Cannot find category: Running
A terminal error occured
```

Резюме

В этой главе я описал возможности Go для обработки ошибок. Я описал тип `error` и показал, как создавать пользовательские ошибки и как использовать вспомогательные функции для создания ошибок с помощью простых сообщений. Я также объяснил панику, то есть то, как обрабатываются неисправимые ошибки. Я объяснил, что решение о том, является ли ошибка неисправимой, может быть субъективным, поэтому Go позволяет восстанавливать паники. Я объяснил процесс восстановления и продемонстрировал, как его можно адаптировать для эффективной работы в горутинах. В следующей главе я начинаю процесс описания стандартной библиотеки Go.

Часть II
Использование стандартной библиотеки Go

16. Обработка строк и регулярные выражения

В этой главе я описываю функции стандартной библиотеки для обработки строковых значений, которые необходимы почти в каждом проекте и которые во многих языках предоставляются в виде методов, определенных для встроенных типов. Но даже несмотря на то, что Go определяет эти функции в стандартной библиотеке, доступен полный набор функций, а также хорошая поддержка работы с регулярными выражениями. В таблице 16-1 эти функции представлены в контексте.

Таблица 16-1 Помещение обработки строк и регулярных выражений в контекст

Вопрос	Ответ
Кто они такие?	Обработка строк включает в себя широкий спектр операций, от обрезки пробелов до разделения строки на компоненты. Регулярные выражения — это шаблоны, которые позволяют кратко определить правила сопоставления строк.
Почему они полезны?	Эти операции полезны, когда приложению необходимо обработать `string` значения. Типичным примером является обработка HTTP-запросов.
Как они используются?	Эти функции содержатся в пакетах `strings` и `regexp`, которые являются частью стандартной библиотеки.
Есть ли подводные камни или ограничения?	Есть некоторые особенности в том, как выполняются некоторые из этих операций, но в основном они ведут себя так, как вы ожидаете.
Есть ли альтернативы?	Использование этих пакетов является необязательным, и их не обязательно использовать. Тем не менее, нет смысла создавать свои собственные реализации этих функций, поскольку стандартная библиотека хорошо написана и тщательно протестирована.

Таблица 16-2 суммирует главу.

Таблица 16-2 Краткое содержание главы

Проблема	Решение	Листинг

Проблема	Решение	Листинг
Сравнить строки	Используйте функцию Contains, EqualFold или Has* в пакете strings.	4
Преобразование строкового регистра	Используйте функцию ToLower, ToUpper, Title или ToTitle в пакете strings.	5, 6
Проверить или изменить регистр символов	Используйте функции, предоставляемые пакетом unicode.	7
Найти содержимое в строках	Используйте функции, предоставляемые пакетом strings или regexp	8, 9, 24–27, 29–32
Разделить строку	Используйте функцию Fields или Split* в пакетах strings и regexp	10–14, 28
Соединить строки	Используйте функцию Join или Repeat в пакете strings	22
Вырезать символы из строки	Используйте функции Trim* в пакете strings	15–18
Выполнить замену	Используйте функцию Replace* или Map в пакете strings, используйте Replacer или используйте функции Replace* в пакете regexp	19–21, 33
Эффективно построить строку	Используйте тип Builder в пакете strings	23

Подготовка к этой главе

Чтобы подготовиться к этой главе, откройте новую командную строку, перейдите в удобное место и создайте каталог с именем stringsandregexp. Запустите команду, показанную в листинге 16-1, чтобы создать файл модуля.

> **Подсказка**
> Вы можете загрузить пример проекта для этой главы — и для всех остальных глав этой книги — с https://github.com/apress/pro-go. См. Главу 2 о том, как получить помощь, если у вас возникнут проблемы с запуском примеров.

```
go mod init stringsandregexp
```

Листинг 16-1 Инициализация модуля

Добавьте файл с именем `main.go` в папку `stringsandregexp` с содержимым, показанным в листинге 16-2.

```
package main

import (
    "fmt"
)

func main() {

    product := "Kayak"

    fmt.Println("Product:", product)
}
```

Листинг 16-2 Содержимое файла main.go в папке stringsandregexp

Используйте командную строку для запуска команды, показанной в листинге 16-3, в папке `stringsandregexp`.

```
go run .
```

Листинг 16-3 Запуск примера проекта

Код будет скомпилирован и выполнен, что приведет к следующему выводу:

```
Product: Kayak
```

Обработка строк

Пакет `strings` предоставляет набор функций для обработки строк. В следующих разделах я опишу наиболее полезные функции пакета `strings` и продемонстрирую их использование.

Сравнение строк

Пакет `strings` предоставляет функции сравнения, как описано в таблице 16-3. Их можно использовать в дополнение к операторам равенства (`==` и `!=`).

Таблица 16-3 Функции для сравнения строк

Функция	Описание
`Contains(s, substr)`	Эта функция возвращает `true`, если строка `s` содержит `substr`, и `false`, если нет.
`ContainsAny(s, substr)`	Эта функция возвращает `true`, если строка `s` содержит любой из символов, содержащихся в строке `substr`.
`ContainsRune(s, rune)`	Эта функция возвращает true, если строка `s` содержит определенную руну (`rune`).
`EqualFold(s1, s2)`	Эта функция выполняет сравнение без учета регистра и возвращает `true`, если строки `s1` и `s2` совпадают.
`HasPrefix(s, prefix)`	Эта функция возвращает значение `true`, если строка `s` начинается с префикса (`prefix`) строки.
`HasSuffix(s, suffix)`	Эта функция возвращает значение `true`, если строка заканчивается суффиксом (`suffix`) строки.

В листинге 16-4 показано использование функций, описанных в таблице 16-3.

```
package main

import (
    "fmt"
    "strings"
)

func main() {

    product := "Kayak"

        fmt.Println("Contains:",  strings.Contains(product,
"yak"))
     fmt.Println("ContainsAny:", strings.ContainsAny(product,
"abc"))
                                  fmt.Println("ContainsRune:",
strings.ContainsRune(product, 'K'))
        fmt.Println("EqualFold:",  strings.EqualFold(product,
"KAYAK"))
        fmt.Println("HasPrefix:",  strings.HasPrefix(product,
"Ka"))
        fmt.Println("HasSuffix:",  strings.HasSuffix(product,
"yak"))
```

```
}
```

Листинг 16-4 Сравнение строк в файле main.go в папке stringsandregexp

Функции в таблице 16-3 выполняют сравнения с учетом регистра, за исключением функции EqualFold. (Сворачивание — это способ, которым Unicode работает с регистром символов, где символы могут иметь разные представления для строчных, прописных и заглавных букв.) Код в листинге 16-4 при выполнении выдает следующий вывод:

```
Contains: true
ContainsAny: true
ContainsRune: true
HasPrefix: true
HasSuffix: true
EqualFold: true
```

ИСПОЛЬЗОВАНИЕ БАЙТ-ОРИЕНТИРОВАННЫХ ФУНКЦИЙ

Для всех функций в пакете strings, которые работают с символами, в пакете bytes есть соответствующая функция, которая работает с байтовым срезом, например:

```
package main

import (
    "fmt"
    "strings"
    "bytes"
)

func main() {

    price := "€100"

                        fmt.Println("Strings      Prefix:",
strings.HasPrefix(price, "€"))
                         fmt.Println("Bytes       Prefix:",
bytes.HasPrefix([]byte(price),
        []byte { 226, 130 }))
}
```

В этом примере показано использование функции HasPrefix, предоставляемой обоими пакетами. Строковая версия пакета работает с символами и проверяет префикс независимо от того, сколько байтов используется символами. Это позволяет мне определить, начинается ли строка price с символа валюты евро. Байтовая версия функции позволяет мне определить, начинается ли переменная price с определенной последовательности байтов, независимо от того, как эти байты относятся к символу. В этой главе я использую функции из пакета strings, потому что они наиболее широко используются. В главе 25 я использую структуру bytes.Buffer, которая является удобным способом хранения двоичных данных в памяти.

Преобразование регистра строк

Пакет strings предоставляет функции, описанные в таблице 16-4, для изменения регистра строк.

Таблица 16-4 Регистровые функции в пакете strings

Функция	Описание
ToLower(str)	Эта функция возвращает новую строку, содержащую символы указанной строки, преобразованные в нижний регистр.
ToUpper(str)	Эта функция возвращает новую строку, содержащую символы указанной строки, преобразованные в нижний регистр.
Title(str)	Эта функция преобразует определенную строку таким образом, чтобы первый символ каждого слова был в верхнем регистре, а остальные символы — в нижнем.
ToTitle(str)	Эта функция возвращает новую строку, содержащую символы указанной строки, сопоставленные с заголовком.

Следует соблюдать осторожность с функциями Title и ToTitle, которые работают не так, как вы могли бы ожидать. Функция Title возвращает строку, подходящую для использования в качестве заголовка, но обрабатывает все слова одинаково, как показано в листинге 16-5.

```
package main

import (
```

```
    "fmt"
    "strings"
)

func main() {

    description := "A boat for sailing"

    fmt.Println("Original:", description)
    fmt.Println("Title:", strings.Title(description))
}
```

Листинг 16-5 Создание заголовка в файле main.go в папке stringsandregexp

Обычно в заглавном регистре артикли, короткие предлоги и союзы не пишутся с заглавной буквы, а это означает, что преобразование этой строки:

```
A boat for sailing
```

обычно преобразуется так:

```
A Boat for Sailing
```

Слово *for* не пишется с заглавной буквы, но другие слова пишутся с большой буквы. Но эти правила сложны, открыты для интерпретации и зависят от языка, поэтому Go использует более простой подход, заключающийся в написании всех слов с заглавной буквы. Эффект можно увидеть, скомпилировав и запустив код из листинга 16-5, который выдает следующий результат:

```
Original: A boat for sailing
Title: A Boat For Sailing
```

В некоторых языках есть символы, внешний вид которых меняется, когда они используются в названии. Unicode определяет три состояния для каждого символа — строчные, прописные и заглавные, а функция `ToTitle` возвращает строку, содержащую только заглавные символы. Это имеет тот же эффект, что и функция `ToUpper` для английского языка, но может давать другие результаты на других языках, что продемонстрировано в листинге 16-6.

```
package main

import (
    "fmt"
    "strings"
)

func main() {

    specialChar := "\u01c9"

                        fmt.Println("Original:",      specialChar,
[]byte(specialChar))

    upperChar := strings.ToUpper(specialChar)
    fmt.Println("Upper:", upperChar, []byte(upperChar))

    titleChar := strings.ToTitle(specialChar)
    fmt.Println("Title:", titleChar, []byte(titleChar))
}
```

Листинг 16-6 Использование регистра заголовков в файле main.go в папке stringsandregexp

Мои ограниченные языковые навыки не распространяются на язык, который требует другого регистра заглавий, поэтому я использовал escape-последовательность Unicode для выбора символа. (Я получил код символа из спецификации Unicode.) При компиляции и выполнении код в листинге 16-6 записывает строчные, прописные и заглавные версии символа вместе с байтами, которые используются для его представления:

```
Original: ǉ [199 137]
Upper: Ǉ [199 135]
Title: ǈ [199 136]
```

Вы можете увидеть разницу в том, как отображается символ, но даже если нет, вы можете увидеть, что для верхнего и заглавного регистра используется другая комбинация значений байтов.

ЛОКАЛИЗАЦИЯ: ВСЕ ИЛИ НИЧЕГО

Локализация продукта требует времени, усилий и ресурсов, и ее должен выполнять кто-то, кто понимает языковые, культурные и денежные правила целевой страны или региона. Если вы не локализуете должным образом, то результат может быть хуже, чем вообще отсутствие локализации.

Именно по этой причине я не описываю детали локализации ни в этой, ни в других своих книгах. Описание функций вне контекста, в котором они будут использоваться, похоже на то, что настраивает читателей на собственную катастрофу. По крайней мере, если продукт не локализован, пользователь знает, где он стоит, и ему не нужно пытаться выяснить, то ли вы просто забыли изменить код валюты, то ли эти цены действительно указаны в долларах США. (Это проблема, с которой я постоянно сталкиваюсь, живя в Соединенном Королевстве.)

Вы *должны* локализовать свои продукты. Ваши пользователи *должны* иметь возможность вести бизнес или выполнять другие операции удобным для них способом. Но вы *должны* отнестись к этому серьезно и выделить время и усилия, необходимые для того, чтобы сделать это должным образом.

Работа с регистром символов

Пакет `unicode` предоставляет функции, которые можно использовать для определения или изменения регистра отдельных символов, как описано в таблице 16-5.

Таблица 16-5 Функции в пакете unicode для регистра символов

Функция	Описание
`IsLower(rune)`	Эта функция возвращает `true`, если указанная руна в нижнем регистре.
`ToLower(rune)`	Эта функция возвращает строчную руну, связанную с указанной руной.
`IsUpper(rune)`	Эта функция возвращает значение `true`, если указанная руна написана в верхнем регистре.
`ToUpper(rune)`	Эта функция возвращает верхнюю руну, связанную с указанной руной.
`IsTitle(rune)`	Эта функция возвращает `true`, если указанная руна является заглавной.
`ToTitle(rune)`	Эта функция возвращает руну в заглавном регистре, связанную с указанной руной.

В листинге 16-7 используются функции, описанные в таблице 16-5, для проверки и изменения регистра рун.

```
package main

import (
    "fmt"
    //"strings"
    "unicode"
)

func main() {

    product := "Kayak"

    for _, char := range product {
               fmt.Println(string(char), "Upper case:",
unicode.IsUpper(char))
    }
}
```

Листинг 16-7 Использование функций регистра рун в файле main.go в папке stringsandregexp

Код в листинге 16-7 перечисляет символы в строке `product`, чтобы определить, являются ли они прописными. Код выдает следующий результат при компиляции и выполнении:

```
K Upper case: true
a Upper case: false
y Upper case: false
a Upper case: false
k Upper case: false
```

Проверка строк

Функции в таблице 16-6 предоставляются пакетом `strings` для проверки строк.

Таблица 16-6 Функции для проверки строк

Функция	Описание

Функция	Описание
Count(s, sub)	Эта функция возвращает int, которое сообщает, сколько раз указанная подстрока встречается в строке s.
Index(s, sub) LastIndex(s, sub)	Эти функции возвращают индекс первого или последнего вхождения указанной строки подстроки в строке s или -1, если вхождения нет.
IndexAny(s, chars) LastIndexAny(s, chars)	Эти функции возвращают первое или последнее вхождение любого символа в указанной строке в пределах строки s или -1, если вхождения нет.
IndexByte(s, b) LastIndexByte(s, b)	Эти функции возвращают индекс первого или последнего вхождения указанного byte в строке s или -1, если вхождения нет.
IndexFunc(s, func) LastIndexFunc(s, func)	Эти функции возвращают индекс первого или последнего вхождения символа в строку s, для которого указанная функция возвращает значение true, как описано в разделе «Проверка строк с помощью пользовательских функций».

В листинге 16-8 показаны функции, описанные в таблице 16-6, некоторые из которых обычно используются для разделения строк на основе их содержимого.

```
package main

import (
    "fmt"
    "strings"
    //"unicode"
)

func main() {

    description := "A boat for one person"

    fmt.Println("Count:", strings.Count(description, "o"))
    fmt.Println("Index:", strings.Index(description, "o"))
     fmt.Println("LastIndex:", strings.LastIndex(description,
"o"))
       fmt.Println("IndexAny:",  strings.IndexAny(description,
"abcd"))
     fmt.Println("LastIndex:", strings.LastIndex(description,
"o"))
```

```
                                    fmt.Println("LastIndexAny:",
strings.LastIndexAny(description, "abcd"))
}
```

Листинг 16-8 Проверка строк в файле main.go в папке stringsandregexp

Сравнения, выполняемые этими функциями, чувствительны к регистру, что означает, что строка, используемая для тестирования в листинге 16-8, содержит, например, `person`, но не `Person`. Для сравнения случаев объедините функции, описанные в таблице 16-6, с функциями из таблиц 16-4 и 16-5. Код в листинге 16-8 выдает следующий результат при компиляции и выполнении:

```
Count: 4
Index: 3
LastIndex: 19
IndexAny: 2
LastIndex: 19
LastIndexAny: 4
```

Проверка строк с помощью пользовательских функций

Функции `IndexFunc` и `LastIndexFunc` используют пользовательскую функцию для проверки строк с помощью пользовательских функций, как показано в листинге 16-9.

```
package main

import (
    "fmt"
    "strings"
)

func main() {

    description := "A boat for one person"

    isLetterB := func (r rune) bool {
        return r == 'B' || r == 'b'
    }

     fmt.Println("IndexFunc:", strings.IndexFunc(description,
isLetterB))
```

```
}
```

Листинг 16-9 Проверка строк с пользовательской функцией в файле main.go в папке stringsandregexp

Пользовательские функции получают `rune` и возвращают `bool` результат, указывающий, соответствует ли символ желаемому условию. Функция `IndexFunc` вызывает пользовательскую функцию для каждого символа в строке до тех пор, пока не будет получен `true` результат, после чего возвращается индекс.

Переменной `isLetterB` назначается пользовательская функция, которая получает руну и возвращает значение `true`, если руна представляет собой прописную или строчную букву `B`. Пользовательская функция передается функции `strings.IndexFunc`, которая производит следующий вывод при компиляции и выполнении кода:

```
IndexFunc: 2
```

Манипулирование строками

Пакет `strings` предоставляет полезные функции для редактирования строк, включая замену некоторых или всех символов или удаление пробелов.

Разделение строк

Первый набор функций, описанный в таблице 16-7, используется для разделения строк. (Существует также полезная функция для разделения строк с помощью регулярных выражений, описанная в разделе «Использование регулярных выражений» далее в этой главе.)

Таблица 16-7 Функции для разделения строк в пакете strings

Функция	Описание
`Fields(s)`	Эта функция разбивает строку на пробельные символы и возвращает срез, содержащий непробельные разделы строки `s`.
`FieldsFunc(s, func)`	Эта функция разбивает строку `s` на символы, для которых пользовательская функция возвращает значение `true`, и возвращает срез, содержащий оставшиеся части строки.

Функция	Описание
Split(s, sub)	Эта функция разбивает строку s на каждое вхождение указанной подстроки, возвращая string срез. Если разделителем является пустая строка, то срез будет содержать строки для каждого символа.
SplitN(s, sub, max)	Эта функция похожа на Split, но принимает дополнительный аргумент типа int, указывающий максимальное количество возвращаемых подстрок. Последняя подстрока результирующего среза будет содержать неразделенную часть исходной строки.
SplitAfter(s, sub)	Эта функция похожа на Split, но включает подстроку, используемую в результатах. См. текст после таблицы для демонстрации.
SplitAfterN(s, sub, max)	Эта функция похожа на SplitAfter, но принимает дополнительный аргумент типа int, указывающий максимальное количество возвращаемых подстрок.

Функции, описанные в таблице 16-7, выполняют ту же основную задачу. Разница между функциями Split и SplitAfter заключается в том, что функция Split исключает подстроку, используемую для разделения, из результатов, как показано в листинге 16-10.

```
package main

import (
    "fmt"
    "strings"
)

func main() {

    description := "A boat for one person"

    splits := strings.Split(description, " ")
    for _, x := range splits {
        fmt.Println("Split >>" + x + "<<")
    }

    splitsAfter := strings.SplitAfter(description, " ")
    for _, x := range splitsAfter {
        fmt.Println("SplitAfter >>" + x + "<<")
    }
}
```

Листинг 16-10 Разделение строк в файле main.go в папке stringsandregexp

Чтобы подчеркнуть разницу, код в листинге 16-10 разбивает одну и ту же строку с помощью функций Split и SplitAfter. Результаты обеих функций перечисляются с использованием циклов for, а сообщения, которые циклы выписывают, заключают результаты в шевроны без пробелов до или после результата. Скомпилируйте и выполните код, и вы увидите следующие результаты:

```
Split >>A<<
Split >>boat<<
Split >>for<<
Split >>one<<
Split >>person<<
SplitAfter >>A <<
SplitAfter >>boat <<
SplitAfter >>for <<
SplitAfter >>one <<
SplitAfter >>person<<
```

Строки разделены пробелом. Как показывают результаты, символ пробела не включается в результаты, полученные функцией Split, но включается в результаты функции SplitAfter.

Ограничение количества результатов

Функции SplitN и SplitAfterN принимают аргумент типа int, указывающий максимальное количество результатов, которые должны быть включены в результаты, как показано в листинге 16-11.

```
package main

import (
    "fmt"
    "strings"
)

func main() {

    description := "A boat for one person"

    splits := strings.SplitN(description, " ", 3)
    for _, x := range splits {
        fmt.Println("Split >>" + x + "<<")
```

```
    }

    // splitsAfter := strings.SplitAfter(description, " ")
    // for _, x := range splitsAfter {
    //     fmt.Println("SplitAfter >>" + x + "<<")
    // }
}
```

Листинг 16-11 Ограничение результатов в файле main.go в папке stringsandregexp

Если строку можно разделить на большее количество строк, чем было указано, то последний элемент результирующего среза будет неразделенным остатком строки. В листинге 16-11 указано не более трех результатов, что означает, что первые два элемента в срезе будут разделены как обычно, а третий элемент будет остатком строки. Скомпилируйте и выполните код, и вы увидите следующий вывод:

```
Split >>A<<
Split >>boat<<
Split >>for one person<<
```

Разделение на пробельные символы

Одним из ограничений функций `Split`, `SplitN`, `SplitAfter` и `SplitAfterN` является то, что они не работают с повторяющимися последовательностями символов, что может стать проблемой при разбиении строки на пробельные символы, как показано в листинге 16-12.

```
package main

import (
    "fmt"
    "strings"
)

func main() {

    description := "This  is  double  spaced"

    splits := strings.SplitN(description, " ", 3)
    for _, x := range splits {
```

```
        fmt.Println("Split >>" + x + "<<")
    }
}
```

Листинг 16-12 Разделение по пробелам в файле main.go в папке stringsandregexp

Слова в исходной строке разделены двойным интервалом, но функция SplitN разделяет только первый символ пробела, что приводит к странным результатам. Скомпилируйте и выполните код, и вы увидите следующий вывод:

```
Split >>This<<
Split >><<
Split >>is  double  spaced<<
```

Второй элемент результирующего среза — пробел. Для обработки повторяющихся пробельных символов функция Fields разбивает строки на любой пробельный символ, как показано в листинге 16-13.

```
package main

import (
    "fmt"
    "strings"
)

func main() {

    description := "This  is  double  spaced"

    splits := strings.Fields(description)
    for _, x := range splits {
        fmt.Println("Field >>" + x + "<<")
    }
}
```

Листинг 16-13 Использование функции полей в файле main.go в папке stringsandregexp

Функция Fields не поддерживает ограничение на количество результатов, но правильно обрабатывает двойные пробелы. Скомпилируйте и запустите проект, и вы увидите следующий вывод:

```
Field >>This<<
Field >>is<<
Field >>double<<
Field >>spaced<<
```

Разделение с использованием пользовательской функции для разделения строк

Функция `FieldsFunc` разделяет строку, передавая каждый символ пользовательской функции и разделяя ее, когда эта функция возвращает значение `true`, как показано в листинге 16-14.

```
package main

import (
    "fmt"
    "strings"
)

func main() {

    description := "This  is  double  spaced"

    splitter := func(r rune) bool {
        return r == ' '
    }

    splits := strings.FieldsFunc(description, splitter)
    for _, x := range splits {
        fmt.Println("Field >>" + x + "<<")
    }
}
```

Листинг 16-14 Разделение с помощью пользовательской функции в файле main.go в папке stringsandregexp

Пользовательская функция получает `rune` и возвращает значение `true`, если эта `rune` должна привести к разделению строки. Функция `FieldsFunc` достаточно умна, чтобы работать с повторяющимися символами, такими как двойные пробелы в листинге 16-14.

Примечание

Я указал пробел в листинге 16-14, чтобы подчеркнуть, что функция FieldsFunc работает с повторяющимися символами. Функция Fields имеет лучший подход, заключающийся в разделении на любой символ, для которого функция IsSpace в пакете unicode возвращает значение true.

Скомпилируйте и запустите проект, и вы увидите следующий вывод:

```
Field >>This<<
Field >>is<<
Field >>double<<
Field >>spaced<<
```

Обрезка строк

Процесс обрезки удаляет начальные и конечные символы из строки и чаще всего используется для удаления пробельных символов. Таблица 16-8 описывает функции, предоставляемые пакетом strings для обрезки.

Таблица 16-8 Функции обрезки строк в пакете strings

Функция	Описание
TrimSpace(s)	Эта функция возвращает строку s без начальных и конечных пробельных символов.
Trim(s, set)	Эта функция возвращает строку, из которой удаляются все начальные или конечные символы, содержащиеся в наборе (set) строк, из строки s.
TrimLeft(s, set)	Эта функция возвращает строку s без какого-либо начального символа, содержащегося в наборе (set) строк. Эта функция соответствует любому из указанных символов — используйте функцию TrimPrefix для удаления полной подстроки.
TrimRight(s, set)	Эта функция возвращает строку s без каких-либо завершающих символов, содержащихся в наборе (set) строк. Эта функция соответствует любому из указанных символов — используйте функцию TrimSuffix для удаления полной подстроки.
TrimPrefix(s, prefix)	Эта функция возвращает строку s после удаления указанной строки префикса. Эта функция удаляет всю строку префикса (prefix) — используйте функцию TrimLeft для удаления символов из набора.
TrimSuffix(s, suffix)	Эта функция возвращает строку s после удаления указанной строки суффикса (suffix). Эта функция удаляет всю строку суффикса — используйте функцию TrimRight для удаления символов из набора.

Функция	Описание
`TrimFunc(s, func)`	Эта функция возвращает строку `s`, из которой удаляются все начальные или конечные символы, для которых пользовательская функция возвращает значение `true`.
`TrimLeftFunc(s, func)`	Эта функция возвращает строку `s`, из которой удаляются все начальные символы, для которых пользовательская функция возвращает значение `true`.
`TrimRightFunc(s, func)`	Эта функция возвращает строку `s`, из которой удаляются все завершающие символы, для которых пользовательская функция возвращает значение `true`.

Обрезка пробелов

Функция `TrimSpace` выполняет наиболее распространенную задачу обрезки, заключающуюся в удалении любых начальных или конечных пробельных символов. Это особенно полезно при обработке пользовательского ввода, где пробелы могут быть введены случайно и вызовут путаницу, если их не удалить, например, при вводе имен пользователей, как показано в листинге 16-15.

```
package main

import (
    "fmt"
    "strings"
)

func main() {

    username := " Alice"
    trimmed := strings.TrimSpace(username)
    fmt.Println("Trimmed:", ">>" + trimmed + "<<")
}
```

Листинг 16-15 Обрезка пробелов в файле main.go в папке stringsandregexp

Пользователь может не осознавать, что нажал пробел при вводе имени, и можно избежать путаницы, обрезав введенное имя перед его использованием. Скомпилируйте и запустите пример проекта, и вы увидите отображаемое усеченное имя:

```
Trimmed: >>Alice<<
```

Обрезка наборов символов

Функции Trim, TrimLeft и TrimRight соответствуют любому символу в указанной строке. В листинге 16-16 показано использование функции Trim. Другие функции работают так же, но обрезают только начало или конец строки.

```
package main

import (
    "fmt"
    "strings"
)

func main() {

    description := "A boat for one person"

    trimmed := strings.Trim(description, "Asno ")

    fmt.Println("Trimmed:", trimmed)
}
```

Листинг 16-16 Обрезка символов в файле main.go в папке stringsandregexp

В листинге 16-16 я указал буквы A, s, n, o и пробел при вызове функции Trim. Функция выполняет сопоставление с учетом регистра, используя любой из символов в наборе, и пропускает любые совпадающие символы из результата. Сопоставление останавливается, как только будет найден символ, не входящий в набор. Процесс выполняется с начала строки для префикса и конца строки для суффикса. Если строка не содержит ни одного из символов набора, функция Trim вернет строку без изменений.

Например, это означает, что буква A и пробел в начале строки будут обрезаны, а буквы s, o и n будут обрезаны с конца строки. Скомпилируйте и выполните проект, и на выходе будет показана обрезанная строка:

```
Trimmed: boat for one per
```

Обрезка подстрок

Функции TrimPrefix и TrimSuffix обрезают подстроки, а не символы из набора, как показано в листинге 16-17.

```
package main

import (
    "fmt"
    "strings"
)

func main() {

    description := "A boat for one person"

     prefixTrimmed := strings.TrimPrefix(description, "A boat
")
    wrongPrefix := strings.TrimPrefix(description, "A hat ")

    fmt.Println("Trimmed:", prefixTrimmed)
    fmt.Println("Not trimmed:", wrongPrefix)
}
```

Листинг 16-17 Обрезка подстрок в файле main.go в папке stringsandregexp

Начало или конец целевой строки должны точно соответствовать указанному префиксу или суффиксу; в противном случае результатом функции обрезки будет исходная строка. В листинге 16-17 я дважды использую функцию TrimPrefix, но только один из них использует префикс, соответствующий началу строки, что дает следующие результаты при компиляции и выполнении кода:

```
Trimmed: for one person
Not trimmed: A boat for one person
```

Обрезка с помощью пользовательских функций

Функции TrimFunc, TrimLeftFunc и TrimRightFunc обрезают строки с помощью пользовательских функций, как показано в листинге 16-18.

```
package main

import (
```

```
    "fmt"
    "strings"
)

func main() {

    description := "A boat for one person"

    trimmer := func(r rune) bool {
        return r == 'A' || r == 'n'
    }

    trimmed := strings.TrimFunc(description, trimmer)
    fmt.Println("Trimmed:", trimmed)
}
```

Листинг 16-18 Обрезка с помощью пользовательской функции в файле main.go в папке stringsandregexp

Пользовательская функция вызывается для символов в начале и в конце строки, и символы будут обрезаться до тех пор, пока функция не вернет значение `false`. Скомпилируйте и выполните пример, и вы получите следующий вывод, в котором первый и последний символы были обрезаны из строки:

```
Trimmed:  boat for one perso
```

Изменение строк

Функции, описанные в таблице 16-9, предоставляются пакетом `strings` для изменения содержимого строк.

Таблица 16-9 Функции для изменения строк в пакете strings

Функция	Описание
`Replace(s, old, new, n)`	Эта функция изменяет строку `s`, заменяя вхождения строки `old` на строку `new`. Максимальное количество заменяемых вхождений определяется аргументом `int n`.
`ReplaceAll(s, old, new)`	Эта функция изменяет строку `s`, заменяя все вхождения строки `old` строкой `new`. В отличие от функции `Replace`, количество заменяемых вхождений не ограничено.

Функция	Описание
`Map(func, s)`	Эта функция генерирует строку, вызывая пользовательскую функцию для каждого символа в строке `s` и объединяя результаты. Если функция выдает отрицательное значение, текущий символ отбрасывается без замены.

Функции `Replace` и `ReplaceAll` находят подстроки и заменяют их. Функция `Replace` позволяет указать максимальное количество изменений, а функция `ReplaceAll` заменит все вхождения найденной подстроки, как показано в листинге 16-19.

```
package main

import (
    "fmt"
    "strings"
)

func main() {

    text := "It was a boat. A small boat."

    replace := strings.Replace(text, "boat", "canoe", 1)
    replaceAll := strings.ReplaceAll(text, "boat", "truck")

    fmt.Println("Replace:", replace)
    fmt.Println("Replace All:", replaceAll)
}
```

Листинг 16-19 Замена подстрок в файле main.go в папке stringsandregexp

В листинге 16-19 функция `Replace` используется для замены одного экземпляра слова `boat`, а функция `ReplaceAll` используется для замены каждого экземпляра. Скомпилируйте и выполните код, и вы увидите следующий вывод:

```
Replace: It was a canoe. A small boat.
Replace All: It was a truck. A small truck.
```

Изменение строк с помощью функции карты Map

Функция `Map` изменяет строки, вызывая функцию для каждого символа и объединяя результаты для формирования новой строки, как показано

в листинге 16-20.

```
package main

import (
    "fmt"
    "strings"
)

func main() {

    text := "It was a boat. A small boat."

    mapper := func(r rune) rune {
        if r == 'b' {
            return 'c'
        }
        return r
    }

    mapped := strings.Map(mapper, text)
    fmt.Println("Mapped:", mapped)
}
```

Листинг 16-20 Использование функции карты в файле main.go в папке stringsandregexp

Функция отображения в листинге 16-20 заменяет символ `b` на символ `c` и передает все остальные символы без изменений. Скомпилируйте и запустите проект, и вы увидите следующие результаты:

```
Mapped: It was a coat. A small coat.
```

Использование заменителя строк

Пакет strings экспортирует тип структуры с именем `Replacer`, который используется для замены строк, предоставляя альтернативу функциям, описанным в таблице 16-10. Листинг 16-21 демонстрирует использование `Replacer`.

```
package main

import (
```

```
    "fmt"
    "strings"
)

func main() {

    text := "It was a boat. A small boat."

    replacer := strings.NewReplacer("boat", "kayak", "small",
"huge")

    replaced := replacer.Replace(text)

    fmt.Println("Replaced:", replaced)
}
```

Листинг 16-21 Использование Replacer в файле main.go в папке stringsandregexp

Таблица 16-10 Методы замены

Функция	Описание
`Replace(s)`	Этот метод возвращает строку, для которой все замены, указанные в конструкторе, были выполнены в строке `s`.
`WriteString(writer, s)`	Этот метод используется для выполнения замен, указанных в конструкторе, и записи результатов в `io.Writer`, описанный в главе 20.

Функция-конструктор с именем `NewReplacer` используется для создания `Replacer` и принимает пары аргументов, которые определяют подстроки и их замены. Таблица 16-10 описывает методы, определенные для типа `Replacer`.

Конструктор, использованный для создания `Replacer` в листинге 16-21, указывает, что экземпляры `boat` должны быть заменены на `kayak`, а экземпляры `small` должны быть заменены на `huge`. Метод `Replace` вызывается для выполнения замены, производя следующий вывод, когда код компилируется и выполняется:

```
Replaced: It was a kayak. A huge kayak.
```

Построение и генерация строк

Пакет strings предоставляет две функции для генерации строк и тип структуры, методы которого можно использовать для эффективного постепенного построения строк. Таблица 16-11 описывает функции.

Таблица 16-11 Функции для генерации строк

Функция	Описание
Join(slice, sep)	Эта функция объединяет элементы в указанном срезе строки с указанной строкой-разделителем, помещенной между элементами.
Repeat(s, count)	Эта функция генерирует строку, повторяя строку s указанное количество раз.

Из этих двух функций наиболее полезна функция Join, поскольку ее можно использовать для рекомбинации разделенных строк, как показано в листинге 16-22.

```
package main

import (
    "fmt"
    "strings"
)

func main() {

    text := "It was a boat. A small boat."

    elements := strings.Fields(text)
    joined := strings.Join(elements, "--")
    fmt.Println("Joined:", joined)
}
```

Листинг 16-22 Разделение и объединение строки в файле main.go в папке stringsandregexp

В этом примере функция Fields используется для разделения строки на пробельные символы и соединения элементов двумя дефисами в качестве разделителя. Скомпилируйте и запустите проект, и вы получите следующий вывод:

```
Joined: It--was--a--boat.--A--small--boat.
```

Строительные строки

Пакет `strings` предоставляет тип `Builder`, который не имеет экспортированных полей, но предоставляет набор методов, которые можно использовать для эффективного постепенного построения строк, как описано в таблице 16-12.

Таблица 16-12 The strings.Builder Methods

Функция	Описание
`WriteString(s)`	Этот метод добавляет строку `s` к строящейся строке.
`WriteRune(r)`	Этот метод добавляет символ `r` к строящейся строке.
`WriteByte(b)`	Этот метод добавляет байт `b` к строящейся строке.
`String()`	Этот метод возвращает строку, созданную компоновщиком.
`Reset()`	Этот метод сбрасывает строку, созданную построителем.
`Len()`	Этот метод возвращает количество байтов, используемых для хранения строки, созданной компоновщиком.
`Cap()`	Этот метод возвращает количество байтов, выделенных компоновщиком.
`Grow(size)`	Этот метод увеличивает количество байтов, используемых компоновщиком для хранения строящейся строки.

Общий шаблон заключается в создании `Builder`; составить строку с помощью функций `WriteString`, `WriteRune` и `WriteByte`; и получите строку, созданную с помощью метода `String`, как показано в листинге 16-23.

```
package main

import (
    "fmt"
    "strings"
)

func main() {

    text := "It was a boat. A small boat."

    var builder strings.Builder

    for _, sub := range strings.Fields(text) {
```

```
        if (sub == "small") {
            builder.WriteString("very ")
        }
        builder.WriteString(sub)
        builder.WriteRune(' ')
    }

    fmt.Println("String:", builder.String())
}
```

Листинг 16-23 Создание строки в файле main.go в папке stringsandregexp

Создание строки с помощью `Builder` более эффективно, чем использование оператора конкатенации для обычных `string` значений, особенно если метод `Grow` используется для предварительного выделения памяти.

> **Осторожно**
> Необходимо соблюдать осторожность при использовании указателей при передаче значений `Builder` в функции и методы и из них; в противном случае повышение эффективности будет потеряно при копировании `Builder`.

Скомпилируйте и запустите проект, и вы получите следующий вывод:

```
String: It was a boat. A very small boat.
```

Использование регулярных выражений

Пакет `regexp` обеспечивает поддержку регулярных выражений, которые позволяют находить сложные шаблоны в строках. Таблица 16-13 описывает основные функции регулярных выражений.

Таблица 16-13 Основные функции, предоставляемые пакетом regexp

Функция	Описание
`Match(pattern, b)`	Эта функция возвращает `bool` значение, указывающее, соответствует ли шаблон байтовому срезу `b`.
`MatchString(patten, s)`	Эта функция возвращает `bool` значение, указывающее, соответствует ли шаблон строке `s`.

Функция	Описание
Compile(pattern)	Эта функция возвращает RegExp, который можно использовать для повторного сопоставления с указанным шаблоном, как описано в разделе «Компиляция и повторное использование шаблонов».
MustCompile(pattern)	Эта функция предоставляет те же возможности, что и Compile, но вызывает панику, как описано в главе 15, если указанный шаблон не может быть скомпилирован.

Примечание

Регулярные выражения, используемые в этом разделе, выполняют базовые сопоставления, но пакет regexp поддерживает расширенный синтаксис шаблонов, который описан по адресу https://pkg.go.dev/regexp/syntax@go1.17.1.

Метод MatchString — самый простой способ определить, соответствует ли строка регулярному выражению, как показано в листинге 16-24.

```
package main

import (
    "fmt"
    //"strings"
    "regexp"
)

func main() {

    description := "A boat for one person"

    match, err := regexp.MatchString("[A-z]oat", description)

    if (err == nil) {
        fmt.Println("Match:", match)
    } else {
        fmt.Println("Error:", err)
    }
}
```

Листинг 16-24 Использование регулярного выражения в файле main.go в папке stringsandregexp

Функция MatchString принимает шаблон регулярного выражения и строку для поиска. Результатом функции MatchString является bool значение, которое true, если есть совпадение, и ошибка, которая будет nil, если не было проблем с выполнением совпадения. Ошибки с регулярными выражениями обычно возникают, если шаблон не может быть обработан.

Шаблон, используемый в листинге 16-24, будет соответствовать любому символу верхнего или нижнего регистра от A до Z, за которым следует символ oat в нижнем регистре. Шаблон будет соответствовать слову boat в строке description, что приведет к следующему результату при компиляции и выполнении кода:

```
Match: true
```

Компиляция и повторное использование шаблонов

Функция MatchString проста и удобна, но все возможности регулярных выражений доступны через функцию Compile, которая компилирует шаблон регулярного выражения, чтобы его можно было использовать повторно, как показано в листинге 16-25.

```
package main

import (
    "fmt"
    "regexp"
)

func main() {

    pattern, compileErr := regexp.Compile("[A-z]oat")

    description := "A boat for one person"
    question := "Is that a goat?"
    preference := "I like oats"

    if (compileErr == nil) {
                                  fmt.Println("Description:",
pattern.MatchString(description))
                                     fmt.Println("Question:",
pattern.MatchString(question))
```

```
                                    fmt.Println("Preference:",
pattern.MatchString(preference))
    } else {
        fmt.Println("Error:", compileErr)
    }
}
```

Листинг 16-25 Компиляция шаблона в файле main.go в папке stringsandregexp

Это более эффективно, потому что шаблон нужно скомпилировать только один раз. Результатом функции Compile является экземпляр типа RegExp, который определяет функцию MatchString. Код в листинге 16-25 выдает следующий результат при компиляции и выполнении:

```
Description: true
Question: true
Preference: false
```

Компиляция шаблона также предоставляет доступ к методам использования функций регулярных выражений, наиболее полезные из которых описаны в таблице 16-14. Методы, описанные в этой главе, работают со строками, но тип RegExp также предоставляет методы, которые используются для обработки байтовых срезов, и методы, работающие со средствами чтения, которые являются частью поддержки ввода-вывода в Go и описаны в главе 20.

Таблица 16-14 Полезные базовые методы регулярных выражений

Функция	Описание
MatchString(s)	Этот метод возвращает true, если строка s соответствует скомпилированному шаблону.
FindStringIndex(s)	Этот метод возвращает int срез, содержащий расположение самого левого совпадения, сделанного скомпилированным шаблоном в строке s. Результат nil означает, что совпадений не было.
FindAllStringIndex(s, max)	Этот метод возвращает срез int срезов, содержащих расположение всех совпадений, сделанных скомпилированным шаблоном в строке s. Результат nil означает, что совпадений не было.
FindString(s)	Этот метод возвращает строку, содержащую самое левое совпадение, сделанное скомпилированным шаблоном в строке s. Пустая строка будет возвращена, если совпадений нет.

Функция	Описание
`FindAllString(s, max)`	Этот метод возвращает срез строки, содержащий совпадения, сделанные скомпилированным шаблоном в строке `s`. Аргумент `int max` указывает максимальное количество совпадений, а `-1` указывает отсутствие ограничения. Если совпадений нет, возвращается nil результат.
`Split(s, max)`	Этот метод разбивает строку `s`, используя совпадения из скомпилированного шаблона в качестве разделителей, и возвращает срез, содержащий разделенные подстроки.

Метод `MatchString` является альтернативой функции, описанной в таблице 16-3, и подтверждает, соответствует ли строка шаблону.

Методы `FindStringIndex` и `FindAllStringIndex` предоставляют позицию индекса совпадений, которую затем можно использовать для извлечения областей строки с использованием нотации диапазона массива/среза, как показано в листинге 16-26. (Обозначение диапазона описано в главе 7.)

```
package main

import (
    "fmt"
    "regexp"
)

func getSubstring(s string, indices []int) string {
    return string(s[indices[0]:indices[1]])
}

func main() {

    pattern := regexp.MustCompile("K[a-z]{4}|[A-z]oat")

    description := "Kayak. A boat for one person."

    firstIndex := pattern.FindStringIndex(description)
    allIndices := pattern.FindAllStringIndex(description, -1)

          fmt.Println("First  index",  firstIndex[0],  "-",
firstIndex[1],
        "=", getSubstring(description, firstIndex))
```

```
    for i, idx := range allIndices {
        fmt.Println("Index", i, "=", idx[0], "-",
            idx[1], "=", getSubstring(description, idx))
    }
}
```

Листинг 16-26 Получение индексов соответствия в файле main.go в папке stringsandregexp

Регулярное выражение в листинге 16-26 выполнит два совпадения со строкой `description`. Метод `FindStringIndex` возвращает только первое совпадение, работая слева направо. Совпадение выражается как `int` срез, где первое значение указывает начальное местоположение совпадения в строке, а второе число указывает количество совпадающих символов.

Метод `FindAllStringIndex` возвращает несколько совпадений и вызывается в листинге 16-26 со значением `-1`, указывающим, что должны быть возвращены все совпадения. Совпадения возвращаются в срезе `int` срезов (это означает, что каждое значение в срезе результата представляет собой срез `int` значений), каждый из которых описывает одно совпадение. В листинге 16-26 индексы используются для извлечения областей из строки с помощью функции с именем `getSubstring`, что дает следующие результаты при компиляции и выполнении:

```
First index 0 - 5 = Kayak
Index 0 = 0 - 5 = Kayak
Index 1 = 9 - 13 = boat
```

Если вам не нужно знать расположение совпадений, методы `FindString` и `FindAllString` более полезны, поскольку их результаты — это подстроки, соответствующие регулярному выражению, как показано в листинге 16-27.

```
package main

import (
    "fmt"
    "regexp"
)
```

```
// func getSubstring(s string, indices []int) string {
//     return string(s[indices[0]:indices[1]])
// }

func main() {

    pattern := regexp.MustCompile("K[a-z]{4}|[A-z]oat")

    description := "Kayak. A boat for one person."

    firstMatch := pattern.FindString(description)
    allMatches := pattern.FindAllString(description, -1)

    fmt.Println("First match:", firstMatch)

    for i, m := range allMatches {
        fmt.Println("Match", i, "=", m)
    }
}
```

Листинг 16-27 Получение подстрок соответствия в файле main.go в папке stringsandregexp

Скомпилируйте и запустите проект, и вы увидите следующий вывод:

```
First match: Kayak
Match 0 = Kayak
Match 1 = boat
```

Разделение строк с помощью регулярного выражения

Метод `Split` разделяет строку, используя совпадения, сделанные регулярным выражением, что может предоставить более гибкую альтернативу функциям разделения, описанным ранее в этой главе, как показано в листинге 16-28.

```
package main

import (
    "fmt"
    "regexp"
)
```

```
func main() {

    pattern := regexp.MustCompile(" |boat|one")

    description := "Kayak. A boat for one person."

    split := pattern.Split(description, -1)

    for _, s := range split {
        if s != "" {
            fmt.Println("Substring:", s)
        }
    }
}
```

Листинг 16-28 Разделение строки в файле main.go в папке stringsandregexp

Регулярное выражение в этом примере соответствует символу пробела или терминам `boat` и `one`. Строка `description` будет разделена при совпадении выражения. Одна странность метода `Split` заключается в том, что он вводит пустую строку в результаты вокруг точки, где были найдены совпадения, поэтому я отфильтровываю эти значения из среза результата в примере. Скомпилируйте и выполните код, и вы увидите следующие результаты:

```
Substring: Kayak.
Substring: A
Substring: for
Substring: person.
```

Использование подвыражений

Подвыражения позволяют получить доступ к частям регулярного выражения, что может упростить извлечение подстрок из соответствующей области. В листинге 16-29 приведен пример того, когда подвыражение может быть полезным.

```
package main

import (
    "fmt"
    "regexp"
```

```
)

func main() {

    pattern := regexp.MustCompile("A [A-z]* for [A-z]* person")

    description := "Kayak. A boat for one person."

    str := pattern.FindString(description)
    fmt.Println("Match:", str)
}
```

Листинг 16-29 Выполнение сопоставления в файле main.go в папке stringsandregexp

Шаблон в этом примере соответствует определенной структуре предложения, что позволяет мне сопоставить интересующую часть строки. Но большая часть структуры предложения статична, а две переменные части шаблона содержат то содержание, которое мне нужно. В этой ситуации метод `FindString` является грубым инструментом, поскольку он соответствует всему шаблону, включая статические области. Скомпилируйте и выполните код, и вы получите следующий вывод:

```
Match: A boat for one person
```

Я могу добавить подвыражения, чтобы идентифицировать важные области содержимого в шаблоне, как показано в листинге 16-30.

```
package main

import (
    "fmt"
    "regexp"
)

func main() {

    pattern := regexp.MustCompile("A ([A-z]*) for ([A-z]*) person")

    description := "Kayak. A boat for one person."
```

```
    subs := pattern.FindStringSubmatch(description)

    for _, s := range subs {
        fmt.Println("Match:", s)
    }
}
```

Листинг 16-30 Использование подвыражений в файле main.go в папке stringsandregexp

Подвыражения обозначаются круглыми скобками. В листинге 16-30 я определил два подвыражения, каждое из которых окружает переменную часть шаблона. Метод `FindStringSubmatch` выполняет ту же задачу, что и `FindString`, но также включает в свой результат подстроки, соответствующие выражениям. Скомпилируйте и выполните код, и вы увидите следующий вывод:

```
Match: A boat for one person
Match: boat
Match: one
```

Таблица 16-15 описывает методы `RegExp` для работы с подвыражениями.

Таблица 16-15 Regexp методы для подвыражений

Функция	Описание
`FindStringSubmatch(s)`	Этот метод возвращает срез, содержащий первое совпадение, сделанное шаблоном, и текст для подвыражений, определяемых шаблоном.
`FindAllStringSubmatch(s, max)`	Этот метод возвращает срез, содержащий все совпадения и текст подвыражений. Аргумент `int` используется для указания максимального количества совпадений. Значение `-1` указывает все совпадения.
`FindStringSubmatchIndex(s)`	Этот метод эквивалентен `FindStringSubmatch`, но возвращает индексы, а не подстроки.
`FindAllStringSubmatchIndex(s, max)`	Этот метод эквивалентен `FindAllStringSubmatch`, но возвращает индексы, а не подстроки.
`NumSubexp()`	Этот метод возвращает количество подвыражений.
`SubexpIndex(name)`	Этот метод возвращает индекс подвыражения с указанным именем или `-1`, если такого подвыражения нет.

Функция	Описание
SubexpNames()	Этот метод возвращает имена подвыражений, выраженные в том порядке, в котором они определены.

Использование именованных подвыражений

Подвыражениям можно давать имена, что усложняет понимание регулярного выражения, но облегчает обработку результатов. В листинге 16-31 показано использование именованных подвыражений.

```
package main

import (
    "fmt"
    "regexp"
)

func main() {

    pattern := regexp.MustCompile(
        "A (?P<type>[A-z]*) for (?P<capacity>[A-z]*) person")

    description := "Kayak. A boat for one person."

    subs := pattern.FindStringSubmatch(description)

    for _, name := range []string { "type", "capacity" } {
                                fmt.Println(name,    "=",
subs[pattern.SubexpIndex(name)])
    }
}
```

Листинг 16-31 Использование именованных подвыражений в файле main.go в папке stringsandregexp

Синтаксис для присвоения имен подвыражениям неудобен: в круглых скобках знак вопроса, за которым следует буква `P` в верхнем регистре, а затем имя в угловых скобках. Шаблон в листинге 16-31 определяет два именованных подвыражения:

```
...
```

```
pattern := regexp.MustCompile("A (?P<type>[A-z]*) for (?P<capacity>[A-z]*) person")
...
```

Подвыражениям присваиваются имена type и capacity. Метод SubexpIndex возвращает позицию именованного подвыражения в результатах, что позволяет мне получить подстроки, соответствующие подвыражениям type и capacity. Скомпилируйте и выполните пример, и вы увидите следующий вывод:

```
type = boat
capacity = one
```

Замена подстрок с помощью регулярного выражения

Последний набор методов RegExp используется для замены подстрок, соответствующих регулярному выражению, как описано в таблице 16-16.

Таблица 16-16 Regexp методы для замены подстрок

Функция	Описание
ReplaceAllString(s, template)	Этот метод заменяет совпадающую часть строки s указанным шаблоном, который расширяется перед включением в результат для включения подвыражений.
ReplaceAllLiteralString(s, sub)	Этот метод заменяет совпадающую часть строки s указанным содержимым, которое включается в результат без расширения для подвыражений.
ReplaceAllStringFunc(s, func)	Этот метод заменяет совпадающую часть строки s результатом, полученным указанной функцией..

Метод ReplaceAllString используется для замены части строки, соответствующей регулярному выражению, шаблоном, который может ссылаться на подвыражения, как показано в листинге 16-32.

```
package main

import (
    "fmt"
    "regexp"
)
```

```
func main() {

    pattern := regexp.MustCompile(
        "A (?P<type>[A-z]*) for (?P<capacity>[A-z]*) person")

    description := "Kayak. A boat for one person."

    template := "(type: ${type}, capacity: ${capacity})"
        replaced  :=  pattern.ReplaceAllString(description,
template)
    fmt.Println(replaced)
}
```

Листинг 16-32 Замена содержимого файла main.go в папке stringsandregexp

Результатом метода `ReplaceAllString` является строка с замененным содержимым. Шаблон может ссылаться на совпадения подвыражений по имени, например, `${type}`, или по положению, например, `${1}`. В листинге часть строки `description`, соответствующая шаблону, будет заменена шаблоном, содержащим совпадения для подвыражений `type` и `capacity`. Скомпилируйте и выполните код, и вы увидите следующий вывод:

```
Kayak. (type: boat, capacity: one).
```

Обратите внимание, что шаблон отвечает только за часть результата метода `ReplaceAllString` в листинге 16-32. Первая часть строки описания — слово `Kayak`, за которым следуют точка и пробел, не соответствует регулярному выражению и включается в результат без изменения.

> **Подсказка**
> Используйте метод `ReplaceAllLiteralString`, если вы хотите заменить содержимое без интерпретации новой подстроки для подвыражений.

Замена совпадающего контента функцией

Метод `ReplaceAllStringFunc` заменяет соответствующий раздел строки содержимым, сгенерированным функцией, как показано в

листинге 16-33.

```
package main

import (
    "fmt"
    "regexp"
)

func main() {

    pattern := regexp.MustCompile(
        "A (?P<type>[A-z]*) for (?P<capacity>[A-z]*) person")

    description := "Kayak. A boat for one person."

      replaced := pattern.ReplaceAllStringFunc(description,
func(s string) string {
        return "This is the replacement content"
    })
    fmt.Println(replaced)
}
```

Листинг 16-33 Замена содержимого функцией в файле main.go в папке stringsandregexp

Результат функции не обрабатывается для ссылок на подвыражения, что вы можете видеть в выводе, полученном при компиляции и выполнении кода:

```
Kayak. This is the replacement content.
```

Резюме

В этой главе я описал стандартные функции библиотеки для обработки строковых значений и применения регулярных выражений, которые предоставляются пакетами `strings`, `unicode` и `regexp`. В следующей главе я опишу связанные функции, которые позволяют форматировать и сканировать строки.

17. Форматирование и сканирование строк

В этой главе я описываю стандартные функции библиотеки для форматирования и сканирования строк. Форматирование — это процесс составления новой строки из одного или нескольких значений данных, а сканирование — это процесс разбора значений из строки. Таблица 17-1 помещает форматирование и сканирование строк в контекст.

Таблица 17-1 Форматирование и сканирование строк в контексте

Вопрос	**Ответ**
Кто они такие?	Форматирование — это процесс объединения значений в строку. Сканирование — это процесс разбора строки на наличие содержащихся в ней значений.
Почему они полезны?	Форматирование строки является распространенным требованием и используется для создания строк для всего, от ведения журнала и отладки до представления информации пользователю. Сканирование полезно для извлечения данных из строк, например из HTTP-запросов или пользовательского ввода.
Как они используются?	Оба набора функций предоставляются через функции, определенные в пакете `fmt`.
Есть ли подводные камни или ограничения?	Шаблоны, используемые для форматирования строк, могут быть трудночитаемыми, и нет встроенной функции, позволяющей создать форматированную строку, к которой автоматически добавляется символ новой строки.
Есть ли альтернативы?	С помощью функций шаблона, описанных в главе 23, можно создавать большие объемы текста и содержимого HTML.

Таблица 17-2 суммирует главу.

Таблица 17-2 Краткое содержание главы

Проблема	**Решение**	**Листинг**
Объединить значения данных, чтобы сформировать строку	Используйте основные функции форматирования, предоставляемые пакетом `fmt`.	5, 6

Проблема	Решение	Листинг
Указать структуру строки	Используйте функции `fmt`, которые используют шаблоны форматирования и используют глаголы форматирования	7–9, 11–18
Изменинить способ представления пользовательских типов данных	Реализовать интерфейс `Stringer`	10
Разобрать строку, чтобы получить содержащиеся в ней значения данных	Используйте функции сканирования, предоставляемые пакетом `fmt`	19–22

Подготовка к этой главе

Чтобы подготовиться к этой главе, откройте новую командную строку, перейдите в удобное место и создайте каталог с именем `usingstrings`. Запустите команду, показанную в листинге 17-1, чтобы создать файл модуля.

> **Подсказка**
> Вы можете загрузить пример проекта для этой главы — и для всех остальных глав этой книги — с `https://github.com/apress/pro-go`. См. Главу 2 о том, как получить помощь, если у вас возникнут проблемы с запуском примеров.

```
go mod init usingstrings
```

Листинг 17-1 Инициализация модуля

Добавьте файл с именем `product.go` в папку `usingstrings` с содержимым, показанным в листинге 17-2.

```
package main

type Product struct {
    Name, Category string
    Price float64
}

var Kayak = Product {
```

```
    Name: "Kayak",
    Category: "Watersports",
    Price: 275,
}

var Products = []Product {
    { "Kayak", "Watersports", 279 },
    { "Lifejacket", "Watersports", 49.95 },
    { "Soccer Ball", "Soccer", 19.50 },
    { "Corner Flags", "Soccer", 34.95 },
    { "Stadium", "Soccer", 79500 },
    { "Thinking Cap", "Chess", 16 },
    { "Unsteady Chair", "Chess", 75 },
    { "Bling-Bling King", "Chess", 1200 },
}
```

Листинг 17-2 Содержимое файла product.go в папке usingstrings

Добавьте файл с именем `main.go` в папку `usingstrings` с содержимым, показанным в листинге 17-3.

```
package main

import "fmt"

func main() {

            fmt.Println("Product:",  Kayak.Name,  "Price:",
Kayak.Price)
}
```

Листинг 17-3 Содержимое файла main.go в папке usingstrings

Используйте командную строку для запуска команды, показанной в листинге 17-4, в папке `usingstrings`.

```
go run .
```

Листинг 17-4 Запуск примера проекта

Код будет скомпилирован и выполнен, что приведет к следующему выводу:

```
Product: Kayak Price: 275
```

Написание строк

Пакет `fmt` предоставляет функции для составления и записи строк. Основные функции описаны в таблице 17-3. Некоторые из этих функций используют модули записи, которые являются частью поддержки Go для ввода/вывода и описаны в главе 20.

Таблица 17-3 Основные функции fmt для составления и записи строк

Функция	Описание
`Print(...vals)`	Эта функция принимает переменное количество аргументов и выводит их значения на стандартный вывод. Пробелы добавляются между значениями, которые не являются строками.
`Println(...vals)`	Эта функция принимает переменное количество аргументов и выводит их значения на стандартный вывод, разделенные пробелами и сопровождаемые символом новой строки.
`Fprint(writer, ...vals)`	Эта функция записывает переменное количество аргументов в указанный модуль записи, который я описываю в главе 20. Между значениями, не являющимися строками, добавляются пробелы.
`Fprintln(writer, ...vals)`	Эта функция записывает переменное количество аргументов в указанный модуль записи, который я описываю в главе 20, за которым следует символ новой строки. Между всеми значениями добавляются пробелы.

Примечание
Стандартная библиотека Go включает в себя пакет шаблонов, описанный в главе 23, который можно использовать для создания больших объемов текста и содержимого HTML.

Функции, описанные в таблице 17-3, добавляют пробелы между значениями в создаваемых ими строках, но делают это непоследовательно. Функции `Println` и `Fprintln` добавляют пробелы между всеми значениями, но функции `Print` и `Fprint` добавляют пробелы только между значениями, которые не являются строками. Это означает, что пары функций в таблице 17-3 отличаются не только добавлением символа новой строки, как показано в листинге 17-5.

```
package main
```

```
import "fmt"

func main() {

            fmt.Println("Product:",   Kayak.Name,   "Price:",
Kayak.Price)
    fmt.Print("Product:", Kayak.Name, "Price:", Kayak.Price,
"\n")
}
```

Листинг 17-5 Запись строк в файл main.go в папку usingstrings

Во многих языках программирования не было бы никакой разницы между строками, созданными операторами в листинге 17-5, потому что я добавил символ новой строки к аргументам, переданным функции `Print`. Но поскольку функция `Print` добавляет пробелы только между парами нестроковых значений, результаты будут другими. Скомпилируйте и выполните код, и вы увидите следующий вывод:

```
Product: Kayak Price: 275
Product:KayakPrice:275
```

Форматирование строк

Я использовал функцию `fmt.Println` для вывода в предыдущих главах. Я использовал эту функцию, потому что она проста, но не обеспечивает контроля над форматированием вывода, что означает, что она подходит для простой отладки, но не для генерации сложных строк или значений форматирования для представления пользователю. Другие функции пакета `fmt`, обеспечивающие управление форматированием, показаны в листинге 17-6.

```
package main

import "fmt"

func main() {

      fmt.Printf("Product:  %v,  Price:  $%4.2f",  Kayak.Name,
Kayak.Price)
}
```

Листинг 17-6 Форматирование строки в файле main.go в папке usingstrings

Функция Printf принимает строку шаблона и ряд значений. Шаблон сканируется на наличие *глаголов*, которые обозначаются знаком процента (символом %), за которым следует спецификатор формата. В шаблоне в листинге 17-6 есть два глагола:

```
...
fmt.Printf("Product: %v, Price: $%4.2f", Kayak.Name,
Kayak.Price)
...
```

Первый глагол — %v, и он указывает представление по умолчанию для типа. Например, для string значения %v просто включает строку в вывод. Глагол %4.2f задает формат для значения с плавающей запятой, с 4 цифрами до десятичной точки и 2 цифрами после. Значения для глаголов шаблона берутся из оставшихся аргументов и используются в том порядке, в котором они указаны. Например, это означает, что команда %v используется для форматирования значения Product.Name, а команда %4.2f используется для форматирования значения Product.Price. Эти значения форматируются, вставляются в строку шаблона и выводятся в консоль, в чем вы можете убедиться, скомпилировав и выполнив код:

```
Product: Kayak, Price: $275.00
```

Таблица 17-4 описывает функции, предоставляемые пакетом fmt, который может форматировать строки. Я описываю глаголы форматирования в разделе «Понимание глаголов форматирования».

Таблица 17-4 Функции fmt для форматирования строк

Функция	Описание
Sprintf(t, ...vals)	Эта функция возвращает строку, созданную путем обработки шаблона t. Остальные аргументы используются в качестве значений для глаголов шаблона.
Printf(t, ...vals)	Эта функция создает строку, обрабатывая шаблон t. Остальные аргументы используются в качестве значений для глаголов шаблона. Строка записывается на стандартный вывод.

Функция	Описание
`Fprintf(writer, t, ...vals)`	Эта функция создает строку, обрабатывая шаблон `t`. Остальные аргументы используются в качестве значений для глаголов шаблона. Строка записывается в модуль `Writer`, который описан в главе 20.
`Errorf(t, ...values)`	Эта функция создает ошибку, обрабатывая шаблон `t`. Остальные аргументы используются в качестве значений для глаголов шаблона. Результатом является значение `error`, метод `Error` которого возвращает отформатированную строку.

В листинге 17-7 я определил функцию, которая использует `Sprintf` для форматирования строкового результата и использует `Errorf` для создания ошибки.

```
package main

import "fmt"

func getProductName(index int) (name string, err error) {
    if (len(Products) > index) {
        name = fmt.Sprintf("Name of product: %v", Products[index].Name)
    } else {
        err = fmt.Errorf("Error for index %v", index)
    }
    return
}

func main() {

    name, _ := getProductName(1)
    fmt.Println(name)

    _, err := getProductName(10)
    fmt.Println(err.Error())
}
```

Листинг 17-7 Использование форматированных строк в файле main.go в папке usingstrings

Обе отформатированные строки в этом примере используют значение `%v`, которое записывает значения в форме по умолчанию.

Скомпилируйте и выполните проект, и вы увидите один результат и одну ошибку, как показано ниже:

```
Name of product: Lifejacket
Error for index 10
```

Понимание глаголов форматирования

Функции, описанные в таблице 17-4, поддерживают в своих шаблонах широкий диапазон команд форматирования. В следующих разделах я опишу наиболее полезные. Я начну с тех глаголов, которые можно использовать с любым типом данных, а затем опишу более конкретные.

Использование глаголов форматирования общего назначения

Глаголы общего назначения могут использоваться для отображения любого значения, как описано в таблице 17-5.

Таблица 17-5 Глаголы форматирования для любого значения

Глагол	Описание
`%v`	Эта команда отображает формат значения по умолчанию. Изменение глагола со знаком плюс (`%+v`) включает имена полей при записи значений структуры.
`%#v`	Эта команда отображает значение в формате, который можно использовать для повторного создания значения в файле кода Go.
`%T`	Эта команда отображает тип значения Go.

В листинге 17-8 я определил пользовательский тип структуры и использовал команды, показанные в таблице, для форматирования значения этого типа.

```go
package main

import "fmt"

func Printfln(template string, values ...interface{}) {
    fmt.Printf(template + "\n", values...)
}
```

```
func main() {

    Printfln("Value: %v", Kayak)
    Printfln("Go syntax: %#v", Kayak)
    Printfln("Type: %T", Kayak)
}
```

Листинг 17-8 Использование глаголов общего назначения в файле main.go в папке usingstrings

Функция `Printf` не добавляет символ новой строки к своему выводу, в отличие от функции `Println`, поэтому я определил функцию `Printfln`, которая добавляет новую строку к шаблону перед вызовом функции `Printf`. Операторы в `main` функции определяют простые строковые шаблоны с глаголами в таблице `17-5`. Скомпилируйте и выполните код, и вы получите следующий вывод:

```
Value: {Kayak Watersports 275}
Go syntax: main.Product{Name:"Kayak", Category:"Watersports",
Price:275}
Type: main.Product
```

Управление форматированием структуры

Go имеет формат по умолчанию для всех типов данных, на которые опирается глагол `%v`. Для структур значение по умолчанию перечисляет значения полей в фигурных скобках. Глагол по умолчанию можно изменить с помощью знака плюс, чтобы включить имена полей в вывод, как показано в листинге 17-9.

```
package main

import "fmt"

func Printfln(template string, values ...interface{}) {
    fmt.Printf(template + "\n", values...)
}

func main() {

    Printfln("Value: %v", Kayak)
    Printfln("Value with fields: %+v", Kayak)
```

```
}
```

Листинг 17-9 Отображение имен полей в файле main.go в папке usingstrings

Скомпилируйте и выполните проект, и вы увидите одно и то же значение `Product`, отформатированное с именами полей и без них:

```
Value: {Kayak Watersports 275}
Value with fields: {Name:Kayak Category:Watersports
Price:275}
```

Пакет `fmt` поддерживает пользовательское форматирование структур через интерфейс с именем `Stringer`, который определяется следующим образом:

```
type Stringer interface {
    String() string
}
```

Метод `String`, заданный интерфейсом `Stringer`, будет использоваться для получения строкового представления любого определяющего его типа, как показано в листинге 17-10, что позволяет указать пользовательское форматирование.

```
package main

import "fmt"

type Product struct {
    Name, Category string
    Price float64
}

// ...variables omitted for brevity...

func (p Product) String() string {
    return fmt.Sprintf("Product: %v, Price: $%4.2f", p.Name,
p.Price)
}
```

Листинг 17-10 Определение пользовательского формата структуры в файле product.go в папке usingstrings

Метод String будет вызываться автоматически, когда требуется строковое представление значения Product. Скомпилируйте и выполните код, и вывод будет использовать пользовательский формат:

```
Value: Product: Kayak, Price: $275.00
Value with fields: Product: Kayak, Price: $275.00
```

Обратите внимание, что пользовательский формат также используется, когда команда %v изменяется для отображения полей структуры.

Подсказка
Если вы определяете метод GoString, который возвращает строку, то ваш тип будет соответствовать интерфейсу GoStringer, который допускает пользовательское форматирование для команды %#v.

ФОРМАТИРОВАНИЕ МАССИВОВ, СРЕЗОВ И КАРТ
Когда массивы и срезы представлены в виде строк, вывод представляет собой набор квадратных скобок, внутри которых находятся отдельные элементы, например:

```
...
[Kayak Lifejacket Paddle]
...
```

Обратите внимание, что запятые не разделяют элементы. Когда карты представлены в виде строк, пары ключ-значение отображаются в квадратных скобках, которым предшествует ключевое слово map, например:

```
...
map[1:Kayak 2:Lifejacket 3:Paddle]
...
```

Интерфейс Stringer можно использовать для изменения формата, используемого для пользовательских типов данных, содержащихся в массиве, срезе или карте. Однако никакие изменения форматов по умолчанию не могут быть внесены, если вы

не используете псевдоним типа, потому что методы должны быть определены в том же пакете, что и тип, к которому они применяются.

Использование команд целочисленного форматирования

Таблица 17-6 описывает команды форматирования для целочисленных значений, независимо от их размера.

Таблица 17-6 Глаголы форматирования для целочисленных значений

Глагол	Описание
`%b`	Эта команда отображает целочисленное значение в виде двоичной строки.
`%d`	Эта команда отображает целочисленное значение в виде десятичной строки. Это формат по умолчанию для целочисленных значений, применяемый при использовании глагола `%v`.
`%o`, `%O`	Эти команды отображают целочисленное значение в виде восьмеричной строки. Глагол `%O` добавляет префикс `0o`.
`%x`, `%X`	Эти команды отображают целочисленное значение в виде шестнадцатеричной строки. Буквы от A до F отображаются в нижнем регистре с помощью глагола `%x` и в верхнем регистре с помощью глагола `%X`.

Листинг 17-11 применяет команды, описанные в Таблице 17-6, к целочисленному значению.

```
package main

import "fmt"

func Printfln(template string, values ...interface{}) {
    fmt.Printf(template + "\n", values...)
}

func main() {

    number := 250

    Printfln("Binary: %b", number)
    Printfln("Decimal: %d", number)
    Printfln("Octal: %o, %O", number, number)
    Printfln("Hexadecimal: %x, %X", number, number)
}
```

Листинг 17-11 Форматирование целочисленного значения в файле main.go в папке usingstrings

Скомпилируйте и запустите проект, и вы получите следующий вывод:

```
Binary: 11111010
Decimal: 250
Octal: 372, 0o372
Hexadecimal: fa, FA
```

Использование глаголов форматирования значений с плавающей запятой

В таблице 17-7 описаны глаголы форматирования для значений с плавающей запятой, которые можно применять как к значениям `float32`, так и к значениям `float64`.

Таблица 17-7 Глаголы форматирования для значений с плавающей запятой

Глагол	Описание
`%b`	Эта команда отображает значение с плавающей запятой с показателем степени и без десятичной точки.
`%e`, `%E`	Эти команды отображают значение с плавающей запятой с показателем степени и десятичным разрядом. `%e` использует индикатор степени в нижнем регистре, а `%E` использует индикатор в верхнем регистре.
`%f`, `%F`	Эти команды отображают значение с плавающей запятой с десятичным разрядом, но без экспоненты. Команды `%f` и `%F` производят одинаковый результат.
`%g`	Этот глагол адаптируется к отображаемому значению. Формат `%e` используется для значений с большими показателями степени, в противном случае используется формат `%f`. Это формат по умолчанию, применяемый при использовании глагола `%v`.
`%G`	Этот глагол адаптируется к отображаемому значению. Формат `%E` используется для значений с большими показателями степени, в противном случае используется формат `%f`.
`%x`, `%X`	Эти команды отображают значение с плавающей запятой в шестнадцатеричном представлении со строчными (`%x`) или прописными (`%X`) буквами.

Листинг 17-12 применяет команды, описанные в таблице 17-7, к значению с плавающей запятой.

```
package main
```

```
import "fmt"

func Printfln(template string, values ...interface{}) {
    fmt.Printf(template + "\n", values...)
}

func main() {
    number := 279.00
    Printfln("Decimalless with exponent: %b", number)
    Printfln("Decimal with exponent: %e", number)
    Printfln("Decimal without exponent: %f", number)
    Printfln("Hexadecimal: %x, %X", number, number)
}
```

Листинг 17-12 Форматирование значения с плавающей запятой в файле main.go в папке usingstrings

Скомпилируйте и запустите проект, и вы увидите следующий вывод:

```
Decimalless with exponent: 4908219906392064p-44
Decimal with exponent: 2.790000e+02
Decimal without exponent: 279.000000
Hexadecimal: 0x1.17p+08, 0X1.17P+08
```

Форматом значений с плавающей запятой можно управлять, изменив глагол, указав ширину (количество символов, используемых для выражения значения) и точность (количество цифр после запятой), как показано в листинге 17-13.

```
package main

import "fmt"

func Printfln(template string, values ...interface{}) {
    fmt.Printf(template + "\n", values...)
}

func main() {
    number := 279.00
    Printfln("Decimal without exponent: >>%8.2f<<", number)
}
```

Листинг 17-13 Управление форматированием в файле main.go в папке usingstrings

Ширина указывается после знака процента, за которым следует точка, за которой следует точность, а затем остальная часть глагола. В листинге 17-13 ширина равна 8 символам, а точность — двум символам, что приводит к следующему результату при компиляции и выполнении кода:

```
Decimal without exponent: >>  279.00<<
```

Я добавил шевроны вокруг отформатированного значения в листинге 17-13, чтобы продемонстрировать, что пробелы используются для заполнения, когда указанное количество символов больше, чем количество символов, необходимое для отображения значения.

Ширина может быть опущена, если вас интересует только точность, как показано в листинге 17-14.

```
package main

import "fmt"

func Printfln(template string, values ...interface{}) {
    fmt.Printf(template + "\n", values...)
}

func main() {
    number := 279.00
    Printfln("Decimal without exponent: >>%.2f<<", number)
}
```

Листинг 17-14 Указание точности в файле main.go в папке usingstrings

Значение ширины опущено, но точка по-прежнему необходима. Формат, указанный в листинге 17-7, дает следующий результат при компиляции и выполнении:

```
Decimal without exponent: >>279.00<<
```

Вывод команд в таблице 17-7 можно изменить с помощью модификаторов, описанных в таблице 17-8.

Таблица 17-8 Модификаторы глаголов форматирования

Модификатор	Описание
`+`	Этот модификатор (знак плюс) всегда печатает знак, положительный или отрицательный, для числовых значений.
`0`	Этот модификатор использует нули, а не пробелы, в качестве заполнения, когда ширина превышает количество символов, необходимое для отображения значения.
`-`	Этот модификатор (символ вычитания) добавляет отступ справа от числа, а не слева.

В листинге 17-15 модификаторы применяются для изменения форматирования целочисленного значения.

```
package main

import "fmt"

func Printfln(template string, values ...interface{}) {
    fmt.Printf(template + "\n", values...)
}

func main() {
    number := 279.00
    Printfln("Sign: >>%+.2f<<", number)
    Printfln("Zeros for Padding: >>%010.2f<<", number)
    Printfln("Right Padding: >>%-8.2f<<", number)
}
```

Листинг 17-15 Изменение форматов в файле main.go в папке usingstrings

Скомпилируйте и запустите проект, и вы увидите влияние модификаторов на отформатированный вывод:

```
Sign: >>+279.00<<
Zeros for Padding: >>0000279.00<<
Right Padding: >>279.00  <<
```

Использование глаголов форматирования строк и символов

Таблица 17-9 описывает глаголы форматирования для строк и рун.

Таблица 17-9 Глаголы форматирования для строк и рун

Глагол	Описание
%s	Этот глагол отображает строку. Это формат по умолчанию, применяемый при использовании глагола %v.
%c	Этот глагол отображает характер. Необходимо соблюдать осторожность, чтобы избежать разделения строк на отдельные байты, как это объясняется в тексте после таблицы.
%U	Эта команда отображает символ в формате Unicode, так что вывод начинается с U+, за которым следует шестнадцатеричный код символа.

Строки легко форматировать, но следует соблюдать осторожность при форматировании отдельных символов. Как я объяснял в главе 7, некоторые символы представляются с использованием нескольких байтов, и вы должны быть уверены, что не пытаетесь форматировать только некоторые байты символа. В листинге 17-16 показано использование глаголов, описанных в таблице 17-9.

```
package main

import "fmt"

func Printfln(template string, values ...interface{}) {
    fmt.Printf(template + "\n", values...)
}

func main() {
    name := "Kayak"
    Printfln("String: %s", name)
    Printfln("Character: %c", []rune(name)[0])
    Printfln("Unicode: %U", []rune(name)[0])
}
```

Листинг 17-16 Форматирование строк и символов в файле main.go в папке usingstrings

Скомпилируйте и запустите проект, и вы увидите следующий отформатированный вывод:

```
String: Kayak
Character: K
Unicode: U+004B
```

Использование глагола форматирования логических значений

Таблица 17-10 описывает команду, которая используется для форматирования логических значений. Это формат bool по умолчанию, что означает, что он будет использоваться глаголом %v.

Таблица 17-10 Глагол форматирования bool

Глагол	Описание
%t	Эта команда форматирует логические значения и отображает значение true или false.

В листинге 17-17 показано использование глагола форматирования bool.

```
package main

import "fmt"

func Printfln(template string, values ...interface{}) {
    fmt.Printf(template + "\n", values...)
}

func main() {
    name := "Kayak"
    Printfln("Bool: %t", len(name) > 1)
    Printfln("Bool: %t", len(name) > 100)
}
```

Листинг 17-17 Форматирование логических значений в файле main.go в папке usingstrings

Скомпилируйте и запустите проект, и вы увидите отформатированный вывод:

```
Bool: true
Bool: false
```

Использование глагола форматирования указателя

Глагол, описанный в таблице 17-11, применяется к указателям.

Таблица 17-11 Глагол форматирования указателя

Глагол	Описание
%p	Эта команда отображает шестнадцатеричное представление места хранения указателя.

В листинге 17-18 показано использование глагола-указателя.

```
package main

import "fmt"

func Printfln(template string, values ...interface{}) {
    fmt.Printf(template + "\n", values...)
}

func main() {
    name := "Kayak"
    Printfln("Pointer: %p", &name)
}
```

Листинг 17-18 Форматирование указателя в файле main.go в папке usingstrings

Скомпилируйте и выполните код, и вы увидите вывод, аналогичный следующему, хотя вы можете увидеть другое расположение:

```
Pointer: 0xc00004a240
```

Сканирование строк

Пакет fmt предоставляет функции для сканирования строк, то есть процесса анализа строк, содержащих значения, разделенные пробелами. Таблица 17-12 описывает эти функции, некоторые из которых используются вместе с функциями, описанными в последующих главах.

Таблица 17-12 Функции fmt для сканирования строк

Функция	Описание

Функция	Описание
Scan(...vals)	Эта функция считывает текст из стандарта и сохраняет значения, разделенные пробелами, в указанные аргументы. Новые строки обрабатываются как пробелы, и функция читает до тех пор, пока не получит значения для всех своих аргументов. Результатом является количество прочитанных значений и error, описывающая любые проблемы.
Scanln(...vals)	Эта функция работает так же, как Scan, но останавливает чтение, когда встречает символ новой строки.
Scanf(template, ...vals)	Эта функция работает так же, как Scan, но использует строку шаблона для выбора значений из получаемых входных данных.
Fscan(reader, ...vals)	Эта функция считывает значения, разделенные пробелами, из указанного средства чтения, описанного в главе 20. Новые строки обрабатываются как пробелы, и функция возвращает количество прочитанных значений и ошибку, описывающую любые проблемы.
Fscanln(reader, ...vals)	Эта функция работает так же, как Fscan, но останавливает чтение, когда встречает символ новой строки.
Fscanf(reader, template, ...vals)	Эта функция работает так же, как Fscan, но использует шаблон для выбора значений из получаемых входных данных.
Sscan(str, ...vals)	Эта функция просматривает указанную строку в поисках значений, разделенных пробелами, которые присваиваются остальным аргументам. Результатом является количество просканированных значений и ошибка, описывающая любые проблемы.
Sscanf(str, template, ...vals)	Эта функция работает так же, как Sscan, но использует шаблон для выбора значений из строки.
Sscanln(str, template, ...vals)	Эта функция работает так же, как Sscanf, но останавливает сканирование строки, как только встречается символ новой строки.

Решение о том, какую функцию сканирования использовать, зависит от источника строки для сканирования, способа обработки новых строк и необходимости использования шаблона. В листинге 17-19 показано основное использование функции Scan, с которого можно начать.

```
package main

import "fmt"

func Printfln(template string, values ...interface{}) {
```

```
    fmt.Printf(template + "\n", values...)
}

func main() {

    var name string
    var category string
    var price float64

    fmt.Print("Enter text to scan: ")
    n, err := fmt.Scan(&name, &category, &price)

    if (err == nil) {
        Printfln("Scanned %v values", n)
        Printfln("Name: %v, Category: %v, Price: %.2f", name,
category, price)
    } else {
        Printfln("Error: %v", err.Error())
    }
}
```

Листинг 17-19 Сканирование строки в файле main.go в папке usingstrings

Функция Scan считывает строку из стандартного ввода и сканирует ее на наличие значений, разделенных пробелами. Значения, извлеченные из строки, присваиваются параметрам в том порядке, в котором они определены. Чтобы функция Scan могла присваивать значения, ее параметры являются указателями.

В листинге 17-19 я определяю переменные name, category и price и использую их в качестве аргументов функции Scan:

```
...
n, err := fmt.Scan(&name, &category, &price)
...
```

При вызове функция Scan считывает строку, извлекает три значения, разделенных пробелами, и присваивает их переменным. Скомпилируйте и запустите проект, и вам будет предложено ввести текст, например:

```
...
Enter text to scan:
```

```
...
```

Введите `Kayak Watersports 279`, что означает слово `Kayak`, за которым следует пробел, за которым следует слово `Watersports`, за которым следует пробел, за которым следует число `279`. Нажмите **Enter**, и строка будет отсканирована, и будет получен следующий результат:

```
Scanned 3 values
Name: Kayak, Category: Watersports, Price: 279.00
```

Функция `Scan` должна преобразовать полученные подстроки в значения Go и сообщит об ошибке, если строка не может быть обработана. Запустите код еще раз, но введите `Kayak Watersports Zero`, и вы получите следующую ошибку:

```
Error: strconv.ParseFloat: parsing "": invalid syntax
```

Строка `Zero` не может быть преобразована в значение Go `float64`, которое является типом параметра `Price`.

СКАНИРОВАНИЕ В СРЕЗ

Если вам нужно просмотреть ряд значений одного типа, естественным подходом будет просмотр среза или массива, например:

```
...
vals := make([]string, 3)
fmt.Print("Enter text to scan: ")
fmt.Scan(vals...)
Printfln("Name: %v", vals)
...
```

Этот код не будет скомпилирован, потому что срез строки не может быть правильно разложен для использования с вариативным параметром. Требуется дополнительный шаг, а именно:

```
...
vals := make([]string, 3)
ivals := make([]interface{}, 3)
```

```
for i := 0; i < len(vals); i++ {
    ivals[i] = &vals[i]
}
fmt.Print("Enter text to scan: ")
fmt.Scan(ivals...)
Printfln("Name: %v", vals)
...
```

Это неудобный процесс, но его можно обернуть вспомогательной функцией, чтобы вам не приходилось каждый раз создавать срез `interface`.

Работа с символами новой строки

По умолчанию сканирование обрабатывает новые строки так же, как пробелы, выступающие в качестве разделителей между значениями. Чтобы увидеть это поведение, запустите проект и, когда появится запрос на ввод, введите `Kayak`, затем пробел, затем `Watersports`, затем клавишу Enter, `279`, а затем снова клавишу Enter. Эта последовательность выдаст следующий результат:

```
Scanned 3 values
Name: Kayak, Category: Watersports, Price: 279.00
```

Функция `Scan` не прекращает поиск значений до тех пор, пока не получит ожидаемое число, а первое нажатие клавиши Enter рассматривается как разделитель, а не как завершение ввода. Функции, имена которых заканчиваются на `ln` в таблице 17-12, такие как `Scanln`, изменяют это поведение. В листинге 17-20 используется функция `Scanln`.

```
package main

import "fmt"

func Printfln(template string, values ...interface{}) {
    fmt.Printf(template + "\n", values...)
}

func main() {
```

```
    var name string
    var category string
    var price float64

    fmt.Print("Enter text to scan: ")
    n, err := fmt.Scanln(&name, &category, &price)

    if (err == nil) {
        Printfln("Scanned %v values", n)
        Printfln("Name: %v, Category: %v, Price: %.2f", name,
category, price)
    } else {
        Printfln("Error: %v", err.Error())
    }
}
```

Листинг 17-20 Использование функции Scanln в файле main.go в папке usingstrings

Скомпилируйте и выполните проект и повторите последовательность ввода. Когда вы впервые нажимаете клавишу Enter, новая строка завершает ввод, оставляя функцию `Scanln` с меньшим количеством значений, чем требуется, и производит следующий вывод:

```
Error: unexpected newline
```

Использование другого источника строк

Функции, описанные в таблице 17-12, сканируют строки из трех источников: стандартного ввода, средства чтения (описанного в главе 20) и значения, переданного в качестве аргумента. Предоставление строки в качестве аргумента является наиболее гибким, поскольку это означает, что строка может возникнуть откуда угодно. В листинге 17-21 я заменил функцию `Scanln` на `Sscan`, которая позволяет мне сканировать строковую переменную.

```
package main

import "fmt"

func Printfln(template string, values ...interface{}) {
    fmt.Printf(template + "\n", values...)
```

```
}

func main() {

    var name string
    var category string
    var price float64

    source := "Lifejacket Watersports 48.95"
    n, err := fmt.Sscan(source, &name, &category, &price)

    if (err == nil) {
        Printfln("Scanned %v values", n)
        Printfln("Name: %v, Category: %v, Price: %.2f", name,
category, price)
    } else {
        Printfln("Error: %v", err.Error())
    }
}
```

Листинг 17-21 Сканирование переменной в файле main.go в папке usingstrings

Первым аргументом функции `Sscan` является сканируемая строка, но во всем остальном процесс сканирования такой же. Скомпилируйте и запустите проект, и вы увидите следующий вывод:

```
Scanned 3 values
Name: Lifejacket, Category: Watersports, Price: 48.95
```

Использование шаблона сканирования

Шаблон можно использовать для поиска значений в строке, содержащей ненужные символы, как показано в листинге 17-22.

```
package main

import "fmt"

func Printfln(template string, values ...interface{}) {
    fmt.Printf(template + "\n", values...)
}

func main() {
```

```
    var name string
    var category string
    var price float64

    source := "Product Lifejacket Watersports 48.95"
    template := "Product %s %s %f"
     n, err := fmt.Sscanf(source, template, &name, &category,
&price)

    if (err == nil) {
        Printfln("Scanned %v values", n)
        Printfln("Name: %v, Category: %v, Price: %.2f", name,
category, price)
    } else {
        Printfln("Error: %v", err.Error())
    }
}
```

Листинг 17-22 Использование шаблона в файле main.go в папке usingstrings

Шаблон, используемый в листинге 17-22, игнорирует термин `Product`, пропуская эту часть строки и позволяя начать сканирование со следующего термина. Скомпилируйте и запустите проект, и вы увидите следующий вывод:

```
Scanned 3 values
Name: Lifejacket, Category: Watersports, Price: 48.95
```

Сканирование с помощью шаблона не так гибко, как использование регулярного выражения, потому что отсканированная строка может содержать только значения, разделенные пробелами. Но использование шаблона все же может быть полезным, если вам нужны только некоторые значения в строке и вы не хотите определять сложные правила сопоставления.

Резюме

В этой главе я описал возможности стандартной библиотеки для форматирования и сканирования строк, обе из которых предоставляются пакетом `fmt`. В следующей главе я опишу

возможности, предоставляемые стандартной библиотекой для математических функций и сортировки срезов.

18. Математические функции и сортировка данных

В этой главе я описываю два набора функций. Во-первых, я описываю поддержку выполнения общих математических задач, включая генерацию случайных чисел. Во-вторых, я описываю возможности сортировки элементов в срезе по порядку. В таблице 18-1 математические функции и функции сортировки представлены в контексте.

Таблица 18-1 Помещение математических функций и сортировки данных в контекст

Вопрос	Ответ
Кто они такие?	Математические функции позволяют выполнять обычные вычисления. Случайные числа — это числа, сгенерированные в последовательности, которую трудно предсказать. Сортировка — это процесс размещения последовательности значений в заданном порядке.
Почему они полезны?	Это функции, которые используются на протяжении всей разработки.
Как они используются?	Эти функции предоставляются в пакетах `math`, `math/rand` и `sort`.
Есть ли подводные камни или ограничения?	Если не инициализировано начальным значением, числа, созданные пакетом `math/rand`, не являются случайными.
Есть ли альтернативы?	Вы можете реализовать оба набора функций с нуля, хотя эти пакеты предоставляются, так что это не требуется.

Таблица 18-2 суммирует главу.

Таблица 18-2 Краткое содержание главы

Проблема	Решение	Листинг
Выполнить общие расчеты	Используйте функции, определенные в пакете `math`	5
Генерация случайных чисел	Используйте функции пакета `math/rand`, позаботившись о том, чтобы предоставить начальное значение.	6–9

Проблема	Решение	Листинг
Перемешать элементы в срезе	Используйте функцию Shuffle	10
Сортировка элементов в срезе	Используйте функции, определенные в пакете sort	11, 12, 15–20
Найти элемент в отсортированном срезе	Используйте функции Search*	13, 14

Подготовка к этой главе

Чтобы подготовиться к этой главе, откройте новую командную строку, перейдите в удобное место и создайте каталог с именем `mathandsorting`. Запустите команду, показанную в листинге 18-1, в папке `mathandsorting`, чтобы создать файл модуля.

> **Подсказка**
> Вы можете загрузить пример проекта для этой главы — и для всех остальных глав этой книги — с https://github.com/apress/pro-go. См. Главу 2 о том, как получить помощь, если у вас возникнут проблемы с запуском примеров.

```
go mod init mathandsorting
```

Листинг 18-1 Инициализация модуля

Добавьте файл с именем `printer.go` в папку `mathandsorting` с содержимым, показанным в листинге 18-2.

```
package main

import "fmt"

func Printfln(template string, values ...interface{}) {
    fmt.Printf(template + "\n", values...)
}
```

Листинг 18-2 Содержимое файла printer.go в папке mathandsorting

Добавьте файл с именем main.go в папку mathandsorting с содержимым, показанным в листинге 18-3.

```
package main

func main() {

    Printfln("Hello, Math and Sorting")
}
```

Листинг 18-3 Содержимое файла main.go в папке mathandsorting

Используйте командную строку для запуска команды, показанной в листинге 18-4, в папке mathandsorting.

```
go run .
```

Листинг 18-4 Запуск примера проекта

Код будет скомпилирован и выполнен, что приведет к следующему выводу:

```
Hello, Math and Sorting
```

Работа с числами

Как я объяснял в главе 4, язык Go поддерживает набор арифметических операторов, которые можно применять к числовым значениям, позволяя выполнять такие базовые задачи, как сложение и умножение. Для более продвинутых операций стандартная библиотека Go включает пакет math, предоставляющий обширный набор функций. Функции, которые наиболее широко используются в типичном проекте, описаны в таблице 18-3. См. документацию пакета по адресу https://golang.org/pkg/math, чтобы узнать о полном наборе функций, включая поддержку более конкретных областей, таких как тригонометрия.

Таблица 18-3 Полезные функции из математического пакета

Функция	Описание

Функция	Описание
Abs(val)	Эта функция возвращает абсолютное значение значения float64, то есть расстояние от нуля без учета направления.
Ceil(val)	Эта функция возвращает наименьшее целое число, равное или превышающее указанное значение float64. Результатом также является значение float64, хотя оно представляет собой целое число.
Copysign(x, y)	Эта функция возвращает значение float64, которое является абсолютным значением x со знаком y.
Floor(val)	Эта функция возвращает наибольшее целое число, которое меньше или равно указанному значению float64. Результатом также является значение float64, хотя оно представляет собой целое число.
Max(x, y)	Эта функция возвращает самое большое из указанных значений float64.
Min(x, y)	Эта функция возвращает наименьшее из указанных значений float64.
Mod(x, y)	Эта функция возвращает остаток x/y.
Pow(x, y)	Эта функция возвращает значение x, возведенное в степень y.
Round(val)	Эта функция округляет указанное значение до ближайшего целого числа, округляя половинные значения в большую сторону. Результатом является значение float64, хотя оно представляет собой целое число.
RoundToEven(val)	Эта функция округляет указанное значение до ближайшего целого числа, округляя половинные значения до ближайшего четного числа. Результатом является значение float64, хотя оно представляет собой целое число.

Все эти функции работают со значениями float64 и выдают результаты float64, что означает, что вы должны явно преобразовывать в другие типы и из них. В листинге 18-5 показано использование функций, описанных в таблице 18-3.

```
package main

import "math"

func main() {

    val1 := 279.00
    val2 := 48.95

    Printfln("Abs: %v", math.Abs(val1))
    Printfln("Ceil: %v", math.Ceil(val2))
    Printfln("Copysign: %v", math.Copysign(val1, -5))
```

```
    Println("Floor: %v", math.Floor(val2))
    Println("Max: %v", math.Max(val1, val2))
    Println("Min: %v", math.Min(val1, val2))
    Println("Mod: %v", math.Mod(val1, val2))
    Println("Pow: %v", math.Pow(val1, 2))
    Println("Round: %v", math.Round(val2))
    Println("RoundToEven: %v", math.RoundToEven(val2))
}
```

Листинг 18-5 Using Functions from the math Package in the main.go File in the mathandsorting Folder

Скомпилируйте и запустите проект, и вы увидите следующий вывод:

```
Abs: 279
Ceil: 49
Copysign: -279
Floor: 48
Max: 279
Min: 48.95
Mod: 34.249999999999986
Pow: 77841
Round: 49
RoundToEven: 49
```

Пакет `math` также предоставляет набор констант для ограничений числовых типов данных, как описано в таблице 18-4.

Таблица 18-4 Предельные константы

Имя	Описание
`MaxInt8` `MinInt8`	Эти константы представляют наибольшее и наименьшее значения, которые могут быть сохранены с использованием `int8`.
`MaxInt16` `MinInt16`	Эти константы представляют наибольшее и наименьшее значения, которые могут быть сохранены с использованием типа `int16`.
`MaxInt32` `MinInt32`	Эти константы представляют наибольшее и наименьшее значения, которые могут быть сохранены с помощью `int32`.
`MaxInt64` `MinInt64`	Эти константы представляют наибольшее и наименьшее значения, которые могут быть сохранены с помощью `int64`.

Имя	Описание
MaxUint8	Эта константа представляет наибольшее значение, которое может быть представлено с помощью uint8. Наименьшее значение равно нулю.
MaxUint16	Эта константа представляет наибольшее значение, которое может быть представлено с помощью uint16. Наименьшее значение равно нулю.
MaxUint32	Эта константа представляет наибольшее значение, которое может быть представлено с помощью uint32. Наименьшее значение равно нулю.
MaxUint64	Эта константа представляет наибольшее значение, которое может быть представлено с помощью uint64. Наименьшее значение равно нулю.
MaxFloat32 MaxFloat64	Эти константы представляют самые большие значения, которые могут быть представлены с использованием значений float32 и float64.
SmallestNonzeroFloat32 SmallestNonzeroFloat64	Эти константы представляют наименьшие ненулевые значения, которые могут быть представлены с использованием значений float32 и float64.

Генерация случайных чисел

Пакет math/rand обеспечивает поддержку генерации случайных чисел. Наиболее полезные функции описаны в таблице 18-5. (Хотя в этом разделе я использую термин *случайный*, числа, создаваемые пакетом math/rand, являются псевдослучайными, что означает, что их не следует использовать там, где важна случайность, например, для генерации криптографических ключей.)

Таблица 18-5 Полезные функции math/rand

Функция	Описание
Seed(s)	Эта функция устанавливает начальное значение, используя указанное значение int64.
Float32()	Эта функция генерирует случайное значение float32 в диапазоне от 0 до 1.
Float64()	Эта функция генерирует случайное значение float64 в диапазоне от 0 до 1.
Int()	Эта функция генерирует случайное int значение.
Intn(max)	Эта функция генерирует случайное int число меньше указанного значения, как описано после таблицы.

Функция	Описание
UInt32()	Эта функция генерирует случайное значение uint32.
UInt64()	Эта функция генерирует случайное значение uint64.
Shuffle(count, func)	Эта функция используется для рандомизации порядка элементов, как описано после таблицы.

Необычность пакета math/rand заключается в том, что он по умолчанию возвращает последовательность предсказуемых значений, как показано в листинге 18-6.

```
package main

import "math/rand"

func main() {

    for i := 0; i < 5; i++ {
        Printfln("Value %v : %v", i, rand.Int())
    }
}
```

Листинг 18-6 Генерация предсказуемых значений в файле main.go в папке mathandsorting

В этом примере вызывается функция Int и выводится значение. Скомпилируйте и выполните код, и вы увидите следующий вывод:

```
Value 0 : 5577006791947779410
Value 1 : 8674665223082153551
Value 2 : 6129484611666145821
Value 3 : 4037200794235010051
Value 4 : 3916589616287113937
```

Код в листинге 18-6 всегда будет выдавать один и тот же набор чисел, потому что начальное начальное значение всегда одно и то же. Чтобы избежать создания одной и той же последовательности чисел, функцию Seed необходимо вызывать с нефиксированным значением, как показано в листинге 18-7.

```
package main
```

```
import (
    "math/rand"
    "time"
)

func main() {
    rand.Seed(time.Now().UnixNano())
    for i := 0; i < 5; i++ {
        Printfln("Value %v : %v", i, rand.Int())
    }
}
```

Листинг 18-7 Установка начального значения в файле main.go в папке mathandsorting

Соглашение состоит в том, чтобы использовать текущее время в качестве начального значения, что делается путем вызова функции `Now`, предоставляемой пакетом `time`, и вызова метода `UnixNano` для результата, который предоставляет значение `int64`, которое можно передать в функцию начального значения. (Я описываю пакет времени в главе 19.) Скомпилируйте и запустите проект, и вы увидите ряд чисел, которые меняются при каждом выполнении программы. Вот результат, который я получил:

```
Value 0 : 8113726196145714527
Value 1 : 3479565125812279859
Value 2 : 8074476402089812953
Value 3 : 3916870404047362448
Value 4 : 8226545715271170755
```

Генерация случайного числа в определенном диапазоне

Функцию `Intn` можно использовать для генерации числа с заданным максимальным значением, как показано в листинге 18-8.

```
package main

import (
    "math/rand"
    "time"
)

func main() {
```

```
    rand.Seed(time.Now().UnixNano())
    for i := 0; i < 5; i++ {
        Printfln("Value %v : %v", i, rand.Intn(10))
    }
}
```

Листинг 18-8 Указание максимального значения в файле main.go в папке mathandsorting

В операторе указано, что все случайные числа должны быть меньше 10. Скомпилируйте и выполните код, и вы увидите вывод, аналогичный следующему, но с другими случайными значениями:

```
Value 0 : 7
Value 1 : 5
Value 2 : 4
Value 3 : 0
Value 4 : 7
```

Не существует функции для указания минимального значения, но можно легко сдвинуть значения, сгенерированные функцией `Intn`, в определенный диапазон, как показано в листинге 18-9.

```
package main

import (
    "math/rand"
    "time"
)

func IntRange(min, max int) int {
    return rand.Intn(max - min) + min
}

func main() {

    rand.Seed(time.Now().UnixNano())
    for i := 0; i < 5; i++ {
        Printfln("Value %v : %v", i, IntRange(10, 20))
    }
}
```

Листинг 18-9 Указание нижней границы в файле main.go в папке mathandsorting

Функция IntRange возвращает случайное число в определенном диапазоне. Скомпилируйте и выполните проект, и вы получите последовательность чисел от 10 до 19, похожую на следующую:

```
Value 0 : 10
Value 1 : 19
Value 2 : 11
Value 3 : 10
Value 4 : 17
```

Перетасовка элементов

Функция Shuffle используется для случайного переупорядочивания элементов, что она делает с помощью пользовательской функции, как показано в листинге 18-10.

```
package main

import (
    "math/rand"
    "time"
)

var names = []string { "Alice", "Bob", "Charlie", "Dora",
"Edith"}

func main() {
    rand.Seed(time.Now().UnixNano())

    rand.Shuffle(len(names), func (first, second int) {
                names[first], names[second] = names[second],
names[first]
    })

    for i, name := range names {
        Println("Index %v: Name: %v", i, name)
    }
}
```

Листинг 18-10 Перетасовка элементов в файле main.go в папке mathandsorting

Аргументами функции Shuffle являются количество элементов и функция, которая меняет местами два элемента, идентифицируемых по

индексу. Функция вызывается для случайной замены элементов. В листинге 18-10 анонимная функция переключает два элемента в срезе names, а это означает, что использование функции Shuffle приводит к перетасовке порядка значений names. Скомпилируйте и выполните проект, и выходные данные будут отображать перетасованный порядок элементов в срезе names, подобно следующему:

```
Index 0: Name: Edith
Index 1: Name: Dora
Index 2: Name: Charlie
Index 3: Name: Alice
Index 4: Name: Bob
```

Сортировка данных

В предыдущем примере было показано, как перетасовать элементы в срезе, но более распространенным требованием является расположение элементов в более предсказуемой последовательности, за которую отвечают функции, предоставляемые пакетом sort. В следующих разделах я опишу встроенные функции сортировки, предоставляемые пакетом, и продемонстрирую их использование.

Сортировка числовых и строковых срезов

Функции, описанные в таблице 18-6, используются для сортировки срезов, содержащих значения int, float64 или string.

Таблица 18-6 Основные функции сортировки

Функция	**Описание**
Float64s(slice)	Эта функция сортирует срез значений float64. Элементы сортируются на месте.
Float64sAreSorted(slice)	Эта функция возвращает значение true, если элементы в указанном срезе float64 упорядочены.
Ints(slice)	Эта функция сортирует срез значений int. Элементы сортируются на месте.
IntsAreSorted(slice)	Эта функция возвращает значение true, если элементы в указанном int срезе упорядочены.
Strings(slice)	Эта функция сортирует срез string значений. Элементы сортируются на месте.

Функция	Описание
StringsAreSorted(slice)	Эта функция возвращает значение true, если элементы в указанном срезе string упорядочены.

Каждый из типов данных имеет собственный набор функций, которые сортируют данные или определяют, были ли они уже отсортированы, как показано в листинге 18-11.

```
package main

import (
    //"math/rand"
    //"time"
    "sort"
)

func main() {

    ints := []int { 9, 4, 2, -1, 10}
    Printfln("Ints: %v", ints)
    sort.Ints(ints)
    Printfln("Ints Sorted: %v", ints)

    floats := []float64 { 279, 48.95, 19.50 }
    Printfln("Floats: %v", floats)
    sort.Float64s(floats)
    Printfln("Floats Sorted: %v", floats)

    strings := []string { "Kayak", "Lifejacket", "Stadium" }
    Printfln("Strings: %v", strings)
    if (!sort.StringsAreSorted(strings)) {
        sort.Strings(strings)
        Printfln("Strings Sorted: %v", strings)
    } else {
        Printfln("Strings Already Sorted: %v", strings)
    }
}
```

Листинг 18-11 Сортировка срезов в файле main.go в папке mathandsorting

В этом примере выполняется сортировка срезов, содержащих значения int и float64. Существует также string срез, который

тестируется с помощью функции StringsAreSorted, чтобы избежать сортировки данных, которые уже упорядочены. Скомпилируйте и запустите проект, и вы получите следующий вывод:

```
Ints: [9 4 2 -1 10]
Ints Sorted: [-1 2 4 9 10]
Floats: [279 48.95 19.5]
Floats Sorted: [19.5 48.95 279]
Strings: [Kayak Lifejacket Stadium]
Strings Already Sorted: [Kayak Lifejacket Stadium]
```

Обратите внимание, что функции в листинге 18-11 сортируют элементы на месте, а не создают новый срез. Если вы хотите создать новый отсортированный срез, вы должны использовать встроенные функции make и copy, как показано в листинге 18-12. Эти функции были представлены в главе 7.

```
package main

import (
    "sort"
)

func main() {

    ints := []int { 9, 4, 2, -1, 10}

    sortedInts := make([]int, len(ints))
    copy(sortedInts, ints)
    sort.Ints(sortedInts)
    Printfln("Ints: %v", ints)
    Printfln("Ints Sorted: %v", sortedInts)
}
```

Листинг 18-12 Создание отсортированной копии среза в файле main.go в папке mathandsorting

Скомпилируйте и запустите проект, и вы получите следующий вывод:

```
Ints: [9 4 2 -1 10]
Ints Sorted: [-1 2 4 9 10]
```

Поиск отсортированных данных

Пакет sort определяет функции, описанные в таблице 18-7, для поиска определенного значения в отсортированных данных.

Таблица 18-7 Функции для поиска отсортированных данных

Функция	Описание
SearchInts(slice, val)	Эта функция ищет в отсортированном срезе указанное значение int. Результатом является индекс указанного значения или, если значение не найдено, индекс, по которому значение может быть вставлено при сохранении порядка сортировки.
SearchFloat64s(slice, val)	Эта функция ищет в отсортированном срезе указанное значение float64. Результатом является индекс указанного значения или, если значение не найдено, индекс, по которому значение может быть вставлено при сохранении порядка сортировки.
SearchStrings(slice, val)	Эта функция ищет в отсортированном срезе указанное string значение. Результатом является индекс указанного значения или, если значение не найдено, индекс, по которому значение может быть вставлено при сохранении порядка сортировки.
Search(count, testFunc)	Эта функция вызывает тестовую функцию для указанного количества элементов. Результатом является индекс, для которого функция возвращает значение true. Если совпадений нет, результатом является индекс, в который можно вставить указанное значение для сохранения порядка сортировки.

Функции, описанные в таблице 18-7, немного неудобны. Когда значение найдено, функции возвращают его положение в срезе. Но необычно, если значение не найдено, результатом является позиция, в которую оно может быть вставлено при сохранении порядка сортировки, как показано в листинге 18-13.

```
package main

import (
    "sort"
)

func main() {

    ints := []int { 9, 4, 2, -1, 10}

    sortedInts := make([]int, len(ints))
```

```
    copy(sortedInts, ints)
    sort.Ints(sortedInts)
    Printfln("Ints: %v", ints)
    Printfln("Ints Sorted: %v", sortedInts)

    indexOf4:= sort.SearchInts(sortedInts, 4)
    indexOf3 := sort.SearchInts(sortedInts, 3)
    Printfln("Index of 4: %v", indexOf4)
    Printfln("Index of 3: %v", indexOf3)
}
```

Листинг 18-13 Поиск отсортированных данных в файле main.go в папке mathandsorting

Скомпилируйте и выполните код, и вы увидите, что поиск значения, находящегося в срезе, дает тот же результат, что и поиск несуществующего значения:

```
Ints: [9 4 2 -1 10]
Ints Sorted: [-1 2 4 9 10]
Index of 4: 2
Index of 3: 2
```

Этим функциям требуется дополнительный тест, чтобы увидеть, является ли значение в месте, возвращаемом этими функциями, тем, которое искали, как показано в листинге 18-14.

```
package main

import (
    "sort"
)

func main() {

    ints := []int { 9, 4, 2, -1, 10}

    sortedInts := make([]int, len(ints))
    copy(sortedInts, ints)
    sort.Ints(sortedInts)
    Printfln("Ints: %v", ints)
    Printfln("Ints Sorted: %v", sortedInts)
```

```
    indexOf4:= sort.SearchInts(sortedInts, 4)
    indexOf3 := sort.SearchInts(sortedInts, 3)
       Printfln("Index  of  4:  %v  (present:  %v)",  indexOf4,
sortedInts[indexOf4] == 4)
       Printfln("Index  of  3:  %v  (present:  %v)",  indexOf3,
sortedInts[indexOf3] == 3)
}
```

Листинг 18-14 Устранение неоднозначности результатов поиска в файле main.go в папке mathandsorting

Скомпилируйте и выполните проект, и вы получите следующие результаты:

```
Ints: [9 4 2 -1 10]
Ints Sorted: [-1 2 4 9 10]
Index of 4: 2 (present: true)
Index of 3: 2 (present: false)
```

Сортировка пользовательских типов данных

Для сортировки пользовательских типов данных в пакете `sort` определен интерфейс со странным названием `Interface`, в котором указаны методы, описанные в таблице 18-8.

Таблица 18-8 Методы, определяемые интерфейсом sort.Interface

Функция	Описание
`Len()`	Этот метод возвращает количество элементов, которые будут отсортированы.
`Less(i, j)`	Этот метод возвращает значение `true`, если элемент с индексом `i` должен появиться в отсортированной последовательности перед элементом `j`. Если `Less(i,j)` и `Less(j, i)` оба `false`, то элементы считаются равными.
`Swap(i, j)`	Этот метод меняет местами элементы по указанным индексам.

Когда тип определяет методы, описанные в таблице `18-8`, его можно сортировать с помощью функций, описанных в таблице `18-9`, которые определяются пакетом `sort`.

Таблица 18-9 Функции для сортировки типов, реализующих интерфейс

Функция	Описание

Функция	Описание
Sort(data)	Эта функция использует методы, описанные в таблице 18-8, для сортировки указанных данных.
Stable(data)	Эта функция использует методы, описанные в таблице 18-8, для сортировки указанных данных без изменения порядка элементов с одинаковым значением.
IsSorted(data)	Эта функция возвращает значение true, если данные отсортированы.
Reverse(data)	Эта функция меняет порядок данных.

Методы, определенные в таблице 18-8, применяются к набору элементов данных, подлежащих сортировке, что означает введение псевдонима типа и функций, которые выполняют преобразования для вызова функций, определенных в таблице 18-9. Для демонстрации добавьте файл с именем productsort.go в папку mathandsorting с кодом, показанным в листинге 18-15.

```
package main

import "sort"

type Product struct {
    Name string
    Price float64
}

type ProductSlice []Product

func ProductSlices(p []Product) {
    sort.Sort(ProductSlice(p))
}

func ProductSlicesAreSorted(p []Product) {
    sort.IsSorted(ProductSlice(p))
}

func (products ProductSlice) Len() int {
    return len(products)
}

func (products ProductSlice) Less(i, j int) bool {
    return products[i].Price < products[j].Price
```

```
}

func (products ProductSlice) Swap(i, j int) {
    products[i], products[j] = products[j], products[i]
}
```

Листинг 18-15 Содержимое файла productsort.go в папке mathandsorting

Тип `ProductSlice` является псевдонимом для среза `Product` и является типом, для которого были реализованы методы интерфейса. В дополнение к методам у меня есть функция `ProductSlices`, которая принимает срез `Product`, преобразует его в тип `ProductSlice` и передает в качестве аргумента функции `Sort`. Существует также функция `ProductSlicesAreSorted`, которая вызывает функцию `IsSorted`. Имена этой функции следуют соглашению, установленному пакетом `sort`: после имени псевдонима следует буква `s`. В листинге 18.16 эти функции используются для сортировки среза значений `Product`.

```
package main

import (
    //"sort"
)

func main() {

    products := []Product {
        { "Kayak", 279} ,
        { "Lifejacket", 49.95 },
        { "Soccer Ball",  19.50 },
    }

    ProductSlices(products)

    for _, p := range products {
        Printfln("Name: %v, Price: %.2f", p.Name, p.Price)
    }
}
```

Листинг 18-16 Сортировка среза в файле main.go в папке mathandsorting

Скомпилируйте и выполните проект, и вы увидите, что выходные данные показывают значения Product, отсортированные в порядке возрастания поля Price:

```
Name: Soccer Ball, Price: 19.50
Name: Lifejacket, Price: 49.95
Name: Kayak, Price: 279.00
```

Сортировка с использованием разных полей

Композицию типов можно использовать для поддержки сортировки одного и того же типа структуры с использованием разных полей, как показано в листинге 18-17.

```
package main

import "sort"

type Product struct {
    Name string
    Price float64
}

type ProductSlice []Product

func ProductSlices(p []Product) {
    sort.Sort(ProductSlice(p))
}

func ProductSlicesAreSorted(p []Product) {
    sort.IsSorted(ProductSlice(p))
}

func (products ProductSlice) Len() int {
    return len(products)
}

func (products ProductSlice) Less(i, j int) bool {
    return products[i].Price < products[j].Price
}

func (products ProductSlice) Swap(i, j int) {
```

```
    products[i], products[j] = products[j], products[i]
}

type ProductSliceName struct { ProductSlice }

func ProductSlicesByName(p []Product) {
    sort.Sort(ProductSliceName{ p })
}

func (p ProductSliceName) Less(i, j int) bool {
    return p.ProductSlice[i].Name < p.ProductSlice[j].Name
}
```

Листинг 18-17 Сортировка различных полей в файле productsort.go в папке mathandsorting

Тип структуры определяется для каждого поля структуры, для которого требуется сортировка, со встроенным полем `ProductSlice`, подобным этому:

```
...
type ProductSliceName struct { ProductSlice }
...
```

Функция композиции типа означает, что методы, определенные для типа `ProductSlice`, повышаются до включающего типа. Определен новый метод `Less` для включающего типа, который будет использоваться для сортировки данных с использованием другого поля, например:

```
...
func (p ProductSliceName) Less(i, j int) bool {
    return p.ProductSlice[i].Name <= p.ProductSlice[j].Name
}
...
```

Последним шагом является определение функции, которая будет выполнять преобразование среза `Product` в новый тип и вызывать функцию `Sort`:

```
...
func ProductSlicesByName(p []Product) {
    sort.Sort(ProductSliceName{ p })
```

```
}
...
```

Результатом дополнений в листинге 18-17 является то, что срезы значений Product можно сортировать по значениям их полей Name, как показано в листинге 18-18.

```
package main

import (
    //"sort"
)

func main() {

    products := []Product {
        { "Kayak", 279} ,
        { "Lifejacket", 49.95 },
        { "Soccer Ball",  19.50 },
    }
    ProductSlicesByName(products)

    for _, p := range products {
        Printfln("Name: %v, Price: %.2f", p.Name, p.Price)
    }
}
```

Листинг 18-18 Сортировка по дополнительным полям в файле main.go в папке mathandsorting

Скомпилируйте и выполните проект, и вы увидите значения Product, отсортированные по полям Name, как показано ниже:

```
Name: Kayak, Price: 279.00
Name: Lifejacket, Price: 49.95
Name: Soccer Ball, Price: 19.50
```

Определение функции сравнения

Альтернативный подход — указать выражение, используемое для сравнения элементов вне функции sort, как показано в листинге 18-19.

```
package main

import "sort"

type Product struct {
    Name string
    Price float64
}

type ProductSlice []Product

// ...types and functions omitted for brevity...

type ProductComparison func(p1, p2 Product) bool

type ProductSliceFlex struct {
    ProductSlice
    ProductComparison
}

func (flex ProductSliceFlex) Less(i, j int) bool {
        return  flex.ProductComparison(flex.ProductSlice[i],
flex.ProductSlice[j])
}

func SortWith(prods []Product, f ProductComparison) {
    sort.Sort(ProductSliceFlex{ prods, f})
}
```

Листинг 18-19 Использование внешнего сравнения в файле productsort.go в папке mathandsorting

Создан новый тип с именем `ProductSliceFlex`, который объединяет данные и функцию сравнения, что позволит этому подходу вписаться в структуру функций, определенных пакетом `sort`. Метод `Less` определен для типа `ProductSliceFlex`, который вызывает функцию сравнения. Последняя часть головоломки — это функция `SortWith`, которая объединяет данные и функцию в значение `ProductSliceFlex` и передает его функции `sort.Sort`. В листинге 18-20 показана сортировка данных с помощью функции сравнения.

```
package main
```

```
import (
    //"sort"
)

func main() {

    products := []Product {
        { "Kayak", 279} ,
        { "Lifejacket", 49.95 },
        { "Soccer Ball",  19.50 },
    }

    SortWith(products, func (p1, p2 Product) bool {
        return p1.Name < p2.Name
    })

    for _, p := range products {
        Printfln("Name: %v, Price: %.2f",  p.Name, p.Price)
    }
}
```

Листинг 18-20 Sorting with a Comparison Function in the main.go File in the mathandsorting Folder

Данные сортируются путем сравнения поля `Name`, и код выдает следующий результат, когда проект компилируется и выполняется:

```
Name: Kayak, Price: 279.00
Name: Lifejacket, Price: 49.95
Name: Soccer Ball, Price: 19.50
```

Резюме

В этой главе я описал возможности, предоставляемые для генерации случайных чисел и перетасовки элементов в срезе. Я также описал противоположные функции, которые сортируют элементы в срезе. В следующей главе я опишу функции стандартной библиотеки для времени, даты и длительности.

19. Даты, время и продолжительность

В этой главе я описываю функции, предоставляемые пакетом `time`, который является частью стандартной библиотеки, отвечающей за представление моментов времени и длительностей. В таблице 19-1 эти функции представлены в контексте.

Таблица 19-1 Помещение дат, времени и продолжительности в контекст

Вопрос	Ответ
Кто они такие?	Функции, предоставляемые пакетом `time`, используются для представления определенных моментов времени и интервалов или длительностей.
Почему они полезны?	Эти функции полезны в любом приложении, которое должно иметь дело с календарем или будильником, а также для разработки любой функции, которая потребует задержек или уведомлений в будущем.
Как они используются?	Пакет `time` определяет типы данных для представления дат и отдельных единиц времени, а также функции для управления ими. Есть также функции, интегрированные в систему каналов Go.
Есть ли подводные камни или ограничения?	Даты могут быть сложными, и необходимо соблюдать осторожность при решении проблем с календарем и часовым поясом.
Есть ли альтернативы?	Это дополнительные функции, и их использование не обязательно.

Таблица 19-2 суммирует главу.

Таблица 19-2 Краткое содержание главы

Проблема	Решение	Листинг
Представить время, дату или продолжительность	Используйте функции и типы, определенные пакетом `time`	5, 13–16
Форматировать даты и время как строки	Используйте функцию `Format` и макет	6–7
Разобрать дату и время из строки	Используйте функцию `Parse`	8–12
Разобрать продолжительность из строки	Используйте функцию `ParseDuration`	17

Проблема	Решение	Листинг
Приостановить выполнение горутины	Используйте функцию `Sleep`	18
Отсрочка выполнения функции	Используйте функцию `AfterFunc`	19
Получать периодические уведомления	Используйте функцию `After`	20–24

Подготовка к этой главе

Чтобы подготовиться к этой главе, откройте новую командную строку, перейдите в удобное место и создайте каталог с именем `dateandtime`. Запустите команду, показанную в листинге 19-1, в папке `dateandtime`, чтобы создать файл модуля.

> **Подсказка**
> Вы можете загрузить пример проекта для этой главы — и для всех остальных глав этой книги — с `https://github.com/apress/pro-go`. См. Главу 2 о том, как получить помощь, если у вас возникнут проблемы с запуском примеров.

```
go mod init datesandtimes
```

Листинг 19-1 Инициализация модуля

Добавьте файл с именем `printer.go` в папку `dateandtimes` с содержимым, показанным в листинге 19-2.

```
package main

import "fmt"

func Printfln(template string, values ...interface{}) {
    fmt.Printf(template + "\n", values...)
}
```

Листинг 19-2 Содержимое файла printer.go в папке dateandtimes

Добавьте файл с именем `main.go` в папку `dateandtimes` с содержимым, показанным в листинге 19-3.

```
package main

func main() {

    Printfln("Hello, Dates and Times")
}
```

Листинг 19-3 Содержимое файла main.go в папке dateandtimes

Используйте командную строку для запуска команды, показанной в листинге 19-4, в папке `dateandtimes`.

```
go run .
```

Листинг 19-4 Запуск примера проекта

Код будет скомпилирован и выполнен, что приведет к следующему выводу:

```
Hello, Dates and Times
```

Представление дат и времени

Пакет `time` предоставляет функции для измерения длительности и выражения даты и времени. В следующих разделах я опишу наиболее полезные из этих функций.

Представление дат и времени

Пакет `time` предоставляет тип `Time`, который используется для представления определенного момента времени. Функции, описанные в таблице 19-3, используются для создания значений `Time`.

Таблица 19-3 Функции в пакете времени для создания значений времени

Функция	Описание
`Now()`	Эта функция создает `Time`, представляющий текущий момент времени.
`Date(y, m, d, h, min, sec, nsec, loc)`	Эта функция создает объект `Time`, представляющий указанный момент времени, который выражается аргументами года, месяца, дня, часа, минуты, секунды, наносекунды и `Location`. (Тип `Location` описан в разделе «Синтаксический анализ значений времени из строк».)
`Unix(sec, nsec)`	Эта функция создает значение `Time` из числа секунд и наносекунд с 1 января 1970 года по Гринвичу, широко известного как время Unix.

Доступ к компонентам Time осуществляется с помощью методов, описанных в таблице 19-4.

Таблица 19-4 Методы доступа к компонентам времени

Функция	Описание
Date()	Этот метод возвращает компоненты года, месяца и дня. Год и день выражаются как значения int, а месяц — как значение Month.
Clock()	Этот метод возвращает компоненты часа, минут и секунд Time.
Year()	Этот метод возвращает компонент года, выраженный как int.
YearDay()	Этот метод возвращает день года, выраженный как int от 1 до 366 (для учета високосных лет).
Month()	Этот метод возвращает компонент месяца, выраженный с использованием типа Month.
Day()	Этот метод возвращает день месяца, выраженный как int.
Weekday()	Этот метод возвращает день недели, выраженный как Weekday.
Hour()	Этот метод возвращает час дня, выраженный как int от 0 до 23.
Minute()	Этот метод возвращает количество минут, прошедших до часа дня, выраженное как int от 0 до 59.
Second()	Этот метод возвращает количество секунд, прошедших до минуты часа, выраженное как int от 0 до 59.
Nanosecond()	Этот метод возвращает количество наносекунд, прошедших до секунды минуты, выраженное как int от 0 до 999 999 999.

Для описания компонентов значения Time определены два типа, как описано в таблице 19-5.

Таблица 19-5 Типы, используемые для описания компонентов времени

Функция	Описание
Month	Этот тип представляет месяц, а пакет time определяет постоянные значения для названий месяцев на английском языке: January, February и т. д. Тип Month определяет метод String, который использует эти имена при форматировании строк.
Weekday	Этот тип представляет день недели, а пакет time определяет постоянные значения для названий дней недели на английском языке: Sunday, Monday и т. д. Тип Weekday определяет метод String, который использует эти имена при форматировании строк.

Используя типы и методы, описанные в таблицах с 19-3 по 19-5, в листинге 19-5 показано, как создавать значения Time и получать доступ к их компонентам.

```
package main

import "time"

func PrintTime(label string, t *time.Time) {
    Printfln("%s: Day: %v: Month: %v Year: %v",
        label, t.Day(), t.Month(), t.Year())
}

func main() {
    current := time.Now()
     specific := time.Date(1995, time.June, 9, 0, 0, 0, 0,
time.Local)
    unix := time.Unix(1433228090, 0)

    PrintTime("Current", &current)
    PrintTime("Specific", &specific)
    PrintTime("UNIX", &unix)
}
```

Листинг 19-5 Создание значений времени в файле main.go в папке dateandtimes

Операторы в функции main создают три разных значения Time с помощью функций, описанных в таблице 19-3. Постоянное значение June используется для создания одного из значений Time, что иллюстрирует использование одного из типов, описанных в таблице 19-5. Значения Time передаются функции PrintTime, которая использует методы из таблицы 19-4 для доступа к компонентам дня, месяца и года для записи сообщения, описывающего каждое Time. Скомпилируйте и выполните проект, и вы увидите результат, аналогичный следующему, с другим временем, возвращаемым функцией Now:

```
Current: Day: 2: Month: June Year: 2021
Specific: Day: 9: Month: June Year: 1995
UNIX: Day: 2: Month: June Year: 2015
```

Последним аргументом функции Date является Location, указывающий местоположение, часовой пояс которого будет использоваться для значения Time. В листинге 19-5 я использовал константу Local, определенную пакетом time, который предоставляет Location для часового пояса системы. Я объясню, как создать значения Location, которые не определяются конфигурацией системы, в разделе «Синтаксический анализ значений времени из строк» далее в этой главе.

Форматирование времени как строк

Метод Format используется для создания форматированных строк из значений Time. Формат строки определяется путем предоставления строки макета, которая показывает, какие компоненты Time требуются, а также порядок и точность, с которыми они должны быть выражены. Таблица 19-6 описывает метод Format для быстрого ознакомления.

Таблица 19-6 Метод Time для создания форматированных строк

Функция	**Описание**
Format(layout)	Этот метод возвращает отформатированную строку, созданную с использованием указанного макета.

В строке макета используется эталонное время, которое составляет 15:04:05 (что означает пять секунд после четырех минут после 15:00) в понедельник, 2 января 2006 г., в часовом поясе MST, что на 7 часов отстает от среднего времени по Гринвичу (GMT).). В листинге 19-6 показано использование эталонного времени для создания форматированных строк.

```
package main

import (
    "time"
    "fmt"
)

func PrintTime(label string, t *time.Time) {
    layout := "Day: 02 Month: Jan Year: 2006"
    fmt.Println(label, t.Format(layout))
}
```

```
func main() {
    current := time.Now()
     specific := time.Date(1995, time.June, 9, 0, 0, 0, 0,
time.Local)
    unix := time.Unix(1433228090, 0)

    PrintTime("Current", &current)
    PrintTime("Specific", &specific)
    PrintTime("UNIX", &unix)
}
```

Листинг 19-6 Форматирование значений времени как файла main.go в папке dateandtimes

Компоновка может смешивать компоненты даты с фиксированными строками, и в этом примере я использовал компоновку для воссоздания формата, использовавшегося в более ранних примерах, с указанием даты ссылки. Скомпилируйте и запустите проект, и вы увидите следующий вывод:

```
Current Day: 03 Month: Jun Year: 2021
Specific Day: 09 Month: Jun Year: 1995
UNIX Day: 02 Month: Jun Year: 2015
```

Пакет `time` определяет набор констант для распространенных форматов времени и даты, показанных в таблице 19-7.

Таблица 19-7 Константы компоновки, определяемые пакетом времени

Функция	Формат исходной даты
ANSIC	Mon Jan _2 15:04:05 2006
UnixDate	Mon Jan _2 15:04:05 MST 2006
RubyDate	Mon Jan 02 15:04:05 -0700 2006
RFC822	02 Jan 06 15:04 MST
RFC822Z	02 Jan 06 15:04 -0700
RFC850	Monday, 02-Jan-06 15:04:05 MST
RFC1123	Mon, 02 Jan 2006 15:04:05 MST
RFC1123Z	Mon, 02 Jan 2006 15:04:05 -0700
RFC3339	2006-01-02T15:04:05Z07:00
RFC3339Nano	2006-01-02T15:04:05.999999999Z07:00

Функция	Формат исходной даты
Kitchen	3:04PM
Stamp	Jan _2 15:04:05
StampMilli	Jan _2 15:04:05.000
StampMicro	Jan _2 15:04:05.000000
StampNano	Jan _2 15:04:05.000000000

Эти константы можно использовать вместо пользовательского макета, как показано в листинге 19-7.

```
package main

import (
    "time"
    "fmt"
)

func PrintTime(label string, t *time.Time) {
    //layout := "Day: 02 Month: Jan Year: 2006"
    fmt.Println(label, t.Format(time.RFC822Z))
}

func main() {
    current := time.Now()
     specific := time.Date(1995, time.June, 9, 0, 0, 0, 0,
time.Local)
    unix := time.Unix(1433228090, 0)

    PrintTime("Current", &current)
    PrintTime("Specific", &specific)
    PrintTime("UNIX", &unix)
}
```

Листинг 19-7 Использование предопределенного макета в файле main.go в папке dateandtimes

Пользовательский макет был заменен макетом RFC822Z, который выдает следующий вывод при компиляции и выполнении проекта:

```
Current 03 Jun 21 08:04 +0100
Specific 09 Jun 95 00:00 +0100
```

```
UNIX 02 Jun 15 07:54 +0100
```

Разбор значений времени из строк

Пакет `time` обеспечивает поддержку создания значений `Time` из строк, как описано в таблице 19-8.

Таблица 19-8 Функции пакета time для разбора строк в значения `Time`

Функция	Описание
`Parse(layout, str)`	Эта функция анализирует строку, используя указанный макет, чтобы создать значение `Time`. Возвращается `error`, указывающая на проблемы с разбором строки.
`ParseInLocation(layout, str, location)`	Эта функция анализирует строку, используя указанный макет и `Location`, если в строку не включен часовой пояс. Возвращается `error`, указывающая на проблемы с разбором строки.

В функциях, описанных в таблице 19-8, используется эталонное время, которое используется для указания формата анализируемой строки. Эталонное время — 15:04:05 (что означает пять секунд после четырех минут после 15:00) в понедельник, 2 января 2006 г., в часовом поясе MST, что на семь часов отстает от GMT.

Компоненты ссылочной даты организованы так, чтобы указать структуру строки даты, которая должна быть проанализирована, как показано в листинге 19-8.

```
package main

import (
    "time"
    "fmt"
)

func PrintTime(label string, t *time.Time) {
    //layout := "Day: 02 Month: Jan Year: 2006"
    fmt.Println(label, t.Format(time.RFC822Z))
}

func main() {
    layout := "2006-Jan-02"
    dates := []string {
        "1995-Jun-09",
```

```
        "2015-Jun-02",
    }

    for _, d := range dates {
        time, err := time.Parse(layout, d)
        if (err == nil) {
            PrintTime("Parsed", &time)
        } else {
            Printfln("Error: %s", err.Error())
        }
    }
}
```

Листинг 19-8 Анализ строки даты в файле main.go в папке dateandtimes

Макет, используемый в этом примере, включает четыре цифры года, три буквы месяца и две цифры дня, разделенные дефисами. Макет передается в функцию `Parse` вместе со строкой для анализа, и функция возвращает значение времени и ошибку, в которой подробно описаны любые проблемы синтаксического анализа. Скомпилируйте и выполните проект, и вы получите следующий вывод, хотя вы можете увидеть другое смещение часового пояса (к которому я вскоре вернусь):

```
Parsed 09 Jun 95 00:00 +0000
Parsed 02 Jun 15 00:00 +0000
```

Использование предопределенных макетов даты

Константы макета, описанные в таблице 19-7, можно использовать для анализа дат, как показано в листинге 19-9.

```
package main

import (
    "time"
    "fmt"
)

func PrintTime(label string, t *time.Time) {
    //layout := "Day: 02 Month: Jan Year: 2006"
    fmt.Println(label, t.Format(time.RFC822Z))
}
```

```
func main() {
    //layout := "2006-Jan-02"
    dates := []string {
        "09 Jun 95 00:00 GMT",
        "02 Jun 15 00:00 GMT",
    }

    for _, d := range dates {
        time, err := time.Parse(time.RFC822, d)
        if (err == nil) {
            PrintTime("Parsed", &time)
        } else {
            Printfln("Error: %s", err.Error())
        }
    }
}
```

Листинг 19-9 Использование предопределенного макета в файле main.go в папке dateandtimes

В этом примере используется константа RFC822 для синтаксического анализа строк даты и получения следующего вывода, хотя вы можете увидеть другое смещение часового пояса:

```
Parsed 09 Jun 95 01:00 +0100
Parsed 02 Jun 15 01:00 +0100
```

Указание разбора местоположения

Функция Parse предполагает, что даты и время, выраженные без часового пояса, определены в универсальном скоординированном времени (UTC). Метод ParseInLocation можно использовать для указания местоположения, которое используется, когда часовой пояс не указан, как показано в листинге 19-10.

```
package main

import (
    "time"
    "fmt"
)

func PrintTime(label string, t *time.Time) {
```

```
    //layout := "Day: 02 Month: Jan Year: 2006"
    fmt.Println(label, t.Format(time.RFC822Z))
}

func main() {
    layout := "02 Jan 06 15:04"
    date := "09 Jun 95 19:30"

    london, lonerr := time.LoadLocation("Europe/London")
    newyork, nycerr := time.LoadLocation("America/New_York")

    if (lonerr == nil && nycerr == nil) {
        nolocation, _ := time.Parse(layout, date)
         londonTime, _ := time.ParseInLocation(layout, date,
london)
        newyorkTime, _ := time.ParseInLocation(layout, date,
newyork)

        PrintTime("No location:", &nolocation)
        PrintTime("London:", &londonTime)
        PrintTime("New York:", &newyorkTime)
    } else {
        fmt.Println(lonerr.Error(), nycerr.Error())
    }
}
```

Листинг 19-10 Указание местоположения в файле main.go в папке dateandtimes

`ParseInLocation` принимает аргумент `time.Location`, указывающий местоположение, часовой пояс которого будет использоваться, если он не включен в проанализированную строку. Значения `Location` можно создать с помощью функций, описанных в таблице 19-9.

Таблица 19-9 Функции для создания локаций

Функция	Описание
`LoadLocation(name)`	Эта функция возвращает `*Location` для указанного имени и `error`, указывающую на наличие проблем.
`LoadLocationFromTZData(name, data)`	Эта функция возвращает `*Location` из байтового среза, содержащего отформатированную базу данных часовых поясов.

Функция	Описание
FixedZone(name, offset)	Эта функция возвращает *Location, который всегда использует указанное имя и смещение от UTC.

Когда место передается функции LoadLocation, возвращаемое местоположение содержит сведения о часовых поясах, используемых в этом местоположении. Названия мест определены в базе данных часовых поясов IANA, https://www.iana.org/time-zones, и перечислены в https://en.wikipedia.org/wiki/List_of_tz_database_time_zones. В примере в листинге 19-10 указаны значения Europe/London и America/New_York, что дает значения Location для Лондона и Нью-Йорка. Скомпилируйте и выполните код, и вы увидите следующий вывод:

```
No location: 09 Jun 95 19:30 +0000
London: 09 Jun 95 19:30 +0100
New York: 09 Jun 95 19:30 -0400
```

Три даты показывают, как строка анализируется с использованием разных часовых поясов. При использовании метода Parse предполагается, что часовой пояс соответствует UTC с нулевым смещением (компонент +0000 выходных данных). Когда используется местоположение в Лондоне, предполагается, что время на один час опережает время в формате UTC, поскольку дата в проанализированной строке попадает в период перехода на летнее время, используемый в Соединенном Королевстве. Точно так же, когда используется местоположение в Нью-Йорке, смещение составляет четыре часа от UTC.

Встраивание базы данных часовых поясов

База данных часовых поясов, используемая для создания значений Location, устанавливается вместе с инструментами Go, что означает, что она может быть недоступна при развертывании скомпилированного приложения. Пакет time/tzdata содержит встроенную версию базы данных, загружаемую функцией инициализации пакета (как описано в главе 12). Чтобы

гарантировать, что данные часового пояса всегда будут доступны, объявите зависимость от пакета следующим образом:

```
...
import (
    "fmt"
    "time"
       _ "time/tzdata"
)
...
```

В пакете нет экспортированных функций, поэтому пустой идентификатор должен использоваться для объявления зависимости без создания ошибки компилятора.

Использование локального местоположения

Если имя места, используемое для создания `Location`, является `Local`, то используется настройка часового пояса компьютера, на котором запущено приложение, как показано в листинге 19-11.

```
package main

import (
    "time"
    "fmt"
)

func PrintTime(label string, t *time.Time) {
    //layout := "Day: 02 Month: Jan Year: 2006"
    fmt.Println(label, t.Format(time.RFC822Z))
}

func main() {

    layout := "02 Jan 06 15:04"
    date := "09 Jun 95 19:30"

    london, lonerr := time.LoadLocation("Europe/London")
    newyork, nycerr := time.LoadLocation("America/New_York")
    local, _ := time.LoadLocation("Local")
```

```
    if (lonerr == nil && nycerr == nil) {
        nolocation, _ := time.Parse(layout, date)
         londonTime, _ := time.ParseInLocation(layout, date,
london)
        newyorkTime, _ := time.ParseInLocation(layout, date,
newyork)
         localTime, _ := time.ParseInLocation(layout, date,
local)

        PrintTime("No location:", &nolocation)
        PrintTime("London:", &londonTime)
        PrintTime("New York:", &newyorkTime)
        PrintTime("Local:", &localTime)
    } else {
        fmt.Println(lonerr.Error(), nycerr.Error())
    }
}
```

Листинг 19-11 Использование местного часового пояса в файле main.go в папке dateandtimes

Результат, полученный в этом примере, будет отличаться в зависимости от вашего местоположения. Я живу в Соединенном Королевстве, а это означает, что мой местный часовой пояс на один час опережает UTC во время перехода на летнее время, что приводит к следующему результату:

```
No location: 09 Jun 95 19:30 +0000
London: 09 Jun 95 19:30 +0100
New York: 09 Jun 95 19:30 -0400
Local: 09 Jun 95 19:30 +0100
```

Непосредственное указание часовых поясов

Использование географических названий — самый надежный способ убедиться, что даты анализируются правильно, поскольку автоматически применяется летнее время. Функцию `FixedZone` можно использовать для создания `Location` с фиксированным часовым поясом, как показано в листинге 19-12.

```
package main

import (
    "time"
```

```
    "fmt"
)

func PrintTime(label string, t *time.Time) {
    //layout := "Day: 02 Month: Jan Year: 2006"
    fmt.Println(label, t.Format(time.RFC822Z))
}

func main() {

    layout := "02 Jan 06 15:04"
    date := "09 Jun 95 19:30"

    london := time.FixedZone("BST", 1 * 60 * 60)
    newyork := time.FixedZone("EDT", -4 * 60 * 60)
    local := time.FixedZone("Local", 0)

    //if (lonerr == nil && nycerr == nil) {
        nolocation, _ := time.Parse(layout, date)
         londonTime, _ := time.ParseInLocation(layout, date,
london)
         newyorkTime, _ := time.ParseInLocation(layout, date,
newyork)
          localTime, _ := time.ParseInLocation(layout, date,
local)

        PrintTime("No location:", &nolocation)
        PrintTime("London:", &londonTime)
        PrintTime("New York:", &newyorkTime)
        PrintTime("Local:", &localTime)
    // } else {
    //     fmt.Println(lonerr.Error(), nycerr.Error())
    // }
}
```

Листинг 19-12 Указание часовых поясов в файле main.go в папке dateandtimes

Аргументами функции `FixedZone` являются имя и количество секунд, смещенных от UTC. В этом примере создаются три фиксированных часовых пояса, один из которых опережает UTC на час, другой отстает на четыре часа, а третий не имеет смещения. Скомпилируйте и запустите проект, и вы увидите следующий вывод:

```
No location: 09 Jun 95 19:30 +0000
London: 09 Jun 95 19:30 +0100
New York: 09 Jun 95 19:30 -0400
Local: 09 Jun 95 19:30 +0000
```

Управление значениями времени

Пакет `time` определяет методы работы со значениями `Time`, как описано в таблице 19-10. Некоторые из этих методов основаны на типе `Duration`, который я опишу в следующем разделе.

Таблица 19-10 Методы работы со значениями Time

Функция	Описание
`Add(duration)`	Этот метод добавляет указанную `Duration` к `Time` и возвращает результат.
`Sub(time)`	Этот метод возвращает значение `Duration`, выражающее разницу между `Time` вызова метода и `Time`, указанным в качестве аргумента.
`AddDate(y, m, d)`	Этот метод добавляет к `Time` указанное количество лет, месяцев и дней и возвращает результат.
`After(time)`	Этот метод возвращает значение `true`, если `Time`, в которое был вызван метод, наступает после `Time`, указанного в качестве аргумента.
`Before(time)`	Этот метод возвращает значение `true`, если время, в которое был вызван метод, предшествует `Time`, указанному в качестве аргумента.
`Equal(time)`	Этот метод возвращает значение `true`, если `Time`, в которое был вызван метод, равно `Time`, указанному в качестве аргумента.
`IsZero()`	Этот метод возвращает значение `true`, если `Time`, в которое был вызван метод, представляет момент нулевого времени, то есть 1 января 1 года, 00:00:00 UTC.
`In(loc)`	Этот метод возвращает значение `Time`, выраженное в указанном `Location`.
`Location()`	Этот метод возвращает `Location`, связанный с `Time`, что фактически позволяет выразить время в другом часовом поясе.
`Round(duration)`	Этот метод округляет `Time` до ближайшего интервала, представленного значением `Duration`.
`Truncate(duration)`	Этот метод округляет `Time` до ближайшего интервала, представленного значением `Duration`.

В листинге 19-13 `Time` анализируется из строки и используются некоторые из методов, описанных в таблице.

```
package main

import (
    "time"
    "fmt"
)

func main() {
    t, err := time.Parse(time.RFC822, "09 Jun 95 04:59 BST")
    if (err == nil) {
        Printfln("After: %v", t.After(time.Now()))
        Printfln("Round: %v", t.Round(time.Hour))
        Printfln("Truncate: %v", t.Truncate(time.Hour))
    } else {
        fmt.Println(err.Error())
    }
}
```

Листинг 19-13 Работа со значением времени в файле main.go в папке dateandtimes

Скомпилируйте и выполните проект, и вы получите следующий вывод, позволяющий изменять формат даты:

```
After: false
Round: 1995-06-09 05:00:00 +0100 BST
Truncate: 1995-06-09 04:00:00 +0100 BST
```

Значения `Time` можно сравнивать с помощью функции `Equal`, которая учитывает разницу в часовых поясах, как показано в листинге 19-14.

```
package main

import (
    //"fmt"
    "time"
)

func main() {
      t1, _ := time.Parse(time.RFC822Z, "09 Jun 95 04:59
+0100")
      t2, _ := time.Parse(time.RFC822Z, "08 Jun 95 23:59
-0400")
```

```
    Printfln("Equal Method: %v", t1.Equal(t2))
    Printfln("Equality Operator: %v", t1 == t2)
}
```

Листинг 19-14 Сравнение значений времени в файле main.go в папке dateandtimes

Значения Time в этом примере отражают один и тот же момент в разных часовых поясах. Функция `Equal` учитывает влияние часовых поясов, чего не происходит при использовании стандартного оператора равенства. Скомпилируйте и запустите проект, и вы получите следующий вывод:

```
Equal Method: true
Equality Operator: false
```

Представление продолжительности

Тип `Duration` является псевдонимом типа `int64` и используется для представления определенного количества миллисекунд. Пользовательские значения `Duration` состоят из постоянных значений `Duration`, определенных в пакете time, описанном в таблице 19-11.

Таблица 19-11 Константы длительности в пакете времени

Функция	Описание
`Hour`	Эта константа представляет 1 час.
`Minute`	Эта константа представляет 1 минуту.
`Second`	Эта константа представляет 1 секунду.
`Millisecond`	Эта константа представляет 1 миллисекунду.
`Microsecond`	Эта константа представляет 1 миллисекунду.
`Nanosecond`	Эта константа представляет 1 наносекунду.

После создания `Duration` его можно проверить с помощью методов, описанных в таблице 19-12.

Таблица 19-12 Методы продолжительности

Функция	Описание
`Hours()`	Этот метод возвращает `float64`, который представляет `Duration` в часах.

Функция	Описание
`Minutes()`	Этот метод возвращает `float64`, который представляет `Duration` в минутах.
`Seconds()`	Этот метод возвращает `float64`, который представляет `Duration` в секундах.
`Milliseconds()`	Этот метод возвращает `float64`, который представляет `Duration` в миллисекундах.
`Microseconds()`	Этот метод возвращает `float64`, который представляет `Duration` в микросекундах.
`Nanoseconds()`	Этот метод возвращает `float64`, который представляет `Duration` в наносекундах.
`Round(duration)`	Этот метод возвращает `Duration`, которая округляется до ближайшего кратного указанной `Duration`.
`Truncate(duration)`	Этот метод возвращает `Duration`, которая округляется в меньшую сторону до ближайшего кратного указанной `Duration`.

В листинге 19-15 показано, как можно использовать константы для создания `Duration`, и используются некоторые методы из таблицы 19-12.

```
package main

import (
    //"fmt"
    "time"
)

func main() {

    var d time.Duration = time.Hour + (30 * time.Minute)

    Printfln("Hours: %v", d.Hours())
    Printfln("Mins: %v", d.Minutes())
    Printfln("Seconds: %v", d.Seconds())
    Printfln("Millseconds: %v", d.Milliseconds())

    rounded := d.Round(time.Hour)
    Printfln("Rounded Hours: %v", rounded.Hours())
    Printfln("Rounded Mins: %v", rounded.Minutes())

    trunc := d.Truncate(time.Hour)
```

```
    Printfln("Truncated  Hours: %v", trunc.Hours())
    Printfln("Rounded Mins: %v", trunc.Minutes())
}
```

Листинг 19-15 Создание и проверка продолжительности в файле main.go в папке dateandtimes

Для параметра `Duration` установлено значение 90 минут, а затем для вывода используются методы `Hours`, `Minutes`, `Seconds` и `Milliseconds`. Методы `Round` и `Truncate` используются для создания новых значений `Duration`, которые записываются в виде часов и минут. Скомпилируйте и запустите проект, и вы получите следующий вывод:

```
Hours: 1.5
Mins: 90
Seconds: 5400
Millseconds: 5400000
Rounded Hours: 2
Rounded Mins: 120
Truncated  Hours: 1
Rounded Mins: 60
```

Обратите внимание, что методы в таблице 19-12 возвращают всю продолжительность, выраженную в определенных единицах измерения, таких как часы или минуты. Это отличается от методов с похожими именами, определенных типом `Time`, которые возвращают только одну часть даты/времени.

Создание продолжительности относительно времени

Пакет `time` определяет две функции, которые можно использовать для создания значений `Duration`, представляющих количество времени между определенным `Time` и текущим `Time`, как описано в таблице 19-13.

Таблица 19-13 Функции времени для создания значений длительности относительно времени

Функция	**Описание**
`Since(time)`	Эта функция возвращает `Duration`, выражающую время, прошедшее с момента указанного значения `Time`.

Функция	Описание
Until(time)	Эта функция возвращает Duration, выражающую время, прошедшее до указанного значения Time.

Перечисление 19-16 демонстрирует использование этих функций.

```
package main

import (
    //"fmt"
    "time"
)

func main() {
    toYears := func(d time.Duration) int {
        return int( d.Hours() / (24 * 365))
    }

    future := time.Date(2051, 0, 0, 0, 0, 0, 0, time.Local)
    past := time.Date(1965, 0, 0, 0, 0, 0, 0, time.Local)

    Printfln("Future: %v", toYears(time.Until(future)))
    Printfln("Past: %v", toYears(time.Since(past)))
}
```

Листинг 19-16 Создание продолжительности относительно времени в файле main.go в папке dateandtimes

В примере используются методы Until и Since, чтобы вычислить, сколько лет осталось до 2051 года и сколько лет прошло с 1965 года. Код в листинге 19-16 при компиляции выдает следующий результат, хотя вы можете увидеть разные результаты в зависимости от того, когда вы запускаете пример:

```
Future: 29
Past: 56
```

Создание длительности из строк

Функция time.ParseDuration анализирует строки для создания значений Duration. Для быстрого ознакомления эта функция описана в таблице 19-14.

Таблица 19-14 Функция для разбора строк в значения длительности

Функция	Описание
ParseDuration(str)	Эта функция возвращает значение Duration и error, указывающую на наличие проблем при синтаксическом анализе указанной строки.

Формат строк, поддерживаемый функцией ParseDuration, представляет собой последовательность числовых значений, за которыми следуют индикаторы единиц измерения, описанные в таблице 19-15.

Таблица 19-15 Индикаторы единиц строки продолжительности

Unit	Описание
h	Эта единица обозначает часы.
m	Эта единица обозначает минуты.
s	Эта единица обозначает секунды.
ms	Эта единица обозначает миллисекунды.
us или μs	Эта единица обозначает микросекунды.
ns	Эта единица обозначает наносекунды.

Между значениями не допускаются пробелы, которые могут быть указаны как целые числа или числа с плавающей запятой. В листинге 19-17 показано создание Duration из строки.

```
package main

import (
    "fmt"
    "time"
)

func main() {
    d, err := time.ParseDuration("1h30m")
    if (err == nil) {
        Printfln("Hours: %v", d.Hours())
        Printfln("Mins: %v", d.Minutes())
        Printfln("Seconds: %v", d.Seconds())
        Printfln("Millseconds: %v", d.Milliseconds())
    } else {
```

```
        fmt.Println(err.Error())
    }
}
```

Листинг 19-17 Разбор строки в файле main.go в папке dateandtimes

Строка указывает 1 час и 30 минут. Скомпилируйте и выполните проект, и будет получен следующий вывод:

```
Hours: 1.5
Mins: 90
Seconds: 5400
Millseconds: 5400000
```

Использование функций времени для горутин и каналов

Пакет `time` предоставляет небольшой набор функций, полезных для работы с горутинами и каналами, как описано в таблице 19-16.

Таблица 19-16 Функции пакета времени

Функция	Описание
`Sleep(duration)`	Эта функция приостанавливает текущую горутину по крайней мере на указанное время.
`AfterFunc(duration, func)`	Эта функция выполняет указанную функцию в своей собственной горутине по истечении указанного времени. Результатом является `*Timer`, метод `Stop` которого можно использовать для отмены выполнения функции до истечения продолжительности.
`After(duration)`	Эта функция возвращает канал, который блокируется на указанное время, а затем возвращает значение `Time`. Подробнее см. в разделе «Получение уведомлений по времени».
`Tick(duration)`	Эта функция возвращает канал, который периодически отправляет значение `Time`, где период указан как продолжительность.

Хотя все эти функции определены в одном пакете, они используются по-разному, как показано в следующих разделах.

Перевод горутины в сон

Функция `Sleep` приостанавливает выполнение текущей горутины на указанное время, как показано в листинге 19-18.

```
package main

import (
    //"fmt"
    "time"
)

func writeToChannel(channel chan <- string) {
    names := []string { "Alice", "Bob", "Charlie", "Dora" }
    for _, name := range names {
        channel <- name
        time.Sleep(time.Second * 1)
    }
    close(channel)
}

func main() {

    nameChannel := make (chan string)

    go writeToChannel(nameChannel)

    for name := range nameChannel {
        Printfln("Read name: %v", name)
    }
}
```

Листинг 19-18 Приостановка горутины в файле main.go в папке dateandtimes

Продолжительность, указанная функцией `Sleep`, — это минимальное количество времени, на которое горутина будет приостановлена, и вам не следует полагаться на точные периоды времени, особенно при меньшей продолжительности. Имейте в виду, что функция `Sleep` приостанавливает горутину, в которой она вызывается, а это значит, что она также приостанавливает `main` горутину, что может создать видимость блокировки приложения. (Если это произойдет, ключ к тому, что вы случайно вызвали функцию `Sleep`, заключается в том, что автоматическое обнаружение взаимоблокировки не вызовет паники.) Скомпилируйте и выполните проект, и вы увидите следующий вывод, который создается с небольшой задержкой между именами:

```
Read name: Alice
Read name: Bob
Read name: Charlie
Read name: Dora
```

Отсрочка выполнения функции

Функция `AfterFunc` используется для отсрочки выполнения функции на указанный период, как показано в листинге 19-19.

```
package main

import (
    //"fmt"
    "time"
)

func writeToChannel(channel chan <- string) {
    names := []string { "Alice", "Bob", "Charlie", "Dora" }
    for _, name := range names {
        channel <- name
        //time.Sleep(time.Second * 1)
    }
    close(channel)
}

func main() {

    nameChannel := make (chan string)

    time.AfterFunc(time.Second * 5, func () {
        writeToChannel(nameChannel)
    })

    for name := range nameChannel {
        Printfln("Read name: %v", name)
    }
}
```

Листинг 19-19 Откладывание функции в файле main.go в папке dateandtimes

Первый аргумент `AfterFunc` — это период задержки, который в данном примере составляет пять секунд. Второй аргумент — это

функция, которая будет выполняться. В этом примере я хочу выполнить функцию `writeToChannel`, но `AfterFunc` принимает только функции без параметров или результатов, поэтому мне приходится использовать простую оболочку. Скомпилируйте и выполните проект, и вы увидите следующие результаты, которые выписываются после пятисекундной задержки:

```
Read name: Alice
Read name: Bob
Read name: Charlie
Read name: Dora
```

Получение уведомлений по времени

Функция `After` ожидает указанное время, а затем отправляет значение `Time` в канал, что является полезным способом использования канала для получения уведомления в заданное время в будущем, как показано в листинге 19-20.

```
package main

import (
    //"fmt"
    "time"
)

func writeToChannel(channel chan <- string) {

    Printfln("Waiting for initial duration...")
    _ = <- time.After(time.Second * 2)
    Printfln("Initial duration elapsed.")

    names := []string { "Alice", "Bob", "Charlie", "Dora" }
    for _, name := range names {
        channel <- name
        time.Sleep(time.Second * 1)
    }
    close(channel)
}

func main() {
```

```
    nameChannel := make (chan string)

    go writeToChannel(nameChannel)

    for name := range nameChannel {
        Printfln("Read name: %v", name)
    }
}
```

Листинг 19-20 Получение уведомления о будущем в файле main.go в папке dateandtimes

Результатом функции `After` является канал, содержащий значения `Time`. Канал блокируется на указанную продолжительность, когда отправляется значение `Time`, указывающее, что продолжительность прошла. В этом примере значение, отправленное по каналу, действует как сигнал и не используется напрямую, поэтому ему присваивается пустой идентификатор, например:

```
...
_ = <- time.After(time.Second * 2)
...
```

Такое использование функции `After` вводит начальную задержку в функции `writeToChannel`. Скомпилируйте и запустите проект, и вы увидите следующий вывод:

```
Waiting for initial duration...
Initial duration elapsed.
Read name: Alice
Read name: Bob
Read name: Charlie
Read name: Dora
```

Эффект в этом примере такой же, как при использовании функции `Sleep`, но разница в том, что функция `After` возвращает канал, который не блокируется до тех пор, пока не будет прочитано значение, а это значит, что можно указать направление, можно выполнить дополнительную работу, а затем может быть выполнено чтение канала, в результате чего канал будет заблокирован только на оставшуюся часть времени.

Использование уведомлений в качестве тайм-аутов в операторах Select

Функцию `After` можно использовать с операторами `select` для предоставления времени ожидания, как показано в листинге 19-21.

```
package main

import (
    //"fmt"
    "time"
)

func writeToChannel(channel chan <- string) {

    Printfln("Waiting for initial duration...")
    _ = <- time.After(time.Second * 2)
    Printfln("Initial duration elapsed.")

    names := []string { "Alice", "Bob", "Charlie", "Dora" }
    for _, name := range names {
        channel <- name
        time.Sleep(time.Second * 3)
    }
    close(channel)
}

func main() {

    nameChannel := make (chan string)

    go writeToChannel(nameChannel)

    channelOpen := true
    for channelOpen {
        Printfln("Starting channel read")
        select {
            case name, ok := <- nameChannel:
                if (!ok) {
                    channelOpen = false
                    break
                } else {
                    Printfln("Read name: %v", name)
```

```
                }
            case <- time.After(time.Second * 2):
                Printfln("Timeout")
        }
    }
}
```

Листинг 19-21 Использование тайм-аута в операторе Select в файле main.go в папке dateandtimes

Оператор `select` будет блокироваться до тех пор, пока один из каналов не будет готов или пока не истечет время таймера. Это работает, потому что оператор `select` будет блокироваться до тех пор, пока один из его каналов не будет готов, и потому что функция `After` создает канал, который блокируется на указанный период. Скомпилируйте и запустите проект, и вы увидите следующий вывод:

```
Waiting for initial duration...
Initial duration elapsed.
Timeout
Read name: Alice
Timeout
Read name: Bob
Timeout
Read name: Charlie
Timeout
Read name: Dora
Timeout
```

Остановка и сброс таймеров

Функция `After` полезна, когда вы уверены, что вам всегда будут нужны уведомления по времени. Если вам нужна возможность отменить уведомление, то вместо нее можно использовать функцию, описанную в таблице 19-17.

Таблица 19-17 Функция пакета time для создания таймера

Функция	Описание
`NewTimer(duration)`	Эта функция возвращает `*Timer` с указанным периодом.

Результатом функции NewTimer является указатель на структуру Timer, которая определяет методы, описанные в таблице 19-18.

Таблица 19-18 Методы, определяемые структурой Timer

Функция	Описание
C	Это поле возвращает канал, по которому Time будет отправлять свое значение Time.
Stop()	Этот метод останавливает таймер. Результатом является логическое значение, которое будет true, если таймер был остановлен, и false, если таймер уже отправил свое сообщение.
Reset(duration)	Этот метод останавливает таймер и сбрасывает его так, чтобы его интервал был заданным значением Duration.

В листинге 19-22 функция NewTimer используется для создания Timer, который сбрасывается до истечения указанного времени.

> **Осторожно**
> Будьте осторожны при остановке таймера. Канал таймера не закрыт, что означает, что чтение из канала будет продолжать блокироваться даже после остановки таймера.

```
package main

import (
    //"fmt"
    "time"
)

func writeToChannel(channel chan <- string) {

    timer := time.NewTimer(time.Minute * 10)

    go func () {
        time.Sleep(time.Second * 2)
        Printfln("Resetting timer")
        timer.Reset(time.Second)
    }()

    Printfln("Waiting for initial duration...")
    <- timer.C
```

```
    Printfln("Initial duration elapsed.")

    names := []string { "Alice", "Bob", "Charlie", "Dora" }
    for _, name := range names {
        channel <- name
        //time.Sleep(time.Second * 3)
    }
    close(channel)
}

func main() {

    nameChannel := make (chan string)

    go writeToChannel(nameChannel)

    for name := range nameChannel {
        Printfln("Read name: %v", name)
    }
}
```

Листинг 19-22 Сброс таймера в файле main.go в папке dateandtimes

`Timer` в этом примере создается с продолжительностью десять минут. Горутина спит в течение двух секунд, а затем сбрасывает таймер, чтобы ее продолжительность составляла две секунды. Скомпилируйте и запустите проект, и вы увидите следующий вывод:

```
Waiting for initial duration...
Resetting timer
Initial duration elapsed.
Read name: Alice
Read name: Bob
Read name: Charlie
Read name: Dora
```

Получение повторяющихся уведомлений

Функция `Tick` возвращает канал, по которому значения `Time` отправляются с заданным интервалом, как показано в листинге 19-23.

```
package main
```

```
import (
    //"fmt"
    "time"
)

func writeToChannel(nameChannel chan <- string) {

    names := []string { "Alice", "Bob", "Charlie", "Dora" }

    tickChannel := time.Tick(time.Second)
    index := 0

    for {
        <- tickChannel
        nameChannel <- names[index]
        index++
        if (index == len(names)) {
            index = 0
        }
    }
}

func main() {

    nameChannel := make (chan string)

    go writeToChannel(nameChannel)

    for name := range nameChannel {
        Printfln("Read name: %v", name)
    }
}
```

Листинг 19-23 Получение повторяющихся уведомлений в файле main.go в папке dateandtimes

Как и прежде, полезность канала, созданного функцией `Tick`, заключается не в передаваемых по нему значениях `Time`, а в периодичности их отправки. В этом примере функция `Tick` используется для создания канала, по которому значения будут отправляться каждую секунду. Канал блокируется, когда нет значения для чтения, что позволяет каналам, созданным с помощью функции

Tick, управлять скоростью, с которой функция writeToChannel генерирует значения. Скомпилируйте и выполните проект, и вы увидите следующий вывод, который повторяется до тех пор, пока программа не будет завершена:

```
Read name: Alice
Read name: Bob
Read name: Charlie
Read name: Dora
Read name: Alice
Read name: Bob
...
```

Функция Tick полезна, когда требуется неопределенная последовательность сигналов. Если требуется фиксированный ряд значений, вместо этого можно использовать функцию, описанную в таблице 19-19.

Таблица 19-19 Функция времени для создания тикера

Функция	**Описание**
NewTicker(duration)	Эта функция возвращает *Ticker с указанным периодом.

Результатом функции NewTicker является указатель на структуру Ticker, которая определяет поле и методы, описанные в таблице 19-20.

Таблица 19-20 Поле и методы, определяемые структурой тикера

Функция	**Описание**
C	Это поле возвращает канал, по которому Ticker будет отправлять значения Time.
Stop()	Этот метод останавливает тикер (но не закрывает канал, возвращаемый полем C).
Reset(duration)	Этот метод останавливает тикер и сбрасывает его так, чтобы его интервал был равен указанной Duration.

В листинге 19-24 функция NewTicker используется для создания Ticker, который останавливается, когда он больше не нужен.

```
package main
```

```
import (
    //"fmt"
    "time"
)

func writeToChannel(nameChannel chan <- string) {

    names := []string { "Alice", "Bob", "Charlie", "Dora" }

    ticker := time.NewTicker(time.Second / 10)
    index := 0
    for {
        <- ticker.C
        nameChannel <- names[index]
        index++
        if (index == len(names)) {
            ticker.Stop()
            close(nameChannel)
            break
        }
    }
}

func main() {

    nameChannel := make (chan string)

    go writeToChannel(nameChannel)

    for name := range nameChannel {
        Printfln("Read name: %v", name)
    }
}
```

Листинг 19-24 Создание тикера в файле main.go в папке dateandtimes

Этот подход полезен, когда приложению необходимо создать несколько тикеров, не оставляя тех, которые больше не требуются для отправки сообщений. Скомпилируйте и запустите проект, и вы увидите следующий вывод:

```
Read name: Alice
Read name: Bob
```

```
Read name: Charlie
Read name: Dora
```

Резюме

В этой главе я описал функции стандартной библиотеки Go для работы со временем, датами и длительностью, включая встроенную поддержку каналов и горутин. В следующей главе я расскажу о средствах чтения и записи, которые представляют собой механизм Go для чтения и записи данных.

20. Чтение и запись данных

В этой главе я описываю два наиболее важных интерфейса, определенных стандартной библиотекой: интерфейсы `Reader` и `Writer`. Эти интерфейсы используются везде, где данные считываются или записываются, а это означает, что любой источник или назначение данных может обрабатываться практически одинаково, так что, например, запись данных в файл точно такая же, как запись данных в сетевое соединение. В таблице 20-1 описаны функции, описанные в этой главе, в контексте.

Таблица 20-1 Помещение средств чтения и записи в контекст

Вопрос	Ответ
Кто они такие?	Эти интерфейсы определяют основные методы, необходимые для чтения и записи данных.
Почему они полезны?	Такой подход означает, что почти любой источник данных можно использовать одинаковым образом, при этом позволяя определять специализированные функции с помощью функций композиции, описанных в главе 13.
Как это используется?	Пакет `io` определяет эти интерфейсы, но их реализации доступны в ряде других пакетов, некоторые из которых подробно описаны в следующих главах.
Есть ли подводные камни или ограничения?	Эти интерфейсы не полностью скрывают детали источников или мест назначения для данных, и часто требуются дополнительные методы, предоставляемые интерфейсами, основанными на `Reader` и `Writer`.
Есть ли альтернативы?	Использование этих интерфейсов необязательно, но их трудно избежать, поскольку они используются во всей стандартной библиотеке.

Таблица 20-2 суммирует главу.

Таблица 20-2 Краткое содержание главы

Проблема	Решение	Листинг
Читать данные	Используйте реализацию интерфейса `Reader`	6

Проблема	Решение	Листинг
Записать данные	Используйте реализацию интерфейса `Writer`	7
Упростить процесс чтения и записи данных	Используйте служебные функции	8
Объединие средств чтения или записи	Используйте специализированные реализации	9–16
Чтение и запись в буфер	Используйте возможности, предоставляемые пакетом `bufio`	17–23
Сканирование и форматирование данных с помощью средств чтения и записи	Используйте функции пакета `fmt`, которые принимают аргументы `Reader` или `Writer`.	24–27

Подготовка к этой главе

Чтобы подготовиться к этой главе, откройте новую командную строку, перейдите в удобное место и создайте каталог с именем `readerandwriters`. Запустите команду, показанную в листинге 20-1, чтобы создать файл модуля.

> **Подсказка**
> Вы можете загрузить пример проекта для этой главы — и для всех остальных глав этой книги — с `https://github.com/apress/pro-go`. См. Главу 2 о том, как получить помощь, если у вас возникнут проблемы с запуском примеров.

```
go mod init readersandwriters
```

Листинг 20-1 Инициализация модуля

Добавьте файл с именем `printer.go` в папку `readerandwriters` с содержимым, показанным в листинге 20-2.

```
package main

import "fmt"

func Printfln(template string, values ...interface{}) {
    fmt.Printf(template + "\n", values...)
```

```
}
```

Листинг 20-2 Содержимое файла printer.go в папке readerandwriters

Добавьте файл с именем `product.go` в папку `readerandwriters` с содержимым, показанным в листинге 20-3.

```
package main

type Product struct {
    Name, Category string
    Price float64
}

var Kayak = Product {
    Name: "Kayak",
    Category: "Watersports",
    Price: 279,
}

var Products = []Product {
    { "Kayak", "Watersports", 279 },
    { "Lifejacket", "Watersports", 49.95 },
    { "Soccer Ball", "Soccer", 19.50 },
    { "Corner Flags", "Soccer", 34.95 },
    { "Stadium", "Soccer", 79500 },
    { "Thinking Cap", "Chess", 16 },
    { "Unsteady Chair", "Chess", 75 },
    { "Bling-Bling King", "Chess", 1200 },
}
```

Листинг 20-3 Содержимое файла product.go в папке readerandwriters

Добавьте файл с именем `main.go` в папку `readerandwriters` с содержимым, показанным в листинге 20-4.

```
package main

func main() {

        Printfln("Product: %v, Price : %v", Kayak.Name,
Kayak.Price)
}
```

Листинг 20-4 Содержимое файла main.go в папке readerandwriters

Используйте командную строку для запуска команды, показанной в листинге 20-5, в папке `readerandwriters`.

```
go run .
```

Листинг 20-5 Запуск примера проекта

Код будет скомпилирован и выполнен, что приведет к следующему выводу:

```
Product: Kayak Price: 275
```

Понимание средств чтения и записи

Интерфейсы `Reader` и `Writer` определяются пакетом `io` и предоставляют абстрактные способы чтения и записи данных без привязки к тому, откуда данные поступают или куда направляются. В следующих разделах я опишу эти интерфейсы и продемонстрирую их использование.

Понимание средств чтения

Интерфейс `Reader` определяет единственный метод, описанный в таблице 20-3.

Таблица 20-3 Reader интерфейс

Функция	Описание
`Read(byteSlice)`	Этот метод считывает данные в указанный `[]byte`. Метод возвращает количество прочитанных байтов, выраженное как `int`, и `error`.

Интерфейс `Reader` не содержит подробностей о том, откуда берутся данные или как они получены — он просто определяет метод `Read`. Детали оставлены на усмотрение типов, реализующих интерфейс, а в стандартной библиотеке есть реализации считывателей для разных источников данных. Один из самых простых считывателей использует `string` в качестве источника данных и показан в листинге 20-6.

```
package main
```

```
import (
    "io"
    "strings"
)

func processData(reader io.Reader) {
    b := make([]byte, 2)
    for {
        count, err := reader.Read(b);
        if (count > 0) {
                            Printfln("Read  %v  bytes:  %v",  count,
string(b[0:count]))
        }
        if err == io.EOF {
            break
        }
    }
}

func main() {
    r := strings.NewReader("Kayak")
    processData(r)
}
```

Листинг 20-6 Использование Reader в файле main.go в папке readerandwriters

Каждый тип Reader создается по-своему, как я продемонстрирую позже в этой и последующих главах. Чтобы создать средство чтения на основе строки, пакет strings предоставляет функцию-конструктор NewReader, которая принимает строку в качестве аргумента:

```
...
r := strings.NewReader("Kayak")
...
```

Чтобы подчеркнуть использование интерфейса, я использую результат функции NewReader в качестве аргумента функции, которая принимает io.Reader. Внутри функции я использую метод Read для чтения байтов данных. Я указываю максимальное количество байтов, которое я хочу получить, устанавливая размер байтового среза, который передается функции Read. Результаты функции Read

показывают, сколько байтов данных было прочитано и произошла ли ошибка.

Пакет io определяет специальную ошибку с именем EOF, которая используется для сигнализации о том, что Reader достигает конца данных. Если результат error функции Read равен ошибке EOF, то я выхожу из цикла for, который считывал данные из Reader:

```
...
if err == io.EOF {
    break
}
...
```

Эффект заключается в том, что цикл for вызывает функцию Read, чтобы получить максимум два байта за раз, и записывает их. При достижении конца строки функция Read возвращает ошибку EOF, что приводит к завершению цикла for. Скомпилируйте и выполните код, и вы получите следующий вывод:

```
Read 2 bytes: Ka
Read 2 bytes: ya
Read 1 bytes: k
```

Понимание средств записи

Интерфейс Writer определяет метод, описанный в таблице 20-4. The Writer interface defines the method described in Table 20-4.

Таблица 20-4 Интерфейс Writer

Функция	Описание
Write(byteSlice)	Этот метод записывает данные из указанного byte среза. Метод возвращает количество записанных байтов и error. Ошибка будет ненулевой, если количество записанных байтов меньше длины среза.

Интерфейс Writer не содержит никаких подробностей о том, как записанные данные хранятся, передаются или обрабатываются, и все это остается на усмотрение типов, реализующих интерфейс. В листинге 20-7 я создал Writer, который создает строку с полученными данными.

```
package main

import (
    "io"
    "strings"
)

func processData(reader io.Reader, writer io.Writer) {
    b := make([]byte, 2)
    for {
        count, err := reader.Read(b);
        if (count > 0) {
            writer.Write(b[0:count])
                      Printfln("Read  %v  bytes:  %v",  count,
string(b[0:count]))
        }
        if err == io.EOF {
            break
        }
    }
}

func main() {
    r := strings.NewReader("Kayak")
    var builder strings.Builder
    processData(r, &builder)
    Printfln("String builder contents: %s", builder.String())
}
```

Листинг 20-7 Использование Writer в файле main.go в папке readerandwriters

Структура `strings.Builder`, описанная в главе 16, реализует интерфейс `io.Writer`, что означает, что я могу записывать байты в `Builder`, а затем вызывать его метод `String` для создания строки из этих байтов.

Модули записи вернут `error`, если не смогут записать все данные в срез. В листинге 20-7 я проверяю результат ошибки и прерываю (`break`) цикл `for`, если возвращается ошибка. Однако, поскольку модуль `Writer` в этом примере строит строку в памяти, вероятность возникновения ошибки мала.

Обратите внимание, что я использую оператор адреса для передачи указателя на `Builder` в функцию `processData`, например:

```
...
processData(r, &builder)
...
```

Как правило, методы Reader и Writer реализуются для указателей, поэтому передача Reader или Writer в функцию не создает копию. Мне не пришлось использовать оператор адреса для Reader в листинге 20-7, потому что результатом функции strings.NewReader является указатель.

Скомпилируйте и выполните проект, и вы получите следующий вывод, показывающий, что байты были прочитаны из одной строки и использованы для создания другой:

```
Read 2 bytes: Ka
Read 2 bytes: ya
Read 1 bytes: k
String builder contents: Kayak
```

Использование служебных функций для программ чтения и записи

Пакет io содержит набор функций, обеспечивающих дополнительные способы чтения и записи данных, как описано в таблице 20-5.

Таблица 20-5 Функции пакета io для чтения и записи данных

Функция	Описание
Copy(w, r)	Эта функция копирует данные из Reader в Writer до тех пор, пока не будет возвращен EOF или не будет обнаружена другая ошибка. Результатом является количество копий байтов и error, используемая для описания любых проблем.
CopyBuffer(w, r, buffer)	Эта функция выполняет ту же задачу, что и Copy, но считывает данные в указанный буфер перед их передачей во Writer.
CopyN(w, r, count)	Эта функция копирует count байтов из Reader в Writer. Результатом является количество копий байтов и error, используемая для описания любых проблем.
ReadAll(r)	Эта функция считывает данные из указанного Reader до тех пор, пока не будет достигнут EOF. Результатом является байтовый срез, содержащий считанные данные и error, которая используется для описания любых проблем.

Функция	Описание
ReadAtLeast(r, byteSlice, min)	Эта функция считывает как минимум указанное количество байтов из устройства чтения, помещая их в байтовый срез. Сообщается об ошибке, если считано меньше байтов, чем указано.
ReadFull(r, byteSlice)	Эта функция заполняет указанный байтовый срез данными. Результатом является количество прочитанных байтов и error. Будет сообщено об ошибке, если EOF будет обнаружен до того, как будет прочитано достаточно байтов для заполнения среза.
WriteString(w, str)	Эта функция записывает указанную строку в модуль записи.

Функции в таблице 20-5 используют методы Read и Write, определенные интерфейсами Reader и Writer, но делают это более удобными способами, избегая необходимости определять цикл for всякий раз, когда вам нужно обработать данные. В листинге 20-8 я использовал функцию Copy для копирования байтов строки примера из Reader и Writer.

```
package main

import (
    "io"
    "strings"
)

func processData(reader io.Reader, writer io.Writer) {
    count, err := io.Copy(writer, reader)
    if (err == nil) {
        Printfln("Read %v bytes", count)
    } else {
        Printfln("Error: %v", err.Error())
    }
}

func main() {
    r := strings.NewReader("Kayak")
    var builder strings.Builder
    processData(r, &builder)
    Printfln("String builder contents: %s", builder.String())
}
```

Листинг 20-8 Копирование данных из файла main.go в папку readerandwriters

Использование функции Copy дает тот же результат, что и в предыдущем примере, но более лаконично. Скомпилируйте и выполните код, и вы получите следующий вывод:

```
Read 5 bytes
String builder contents: Kayak
```

Использование специализированных средств чтения и записи

В дополнение к базовым интерфейсам Reader и Writer пакет io предоставляет несколько специализированных реализаций, описанных в таблице 20-6 и продемонстрированных в следующих разделах.

Таблица 20-6 Функции пакета io для специализированных средств чтения и записи

Функция	Описание
Pipe()	Эта функция возвращает PipeReader и PipeWriter, которые можно использовать для соединения функций, требующих Reader и Writer, как описано в разделе «Использование каналов».
MultiReader(...readers)	Эта функция определяет переменный параметр, который позволяет указать произвольное количество значений Reader. В результате получается Reader, которое передает содержимое каждого из своих параметров в той последовательности, в которой они определены, как описано в разделе «Объединение нескольких средств чтения».
MultiWriter(...writers)	Эта функция определяет переменный параметр, который позволяет указать произвольное количество значений Writer. Результатом является Writer, который отправляет одни и те же данные всем указанным модулям записи, как описано в разделе «Объединение нескольких средств записи».
LimitReader(r, limit)	Эта функция создает Reader, который будет завершать работу после указанного количества байтов, как описано в разделе «Ограничение чтения данных».

Использование пайпов

Каналы используются для соединения кода, потребляющего данные через Reader, и кода, создающего код через Writer. Добавьте файл с именем data.go в папку readerandwriters с содержимым, показанным в листинге 20-9.

```
package main

import "io"

func GenerateData(writer io.Writer) {
    data := []byte("Kayak, Lifejacket")
    writeSize := 4
    for i := 0; i < len(data); i += writeSize {
        end := i + writeSize;
        if (end > len(data)) {
            end = len(data)
        }
        count, err := writer.Write(data[i: end])
                 Printfln("Wrote  %v  byte(s):  %v",  count,
string(data[i: end]))
        if (err != nil)  {
            Printfln("Error: %v", err.Error())
        }
    }
}

func ConsumeData(reader io.Reader) {
    data := make([]byte, 0, 10)
    slice := make([]byte, 2)
    for {
        count, err := reader.Read(slice)
        if (count > 0) {
            Printfln("Read data: %v", string(slice[0:count]))
            data = append(data, slice[0:count]...)
        }
        if (err == io.EOF) {
            break
        }
    }
    Printfln("Read data: %v", string(data))
}
```

Листинг 20-9 Содержимое файла data.go в папке readerandwriters

Функция `GenerateData` определяет параметр `Writer`, который используется для записи байтов из строки. Функция `ConsumeData` определяет параметр `Reader`, который используется для чтения байтов данных, которые затем используются для создания строки.

Реальным проектам не нужно считывать байты из одной строки только для того, чтобы создать другую, но это обеспечивает хорошую демонстрацию работы каналов, как показано в листинге 20-10.

```
package main

import (
    "io"
    //"strings"
)

// func processData(reader io.Reader, writer io.Writer) {
//     count, err := io.Copy(writer, reader)
//     if (err == nil) {
//         Printfln("Read %v bytes", count)
//     } else {
//         Printfln("Error: %v", err.Error())
//     }
// }

func main() {
    pipeReader, pipeWriter := io.Pipe()
    go func() {
        GenerateData(pipeWriter)
        pipeWriter.Close()
    }()
    ConsumeData(pipeReader)
}
```

Листинг 20-10 Использование каналов в файле main.go в папке readerandwriters

Функция `io.Pipe` возвращает `PipeReader` и `PipeWriter`. Структуры `PipeReader` и `PipeWriter` реализуют интерфейс `Closer`, который определяет метод, показанный в таблице 20-7.

Таблица 20-7 Closer метод

Функция	Описание
`Close()`	Этот метод закрывает средство чтения или записи. Детали зависят от реализации, но, как правило, любые последующие операции чтения из закрытого `Reader` будут возвращать нулевые байты и ошибку `EOF`, в то время как любые последующие записи в закрытый `Writer` будут возвращать ошибку.

Поскольку PipeWriter реализует интерфейс Writer, я могу использовать его в качестве аргумента функции GenerateData, а затем вызвать метод Close после завершения функции, чтобы считыватель получил EOF, например:

```
...
GenerateData(pipeWriter)
pipeWriter.Close()
...
```

Каналы являются синхронными, поэтому метод PipeWriter.Write будет блокироваться до тех пор, пока данные не будут прочитаны из канала. Это означает, что PipeWriter необходимо использовать в другой горутине, отличной от программы чтения, чтобы предотвратить взаимоблокировку приложения:

```
...
go func() {
    GenerateData(pipeWriter)
    pipeWriter.Close()
}()
...
```

Обратите внимание на круглые скобки в конце этого утверждения. Они необходимы при создании горутины для анонимной функции, но их легко забыть.

Структура PipeReader реализует интерфейс Reader, что означает, что я могу использовать ее в качестве аргумента функции ConsumeData. Функция ConsumeData выполняется в main горутине, а это означает, что приложение не завершится, пока функция не завершится.

В результате данные записываются в канал с помощью PipeWriter и считываются из канала с помощью PipeReader. Когда функция GenerateData завершена, метод Close вызывается в PipeWriter, что приводит к следующему чтению PipeReader для создания EOF. Скомпилируйте и запустите проект, и вы получите следующий вывод:

```
Read data: Ka
Wrote 4 byte(s): Kaya
```

```
Read data: ya
Read data: k,
Wrote 4 byte(s): k, L
Read data:  L
Read data: if
Wrote 4 byte(s): ifej
Read data: ej
Read data: ac
Wrote 4 byte(s): acke
Read data: ke
Wrote 1 byte(s): t
Read data: t
Read data: Kayak, Lifejacket
```

Вывод подчеркивает тот факт, что каналы синхронны. Функция `GenerateData` вызывает метод `Write` модуля записи и затем блокируется до тех пор, пока данные не будут прочитаны. Вот почему первое сообщение в выводе исходит от считывателя: считыватель потребляет данные по два байта за раз, а это означает это означает, что требуются две операции чтения перед первоначальным вызовом метода `Write`, который используется для отправки четырех байтов, завершается, и отображается сообщение от функции `GenerateData`.

Улучшение примера

В листинге 20-10 я вызвал метод `Close` для `PipeWriter` в горутине, которая выполняет функцию `GenerateData`. Это работает, но я предпочитаю проверять, реализует ли `Writer` интерфейс `Closer` в коде, производящем данные, как показано в листинге 20-11.

```
...
func GenerateData(writer io.Writer) {
    data := []byte("Kayak, Lifejacket")
    writeSize := 4
    for i := 0; i < len(data); i += writeSize {
        end := i + writeSize;
        if (end > len(data)) {
            end = len(data)
        }
        count, err := writer.Write(data[i: end])
                 Printfln("Wrote  %v  byte(s):  %v",  count,
string(data[i: end]))
```

```
        if (err != nil)  {
            Println("Error: %v", err.Error())
        }
    }
    if closer, ok := writer.(io.Closer); ok {
        closer.Close()
    }
}
...
```

Листинг 20-11 Закрытие Writer в файле data.go в папке readerandwriters

Этот подход предоставляет согласованные обработчики `Writer`, определяющие метод `Close`, который включает в себя некоторые из наиболее полезных типов, описанных в следующих главах. Это также позволяет мне изменить горутину так, чтобы она выполняла функцию `GenerateData` без необходимости использования анонимной функции, как показано в листинге 20-12.

```
package main

import (
    "io"
    //"strings"
)

func main() {
    pipeReader, pipeWriter := io.Pipe()
    go GenerateData(pipeWriter)
    ConsumeData(pipeReader)
}
```

Листинг 20-12 Упрощение кода в файле main.go в папке readerandwriters

Этот пример дает тот же результат, что и код в листинге 20-10.

Объединение нескольких средств чтения

Функция `MultiReader` концентрирует входные данные от нескольких считывателей, чтобы их можно было обрабатывать последовательно, как показано в листинге 20-13.

```
package main
```

```
import (
    "io"
    "strings"
)

func main() {

    r1 := strings.NewReader("Kayak")
    r2 := strings.NewReader("Lifejacket")
    r3 := strings.NewReader("Canoe")

    concatReader := io.MultiReader(r1, r2, r3)

    ConsumeData(concatReader)
}
```

Листинг 20-13 Объединение Readers в файле main.go в папке readerandwriters

Reader, возвращаемый функцией MultiReader, отвечает на метод Read содержимым из одного из базовых значений Reader. Когда первый Reader возвращает EOF, содержимое считывается со второго Reader. Этот процесс продолжается до тех пор, пока последний базовый Reader не вернет EOF. Скомпилируйте и выполните код, и вы увидите следующий вывод:

```
Read data: Ka
Read data: ya
Read data: k
Read data: Li
Read data: fe
Read data: ja
Read data: ck
Read data: et
Read data: Ca
Read data: no
Read data: e
Read data: KayakLifejacketCanoe
```

Объединение нескольких средств записи

Функция MultiWriter объединяет несколько модулей записи, чтобы данные отправлялись всем им, как показано в листинге 20-14.

```
package main

import (
    "io"
    "strings"
)

func main() {

    var w1 strings.Builder
    var w2 strings.Builder
    var w3 strings.Builder

    combinedWriter := io.MultiWriter(&w1, &w2, &w3)

    GenerateData(combinedWriter)

    Printfln("Writer #1: %v", w1.String())
    Printfln("Writer #2: %v", w2.String())
    Printfln("Writer #3: %v", w3.String())
}
```

Листинг 20-14 Объединение писателей в файле main.go в папке readerandwriters

Средства записи в этом примере — это значения `string.Builder`, описанные в главе 16 и реализующие интерфейс `Writer`. Функция `MultiWriter` используется для создания модуля записи, поэтому вызов метода `Write` приведет к записи одних и тех же данных в три отдельных модуля записи. Скомпилируйте и запустите проект, и вы увидите следующий вывод:

```
Wrote 4 byte(s): Kaya
Wrote 4 byte(s): k, L
Wrote 4 byte(s): ifej
Wrote 4 byte(s): acke
Wrote 1 byte(s): t
Writer #1: Kayak, Lifejacket
Writer #2: Kayak, Lifejacket
Writer #3: Kayak, Lifejacket
```

Повторение данных чтения во Writer

Функция TeeReader возвращает Reader, который повторяет полученные данные в Writer, как показано в листинге 20-15.

```
package main

import (
    "io"
    "strings"
)

func main() {

    r1 := strings.NewReader("Kayak")
    r2 := strings.NewReader("Lifejacket")
    r3 := strings.NewReader("Canoe")

    concatReader := io.MultiReader(r1, r2, r3)

    var writer strings.Builder
    teeReader := io.TeeReader(concatReader, &writer);

    ConsumeData(teeReader)
    Printfln("Echo data: %v", writer.String())
}
```

Листинг 20-15 Эхо данных в файле main.go в папке readerandwriters

Функция TeeReader используется для создания Reader, который будет повторять данные в strings.Builder, описанный в главе 16 и реализующий интерфейс Writer. Скомпилируйте и выполните проект, и вы увидите следующий вывод, включающий эхо-данные:

```
Read data: Ka
Read data: ya
Read data: k
Read data: Li
Read data: fe
Read data: ja
Read data: ck
Read data: et
Read data: Ca
Read data: no
```

```
Read data: e
Read data: KayakLifejacketCanoe
Echo data: KayakLifejacketCanoe
```

Ограничение чтения данных

Функция LimitReader используется для ограничения количества данных, которые могут быть получены от Reader, как показано в листинге 20-16.

```
package main

import (
    "io"
    "strings"
)

func main() {

    r1 := strings.NewReader("Kayak")
    r2 := strings.NewReader("Lifejacket")
    r3 := strings.NewReader("Canoe")

    concatReader := io.MultiReader(r1, r2, r3)

    limited := io.LimitReader(concatReader, 5)
    ConsumeData(limited)
}
```

Листинг 20-16 Ограничение данных в файле main.go в папке readerandwriters

Первым аргументом функции LimitReader является Reader, который будет предоставлять данные. Второй аргумент — это максимальное количество байтов, которые можно прочитать. Reader, возвращаемый функцией LimitReader, отправит EOF при достижении предела, если базовый считыватель не отправит EOF первым. В листинге 20-16 я установил ограничение в 5 байтов, что дает следующий вывод, когда проект компилируется и выполняется:

```
Read data: Ka
Read data: ya
Read data: k
```

```
Read data: Kayak
```

Буферизация данных

Пакет bufio обеспечивает поддержку добавления буферов для чтения и записи. Чтобы увидеть, как данные обрабатываются без буфера, добавьте файл с именем custom.go в папку readerandwriters с содержимым, показанным в листинге 20-17.

```
package main

import "io"

type CustomReader struct {
    reader io.Reader
    readCount int
}

func NewCustomReader(reader io.Reader) *CustomReader {
    return &CustomReader { reader, 0 }
}

func (cr *CustomReader) Read(slice []byte) (count int, err error) {
    count, err = cr.reader.Read(slice)
    cr.readCount++
    Printfln("Custom Reader: %v bytes", count)
    if (err == io.EOF) {
        Printfln("Total Reads: %v", cr.readCount)
    }
    return
}
```

Листинг 20-17 Содержимое файла custom.go в папке readerandwriters

Код в листинге 20-17 определяет тип структуры с именем CustomReader, который действует как оболочка для Reader. Реализация метода Read генерирует выходные данные, сообщающие, сколько данных считано и сколько операций чтения выполнено в целом. В листинге 20-18 новый тип используется в качестве оболочки строкового Reader.

```
package main

import (
    "io"
    "strings"
)

func main() {

    text := "It was a boat. A small boat."

                          var      reader      io.Reader      =
NewCustomReader(strings.NewReader(text))
    var writer strings.Builder
    slice := make([]byte, 5)

    for {
        count, err := reader.Read(slice)
        if (count > 0) {
            writer.Write(slice[0:count])
        }
        if (err != nil) {
            break
        }
    }

    Printfln("Read data: %v", writer.String())
}
```

Листинг 20-18 Использование Reader Wrapper в файле main.go в папке readerandwriters

Функция `NewCustomreader` используется для создания `CustomReader`, который считывает из строки и использует цикл `for` для получения данных с использованием среза байтов. Скомпилируйте и запустите проект, и вы увидите, как читаются данные:

```
Custom Reader: 5 bytes
Custom Reader: 5 bytes
Custom Reader: 5 bytes
Custom Reader: 5 bytes
Custom Reader: 5 bytes
Custom Reader: 3 bytes
Custom Reader: 0 bytes
```

```
Total Reads: 7
Read data: It was a boat. A small boat.
```

Именно размер байтового среза, передаваемого функции Read, определяет, как потребляются данные. В этом случае размер среза равен пяти, что означает, что при каждом вызове функции чтения считывается не более пяти байтов. Есть два чтения, которые не получили 5 байтов данных. Предпоследнее чтение произвело три байта, потому что исходные данные не делятся точно на пять, и осталось три байта данных. Окончательное чтение вернуло нулевые байты, но получило ошибку EOF, указывающую, что был достигнут конец данных.

Всего для чтения 28 байт потребовалось 7 операций чтения. (Я выбрал исходные данные таким образом, чтобы все символы в строке требовали одного байта, но вы можете увидеть другое количество чтений, если вы измените пример, чтобы ввести символы, для которых требуется несколько байтов.)

Чтение небольших объемов данных может быть проблематичным, если с каждой операцией связаны большие накладные расходы. Это не проблема при чтении строки, хранящейся в памяти, но чтение данных из других источников данных, таких как файлы, может быть более затратным, и может быть предпочтительнее выполнять меньшее количество больших чтений. Это делается путем введения буфера, в который считывается большой объем данных для обслуживания нескольких меньших запросов данных. В таблице 20-8 описаны функции, предоставляемые пакетом bufio, которые создают буферизованные считыватели.

Таблица 20-8 Функции bufio для создания буферизованных ридеров

Функция	**Описание**
NewReader(r)	Эта функция возвращает буферизованный Reader с размером буфера по умолчанию (который на момент написания составляет 4096 байт).
NewReaderSize(r, size)	Эта функция возвращает буферизованный Reader с указанным размером буфера.

Результаты, полученные NewReader и NewReaderSize, реализуют интерфейс Reader, но вводят буфер, который может уменьшить

количество операций чтения, выполняемых для базового источника данных. Листинг 20-19 демонстрирует введение в пример буфера.

```
package main

import (
    "io"
    "strings"
    "bufio"
)

func main() {

    text := "It was a boat. A small boat."

    var reader io.Reader = NewCustomReader(strings.NewReader(text))
    var writer strings.Builder
    slice := make([]byte, 5)

    reader = bufio.NewReader(reader)

    for {
        count, err := reader.Read(slice)
        if (count > 0) {
            writer.Write(slice[0:count])
        }
        if (err != nil) {
            break
        }
    }

    Printfln("Read data: %v", writer.String())
}
```

Листинг 20-19 Использование буфера в файле main.go в папке readerandwriters

Я использовал функцию `NewReader`, которая создает `Reader` с размером буфера по умолчанию. Буферизованный `Reader` заполняет свой буфер и использует содержащиеся в нем данные для ответа на вызовы метода `Read`. Скомпилируйте и выполните проект, чтобы увидеть эффект от введения буфера:

```
Custom Reader: 28 bytes
Custom Reader: 0 bytes
Total Reads: 2
Read data: It was a boat. A small boat.
```

Размер буфера по умолчанию составляет 4096 байт, что означает, что буферизованный считыватель смог прочитать все данные за одну операцию чтения, а также дополнительное чтение для получения результата EOF. Введение буфера снижает накладные расходы, связанные с операциями чтения, хотя и за счет памяти, используемой для буферизации данных.

Использование дополнительных методов буферизованного чтения

Функции NewReader и NewReaderSize возвращают значения bufio.Reader, которые реализуют интерфейс io.Reader и могут использоваться в качестве вставных оболочек для других типов методов Reader, органично вводя буфер чтения.

Структура bufio.Reader определяет дополнительные методы, напрямую использующие буфер, как описано в таблице 20-9.

Таблица 20-9 Методы, определенные буферизованным считывателем

Функция	Описание
Buffered()	Этот метод возвращает int число, указывающее количество байтов, которые можно прочитать из буфера.
Discard(count)	Этот метод отбрасывает указанное количество байтов.
Peek(count)	Этот метод возвращает указанное количество байтов, не удаляя их из буфера, то есть они будут возвращены последующими вызовами метода Read.
Reset(reader)	Этот метод отбрасывает данные в буфере и выполняет последующие чтения из указанного Reader.
Size()	Этот метод возвращает размер буфера, выраженный int.

В листинге 20-20 показано использование методов Size и Buffered для сообщения размера буфера и количества содержащихся в нем данных.

```
package main
```

```
import (
    "io"
    "strings"
    "bufio"
)

func main() {

    text := "It was a boat. A small boat."

                        var     reader      io.Reader      =
NewCustomReader(strings.NewReader(text))
    var writer strings.Builder
    slice := make([]byte, 5)

    buffered := bufio.NewReader(reader)

    for {
        count, err := buffered.Read(slice)
        if (count > 0) {
            Printfln("Buffer size: %v, buffered: %v",
                buffered.Size(), buffered.Buffered())
            writer.Write(slice[0:count])
        }
        if (err != nil) {
            break
        }
    }

    Printfln("Read data: %v", writer.String())
}
```

Листинг 20-20 Работа с буфером в файле main.go в папке readerandwriters

Скомпилируйте и выполните проект, и вы увидите, что каждая операция чтения потребляет часть буферизованных данных:

```
Custom Reader: 28 bytes
Buffer size: 4096, buffered: 23
Buffer size: 4096, buffered: 18
Buffer size: 4096, buffered: 13
Buffer size: 4096, buffered: 8
Buffer size: 4096, buffered: 3
```

```
Buffer size: 4096, buffered: 0
Custom Reader: 0 bytes
Total Reads: 2
Read data: It was a boat. A small boat.
```

Выполнение буферизованной записи

Пакет bufio также поддерживает создание модулей записи, использующих буфер, с помощью функций, описанных в таблице 20-10.

Таблица 20-10 Функции bufio для создания буферизованных модулей записи

Функция	Описание
NewWriter(w)	Эта функция возвращает буферизованный Writer с размером буфера по умолчанию (который на момент записи составляет 4096 байт).
NewWriterSize(w, size)	Эта функция возвращает буферизованный Writer с указанным размером буфера.

Результаты, полученные функциями, описанными в таблице 20-10, реализуют интерфейс Writer, что означает, что их можно использовать для беспрепятственного введения буфера для записи. Конкретный тип данных, возвращаемый этими функциями, — bufio.Writer, который определяет методы, описанные в таблице 20-11, для управления буфером и его содержимым.

Таблица 20-11 Методы, определяемые структурой bufio.Writer

Функция	Описание
Available()	Этот метод возвращает количество доступных байтов в буфере.
Buffered()	Этот метод возвращает количество байтов, записанных в буфер.
Flush()	Этот метод записывает содержимое буфера в базовый Writer.
Reset(writer)	Этот метод отбрасывает данные в буфере и выполняет последующую запись в указанный Writer.
Size()	Этот метод возвращает емкость буфера в байтах.

В листинге 20-21 определяется пользовательский Writer, который сообщает о своих операциях и показывает влияние буфера. Это аналог Reader, созданного в предыдущем разделе.

```
package main

import "io"

// ...reader type and functions omitted for brevity...

type CustomWriter struct {
    writer io.Writer
    writeCount int
}

func NewCustomWriter(writer io.Writer) * CustomWriter {
    return &CustomWriter{ writer, 0}
}

func (cw *CustomWriter) Write(slice []byte) (count int, err
error) {
    count, err = cw.writer.Write(slice)
    cw.writeCount++
    Printfln("Custom Writer: %v bytes", count)
    return
}

func (cw *CustomWriter) Close() (err error) {
    if closer, ok := cw.writer.(io.Closer); ok {
        closer.Close()
    }
    Printfln("Total Writes: %v", cw.writeCount)
    return
}
```

Листинг 20-21 Определение пользовательского модуля записи в файле custom.go в папке readerandwriters

Конструктор `NewCustomWriter` заключает в `Writer` структуру `CustomWriter`, которая сообщает о его операциях записи. В листинге 20-22 показано, как операции записи выполняются без буферизации.

```
package main

import (
    //"io"
    "strings"
```

```
    //"bufio"
)

func main() {

    text := "It was a boat. A small boat."

    var builder strings.Builder
    var writer = NewCustomWriter(&builder)
    for i := 0; true; {
        end := i + 5
        if (end >= len(text)) {
            writer.Write([]byte(text[i:]))
            break
        }
        writer.Write([]byte(text[i:end]))
        i = end
    }
    Printfln("Written data: %v", builder.String())
}
```

Листинг 20-22 Выполнение небуферизованной записи в файле main.go в папке readerandwriters

В примере за один раз записывается по пять байтов в `Writer`, который поддерживается `Builder` из пакета `strings`. Скомпилируйте и выполните проект, и вы увидите эффект от каждого вызова метода `Write`:

```
Custom Writer: 5 bytes
Custom Writer: 5 bytes
Custom Writer: 5 bytes
Custom Writer: 5 bytes
Custom Writer: 5 bytes
Custom Writer: 3 bytes
Written data: It was a boat. A small boat.
```

Буферизованный `Writer` хранит данные в буфере и передает их базовому `Writer` только тогда, когда буфер заполнен или когда вызывается метод `Flush`. В листинге 20-23 в пример вводится буфер.

```
package main
```

```
import (
    //"io"
    "strings"
    "bufio"
)

func main() {

    text := "It was a boat. A small boat."

    var builder strings.Builder
                                  var      writer       =
bufio.NewWriterSize(NewCustomWriter(&builder), 20)
    for i := 0; true; {
        end := i + 5
        if (end >= len(text)) {
            writer.Write([]byte(text[i:]))
            writer.Flush()
            break
        }
        writer.Write([]byte(text[i:end]))
        i = end
    }
    Printfln("Written data: %v", builder.String())
}
```

Листинг 20-23 Использование буферизованного модуля записи в файле main.go в папке readerandwriters

Переход к буферизованному `Writer` не совсем плавный, потому что важно вызвать метод `Flush`, чтобы убедиться, что все данные записаны. Буфер, который я выбрал в листинге 20-23, имеет размер 20 байт, что намного меньше, чем буфер по умолчанию, и слишком мало, чтобы иметь эффект в реальных проектах, но он идеально подходит для демонстрации того, как введение буфера снижает количество операций записи. операции в примере. Скомпилируйте и запустите проект, и вы увидите следующий вывод:

```
Custom Writer: 20 bytes
Custom Writer: 8 bytes
Written data: It was a boat. A small boat.
```

Форматирование и сканирование с помощью средств чтения и записи

В главе 17 я описал возможности форматирования и сканирования, предоставляемые пакетом fmt, и продемонстрировал их использование со строками. Как я уже отмечал в этой главе, пакет fmt поддерживает применение этих функций к модулям чтения и записи, как описано в следующих разделах. Я также описываю, как функции из пакета strings можно использовать с Writer.

Сканирование значений из считывателя

Пакет fmt предоставляет функции для сканирования значений из Reader и преобразования их в различные типы, как показано в листинге 20-24. (Использование функции для сканирования значений не является обязательным требованием, и я сделал это только для того, чтобы подчеркнуть, что процесс сканирования работает на любом устройстве Reader.)

```
package main

import (
    "io"
    "strings"
    //"bufio"
    "fmt"
)

func scanFromReader(reader io.Reader, template string,
        vals ...interface{}) (int, error) {
    return fmt.Fscanf(reader, template, vals...)
}

func main() {

    reader := strings.NewReader("Kayak Watersports $279.00")

    var name, category string
    var price float64
    scanTemplate := "%s %s $%f"
```

```
    _, err := scanFromReader(reader, scanTemplate, &name,
&category, &price)
    if (err != nil) {
        Printfln("Error: %v", err.Error())
    } else {
        Printfln("Name: %v", name)
        Printfln("Category: %v", category)
        Printfln("Price: %.2f", price)
    }
}
```

Листинг 20-24 Сканирование с устройства чтения в файле main.go в папке readerandwriters

Процесс сканирования считывает байты из `Reader` и использует шаблон сканирования для анализа полученных данных. Шаблон сканирования в листинге 20-24 содержит две строки и значение `float64`, а компиляция и выполнение кода приводит к следующему результату:

```
Name: Kayak
Category: Watersports
Price: 279.00
```

Полезным приемом при использовании `Reader` является постепенное сканирование данных с использованием цикла, как показано в листинге 20-25. Этот подход хорошо работает, когда байты поступают с течением времени, например, при чтении из HTTP-соединения (которое я описываю в главе 25).

```
package main

import (
    "io"
    "strings"
    //"bufio"
    "fmt"
)

func scanFromReader(reader io.Reader, template string,
        vals ...interface{}) (int, error) {
    return fmt.Fscanf(reader, template, vals...)
}
```

```
func  scanSingle(reader  io.Reader,  val  interface{})  (int,
error) {
    return fmt.Fscan(reader, val)
}

func main() {

    reader := strings.NewReader("Kayak Watersports $279.00")

    for {
        var str string
        _, err := scanSingle(reader, &str)
        if (err != nil) {
            if (err != io.EOF) {
                Printfln("Error: %v", err.Error())
            }
            break
        }
        Printfln("Value: %v", str)
    }
}
```

Листинг 20-25 Постепенное сканирование в файле main.go в папке readerandwriters

Цикл for вызывает функцию scanSingle, которая использует функцию Fscan для чтения строки из Reader. Значения считываются до тех пор, пока не будет возвращен EOF, после чего цикл завершается. Скомпилируйте и запустите проект, и вы получите следующий вывод:

```
Value: Kayak
Value: Watersports
Value: $279.00
```

Запись отформатированных строк в Writer

Пакет fmt также предоставляет функции для записи форматированных строк в Writer, как показано в листинге 20-26. (Использование функции для форматирования строк не является обязательным, и я сделал это только для того, чтобы подчеркнуть, что форматирование работает с любым Reader.)

```
package main
```

```
import (
    "io"
    "strings"
    //"bufio"
    "fmt"
)

// func scanFromReader(reader io.Reader, template string,
//         vals ...interface{}) (int, error) {
//     return fmt.Fscanf(reader, template, vals...)
// }

// func scanSingle(reader io.Reader, val interface{}) (int,
error) {
//     return fmt.Fscan(reader, val)
// }

func writeFormatted(writer io.Writer, template string, vals
...interface{}) {
    fmt.Fprintf(writer, template, vals...)
}

func main() {

    var writer strings.Builder
    template := "Name: %s, Category: %s, Price: $%.2f"

    writeFormatted(&writer, template, "Kayak", "Watersports",
float64(279))

    fmt.Println(writer.String())
}
```

Листинг 20-26 Запись форматированной строки в файл main.go в папке readerandwriters

Функция `writeFormatted` использует функцию `fmt.Fprintf` для записи строки, отформатированной с помощью шаблона, в `Writer`. Скомпилируйте и запустите проект, и вы увидите следующий вывод:

```
Name: Kayak, Category: Watersports, Price: $279.00
```

Использование Replacer с Writer

Структуру strings.Replacer можно использовать для замены строки и вывода измененного результата в Writer, как показано в листинге 20-27.

```
package main

import (
    "io"
    "strings"
    //"bufio"
    "fmt"
)

func writeReplaced(writer io.Writer, str string, subs
...string) {
    replacer := strings.NewReplacer(subs...)
    replacer.WriteString(writer, str)
}

func main() {

    text := "It was a boat. A small boat."
    subs := []string { "boat", "kayak", "small", "huge" }

    var writer strings.Builder
    writeReplaced(&writer, text, subs...)
    fmt.Println(writer.String())
}
```

Листинг 20-27 Использование Replacer в файле main.go в папке readerandwriters

Метод WriteString выполняет свои замены и записывает измененную строку. Скомпилируйте и выполните код, и вы получите следующий вывод:

```
It was a kayak. A huge kayak.
```

Резюме

В этой главе я описал интерфейсы Reader и Writer, которые используются во всей стандартной библиотеке везде, где данные

читаются или записываются. Я описываю методы, которые определяют эти интерфейсы, объясняю использование доступных специализированных реализаций и показываю, как достигается буферизация, форматирование и сканирование. В следующей главе я опишу поддержку обработки данных JSON, в которой используются функции, описанные в этой главе.

21. Работа с данными JSON

В этой главе я описываю стандартную библиотеку Go, поддерживающую формат JavaScript Object Notation (JSON). JSON стал стандартом де-факто для представления данных, в основном потому, что он прост и работает на разных платформах. См. `http://json.org` для краткого описания формата данных, если вы раньше не сталкивались с JSON. JSON часто встречается в качестве формата данных, используемого в веб-службах RESTful, которые я продемонстрирую в третьей части. В таблице 21-1 функции JSON представлены в контексте.

Таблица 21-1 Работа с данными JSON в контексте

Вопрос	**Ответ**
Что это?	Данные JSON являются стандартом де-факто для обмена данными, особенно в HTTP-приложениях.
Почему это полезно?	JSON достаточно прост для поддержки любого языка, но может представлять относительно сложные данные.
Как это используется?	Пакет `encoding/json` обеспечивает поддержку кодирования и декодирования данных JSON.
Есть ли подводные камни или ограничения?	Не все типы данных Go могут быть представлены в формате JSON, поэтому разработчик должен помнить о том, как будут выражаться типы данных Go.
Есть ли альтернативы?	Доступно множество других кодировок данных, некоторые из которых поддерживаются стандартной библиотекой Go.

Таблица 21-2 суммирует главу.

Таблица 21-2 Краткое содержание главы

Проблема	**Решение**	**Листинг**
Кодировать данные JSON	Создайте `Encoder` с помощью `Writer` и вызовите метод `Encode`.	2–7, 14, 15
Кодирование управляющей структуры	Используйте теги структуры JSON или реализуйте интерфейс `Marshaler`	8–13, 16
Декодировать данные JSON	Создайте `Decoder` с помощью `Reader` и вызовите метод `Decode`.	17–25
Расшифровка управляющей структуры	Используйте теги структуры JSON или реализуйте интерфейс `Unmarshaler`.	26–28

Подготовка к этой главе

В этой главе я продолжаю использовать проект `readerandwriters`, созданный в главе 20. Для подготовки к этой главе не требуется никаких изменений. Откройте новую командную строку, перейдите в папку `readerandwriters` и выполните команду, показанную в листинге 21-1, чтобы скомпилировать и выполнить проект.

Подсказка
Вы можете загрузить пример проекта для этой главы — и для всех остальных глав этой книги — с `https://github.com/apress/pro-go`. См. главу 2 о том, как получить помощь, если у вас возникнут проблемы с запуском примеров.

```
go run .
```

Листинг 21-1 Запуск примера проекта

Скомпилированный проект выдает следующий результат при выполнении:

```
It was a kayak. A huge kayak.
```

Чтение и запись данных JSON

Пакет `encoding/json` обеспечивает поддержку кодирования и декодирования данных JSON, как показано в следующих разделах. Для справки в таблице 21-3 описаны функции конструктора, которые используются для создания структур для кодирования и декодирования данных JSON и которые подробно описаны далее.

Таблица 21-3 Функции конструктора encoding/json для данных JSON

Функция	Описание
`NewEncoder(writer)`	Эта функция возвращает `Encoder`, который можно использовать для кодирования данных JSON и записи их в указанный `Writer`.
`NewDecoder(reader)`	Эта функция возвращает `Decoder`, который можно использовать для чтения данных JSON из указанного `Reader` и их декодирования.

Примечание
Стандартная библиотека Go включает пакеты для других форматов данных, включая XML и CSV. Подробнее см. `https://golang.org/pkg/encoding`..

Пакет `encoding/json` также предоставляет функции для кодирования и декодирования JSON без использования `Reader` или `Writer`, описанные в таблице 21-4.

Таблица 21-4 Функции для создания и анализа данных JSON

Функция	Описание
`Marshal(value)`	Эта функция кодирует указанное значение как JSON. Результатом является содержимое JSON, выраженное в виде среза байта, и `error`, указывающая на наличие проблем с кодировкой.
`Unmarshal(byteSlice, val)`	Эта функция анализирует данные JSON, содержащиеся в указанном срезе байтов, и присваивает результат указанному значению.

Кодирование данных JSON

Функция-конструктор `NewEncoder` используется для создания `Encoder`, который можно использовать для записи данных JSON в `Writer`, используя методы, описанные в таблице 21-5.

Таблица 21-5 Методы кодировщика

Функция	Описание
`Encode(val)`	Этот метод кодирует указанное значение как JSON и записывает его в `Writer`.
`SetEscapeHTML(on)`	Этот метод принимает `bool` аргумент, который, если он равен `true`, кодирует символы, экранирование которых в HTML было бы опасным. По умолчанию эти символы экранируются.
`SetIndent(prefix, indent)`	Этот метод задает префикс и отступ, которые применяются к имени каждого поля в выходных данных JSON.

На любом языке, кроме JavaScript, типы данных, выраженные JSON, не совпадают в точности с собственными типами данных. В таблице 21-6 показано, как основные типы данных Go представлены в JSON.

Таблица 21-6 Выражение основных типов данных Go в JSON

Тип данных	Описание
`bool`	Значения Go `bool` выражаются как JSON `true` или `false`.
`string`	Строковые значения Go выражаются в виде строк JSON. По умолчанию небезопасные символы HTML экранируются.
`float32`, `float64`	Значения Go с плавающей запятой выражаются в виде чисел JSON.
`int`, `int<size>`	Целочисленные значения Go выражаются в виде чисел JSON.
`uint`, `uint<size>`	Целочисленные значения Go выражаются в виде чисел JSON.
`byte`	Байты Go выражаются в виде чисел JSON.
`rune`	Руны Go выражаются в виде чисел JSON.
`nil`	Значение Go `nil` выражается как `null` значение JSON.
`Pointers`	Кодер JSON следует указателям и кодирует значение в месте расположения указателя.

В листинге 21-2 показан процесс создания кодировщика JSON и кодирования некоторых основных типов Go.

```
package main

import (
    //"io"
    "strings"
    "fmt"
    "encoding/json"
)
```

```
// func writeReplaced(writer io.Writer, str string, subs ...string) {
//     replacer := strings.NewReplacer(subs...)
//     replacer.WriteString(writer, str)
// }

func main() {

    var b bool = true
    var str string = "Hello"
    var fval float64 = 99.99
    var ival int = 200
    var pointer *int = &ival

    var writer strings.Builder
    encoder := json.NewEncoder(&writer)

    for _, val := range []interface{} {b, str, fval, ival, pointer} {
        encoder.Encode(val)
    }

    fmt.Print(writer.String())
}
```

Листинг 21-2 Кодирование данных JSON в файле main.go в папке readerandwriters

В листинге 21-2 определен ряд переменных различных основных типов. Конструктор `NewEncoder` используется для создания `Encoder`, а цикл `for` используется для кодирования каждого значения в виде JSON. Данные записываются в `Builder`, чей метод `String` вызывается для отображения JSON. Скомпилируйте и запустите проект, и вы увидите следующий вывод:

```
true
"Hello"
99.99
200
200
```

Обратите внимание, что я использовал функцию `fmt.Print` для получения вывода в листинге 21-2. `Encoder` JSON добавляет символ новой строки после кодирования каждого значения.

Кодирование массивов и срезов

Срезы и массивы Go кодируются как массивы JSON, за исключением того, что срезы байтов выражаются в виде строк в кодировке base64. Однако байтовые массивы кодируются как массив чисел JSON. В листинге 21-3 показана поддержка массивов и срезов, включая байты.

```
package main
```

```
import (
    "strings"
    "fmt"
    "encoding/json"
)

func main() {

    names := []string {"Kayak", "Lifejacket", "Soccer Ball"}
    numbers := [3]int { 10, 20, 30}
    var byteArray [5]byte
    copy(byteArray[0:], []byte(names[0]))
    byteSlice := []byte(names[0])

    var writer strings.Builder
    encoder := json.NewEncoder(&writer)

    encoder.Encode(names)
    encoder.Encode(numbers)
    encoder.Encode(byteArray)
    encoder.Encode(byteSlice)

    fmt.Print(writer.String())
}
```

Листинг 21-3 Кодирование срезов и массивов в файле main.go в папке readerandwriters

Encoder выражает каждый массив в синтаксисе JSON, за исключением среза байтов. Скомпилируйте и запустите проект, и вы увидите следующий вывод:

```
["Kayak","Lifejacket","Soccer Ball"]
[10,20,30]
[75,97,121,97,107]
"S2F5YWs="
```

Обратите внимание, что байтовые массивы и байтовые срезы обрабатываются по-разному, даже если их содержимое одинаково.

Кодирование карт

Карты Go кодируются как объекты JSON, а ключи карты используются в качестве ключей объекта. Значения, содержащиеся в карте, кодируются в зависимости от их типа. В листинге 21-4 кодируется карта, содержащая значения float64.

> **Подсказка**
> Карты также могут быть полезны для создания пользовательских представлений данных Go в формате JSON, как описано в разделе «Создание полностью настраиваемых кодировок JSON».

```
package main

import (
    "strings"
    "fmt"
    "encoding/json"
)

func main() {

    m := map[string]float64 {
        "Kayak": 279,
        "Lifejacket": 49.95,
    }

    var writer strings.Builder
    encoder := json.NewEncoder(&writer)

    encoder.Encode(m)

    fmt.Print(writer.String())
}
```

Листинг 21-4 Кодирование карты в файле main.go в папке readerandwriters

Скомпилируйте и выполните проект, и вы увидите следующий вывод, показывающий, как ключи и значения в карте закодированы как объект JSON:

```
{"Kayak":279,"Lifejacket":49.95}
```

Кодирование структур

`Encoder` выражает значения структуры в виде объектов JSON, используя имена полей экспортированной структуры в качестве ключей объекта и значения полей в качестве значений объекта, как показано в листинге 21-5. Неэкспортированные поля игнорируются.

```
package main

import (
    "strings"
    "fmt"
    "encoding/json"
)

func main() {

    var writer strings.Builder
    encoder := json.NewEncoder(&writer)
    encoder.Encode(Kayak)
    fmt.Print(writer.String())
```

```
}
```

Листинг 21-5 Кодирование структуры в файле main.go в папке readerandwriters

В этом примере кодируется значение структуры Product с именем Kayak, которое было определено в главе 20. Структура Product определяет экспортированные поля Name, Category и Price, и их можно увидеть в выводе, полученном при компиляции и выполнении проекта:

```
{"Name":"Kayak","Category":"Watersports","Price":279}
```

Понимание эффекта продвижения в JSON при кодировании

Когда структура определяет встроенное поле, которое также является структурой, поля встроенной структуры продвигаются вперед и кодируются, как если бы они были определены включающим типом. Добавьте файл с именем discount.go в папку readerandwriters с содержимым, показанным в листинге 21-6.

```
package main

type DiscountedProduct struct {
    *Product
    Discount float64
}
```

Листинг 21-6 Содержимое файла discount.go в папке readerandwriters

Тип структуры DiscountedProduct определяет встроенное поле Product. В листинге 21-7 создается и кодируется DiscountedProduct как JSON.

```
package main

import (
    "strings"
    "fmt"
    "encoding/json"
)

func main() {

    var writer strings.Builder
    encoder := json.NewEncoder(&writer)
    dp := DiscountedProduct {
        Product: &Kayak,
        Discount: 10.50,
    }
    encoder.Encode(&dp)
    fmt.Print(writer.String())
}
```

Листинг 21-7 Кодирование структуры со встроенным полем в файле main.go в папке readerandwriters

Encoder продвигает поля Product в выходных данных JSON, как показано в выходных данных, когда проект компилируется и выполняется:

```
{"Name":"Kayak","Category":"Watersports","Price":279,"Discount":10.5}
```

Обратите внимание, что в листинге 21-7 кодируется указатель на значение структуры. Функция Encode следует за указателем и кодирует значение в его местоположении, что означает, что код в листинге 21-7 кодирует значение DiscountedProduct без создания копии.

Настройка JSON-кодирования структур

Способ кодирования структуры можно настроить с помощью *тегов структуры*, которые представляют собой строковые литералы, следующие за полями. Структурные теги являются частью поддержки Go для рефлексии, которую я описываю в главе 28, но для этой главы достаточно знать, что теги следуют за полями и могут использоваться для изменения двух аспектов того, как поле кодируется в JSON, как показано ниже. в листинге 21-8.

```
package main

type DiscountedProduct struct {
    *Product `json:"product"`
    Discount float64
}
```

Листинг 21-8 Использование тега структуры в файле discount.go в папке readerandwriters

Тег структуры имеет определенный формат, показанный на рисунке 21-1. За термином json следует двоеточие, за которым следует имя, которое следует использовать при кодировании поля, заключенное в двойные кавычки. Весь тег заключен в обратные кавычки.

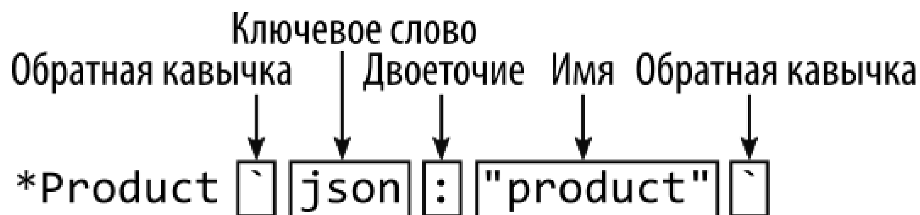

Рисунок 21-1 Тип структуры

Тег в листинге 21-8 указывает название product для встроенного поля. Скомпилируйте и выполните проект, и вы увидите следующий вывод, показывающий, что использование тега предотвратило продвижение поля:

```
{"product":
{"Name":"Kayak","Category":"Watersports","Price":279},"Discount":10.5}
```

Пропуск поля

Encoder пропускает поля, отмеченные тегом, указывающим дефис (символ -) для имени, как показано в листинге 21-9.

```
package main

type DiscountedProduct struct {
    *Product `json:"product"`
    Discount float64 `json:"-"`
}
```

Листинг 21-9 Пропуск поля в файле discount.go в папке readerandwriters

Новый тег указывает Encoder пропустить поле Discount при создании JSON-представления значения DIScountedProduct. Скомпилируйте и запустите проект, и вы увидите следующий вывод:

```
{"product":{"Name":"Kayak","Category":"Watersports","Price":279}}
```

Пропуск неназначенных полей

По умолчанию Encoder JSON включает поля структуры, даже если им не присвоено значение, как показано в листинге 21-10.

```
package main

import (
    "strings"
    "fmt"
    "encoding/json"
)

func main() {

    var writer strings.Builder
    encoder := json.NewEncoder(&writer)

    dp := DiscountedProduct {
        Product: &Kayak,
        Discount: 10.50,
    }
    encoder.Encode(&dp)

    dp2 := DiscountedProduct { Discount: 10.50 }
    encoder.Encode(&dp2)

    fmt.Print(writer.String())
}
```

Листинг 21-10 Неназначенное поле в файле main.go в папке readerandwriters

Скомпилируйте и выполните код, и вы увидите обработку по умолчанию для nil полей:

```
{"product":{"Name":"Kayak","Category":"Watersports","Price":279}}
{"product":null}
```

Чтобы исключить nil поле, к тегу поля добавляется ключевое слово omitempty, как показано в листинге 21-11.

```
package main

type DiscountedProduct struct {
    *Product `json:"product,omitempty"`
    Discount float64 `json:"-"`
}
```

Листинг 21-11 Пропуск нулевого поля в файле discount.go в папке readerandwriters

Ключевое слово omitempty отделяется от имени поля запятой, но без пробелов. Скомпилируйте и выполните код, и вы увидите вывод без пустого поля:

```
{"product":{"Name":"Kayak","Category":"Watersports","Price":279}}
{}
```

Чтобы пропустить пустое поле без изменения имени или продвижения поля, укажите ключевое слово omitempty без имени, как показано в листинге 21-12.

```
package main

type DiscountedProduct struct {
    *Product `json:",omitempty"`
    Discount float64 `json:"-"`
}
```

Листинг 21-12 Пропуск поля в файле discount.go в папке readerandwriters

Encoder продвигает поля Product, если встроенному полю присвоено значение, и пропускает поле, если значение не присвоено. Скомпилируйте и запустите проект, и вы увидите следующий вывод:

```
{"Name":"Kayak","Category":"Watersports","Price":279}
{}
```

Принудительное кодирование полей как строк

Теги структуры можно использовать для принудительного кодирования значения поля в виде строки, переопределяя обычную кодировку для типа поля, как показано в листинге 21-13.

```
package main
```

```
type DiscountedProduct struct {
    *Product `json:",omitempty"`
    Discount float64 `json:",string"`
}
```

Листинг 21-13 Принудительная оценка строки в файле discount.go в папке readerandwriters

Добавление ключевого слова `string` переопределяет кодировку по умолчанию и создает строку для поля `Discount`, которую можно увидеть в выводе, который создается при компиляции и выполнении проекта:

```
{"Name":"Kayak","Category":"Watersports","Price":279,"Discount":"10.5"}
{"Discount":"10.5"}
```

Интерфейсы кодирования

Кодировщик JSON можно использовать для значений, присвоенных переменным интерфейса, но кодируется динамический тип. Добавьте файл с именем `interface.go` в папку `readerandwriters` с содержимым, показанным в листинге 21-14.

```
package main

type Named interface { GetName() string }

type Person struct { PersonName string}
func (p *Person) GetName() string { return p.PersonName}

func (p *DiscountedProduct) GetName() string { return p.Name}
```

Листинг 21-14 Содержимое файла interface.go в папке readerandwriters

Этот файл определяет простой интерфейс и структуру, которая его реализует, а также определяет метод для структуры `DiscountedProduct`, который также реализует интерфейс. В листинге 21-15 кодировщик JSON используется для кодирования среза интерфейса.

```
package main

import (
    "strings"
    "fmt"
    "encoding/json"
)

func main() {

    var writer strings.Builder
    encoder := json.NewEncoder(&writer)

    dp := DiscountedProduct {
```

```
        Product: &Kayak,
        Discount: 10.50,
    }

    namedItems := []Named { &dp, &Person{ PersonName: "Alice"}}
    encoder.Encode(namedItems)

    fmt.Print(writer.String())
}
```

Листинг 21-15 Кодирование среза интерфейса в файле main.go в папке readerandwriters

Срез `Named` значений содержит различные динамические типы, которые можно увидеть, скомпилировав и выполнив проект:

```
[{"Name":"Kayak","Category":"Watersports","Price":279,"Discount":"10.5"},
 {"PersonName":"Alice"}]
```

Никакой аспект интерфейса не используется для адаптации JSON, и все экспортируемые поля каждого значения в срезе включаются в JSON. Это может быть полезной функцией, но при декодировании такого типа JSON следует соблюдать осторожность, поскольку каждое значение может иметь разный набор полей, как я объясню в разделе «Декодирование массивов».

Создание полностью настраиваемых кодировок JSON

`Encoder` проверяет, реализует ли структура интерфейс `Marshaler`, который обозначает тип, имеющий пользовательскую кодировку и определяющий метод, описанный в таблице 21-7.

Таблица 21-7 Метод Marshaler

Функция	Описание
`MarshalJSON()`	Этот метод вызывается для создания JSON-представления значения и возвращает байтовый срез, содержащий JSON и `error`, указывающую на проблемы с кодировкой.

В листинге 21-16 реализован интерфейс `Marshaler` для указателей на тип структуры `DiscountedProduct`.

```
package main

import "encoding/json"

type DiscountedProduct struct {
    *Product `json:",omitempty"`
    Discount float64 `json:",string"`
}

func (dp *DiscountedProduct) MarshalJSON() (jsn []byte, err error) {
    if (dp.Product != nil) {
```

```
        m := map[string]interface{} {
            "product": dp.Name,
            "cost": dp.Price - dp.Discount,
        }
        jsn, err = json.Marshal(m)
    }
    return
}
```

Листинг 21-16 Реализация интерфейса Marshaler в файле discount.go в папке readerandwriters.

Метод MarshalJSON может генерировать JSON любым удобным для проекта способом, но я считаю, что наиболее надежным подходом является использование поддержки карт кодирования. Я определяю карту со string ключами и использую пустой интерфейс для значений. Это позволяет мне построить JSON, добавив к карте пары ключ-значение, а затем передать карту функции Marshal, описанной в таблице 21-7, которая использует встроенную поддержку для кодирования каждого из значений, содержащихся в карте. Скомпилируйте и запустите проект, и вы увидите следующий вывод:

```
[{"cost":268.5,"product":"Kayak"},{"PersonName":"Alice"}]
```

Декодирование данных JSON

Функция-конструктор NewDecoder создает Decoder, который можно использовать для декодирования данных JSON, полученных от Reader, с использованием методов, описанных в Таблице 21-8.

Таблица 21-8 Метод Decoder

Функция	Описание
Decode(value)	Этот метод считывает и декодирует данные, которые используются для создания указанного значения. Метод возвращает error, указывающую на проблемы с декодированием данных до требуемого типа или EOF.
DisallowUnknownFields()	По умолчанию при декодировании типа структуры Decoder игнорирует любой ключ в данных JSON, для которого нет соответствующего поля структуры. Вызов этого метода приводит к тому, что Decode возвращает ошибку, а не игнорирует ключ.
UseNumber()	По умолчанию числовые значения JSON декодируются в значения float64. При вызове этого метода вместо этого используется тип Number, как описано в разделе «Расшифровка числовых значений».

Листинг 21-17 демонстрирует декодирование основных типов данных.

```
package main

import (
    "strings"
    //"fmt"
    "encoding/json"
```

```
    "io"
)

func main() {

    reader := strings.NewReader(`true "Hello" 99.99 200`)

    vals := []interface{} { }

    decoder := json.NewDecoder(reader)

    for {
        var decodedVal interface{}
        err := decoder.Decode(&decodedVal)
        if (err != nil) {
            if (err != io.EOF) {
                Printfln("Error: %v", err.Error())
            }
            break
        }
        vals = append(vals, decodedVal)
    }

    for _, val := range vals {
        Printfln("Decoded (%T): %v", val, val)
    }
}
```

Листинг 21-17 Декодирование основных типов данных в файле main.go в папке readerandwriters

Я создаю `Reader`, который будет создавать данные из строки, содержащей последовательность значений, разделенных пробелами (спецификация JSON позволяет разделять значения пробелами или символами новой строки).

Первым шагом в декодировании данных является создание `Decoder`, который принимает `Reader`. Я хочу декодировать несколько значений, поэтому вызываю метод `Decode` внутри цикла `for`. Декодер может выбрать подходящий тип данных Go для значений JSON, и это достигается путем предоставления указателя на пустой интерфейс в качестве аргумента метода `Decode`, например:

```
...
var decodedVal interface{}
err := decoder.Decode(&decodedVal)
...
```

Метод `Decode` возвращает `error`, которая указывает на проблемы с декодированием, но также используется для обозначения конца данных с помощью ошибки `io.EOF`. Цикл `for` повторно декодирует значения до тех пор, пока не завершится EOF, а затем я использую другой цикл for для записи каждого декодированного типа и значения, используя глаголы форматирования, описанные в

главе 17. Скомпилируйте и выполните проект, и вы увидите декодированные значения:

```
Decoded (bool): true
Decoded (string): Hello
Decoded (float64): 99.99
Decoded (float64): 200
```

Расшифровка числовых значений

JSON использует один тип данных для представления как значений с плавающей запятой, так и целых чисел. `Decoder` декодирует эти числовые значения как значения `float64`, что можно увидеть в выходных данных предыдущего примера.

Это поведение можно изменить, вызвав метод `UseNumber` в `Decoder`, который приводит к декодированию числовых значений JSON в тип `Number`, определенный в пакете `encoding/json`. Тип `Number` определяет методы, описанные в таблице 21-9.

Таблица 21-9 Методы, определяемые числовым типом

Функция	**Описание**
`Int64()`	Этот метод возвращает декодированное значение как `int64` и `error`, которая указывает, что значение не может быть преобразовано.
`Float64()`	Этот метод возвращает декодированное значение в виде `float64` и `error`, которая указывает, что значение не может быть преобразовано.
`String()`	Этот метод возвращает непреобразованную строку из данных JSON.

Методы в таблице 21-9 используются последовательно. Не все числовые значения JSON могут быть выражены как значения Go `int64`, поэтому этот метод обычно вызывается первым. Если попытка преобразования в целое число не удалась, можно вызвать метод `Float64`. Если число не может быть преобразовано ни в один из типов Go, то можно использовать метод `String` для получения непреобразованной строки из данных JSON. Эта последовательность показана в листинге 21-18.

```
package main

import (
    "strings"
    //"fmt"
    "encoding/json"
    "io"
)

func main() {

    reader := strings.NewReader(`true "Hello" 99.99 200`)

    vals := []interface{} { }
```

```
    decoder := json.NewDecoder(reader)
    decoder.UseNumber()

    for {
        var decodedVal interface{}
        err := decoder.Decode(&decodedVal)
        if (err != nil) {
            if (err != io.EOF) {
                Printfln("Error: %v", err.Error())
            }
            break
        }
        vals = append(vals, decodedVal)
    }

    for _, val := range vals {
        if num, ok := val.(json.Number); ok {
            if ival, err := num.Int64(); err == nil {
                Printfln("Decoded Integer: %v", ival)
            } else if fpval, err := num.Float64(); err == nil {
                Printfln("Decoded Floating Point: %v", fpval)
            } else {
                Printfln("Decoded String: %v", num.String())
            }
        } else {
            Printfln("Decoded (%T): %v", val, val)
        }
    }
}
```

Листинг 21-18 Расшифровка чисел в файле main.go в папке readerandwriters

Скомпилируйте и выполните код, и вы увидите, что одно из значений JSON было преобразовано в значение `int64`:

```
Decoded (bool): true
Decoded (string): Hello
Decoded Floating Point: 99.99
Decoded Integer: 200
```

Указание типов для декодирования

В предыдущих примерах методу `Decode` передавалась пустая переменная интерфейса, например:

```
...
var decodedVal interface{}
err := decoder.Decode(&decodedVal)
...
```

Это позволяет Decoder выбрать тип данных Go для декодируемого значения JSON. Если вы знаете структуру данных JSON, которые вы декодируете, вы можете указать Decoder использовать определенные типы Go, используя переменные этого типа для получения декодированного значения, как показано в листинге 21-19.

```
package main

import (
    "strings"
    //"fmt"
    "encoding/json"
    //"io"
)

func main() {

    reader := strings.NewReader(`true "Hello" 99.99 200`)

    var bval bool
    var sval string
    var fpval float64
    var ival int

    vals := []interface{} { &bval, &sval, &fpval, &ival }

    decoder := json.NewDecoder(reader)

    for i := 0; i < len(vals); i++ {
        err := decoder.Decode(vals[i])
        if err != nil {
            Printfln("Error: %v", err.Error())
            break
        }
    }

    Printfln("Decoded (%T): %v", bval, bval)
    Printfln("Decoded (%T): %v", sval, sval)
    Printfln("Decoded (%T): %v", fpval, fpval)
    Printfln("Decoded (%T): %v", ival, ival)
}
```

Листинг 21-19 Указание типов для декодирования в файле main.go в папке readerandwriters

В листинге 21-19 указаны типы данных, которые следует использовать для декодирования, и для удобства они сгруппированы в срез. Значения декодируются в целевые типы, которые можно увидеть в выводе, отображаемом при компиляции и выполнении проекта:

```
Decoded (bool): true
Decoded (string): Hello
```

```
Decoded (float64): 99.99
Decoded (int): 200
```

`Decoder` вернет ошибку, если не сможет декодировать значение JSON в указанный тип. Этот метод следует использовать только в том случае, если вы уверены, что понимаете данные JSON, которые будут декодированы.

Декодирование массивов

`Decoder` обрабатывает массивы автоматически, но следует соблюдать осторожность, поскольку JSON позволяет массивам содержать значения разных типов, что противоречит строгим правилам типов, применяемым в Go. В листинге 21-20 показано декодирование массива.

```
package main

import (
    "strings"
    //"fmt"
    "encoding/json"
    "io"
)

func main() {

    reader := strings.NewReader(`[10,20,30]["Kayak","Lifejacket",279]`)

    vals := []interface{} { }

    decoder := json.NewDecoder(reader)

    for {
        var decodedVal interface{}
        err := decoder.Decode(&decodedVal)
        if (err != nil) {
            if (err != io.EOF) {
                Printfln("Error: %v", err.Error())
            }
            break
        }
        vals = append(vals, decodedVal)
    }

    for _, val := range vals {
        Printfln("Decoded (%T): %v", val, val)
    }
}
```

Листинг 21-20 Декодирование массива в файле main.go в папке readerandwriters

Исходные данные JSON содержат два массива, один из которых содержит только числа, а другой содержит числа и строки. Decoder не пытается выяснить, можно ли представить массив JSON с помощью одного типа Go, и декодирует каждый массив в пустой срез интерфейса:

```
Decoded ([]interface {}): [10 20 30]
Decoded ([]interface {}): [Kayak Lifejacket 279]
```

Каждое значение вводится на основе значения JSON, но тип среза — пустой интерфейс. Если вы заранее знаете структуру данных JSON и декодируете массив, содержащий один тип данных JSON, то вы можете передать Go-срез нужного типа в метод Decode, как показано в листинге 21-21.

```
package main

import (
    "strings"
    //"fmt"
    "encoding/json"
    //"io"
)

func main() {

    reader := strings.NewReader(`[10,20,30]["Kayak","Lifejacket",279]`)

    ints := []int {}
    mixed := []interface{} {}

    vals := []interface{} { &ints, &mixed}

    decoder := json.NewDecoder(reader)

    for i := 0; i < len(vals); i++ {
        err := decoder.Decode(vals[i])
        if err != nil {
            Printfln("Error: %v", err.Error())
            break
        }
    }

    Printfln("Decoded (%T): %v", ints, ints)
    Printfln("Decoded (%T): %v", mixed, mixed)
}
```

Листинг 21-21 Указание типа декодированного массива в файле main.go в папке readerandwriters

Я могу указать срез int для декодирования первого массива в данных JSON, потому что все значения могут быть представлены как значения Go int. Второй массив содержит смесь значений, что означает, что я должен указать пустой

интерфейс в качестве целевого типа. Синтаксис буквального среза неудобен при использовании пустого интерфейса, поскольку требуются два набора фигурных скобок:

```
...
mixed := []interface{} {}
...
```

Пустой тип интерфейса включает пустые фигурные скобки (`interface{}`), а также указание пустого среза (`{}`). Скомпилируйте и запустите проект, и вы увидите, что первый массив JSON был декодирован в `int` срез:

```
Decoded ([]int): [10 20 30]
Decoded ([]interface {}): [Kayak Lifejacket 279]
```

Декодирование карт

Объекты JavaScript выражаются в виде пар ключ-значение, что упрощает их декодирование в карты Go, как показано в листинге 21-22.

```
package main

import (
    "strings"
    //"fmt"
    "encoding/json"
    //"io"
)

func main() {

    reader := strings.NewReader(`{"Kayak" : 279, "Lifejacket" : 49.95}`)

    m := map[string]interface{} {}

    decoder := json.NewDecoder(reader)

    err := decoder.Decode(&m)
    if err != nil {
        Printfln("Error: %v", err.Error())
    } else {
        Printfln("Map: %T, %v", m, m)
        for k, v := range m {
            Printfln("Key: %v, Value: %v", k, v)
        }
    }
}
```

Листинг 21-22 Расшифровка карты в файле main.go в папке readerandwriters

Самый безопасный подход — определить карту со строковыми ключами и пустыми значениями интерфейса, что гарантирует, что все пары ключ-значение в данных JSON могут быть декодированы в карту, как показано в листинге 21-22. После декодирования JSON цикл `for` используется для перечисления содержимого карты, что приводит к следующему результату при компиляции и выполнении проекта:

```
Map: map[string]interface {}, map[Kayak:279 Lifejacket:49.95]
Key: Kayak, Value: 279
Key: Lifejacket, Value: 49.95
```

Один объект JSON может использоваться для нескольких типов данных в качестве значений, но если вы заранее знаете, что будете декодировать объект JSON с одним типом значения, то вы можете быть более конкретными при определении карты, в которую будут помещаться данные. быть декодирован, как показано в листинге 21-23.

```
package main

import (
    "strings"
    //"fmt"
    "encoding/json"
    //"io"
)

func main() {

    reader := strings.NewReader(`{"Kayak" : 279, "Lifejacket" : 49.95}`)

    m := map[string]float64 {}

    decoder := json.NewDecoder(reader)

    err := decoder.Decode(&m)
    if err != nil {
        Printfln("Error: %v", err.Error())
    } else {
        Printfln("Map: %T, %v", m, m)
        for k, v := range m {
            Printfln("Key: %v, Value: %v", k, v)
        }
    }
}
```

Листинг 21-23 Использование определенного типа значения в файле main.go в папке readerandwriters

Все значения в объекте JSON могут быть представлены с использованием типа Go `float64`, поэтому в листинге 21-23 тип карты изменен на `map[string]float64`.

Скомпилируйте и запустите проект, и вы увидите изменение типа карты:

```
Map: map[string]float64, map[Kayak:279 Lifejacket:49.95]
Key: Kayak, Value: 279
Key: Lifejacket, Value: 49.95
```

Декодирование структур

Структура ключ-значение объектов JSON может быть декодирована в значения структуры Go, как показано в листинге 21-24, хотя для этого требуется больше знаний о данных JSON, чем для декодирования данных в карту.

> **Декодирование типов интерфейса**
> Как я объяснял ранее в этой главе, кодировщик JSON работает с интерфейсами, кодируя значение с использованием экспортируемых полей динамического типа. Это связано с тем, что JSON имеет дело с парами ключ-значение и не имеет возможности выразить методы. Как следствие, вы не можете напрямую декодировать интерфейсную переменную из JSON. Вместо этого вы должны декодировать структуру или карту, а затем присвоить созданное значение переменной интерфейса..

```
package main

import (
    "strings"
    //"fmt"
    "encoding/json"
    "io"
)

func main() {

    reader := strings.NewReader(`
        {"Name":"Kayak","Category":"Watersports","Price":279}
        {"Name":"Lifejacket","Category":"Watersports" }
               {"name":"Canoe","category":"Watersports",  "price":  100,
"inStock": true }
    `)

    decoder := json.NewDecoder(reader)

    for {
        var val Product
        err := decoder.Decode(&val)
        if err != nil {
            if err != io.EOF {
                Printfln("Error: %v", err.Error())
            }
```

```
            break
        } else {
            Printfln("Name: %v, Category: %v, Price: %v",
                val.Name, val.Category, val.Price)
        }
    }
}
```

Листинг 21-24 Декодирование в структуру в файле main.go в папке readerandwriters

Decoder декодирует объект JSON и использует ключи для установки значений экспортируемых полей структуры. Использование заглавных букв в полях и ключах JSON не обязательно должно совпадать, и Decoder будет игнорировать любой ключ JSON, для которого нет поля структуры, и любое поле структуры, для которого нет ключа JSON. Объекты JSON в листингах 21-24 содержат другой регистр заглавных букв и имеют больше или меньше ключей, чем поля структуры Product. Decoder обрабатывает данные как можно лучше, выдавая следующий результат, когда проект компилируется и выполняется:

```
Name: Kayak, Category: Watersports, Price: 279
Name: Lifejacket, Category: Watersports, Price: 0
Name: Canoe, Category: Watersports, Price: 100
```

Запрет неиспользуемых ключей

По умолчанию Decoder будет игнорировать ключи JSON, для которых нет соответствующего поля структуры. Это поведение можно изменить, вызвав метод DisallowUnknownFields, как показано в листинге 21-25, который вызывает ошибку при обнаружении такого ключа.

```
...
decoder := json.NewDecoder(reader)
decoder.DisallowUnknownFields()
...
```

Листинг 21-25 Запрет неиспользуемых ключей в файле main.go в папке readerandwriters

Один из объектов JSON, определенных в листинге 21-25, содержит ключ inStock, для которого нет соответствующего поля Product. Обычно этот ключ игнорируется, но поскольку был вызван метод DisallowUnknownFields, при декодировании этого объекта возникает ошибка, которую можно увидеть в выводе:

```
Name: Kayak, Category: Watersports, Price: 279
Name: Lifejacket, Category: Watersports, Price: 0
Error: json: unknown field "inStock"
```

Использование структурных тегов для управления декодированием

Ключи, используемые в объекте JSON, не всегда совпадают с полями, определенными в структурах проекта Go. Когда это происходит, можно

использовать теги структуры для сопоставления данных JSON и структуры, как показано в листинге 21-26.

```
package main

import "encoding/json"

type DiscountedProduct struct {
    *Product `json:",omitempty"`
    Discount float64 `json:"offer,string"`
}

func (dp *DiscountedProduct) MarshalJSON() (jsn []byte, err error) {
    if (dp.Product != nil) {
        m := map[string]interface{} {
            "product": dp.Name,
            "cost": dp.Price - dp.Discount,
        }
        jsn, err = json.Marshal(m)
    }
    return
}
```

Листинг 21-26 Использование структурных тегов в файле discount.go в папке readerandwriters

Тег, примененный к полю `Discount`, сообщает `Decoder`, что значение для этого поля должно быть получено из ключа JSON с именем `offer` и что значение будет проанализировано из строки, а не числа JSON, которое обычно ожидается для Go `float64`. В листинге 21-27 строка JSON декодируется в значение структуры `DiscountedProduct`.

```
package main

import (
    "strings"
    //"fmt"
    "encoding/json"
    "io"
)

func main() {

    reader := strings.NewReader(`
        {"Name":"Kayak","Category":"Watersports","Price":279, "Offer":
"10"}`)

    decoder := json.NewDecoder(reader)

    for {
        var val DiscountedProduct
```

```
        err := decoder.Decode(&val)
        if err != nil {
            if err != io.EOF {
                Printfln("Error: %v", err.Error())
            }
            break
        } else {
            Printfln("Name: %v, Category: %v, Price: %v, Discount: %v",
                val.Name, val.Category, val.Price, val.Discount)
        }
    }
}
```

Листинг 21-27 Декодирование структуры с тегом в файле main.go в папке readerandwriters

Скомпилируйте и выполните проект, и вы увидите, как тег структуры использовался для управления декодированием данных JSON:

```
Name: Kayak, Category: Watersports, Price: 279, Discount: 10
```

Создание полностью настраиваемых декодеров JSON

`Decoder` проверяет, реализует ли структура интерфейс `Unmarshaler`, обозначающий тип с пользовательской кодировкой и определяющий метод, описанный в таблице 21-10.

Таблица 21-10 Метод Unmarshaler

Функция	Описание
`UnmarshalJSON(byteSlice)`	Этот метод вызывается для декодирования данных JSON, содержащихся в указанном байтовом срезе. Результатом является error, указывающая на проблемы с кодировкой.

В листинге 21-28 реализован интерфейс для указателей на тип структуры `DiscountedProduct`.

```
package main

import (
    "encoding/json"
    "strconv"
)

type DiscountedProduct struct {
    *Product `json:",omitempty"`
    Discount float64 `json:"offer,string"`
}

func (dp *DiscountedProduct) MarshalJSON() (jsn []byte, err error) {
    if (dp.Product != nil) {
        m := map[string]interface{} {
```

```
            "product": dp.Name,
            "cost": dp.Price - dp.Discount,
        }
        jsn, err = json.Marshal(m)
    }
    return
}

func (dp *DiscountedProduct) UnmarshalJSON(data []byte) (err error) {

    mdata := map[string]interface{} {}
    err = json.Unmarshal(data, &mdata)

    if (dp.Product == nil) {
        dp.Product = &Product{}
    }

    if (err == nil) {
        if name, ok := mdata["Name"].(string); ok {
            dp.Name = name
        }
        if category, ok := mdata["Category"].(string); ok {
            dp.Category = category
        }
        if price, ok := mdata["Price"].(float64); ok {
            dp.Price = price
        }
        if discount, ok := mdata["Offer"].(string); ok {
            fpval, fperr := strconv.ParseFloat(discount, 64)
            if (fperr == nil) {
                dp.Discount = fpval
            }
        }
    }
    return
}
```

Листинг 21-28 Определение пользовательского декодера в файле discount.go в папке readerandwriters

Эта реализация метода `UnmarshalJSON` использует метод `Unmarshal` для декодирования данных JSON в карту, а затем проверяет тип каждого значения, необходимого для структуры `DiscountedProduct`. Скомпилируйте и запустите проект, и вы увидите пользовательскую расшифровку:

```
Name: Kayak, Category: Watersports, Price: 279, Discount: 10
```

Резюме

В этой главе я описал поддержку Go для работы с данными JSON, которая опирается на интерфейсы `Reader` и `Writer`, описанные в главе 20. Эти интерфейсы

последовательно используются во всей стандартной библиотеке, как вы увидите в следующей главе, где я объясняю как файлы могут быть прочитаны и записаны.

22. Работа с файлами

В этой главе я описываю возможности, предоставляемые стандартной библиотекой Go для работы с файлами и каталогами. Go работает на нескольких платформах, а стандартная библиотека использует независимый от платформы подход, поэтому код можно писать без необходимости разбираться в файловых системах, используемых разными операционными системами. Таблица 22-1 рассматривает работу с файлами в контексте.

Таблица 22-1 Работа с файлами в контексте

Вопрос	Ответ
Кто они такие?	Эти функции обеспечивают доступ к файловой системе, чтобы файлы можно было читать и записывать.
Почему они полезны?	Файлы используются для всего, от ведения журнала до файлов конфигурации.
Как они используются?	Доступ к этим функциям осуществляется через пакет `os`, который обеспечивает независимый от платформы доступ к файловой системе.
Есть ли подводные камни или ограничения?	Необходимо учитывать базовую файловую систему, особенно при работе с путями.
Есть ли альтернативы?	Go поддерживает альтернативные способы хранения данных, например базы данных, но альтернативных механизмов доступа к файлам нет.

Таблица 22-2 суммирует содержание главы.

Таблица 22-2 Краткое содержание главы

Проблема	Решение	Листинг
Прочитать содержимое файла	Use the `ReadFile` function	6–8
Контролировать способ чтения файлов	Получите структуру `File` и используйте предоставляемые ею функции	9–10
Написать содержимое файла	Используйте функцию `WriteFile`	11

Проблема	Решение	Листинг
Контролировать способ записи файлов	Получите структуру File и используйте предоставляемые ею функции	12, 13
Создать новые файлы	Используйте функцию Create или CreateTemp	14
Работа с путями к файлам	Используйте функции в пакете path/filepath или используйте общие расположения, для которых есть функции в пакете os.	15
Управление файлами и каталогами	Используйте функции, предоставляемые пакетом os	16–17, 19, 20
Определить, существует ли файл	Проверьте error, возвращаемую функцией Stat	18

Подготовка к этой главе

Чтобы подготовиться к этой главе, откройте новую командную строку, перейдите в удобное место и создайте каталог с именем files. Запустите команду, показанную в листинге 22-1, чтобы создать файл модуля.

> **Подсказка**
> Вы можете загрузить пример проекта для этой главы — и для всех остальных глав этой книги — с https://github.com/apress/pro-go. См. Главу 2 о том, как получить помощь, если у вас возникнут проблемы с запуском примеров.

```
go mod init files
```

Листинг 22-1 Инициализация модуля

Добавьте файл с именем printer.go в папку файлов с содержимым, показанным в листинге 22-2.

```
package main

import "fmt"
```

```
func Println(template string, values ...interface{}) {
    fmt.Printf(template + "\n", values...)
}
```

Листинг 22-2 Содержимое файла printer.go в папке с файлами

Добавьте файл с именем `product.go` в папку `files` с содержимым, показанным в листинге 22-3.

```
package main

type Product struct {
    Name, Category string
    Price float64
}

var Products = []Product {
    { "Kayak", "Watersports", 279 },
    { "Lifejacket", "Watersports", 49.95 },
    { "Soccer Ball", "Soccer", 19.50 },
    { "Corner Flags", "Soccer", 34.95 },
    { "Stadium", "Soccer", 79500 },
    { "Thinking Cap", "Chess", 16 },
    { "Unsteady Chair", "Chess", 75 },
    { "Bling-Bling King", "Chess", 1200 },
}
```

Листинг 22-3 Содержимое файла product.go в папке с файлами

Добавьте файл с именем `main.go` в папку `files`, содержимое которого показано в листинге 22-4.

```
package main

func main() {
    for _, p := range Products {
        Println("Product: %v, Category: %v, Price: $%.2f",
            p.Name, p.Category, p.Price)
    }
}
```

Листинг 22-4 Содержимое файла main.go в папке с файлами

Используйте командную строку для запуска команды, показанной в листинге 22-5, в папке files.

```
go run .
```

Листинг 22-5 Запуск примера проекта

Код будет скомпилирован и выполнен, что приведет к следующему выводу:

```
Product: Kayak, Category: Watersports, Price: $279.00
Product: Lifejacket, Category: Watersports, Price: $49.95
Product: Soccer Ball, Category: Soccer, Price: $19.50
Product: Corner Flags, Category: Soccer, Price: $34.95
Product: Stadium, Category: Soccer, Price: $79500.00
Product: Thinking Cap, Category: Chess, Price: $16.00
Product: Unsteady Chair, Category: Chess, Price: $75.00
Product: Bling-Bling King, Category: Chess, Price: $1200.00
```

Чтение файлов

Ключевым пакетом при работе с файлами является пакет os. Этот пакет обеспечивает доступ к функциям операционной системы, включая файловую систему, таким образом, что скрывает большую часть деталей реализации, а это означает, что одни и те же функции могут использоваться для достижения одинаковых результатов независимо от используемой операционной системы.

Нейтральный подход, принятый пакетом os, приводит к некоторым компромиссам и склоняется к UNIX/Linux, а не, скажем, к Windows. Но даже в этом случае функции, предоставляемые пакетом os, надежны и надежны и позволяют писать код Go, который можно использовать на разных платформах без модификации. Таблица 22-3 описывает функции, предоставляемые пакетом os для чтения файлов.

Таблица 22-3 Функции пакета os для чтения файлов

Функция	Описание
ReadFile(name)	Эта функция открывает указанный файл и читает его содержимое. Результатом является байтовый срез, содержащий контент файла, и error, указывающая на проблемы с открытием или чтением файла.

Функция	Описание
`Open(name)`	Эта функция открывает указанный файл для чтения. Результатом является структура `File` и `error`, указывающая на проблемы с открытием файла.

Чтобы подготовиться к примерам в этой части главы, добавьте файл с именем `config.json` в папку `files` с содержимым, показанным в листинге 22-6.

```
{
    "Username": "Alice",
    "AdditionalProducts": [
        {"name": "Hat", "category": "Skiing", "price": 10},
             {"name":  "Boots", "category":"Skiing", "price":
220.51 },
            {"name":  "Gloves", "category":"Skiing", "price":
40.20 }
    ]
}
```

Листинг 22-6 Содержимое файла config.json в папке files

Одной из наиболее распространенных причин чтения файла является загрузка данных конфигурации. Формат JSON хорошо подходит для файлов конфигурации, поскольку он прост в обработке, имеет хорошую поддержку в стандартной библиотеке Go (как показано в главе 21) и может представлять сложные структуры.

Использование функции удобства чтения

Функция `ReadFile` обеспечивает удобный способ чтения всего содержимого файла в байтовый срез за один шаг. Добавьте файл с именем `readconfig.go` в папку файлов с содержимым, показанным в листинге 22-7.

```
package main

import "os"

func LoadConfig() (err error) {
    data, err := os.ReadFile("config.json")
    if (err == nil) {
        Println(string(data))
```

```
    }
    return
}

func init() {
    err := LoadConfig()
    if (err != nil) {
        Printfln("Error Loading Config: %v", err.Error())
    }
}
```

Листинг 22-7 одержимое файла readconfig.go в папке files

Функция `LoadConfig` использует функцию `ReadFile` для чтения содержимого файла `config.json`. Файл будет прочитан из текущего рабочего каталога при выполнении приложения, а это значит, что я могу открыть файл только по его имени.

Содержимое файла возвращается как байтовый срез, который преобразуется в `string` и записывается. Функция `LoadConfig` вызывается функцией инициализации, которая обеспечивает чтение файла конфигурации. Скомпилируйте и выполните код, и вы увидите содержимое файла `config.json` в выводе, созданном приложением:

```
{
    "Username": "Alice",
    "AdditionalProducts": [
        {"name": "Hat", "category": "Skiing", "price": 10},
            {"name":  "Boots",  "category":"Skiing",  "price":
220.51 },
            {"name":  "Gloves",  "category":"Skiing",  "price":
40.20 }
    ]
}
Product: Kayak, Category: Watersports, Price: $279.00
Product: Lifejacket, Category: Watersports, Price: $49.95
Product: Soccer Ball, Category: Soccer, Price: $19.50
Product: Corner Flags, Category: Soccer, Price: $34.95
Product: Stadium, Category: Soccer, Price: $79500.00
Product: Thinking Cap, Category: Chess, Price: $16.00
Product: Unsteady Chair, Category: Chess, Price: $75.00
Product: Bling-Bling King, Category: Chess, Price: $1200.00
```

Декодирование данных JSON

Для примера файла конфигурации получение содержимого файла в виде строки не является идеальным, и более полезным подходом будет анализ содержимого в виде JSON, что можно легко сделать, упаковав байтовые данные, чтобы к ним можно было получить доступ. через `Reader`, как показано в листинге 22-8.

```
package main

import (
    "os"
    "encoding/json"
    "strings"
)

type ConfigData struct {
    UserName string
    AdditionalProducts []Product
}

var Config ConfigData

func LoadConfig() (err error) {
    data, err := os.ReadFile("config.json")
    if (err == nil) {
        decoder :=
json.NewDecoder(strings.NewReader(string(data)))
        err = decoder.Decode(&Config)
    }
    return
}

func init() {
    err := LoadConfig()
    if (err != nil) {
        Printfln("Error Loading Config: %v", err.Error())
    } else {
        Printfln("Username: %v", Config.UserName)
        Products = append(Products,
Config.AdditionalProducts...)
    }
}
```

Листинг 22-8 Декодирование данных JSON в файле readconfig.go в папке files

Я мог бы декодировать данные JSON в файле `config.json` в карту, но я применил более структурированный подход в листинге 22-8 и определил тип структуры, поля которого соответствуют структуре данных конфигурации, что, как мне кажется, упрощает задачу. использовать данные конфигурации в реальных проектах. После декодирования данных конфигурации я записываю значение поля `UserName` и добавляю значения `Product` к срезу, определенному в файле `product.go`. Скомпилируйте и запустите проект, и вы увидите следующий вывод:

```
Username: Alice
Product: Kayak, Category: Watersports, Price: $279.00
Product: Lifejacket, Category: Watersports, Price: $49.95
Product: Soccer Ball, Category: Soccer, Price: $19.50
Product: Corner Flags, Category: Soccer, Price: $34.95
Product: Stadium, Category: Soccer, Price: $79500.00
Product: Thinking Cap, Category: Chess, Price: $16.00
Product: Unsteady Chair, Category: Chess, Price: $75.00
Product: Bling-Bling King, Category: Chess, Price: $1200.00
Product: Hat, Category: Skiing, Price: $10.00
Product: Boots, Category: Skiing, Price: $220.51
Product: Gloves, Category: Skiing, Price: $40.20
```

Использование файловой структуры для чтения файла

Функция `Open` открывает файл для чтения и возвращает значение `File`, представляющее открытый файл, и ошибку, которая используется для обозначения проблем с открытием файла. Структура `File` реализует интерфейс `Reader`, который упрощает чтение и обработку примера данных JSON без чтения всего файла в байтовый срез, как показано в листинге 22-9.

> **Использование стандартного ввода, вывода и ошибки**
> Пакет `os` определяет три переменные `*File` с именами `Stdin`, `Stdout` и `Stderr`, которые обеспечивают доступ к `Stdin`, `Stdout` и `Stderr`.

```
package main
```

```
import (
    "os"
    "encoding/json"
    //"strings"
)

type ConfigData struct {
    UserName string
    AdditionalProducts []Product
}

var Config ConfigData

func LoadConfig() (err error) {
    file, err := os.Open("config.json")
    if (err == nil) {
        defer file.Close()
        decoder := json.NewDecoder(file)
        err = decoder.Decode(&Config)
    }
    return
}

func init() {
    err := LoadConfig()
    if (err != nil) {
        Printfln("Error Loading Config: %v", err.Error())
    } else {
        Printfln("Username: %v", Config.UserName)
                                Products    =    append(Products,
Config.AdditionalProducts...)
    }
}
```

Листинг 22-9 Чтение файла конфигурации в файле readconfig.go в папке files

Структура `File` также реализует интерфейс `Closer`, описанный в главе 21, который определяет метод `Close`. Ключевое слово defer можно использовать для вызова метода `Close` после завершения закрывающей функции, например:

```
...
defer file.Close()
```

```
...
```

Вы можете просто вызвать метод `Close` в конце функции, если хотите, но использование ключевого слова `defer` гарантирует, что файл будет закрыт, даже если функция вернется раньше. Результат такой же, как и в предыдущем примере, который вы можете увидеть, скомпилировав и выполнив проект.

```
Username: Alice
Product: Kayak, Category: Watersports, Price: $279.00
Product: Lifejacket, Category: Watersports, Price: $49.95
Product: Soccer Ball, Category: Soccer, Price: $19.50
Product: Corner Flags, Category: Soccer, Price: $34.95
Product: Stadium, Category: Soccer, Price: $79500.00
Product: Thinking Cap, Category: Chess, Price: $16.00
Product: Unsteady Chair, Category: Chess, Price: $75.00
Product: Bling-Bling King, Category: Chess, Price: $1200.00
Product: Hat, Category: Skiing, Price: $10.00
Product: Boots, Category: Skiing, Price: $220.51
Product: Gloves, Category: Skiing, Price: $40.20
```

Чтение из определенного места

Структура `File` определяет методы помимо тех, что требуются интерфейсу `Reader`, которые позволяют выполнять чтение в определенном месте в файле, как описано в таблице 22-4.

Таблица 22-4 Методы, определенные файловой структурой для чтения в определенном месте

Функция	Описание
`ReadAt(slice, offset)`	Этот метод определяется интерфейсом `ReaderAt` и выполняет чтение в конкретный срез в указанном смещении позиции в файле.
`Seek(offset, how)`	Этот метод определяется интерфейсом `Seeker` и перемещает смещение в файл для следующего чтения. Смещение определяется комбинацией двух аргументов: первый аргумент указывает количество байтов для смещения, а второй аргумент определяет, как применяется смещение — значение `0` означает, что смещение относительно начала файла, значение `1` означает, что смещение относительно текущей позиции чтения, а значение `2` означает, что смещение относительно конца файла.

В листинге 22-10 показано использование методов из таблицы 22-4 для чтения определенных разделов данных из файла, которые затем объединяются в строку JSON и декодируются.

```
package main

import (
    "os"
    "encoding/json"
    //"strings"
)

type ConfigData struct {
    UserName string
    AdditionalProducts []Product
}

var Config ConfigData

func LoadConfig() (err error) {
    file, err := os.Open("config.json")
    if (err == nil) {
        defer file.Close()

        nameSlice := make([]byte, 5)
        file.ReadAt(nameSlice, 20)
        Config.UserName = string(nameSlice)

        file.Seek(55, 0)
        decoder := json.NewDecoder(file)
        err = decoder.Decode(&Config.AdditionalProducts)
    }
    return
}

func init() {
    err := LoadConfig()
    if (err != nil) {
        Printfln("Error Loading Config: %v", err.Error())
    } else {
        Printfln("Username: %v", Config.UserName)
                                Products    =    append(Products,
Config.AdditionalProducts...)
    }
}
```

Листинг 22-10 Чтение из определенных мест в файле readconfig.go в папке files

Чтение из определенных мест требует знания структуры файла. В этом примере я знаю расположение данных, которые я хочу прочитать, что позволяет мне использовать метод ReadAt для чтения значения имени пользователя и метод Seek для перехода к началу данных о продукте. Скомпилируйте и запустите проект, и вы увидите следующий вывод:

```
Username: Alice
Product: Kayak, Category: Watersports, Price: $279.00
Product: Lifejacket, Category: Watersports, Price: $49.95
Product: Soccer Ball, Category: Soccer, Price: $19.50
Product: Corner Flags, Category: Soccer, Price: $34.95
Product: Stadium, Category: Soccer, Price: $79500.00
Product: Thinking Cap, Category: Chess, Price: $16.00
Product: Unsteady Chair, Category: Chess, Price: $75.00
Product: Bling-Bling King, Category: Chess, Price: $1200.00
Product: Hat, Category: Skiing, Price: $10.00
Product: Boots, Category: Skiing, Price: $220.51
Product: Gloves, Category: Skiing, Price: $40.20
```

Если вы получаете сообщение об ошибке из этого примера, вероятной причиной является то, что местоположения, указанные в листинге 22-10, не соответствуют структуре вашего файла JSON. В качестве первого шага, особенно в Linux, убедитесь, что вы сохранили файл с символами CR и LR, что вы можете сделать в Visual Studio Code, щелкнув индикатор LR в нижней части окна.

Запись в файлы

Пакет os также включает функции для записи файлов, как описано в таблице 22-5. Эти функции более сложны в использовании, чем их аналоги, связанные с чтением, поскольку требуется больше параметров конфигурации.

Таблица 22-5 Функция пакета os для записи файлов

Функция	**Описание**

Функция	Описание
WriteFile(name, slice, modePerms)	Эта функция создает файл с указанным именем, режимом и разрешениями и записывает содержимое указанного среза байтов. Если файл уже существует, его содержимое будет заменено байтовым срезом. Результатом является ошибка, сообщающая о любых проблемах с созданием файла или записью данных.
OpenFile(name, flag, modePerms)	Функция открывает файл с указанным именем, используя флаги для управления открытием файла. Если создается новый файл, применяются указанный режим и разрешения. Результатом является значение File, обеспечивающее доступ к содержимому файла, и ошибка, указывающая на проблемы с открытием файла.

Использование функции удобства записи

Функция WriteFile предоставляет удобный способ записи всего файла за один шаг и создаст файл, если он не существует. В листинге 22-11 показано использование функции WriteFile.

```
package main

import (
    "fmt"
    "time"
    "os"
)

func main() {

    total := 0.0
    for _, p := range Products {
        total += p.Price
    }

    dataStr := fmt.Sprintf("Time: %v, Total: $%.2f\n",
        time.Now().Format("Mon 15:04:05"), total)

    err := os.WriteFile("output.txt", []byte(dataStr), 0666)
    if (err == nil) {
        fmt.Println("Output file created")
    } else {
        Printfln("Error: %v", err.Error())
    }
}
```

Листинг 22-11 Запись файла в файл main.go в папку files

Первые два аргумента функции WriteFile — это имя файла и байтовый срез, содержащий данные для записи. Третий аргумент объединяет две настройки файла: режим файла и права доступа к файлу, как показано на рисунке 22-1.

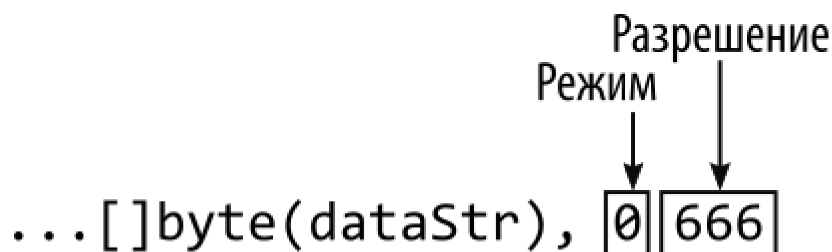

Рисунок 22-1 Режим файла и права доступа к файлу

Файловый режим используется для указания особых характеристик файла, но для обычных файлов используется нулевое значение, как в примере. Вы можете найти список значений файловых режимов и их настройки по адресу https://golang.org/pkg/io/fs/#FileMode, но они не требуются в большинстве проектов, и я не описываю их в этой книге.

Права доступа к файлам используются более широко и следуют стилю разрешений файлов UNIX, состоящему из трех цифр, которые устанавливают доступ для владельца файла, группы и других пользователей. Каждая цифра представляет собой сумму разрешений, которые должны быть предоставлены, где чтение имеет значение 4, запись имеет значение 2, а выполнение имеет значение 1. Эти значения складываются вместе, чтобы разрешение на чтение и запись файла устанавливается путем сложения значений 4 и 2 для получения разрешения 6. В листинге 22-11 я хочу создать файл, который могут читать и записывать все пользователи, поэтому я использую значение 6 для всех трех параметров, получая разрешение 666.

Функция WriteFile создает файл, если он еще не существует, что можно увидеть, скомпилировав и выполнив проект, который выдает следующий результат:

```
Username: Alice
Output file created
```

Изучите содержимое папки files, и вы увидите, что был создан файл с именем output.txt с содержимым, подобным следующему, хотя вы увидите другую отметку времени:

```
Time: Sun 07:05:06, Total: $81445.11
```

Если указанный файл уже существует, метод WriteFile заменяет его содержимое, в чем можно убедиться, повторно запустив скомпилированную программу. После завершения выполнения исходное содержимое будет заменено новой отметкой времени:

```
Time: Sun 07:08:21, Total: $81445.11
```

Использование файловой структуры для записи в файл

Функция OpenFile открывает файл и возвращает значение File. В отличие от функции Open, функция OpenFile принимает один или несколько флагов, указывающих, как следует открывать файл. Флаги определены как константы в пакете os, как описано в таблице 22-6. Следует соблюдать осторожность с этими флагами, не все из которых поддерживаются каждой операционной системой.

Таблица 22-6 Флаги открытия файлов

Функция	Описание
O_RDONLY	Этот флаг открывает файл только для чтения, чтобы его можно было читать, но не записывать.
O_WRONLY	Этот флаг открывает файл только для записи, чтобы в него можно было писать, но нельзя было читать.
O_RDWR	Этот флаг открывает файл для чтения и записи, чтобы в него можно было записывать и читать.
O_APPEND	Этот флаг будет добавлять записи в конец файла.
O_CREATE	Этот флаг создаст файл, если он не существует.
O_EXCL	Этот флаг используется в сочетании с O_CREATE для обеспечения создания нового файла. Если файл уже существует, этот флаг вызовет ошибку.
O_SYNC	Этот флаг включает синхронную запись, так что данные записываются на устройство хранения до возврата из функции/метода записи.
O_TRUNC	Этот флаг усекает существующее содержимое в файле.

Флаги объединяются с помощью побитового оператора ИЛИ, как показано в листинге 22-12.

```
package main

import (
    "fmt"
    "time"
    "os"
)

func main() {

    total := 0.0
    for _, p := range Products {
        total += p.Price
    }

    dataStr := fmt.Sprintf("Time: %v, Total: $%.2f\n",
        time.Now().Format("Mon 15:04:05"), total)

    file, err := os.OpenFile("output.txt",
        os.O_WRONLY | os.O_CREATE | os.O_APPEND, 0666)
    if (err == nil) {
        defer file.Close()
        file.WriteString(dataStr)
    } else {
        Printfln("Error: %v", err.Error())
    }
}
```

Листинг 22-12 Запись в файл в файле main.go в папке files

Я объединил флаг O_WRONLY, чтобы открыть файл для записи, файл O_CREATE для создания, если он еще не существует, и флаг O_APPEND для добавления любых записанных данных в конец файла.

Структура File определяет методы, описанные в таблице 22-7, для записи данных в файл после его открытия.

Таблица 22-7 Файловые методы для записи данных

Функция	Описание

Функция	Описание
Seek(offset, how)	Этот метод устанавливает местоположение для последующих операций.
Write(slice)	Этот метод записывает содержимое указанного среза байтов в файл. Результатом является количество записанных байтов и ошибка, указывающая на проблемы с записью данных.
WriteAt(slice, offset)	Этот метод записывает данные среза в указанное место и является аналогом метода ReadAt.
WriteString(str)	Этот метод записывает строку в файл. Это удобный метод, который преобразует строку в байтовый срез, вызывает метод Write и возвращает полученные результаты.

В листинге 22-12 я использовал удобный метод WriteString для записи строки в файл. Скомпилируйте и запустите проект, и вы увидите дополнительное сообщение в конце файла output.txt, как только программа будет завершена:

```
Time: Sun 07:08:21, Total: $81445.11
Time: Sun 07:49:14, Total: $81445.11
```

Запись данных JSON в файл

Структура File реализует интерфейс Writer, который позволяет использовать файл с функциями форматирования и обработки строк, описанными в предыдущих главах. Это также означает, что функции JSON, описанные в главе 21, можно использовать для записи данных JSON в файл, как показано в листинге 22-13.

```
package main

import (
    // "fmt"
    // "time"
    "os"
    "encoding/json"
)

func main() {

    cheapProducts := []Product {}
    for _, p := range Products {
```

```
        if (p.Price < 100) {
            cheapProducts = append(cheapProducts, p)
        }
    }

      file, err := os.OpenFile("cheap.json", os.O_WRONLY |
os.O_CREATE, 0666)
    if (err == nil) {
        defer file.Close()
        encoder := json.NewEncoder(file)
        encoder.Encode(cheapProducts)
    } else {
        Printfln("Error: %v", err.Error())
    }
}
```

Листинг 22-13 Запись данных JSON в файл в файле main.go в папке files

В этом примере выбираются значения Product со значением Price меньше 100, помещаются в срез и используется JSON Encoder для записи этого среза в файл с именем cheap.json. Скомпилируйте и запустите проект, и как только выполнение будет завершено, вы увидите файл с именем cheap.json в папке файлов со следующим содержимым, которое я отформатировал для размещения на странице:

```
[{"Name":"Lifejacket","Category":"Watersports","Price":49.95},
 {"Name":"Soccer Ball","Category":"Soccer","Price":19.5},
 {"Name":"Corner Flags","Category":"Soccer","Price":34.95},
 {"Name":"Thinking Cap","Category":"Chess","Price":16},
 {"Name":"Unsteady Chair","Category":"Chess","Price":75},
 {"Name":"Hat","Category":"Skiing","Price":10},
 {"Name":"Gloves","Category":"Skiing","Price":40.2}]
```

Использование удобных функций для создания новых файлов

Хотя можно использовать функцию OpenFile для создания новых файлов, как показано в предыдущем разделе, пакет os также предоставляет некоторые полезные удобные функции, как описано в таблице 22-8.

Таблица 22-8 Функции пакета os для создания файлов

Функция	Описание
Create(name)	Эта функция эквивалентна вызову OpenFile с флагами O_RDWR, O_CREATE и O_TRUNC. Результатом является File, который можно использовать для чтения и записи, и error, которая используется для обозначения проблем при создании файла. Обратите внимание, что эта комбинация флагов означает, что если файл с указанным именем существует, он будет открыт, а его содержимое будет удалено.
CreateTemp(dirName, fileName)	Эта функция создает новый файл в каталоге с указанным именем. Если имя представляет собой пустую строку, то используется системный временный каталог, полученный с помощью функции TempDir (описано в таблице 22-9). Файл создается с именем, которое содержит случайную последовательность символов, как показано в тексте после таблицы. Файл открывается с флагами O_RDWR, O_CREATE и O_EXCL. Файл не удаляется при закрытии.

Функция CreateTemp может быть полезна, но важно понимать, что цель этой функции — генерировать случайное имя файла и что во всех других отношениях создаваемый файл является обычным файлом. Созданный файл не удаляется автоматически и останется на устройстве хранения после выполнения приложения.

В листинге 22-14 показано использование функции CreateTemp и показано, как можно управлять расположением рандомизированного компонента имени.

```
package main

import (
    // "fmt"
    // "time"
    "os"
    "encoding/json"
)

func main() {

    cheapProducts := []Product {}
    for _, p := range Products {
        if (p.Price < 100) {
            cheapProducts = append(cheapProducts, p)
        }
    }
```

```
    file, err := os.CreateTemp(".", "tempfile-*.json")
    if (err == nil) {
        defer file.Close()
        encoder := json.NewEncoder(file)
        encoder.Encode(cheapProducts)
    } else {
        Printfln("Error: %v", err.Error())
    }
}
```

Листинг 22-14 Создание временного файла в файле main.go в папке files

Местоположение временного файла указывается с точкой, что означает текущий рабочий каталог. Как отмечено в таблице 22-8, если используется пустая строка, то файл будет создан во временном каталоге по умолчанию, который получается с помощью функции `TempDir`, описанной в таблице 22-9. Имя файла может включать звездочку (`*`), и если она присутствует, ее заменяет случайная часть имени файла. Если имя файла не содержит звездочки, то в конце имени будет добавлена случайная часть имени файла.

Скомпилируйте и запустите проект, и после завершения выполнения вы увидите новый файл в папке `files`. Файл в моем проекте называется `tempfile-1732419518.json`, но ваше имя файла будет другим, и вы будете видеть новый файл и уникальное имя каждый раз при выполнении программы.

Работа с путями к файлам

В примерах, приведенных до сих пор в этой главе, использовались файлы, находящиеся в текущем рабочем каталоге, который обычно является местом, из которого запускается скомпилированный исполняемый файл. Если вы хотите читать и записывать файлы в других местах, вы должны указать пути к файлам. Проблема в том, что не все операционные системы, поддерживающие Go, одинаково поддерживают экспресс-пути к файлам. Например, путь к файлу с именем `mydata.json` в моем домашнем каталоге в системе Linux может быть выражен следующим образом:

```
/home/adam/mydata.json
```

Обычно я развертываю свои проекты в Linux, но предпочитаю разрабатывать в Windows, где путь к тому же самому в моем домашнем каталоге выражается следующим образом:

```
C:\Users\adam\mydata.json
```

Windows более гибкая, чем можно было бы ожидать, а низкоуровневые API-интерфейсы, вызываемые функциями Go, такими как `OpenFile`, не зависят от разделителей файлов и принимают как обратную, так и прямую косую черту. Это означает, что я могу указать путь к файлу как `c:/users/adam/mydata.json` или даже `/users/adam/mydata.json` при написании кода Go, и Windows все равно откроет файл правильно. Но разделитель файлов — это только одно из отличий между платформами. Тома обрабатываются по-разному, и существуют разные места для хранения файлов по умолчанию. Так, например, я мог бы прочитать свой гипотетический файл данных, используя `/home/adam.mydata.json` или `/users/mydata.json`, но правильный выбор будет зависеть от того, какую операционную систему я использую. А по мере того, как Go портируется на большее количество платформ, будет более широкий диапазон возможных местоположений. Для решения этой проблемы пакет `os` предоставляет набор функций, которые возвращают пути к общим местоположениям, как описано в таблице 22-9.

Таблица 22-9 Общие функции определения местоположения, определенные пакетом os

Функция	Описание
`Getwd()`	Эта функция возвращает текущий рабочий каталог, выраженный в виде строки, и `error`, указывающую на проблемы с получением значения.
`UserHomeDir()`	Эта функция возвращает домашний каталог пользователя и ошибку, указывающую на проблемы с получением пути.
`UserCacheDir()`	Эта функция возвращает каталог по умолчанию для пользовательских кэшированных данных и ошибку, указывающую на проблемы с получением пути.
`UserConfigDir()`	Эта функция возвращает каталог по умолчанию для пользовательских данных конфигурации и ошибку, указывающую на проблемы с получением пути.
`TempDir()`	Эта функция возвращает каталог по умолчанию для временных файлов и ошибку, указывающую на проблемы с получением пути.

Получив путь, вы можете обращаться с ним как со строкой и просто добавлять к нему дополнительные сегменты или, во избежание ошибок, использовать функции, предоставляемые пакетом `path/filepath` для манипулирования путями, наиболее полезные из которых описаны в таблице 22-10.

Таблица 22-10 Функции path/filepath для путей

Функция	Описание
`Abs(path)`	Эта функция возвращает абсолютный путь, что полезно, если у вас есть относительный путь, например имя файла.
`IsAbs(path)`	Эта функция возвращает `true`, если указанный путь является абсолютным.
`Base(path)`	Эта функция возвращает последний элемент пути.
`Clean(path)`	Эта функция очищает строки пути, удаляя повторяющиеся разделители и относительные ссылки.
`Dir(path)`	Эта функция возвращает все элементы пути, кроме последнего.
`EvalSymlinks(path)`	Эта функция оценивает символическую ссылку и возвращает результирующий путь.
`Ext(path)`	Эта функция возвращает расширение файла из указанного пути, который считается суффиксом после последней точки в строке пути.
`FromSlash(path)`	Эта функция заменяет каждую косую черту символом разделителя файлов платформы.
`ToSlash(path)`	Эта функция заменяет разделитель файлов платформы косой чертой.
`Join(...elements)`	Эта функция объединяет несколько элементов, используя файловый разделитель платформы.
`Match(pattern, path)`	Эта функция возвращает значение `true`, если путь соответствует указанному шаблону.
`Split(path)`	Эта функция возвращает компоненты по обе стороны от конечного разделителя пути в указанном пути.
`SplitList(path)`	Эта функция разбивает путь на компоненты, которые возвращаются в виде среза строки.
`VolumeName(path)`	Эта функция возвращает компонент тома указанного пути или пустую строку, если путь не содержит тома.

В листинге 22-15 показан процесс запуска с путем, возвращаемым одной из вспомогательных функций, описанных в таблице 22-10, и манипулирования им с помощью функций из таблицы 22-9.

```
package main

import (
    // "fmt"
    // "time"
    "os"
    //"encoding/json"
    "path/filepath"
)

func main() {
    path, err := os.UserHomeDir()
    if (err == nil) {
                          path   =   filepath.Join(path,   "MyApp",
"MyTempFile.json")
    }

    Printfln("Full path: %v", path)
    Printfln("Volume name: %v", filepath.VolumeName(path))
    Printfln("Dir component: %v", filepath.Dir(path))
    Printfln("File component: %v", filepath.Base(path))
    Printfln("File extension: %v", filepath.Ext(path))
}
```

Листинг 22-15 Работа с путем в файле main.go в папке files

Этот пример начинается с пути, возвращаемого функцией `UserHomeDir`, используется функция `Join` для добавления дополнительных сегментов, а затем записываются разные части пути. Результаты, которые вы получите, будут зависеть от вашего имени пользователя и вашей платформы. Вот вывод, который я получил на своем компьютере с Windows:

```
Username: Alice
Full path: C:\Users\adam\MyApp\MyTempFile.json
Volume name: C:
Dir component: C:\Users\adam\MyApp
File component: MyTempFile.json
File extension: .json
```

Вот результат, который я получил на своей тестовой машине с Ubuntu:

```
Username: Alice
Full path: /home/adam/MyApp/MyTempFile.json
Volume name:
Dir component: /home/adam/MyApp
File component: MyTempFile.json
File extension: .json
```

Управление файлами и каталогами

Функции, описанные в предыдущем разделе, обрабатывают пути, но это всего лишь строки. Когда я добавлял сегменты к пути в листинге 22-15, результатом была просто еще одна строка, а в файловой системе не было соответствующих изменений. Для внесения таких изменений пакет `os` предоставляет функции, описанные в таблице 22-11.

Таблица 22-11 Функции пакета os для управления файлами и каталогами

Функция	Описание
`Chdir(dir)`	Эта функция изменяет текущий рабочий каталог на указанный каталог. Результатом является `error`, указывающая на проблемы с внесением изменений.
`Mkdir(name, modePerms)`	Эта функция создает каталог с указанным именем и режимом/разрешениями. Результатом является `error`, равная нулю, если каталог создан, или описывающая проблему, если она возникает.
`MkdirAll(name, modePerms)`	Эта функция выполняет ту же задачу, что и `Mkdir`, но создает любые родительские каталоги по указанному пути.
`MkdirTemp(parentDir, name)`	Эта функция похожа на `CreateTemp`, но создает каталог, а не файл. Случайная строка добавляется в конец указанного имени или вместо звездочки, и новый каталог создается в указанном родительском каталоге. Результатами являются имя каталога и `error`, указывающая на проблемы.
`Remove(name)`	Эта функция удаляет указанный файл или каталог. Результатом является `error`, которая описывает любые возникающие проблемы.
`RemoveAll(name)`	Эта функция удаляет указанный файл или каталог. Если имя указывает каталог, то все содержащиеся в нем дочерние элементы также удаляются. Результатом является `error`, которая описывает любые возникающие проблемы.
`Rename(old, new)`	Эта функция переименовывает указанный файл или папку. Результатом является ошибка, которая описывает любые возникающие проблемы.

Функция	Описание
Symlink(old, new)	Эта функция создает символическую ссылку на указанный файл. Результатом является ошибка, которая описывает любые возникающие проблемы.

В листинге 22-16 функция MkdirAll используется для обеспечения создания каталогов, необходимых для пути к файлу, чтобы при попытке создать файл не возникала ошибка.

```
package main

import (
    // "fmt"
    // "time"
    "os"
    "encoding/json"
    "path/filepath"
)

func main() {
    path, err := os.UserHomeDir()
    if (err == nil) {
                  path   =   filepath.Join(path,   "MyApp",
"MyTempFile.json")
    }

    Printfln("Full path: %v", path)

    err = os.MkdirAll(filepath.Dir(path), 0766)
    if (err == nil) {
              file,  err  :=  os.OpenFile(path,  os.O_CREATE  |
os.O_WRONLY, 0666)
        if (err == nil) {
            defer file.Close()
            encoder := json.NewEncoder(file)
            encoder.Encode(Products)
        }
    }
    if (err != nil) {
        Printfln("Error %v", err.Error())
    }
}
```

Листинг 22-16 Создание каталогов в файле main.go в папке files

Чтобы убедиться, что каталоги в моем пути существуют, я использую функцию `filepath.Dir` и передаю результат функции `os.MkdirAll`. Затем я могу создать файл, используя функцию `OpenFile` и указав флаг `O_CREATE`. Я использую `File` как `Writer` для JSON `Encoder` и записываю содержимое среза `Product`, определенного в листинге 22-3, в новый файл. Отложенный оператор `Close` закрывает файл. Скомпилируйте и выполните проект, и вы увидите, что в вашей домашней папке был создан каталог с именем `MyApp`, содержащий файл JSON с именем `MyTempFile.json`. Файл будет содержать следующие данные JSON, которые я отформатировал так, чтобы они поместились на странице:

```
[{"Name":"Lifejacket","Category":"Watersports","Price":49.95},
 {"Name":"Soccer Ball","Category":"Soccer","Price":19.5},
 {"Name":"Corner Flags","Category":"Soccer","Price":34.95},
 {"Name":"Thinking Cap","Category":"Chess","Price":16},
 {"Name":"Unsteady Chair","Category":"Chess","Price":75},
 {"Name":"Hat","Category":"Skiing","Price":10},
 {"Name":"Gloves","Category":"Skiing","Price":40.2}]
```

Изучение файловой системы

Если вы знаете расположение нужных вам файлов, вы можете просто создать пути, используя функции, описанные в предыдущем разделе, и использовать их для открытия файлов. Если ваш проект основан на обработке файлов, созданных другим процессом, вам необходимо изучить файловую систему. Пакет os предоставляет функцию, описанную в таблице 22-12.

Таблица 22-12 Функция пакета os для вывода каталогов

Функция	**Описание**
`ReadDir(name)`	Эта функция читает указанный каталог и возвращает срез `DirEntry`, каждый из которых описывает элемент в каталоге.

Результатом функции `ReadDir` является срез значений, которые реализуют интерфейс `DirEntry`, определяющий методы, описанные в таблице 22-13.

Таблица 22-13 Методы, определенные интерфейсом DirEntry

Функция	Описание
`Name()`	Этот метод возвращает имя файла или каталога, описанного значением `DirEntry`.
`IsDir()`	Этот метод возвращает значение `true`, если значение `DirEntry` представляет каталог.
`Type()`	Этот метод возвращает значение `FileMode`, которое является псевдонимом `uint32`, который описывает файл больше и права доступа к файлу или каталогу, представленные значением `DirEntry`.
`Info()`	Этот метод возвращает значение `FileInfo`, которое предоставляет дополнительные сведения о файле или каталоге, представленном значением `DirEntry`.

Интерфейс `FileInfo`, являющийся результатом метода `Info`, используется для получения сведений о файле или каталоге. Наиболее полезные методы, определенные интерфейсом `FileInfo`, описаны в таблице 22-14.

Таблица 22-14 Полезные методы, определенные интерфейсом FileInfo

Функция	Описание
`Name()`	Этот метод возвращает строку, содержащую имя файла или каталога.
`Size()`	Этот метод возвращает размер файла, выраженный в виде значения `int64`.
`Mode()`	Этот метод возвращает режим файла и настройки разрешений для файла или каталога.
`ModTime()`	Этот метод возвращает время последнего изменения файла или каталога.

Вы также можете получить значение `FileInfo` для одного файла, используя функцию, описанную в таблице 22-15.

Таблица 22-15 Функция пакета os для проверки файла

Функция	Описание
`Stat(path)`	Эта функция принимает строку пути. Он возвращает значение `FileInfo`, описывающее файл, и `error`, указывающую на проблемы с проверкой файла.

В листинге 22-17 используется функция `ReadDir` для перечисления содержимого папки проекта.

```
package main
```

```
import (
    // "fmt"
    // "time"
    "os"
    //"encoding/json"
    //"path/filepath"
)

func main() {
    path, err := os.Getwd()
    if (err == nil) {
        dirEntries, err := os.ReadDir(path)
        if (err == nil) {
            for _, dentry := range dirEntries {
                       Printfln("Entry name: %v, IsDir: %v",
dentry.Name(), dentry.IsDir())
            }
        }
    }
    if (err != nil) {
        Printfln("Error %v", err.Error())
    }
}
```

Листинг 22-17 Перечисление файлов в файле main.go в папке files

Цикл `for` используется для перечисления значений `DirEntry`, возвращаемых функцией `ReadDir`, и записываются результаты функций `Name` и `IsDir`. Скомпилируйте и выполните проект, и вы увидите вывод, аналогичный следующему, с учетом различий в именах файлов, созданных с помощью функции `CreateTemp`:

```
Username: Alice
Entry name: cheap.json, IsDir: false
Entry name: config.go, IsDir: false
Entry name: config.json, IsDir: false
Entry name: go.mod, IsDir: false
Entry name: main.go, IsDir: false
Entry name: output.txt, IsDir: false
Entry name: product.go, IsDir: false
Entry name: tempfile-1732419518.json, IsDir: false
```

Определение существования файла

Пакет os определяет функцию с именем IsNotExist, принимает ошибку и возвращает true, если это означает, что ошибка указывает на то, что файл не существует, как показано в листинге 22-18.

```
package main

import (
    // "fmt"
    // "time"
    "os"
    // "encoding/json"
    // "path/filepath"
)

func main() {

          targetFiles := []string { "no_such_file.txt",
"config.json" }
    for _, name := range targetFiles {
        info, err := os.Stat(name)
        if os.IsNotExist(err) {
            Printfln("File does not exist: %v", name)
        } else if err != nil  {
            Printfln("Other error: %v", err.Error())
        } else {
                 Printfln("File %v, Size: %v", info.Name(),
info.Size())
        }
    }
}
```

Листинг 22-18 Проверка существования файла в файле main.go в папке files

Ошибка, возвращаемая функцией Stat, передается функции IsNotExist, что позволяет идентифицировать несуществующие файлы. Скомпилируйте и запустите проект, и вы получите следующий вывод:

```
Username: Alice
File does not exist: no_such_file.txt
File config.json, Size: 262
```

Поиск файлов с помощью шаблона

Пакет path/filepath определяет функцию Glob, которая возвращает все имена в каталоге, соответствующие указанному шаблону. Функция описана в таблице 22-16 для быстрого ознакомления.

Таблица 22-16 Функция path/filepath для поиска файлов с шаблоном

Функция	Описание
Match(pattern, name)	Эта функция сопоставляет один путь с шаблоном. Результатом является bool значение, указывающее на наличие совпадения, и error, указывающая на проблемы с шаблоном или с выполнением совпадения.
Glob(pathPatten)	Эта функция находит все файлы, соответствующие указанному шаблону. Результатом является string срез, содержащий совпавшие пути и ошибку, указывающую на проблемы с выполнением поиска.

Шаблоны, используемые функциями в таблице 22-16, используют синтаксис, описанный в таблице 22-17.

Таблица 22-17 Синтаксис шаблона поиска для функций path/filepath

Термин	Описание
*	Этот термин соответствует любой последовательности символов, кроме разделителя пути.
?	Этот термин соответствует любому одиночному символу, за исключением разделителя пути.
[a-Z]	Этот термин соответствует любому символу в указанном диапазоне.

В листинге 22-19 функция Glob используется для получения путей файлов JSON в текущем рабочем каталоге.

```
package main

import (
    // "fmt"
    // "time"
    "os"
    // "encoding/json"
    "path/filepath"
)

func main() {
```

```
    path, err := os.Getwd()
    if (err == nil) {
            matches, err := filepath.Glob(filepath.Join(path,
"*.json"))
        if (err == nil) {
            for _, m := range matches {
                Printfln("Match: %v", m)
            }
        }
    }

    if (err != nil) {
        Printfln("Error %v", err.Error())
    }
}
```

Листинг 22-19 Расположение файлов в файле main.go в папке files

Я создаю шаблон поиска с помощью функций `Getwd` и `Join` и записываю пути, которые поворачиваются с помощью функции `Glob`. Скомпилируйте и выполните проект, и вы увидите следующий вывод, хотя и отражающий расположение папки вашего проекта:

```
Username: Alice
Match: C:\files\cheap.json
Match: C:\files\config.json
Match: C:\files\tempfile-1732419518.json
```

Обработка всех файлов в каталоге

Альтернативой использованию шаблонов является перечисление всех файлов в определенном месте, что можно сделать с помощью функции, описанной в таблице 22-18, которая определена в пакете `path/filepath`.

Таблица 22-18 Функция, предоставляемая пакетом path/filepath

Функция	Описание
`WalkDir(directory, func)`	Эта функция вызывает указанную функцию для каждого файла и каталога в указанном каталоге.

Функция обратного вызова, вызываемая `WalkDir`, получает строку, содержащую путь, значение `DirEntry`, предоставляющее сведения о

файле или каталоге, и ошибку, указывающую на проблемы с доступом к этому файлу или каталогу. Результатом функции обратного вызова является ошибка, которая не позволяет функции WalkDir войти в текущий каталог, возвращая специальное значение SkipDir. В листинге 22-20 показано использование функции WalkDir.

```
package main

import (
    // "fmt"
    // "time"
    "os"
    //"encoding/json"
    "path/filepath"
)

func callback(path string, dir os.DirEntry, dirErr error)
(err error) {
    info, _ := dir.Info()
    Printfln("Path %v, Size: %v", path, info.Size())
    return
}

func main() {

    path, err := os.Getwd()
    if (err == nil) {
        err = filepath.WalkDir(path, callback)
    } else {
        Printfln("Error %v", err.Error())
    }
}
```

Листинг 22-20 рогулка по каталогу в файле main.go в папке files

В этом примере функция WalkDir используется для перечисления содержимого текущего рабочего каталога и записи пути и размера каждого найденного файла. Скомпилируйте и запустите проект, и вы увидите вывод, подобный следующему:

```
Username: Alice
Path C:\files, Size: 4096
```

```
Path C:\files\cheap.json, Size: 384
Path C:\files\config.json, Size: 262
Path C:\files\go.mod, Size: 28
Path C:\files\main.go, Size: 467
Path C:\files\output.txt, Size: 74
Path C:\files\product.go, Size: 679
Path C:\files\readconfig.go, Size: 870
Path C:\files\tempfile-1732419518.json, Size: 384
```

Резюме

В этой главе я описываю поддержку стандартной библиотеки для работы с файлами. Я описываю удобные функции для чтения и записи файлов, объясняю использование структуры `File` и демонстрирую, как исследовать файловую систему и управлять ею. В следующей главе я объясню, как создавать и использовать HTML и текстовые шаблоны.

23. Использование HTML и текстовых шаблонов

В этой главе я описываю пакеты стандартных библиотек, которые используются для создания HTML и текстового содержимого из шаблонов. Эти пакеты шаблонов полезны при создании больших объемов контента и имеют обширную поддержку для создания динамического контента. В таблице 223-1 HTML и текстовые шаблоны представлены в контексте.

Таблица 23-1 Помещение HTML и текстовых шаблонов в контекст

Вопрос	Ответ
Кто они такие?	Эти шаблоны позволяют динамически генерировать HTML и текстовый контент из значений данных Go.
Почему они полезны?	Шаблоны полезны, когда требуется большое количество контента, так что определение контента в виде строк было бы неуправляемым.
Как они используются?	Шаблоны представляют собой HTML или текстовые файлы, снабженные инструкциями для механизма обработки шаблонов. При отображении шаблона инструкции обрабатываются для создания HTML или текстового содержимого.
Есть ли подводные камни или ограничения?	Синтаксис шаблона нелогичен и не проверяется компилятором Go. Это означает, что необходимо соблюдать осторожность, чтобы использовать правильный синтаксис, что может быть разочаровывающим процессом.
Есть ли альтернативы?	Шаблоны необязательны, и с помощью строк можно создавать меньшее количество контента.

Таблица 23-2 суммирует главу.

Таблица 23-2 Краткое содержание главы

Проблема	Решение	Листинг
Создать HTML-документ	Определите шаблон HTML с действиями, которые включают значения данных в выходные данные. Загрузите и выполните шаблоны, предоставив данные для действий.	6–10

Проблема	**Решение**	**Листинг**
Перечислить загруженные шаблоны	Перечислите результаты метода `Templates`.	11
Найти конкретный шаблон	Используйте метод `Lookup`.	12
Создать динамический контент	Используйте действие шаблона.	13, 21
Форматировать значение данных	Используйте функции форматирования.	14–16
Подавить пробелы	Добавьте дефисы в шаблон.	17–19
Обработать срез	Используйте функции среза.	22
Условное выполнение содержимого шаблона	Используйте условные действия и функции.	23-24
Создать вложенный шаблон	Используйте действия `define` и `template`.	25–27
Определить шаблон по умолчанию	Используйте действия `block` и `template`.	28–30
Создание функций для использования в шаблоне	Определите функции шаблона.	31–32, 35, 36
Отключить кодирование результатов функции	Возвращает один из псевдонимов типов, определенных пакетом `html/template`.	33, 34
Сохранение значений данных для последующего использования в шаблоне	Определите переменные шаблона.	37–40
Создать текстовый документ	Используйте пакет `text/template`.	41, 42

Подготовка к этой главе

Чтобы подготовиться к этой главе, откройте новую командную строку, перейдите в удобное место и создайте каталог с именем `htmltext`. Запустите команду, показанную в листинге 23-1, чтобы создать файл модуля.

Подсказка

> Вы можете загрузить пример проекта для этой главы — и для всех остальных глав этой книги — с `https://github.com/apress/pro-go`. См. Главу 2 о том, как получить помощь, если у вас возникнут проблемы с запуском примеров.

```
go mod init htmltext
```

Листинг 23-1 Инициализация модуля

Добавьте файл с именем `printer.go` в папку `htmltext` с содержимым, показанным в листинге `23-2`.

```
package main

import "fmt"

func Printfln(template string, values ...interface{}) {
    fmt.Printf(template + "\n", values...)
}
```

Листинг 23-2 Содержимое файла printer.go в папке htmltext

Добавьте файл с именем `product.go` в папку `htmltext` с содержимым, показанным в листинге 23-3.

```
package main

type Product struct {
    Name, Category string
    Price float64
}

var Kayak = Product {
    Name: "Kayak",
    Category: "Watersports",
    Price: 279,
}

var Products = []Product {
    { "Kayak", "Watersports", 279 },
    { "Lifejacket", "Watersports", 49.95 },
    { "Soccer Ball", "Soccer", 19.50 },
```

```
    { "Corner Flags", "Soccer", 34.95 },
    { "Stadium", "Soccer", 79500 },
    { "Thinking Cap", "Chess", 16 },
    { "Unsteady Chair", "Chess", 75 },
    { "Bling-Bling King", "Chess", 1200 },
}

func (p *Product) AddTax() float64 {
    return p.Price * 1.2
}

func (p * Product) ApplyDiscount(amount float64) float64 {
    return p.Price - amount
}
```

Листинг 23-3 Содержимое файла product.go в папке htmltext

Добавьте файл с именем `main.go` в папку `htmltext` с содержимым, показанным в листинге 23-4.

```
package main

func main() {
    for _, p := range Products {
        Println("Product: %v, Category: %v, Price: $%.2f",
            p.Name, p.Category, p.Price)
    }
}
```

Листинг 23-4 Содержимое файла main.go в папке usingstrings

Используйте командную строку для запуска команды, показанной в листинге 23-5, в папке `htmltext`.

```
go run .
```

Листинг 23-5 Запуск примера проекта

Код будет скомпилирован и выполнен, что приведет к следующему выводу:

```
Product: Kayak, Category: Watersports, Price: $279.00
Product: Lifejacket, Category: Watersports, Price: $49.95
Product: Soccer Ball, Category: Soccer, Price: $19.50
```

```
Product: Corner Flags, Category: Soccer, Price: $34.95
Product: Stadium, Category: Soccer, Price: $79500.00
Product: Thinking Cap, Category: Chess, Price: $16.00
Product: Unsteady Chair, Category: Chess, Price: $75.00
Product: Bling-Bling King, Category: Chess, Price: $1200.00
```

Создание HTML-шаблонов

Пакет `html/template` обеспечивает поддержку создания шаблонов, которые обрабатываются с использованием структуры данных для создания динамического вывода HTML. Создайте папку `htmltext/templates` и добавьте в нее файл с именем `template.html` с содержимым, показанным в листинге 23-6.

> **Примечание**
> Примеры в этой главе создают фрагменты HTML. См. третью часть для примеров, которые создают полные HTML-документы.

```
<h1>Template Value: {{ . }}</h1>
```

Листинг 23-6 Содержимое файла template.html в папке templates

Шаблоны содержат статическое содержимое, смешанное с выражениями, заключенными в двойные фигурные скобки, которые называются *действиями*. Шаблон в листинге 23-6 использует самое простое действие — точку (символ `.`), которое выводит данные, используемые для выполнения шаблона, которые я объясню в следующем разделе.

Проект может содержать несколько файлов шаблонов. Добавьте файл с именем `extras.html` в папку шаблонов с содержимым, показанным в листинге 23-7.

```
<h1>Extras Template Value: {{ . }}</h1>
```

Листинг 23-7 Содержимое файла extras.html в папке templates

Новый шаблон использует то же действие, что и предыдущий пример, но имеет другое статическое содержимое, чтобы было ясно, какой шаблон был выполнен в следующем разделе. После того, как я

описал основные приемы использования шаблонов, я представлю более сложные шаблонные действия.

Загрузка и выполнение шаблонов

Использование шаблонов — это двухэтапный процесс. Сначала файлы шаблонов загружаются и обрабатываются для создания значений `Template`. Таблица 23-3 описывает функции, используемые для загрузки файлов шаблонов.

Таблица 23-3 Функции html/template для загрузки файлов шаблонов

Функция	Описание
`ParseFiles(...files)`	Эта функция загружает один или несколько файлов, которые указаны по имени. Результатом является `Template`, который можно использовать для создания контента, и `error`, сообщающая о проблемах с загрузкой шаблонов.
`ParseGlob(pattern)`	Эта функция загружает один или несколько файлов, выбранных с помощью шаблона. Результатом является `Template`, который можно использовать для создания контента, и `error`, сообщающая о проблемах с загрузкой шаблонов.

Если вы последовательно называете файлы шаблонов, вы можете использовать функцию `ParseGlob` для их загрузки с помощью простого шаблона. Если вам нужны определенные файлы или файлы не имеют последовательных имен, вы можете указать отдельные файлы с помощью функции `ParseFiles`.

После загрузки файлов шаблонов значение `Template`, возвращаемое функциями из таблицы 23-3, используется для выбора шаблона и его выполнения для создания контента с использованием методов, описанных в таблице 23-4.

Таблица 23-4 Шаблонные методы для выбора и выполнения шаблонов

Функция	Описание
`Templates()`	Эта функция возвращает срез, содержащий указатели на загруженные значения `Template`.
`Lookup(name)`	Эта функция возвращает `*Template` для указанного загруженного шаблона.
`Name()`	Этот метод возвращает имя `Template`.

Функция	Описание
Execute(writer, data)	Эта функция выполняет Template, используя указанные данные, и записывает вывод в указанный Writer.
ExecuteTemplate(writer, templateName, data)	Эта функция выполняет шаблон с указанным именем и данными и записывает вывод в указанный Writer.

В листинге 23-8 я загружаю и выполняю шаблон.

```
package main

import (
    "html/template"
    "os"
)

func main() {
    t, err := template.ParseFiles("templates/template.html")
    if (err == nil) {
        t.Execute(os.Stdout, &Kayak)
    } else {
        Printfln("Error: %v", err.Error())
    }
}
```

Листинг 23-8 Загрузка и выполнение шаблона в файле main.go в папке htmltext

Я использовал функцию ParseFiles для загрузки одного шаблона. Результатом функции ParseFiles является Template, для которого я вызвал метод Execute, указав стандартный вывод в качестве Writer и Product в качестве данных для обработки шаблона.

Содержимое файла template.html обрабатывается, и содержащееся в нем действие выполняется, вставляя аргумент данных, переданный методу Execute, в выходные данные, отправляемые в Writer. Скомпилируйте и запустите проект, и вы увидите следующий вывод:

```
<h1>Template Value: {Kayak Watersports 279}</h1>
```

Выходные данные шаблона включают строковое представление структуры Product. Далее в этой главе я опишу более полезные способы создания содержимого из значений структуры.

Загрузка нескольких шаблонов

Существует два подхода к работе с несколькими шаблонами. Первый заключается в создании отдельного значения Template для каждого из них и выполнении их отдельно, как показано в листинге 23-9.

```
package main

import (
    "html/template"
    "os"
)

func main() {
                                    t1,         err1         :=
template.ParseFiles("templates/template.html")
    t2, err2 := template.ParseFiles("templates/extras.html")
    if (err1 == nil && err2 == nil) {
        t1.Execute(os.Stdout, &Kayak)
        os.Stdout.WriteString("\n")
        t2.Execute(os.Stdout, &Kayak)
    } else {
        Printfln("Error: %v %v", err1.Error(), err2.Error())
    }
}
```

Листинг 23-9 Использование отдельных шаблонов в файле main.go в папке htmltext

Обратите внимание, что я написал символ новой строки между выполнением шаблонов. Вывод шаблона — это именно то, что содержится в файле. Ни один из файлов в каталоге templates не содержит символа новой строки, поэтому мне пришлось добавить его в вывод, чтобы отделить содержимое, создаваемое шаблонами. Скомпилируйте и запустите проект, и вы получите следующий вывод:

```
<h1>Template Value: {Kayak Watersports 279}</h1>
<h1>Extras Template Value: {Kayak Watersports 279}</h1>
```

Использование отдельных значений Template — самый простой подход, но альтернативой является загрузка нескольких файлов в одно значение Template, а затем указание имени шаблона, который вы хотите выполнить, как показано в листинге 23-10.

```
package main

import (
    "html/template"
    "os"
)

func main() {
                            allTemplates,     err1     :=
template.ParseFiles("templates/template.html",
        "templates/extras.html")
    if (err1 == nil) {
                      allTemplates.ExecuteTemplate(os.Stdout,
"template.html", &Kayak)
        os.Stdout.WriteString("\n")
                      allTemplates.ExecuteTemplate(os.Stdout,
"extras.html", &Kayak)
    } else {
        Printfln("Error: %v %v", err1.Error())
    }
}
```

Листинг 23-10 Использование комбинированного шаблона в файле main.go в папке htmltext

Когда несколько файлов загружаются с `ParseFiles`, результатом является значение `Template`, для которого можно вызвать метод `ExecuteTemplate` для выполнения указанного шаблона. Имя файла используется в качестве имени шаблона, что означает, что шаблоны в этом примере называются `template.html` и `extras.html`.

> **Примечание**
> Вы можете вызвать метод `Execute` для `Template`, возвращаемого функцией `ParseFiles` или `ParseGlob`, и первый загруженный шаблон будет выбран и использован для создания выходных данных. Будьте осторожны при использовании функции `ParseGlob`, потому что первый загруженный шаблон — и, следовательно, шаблон, который будет выполнен — может оказаться не тем файлом, который вы ожидаете.

Вам не обязательно использовать расширения файлов для файлов шаблонов, но я сделал это, чтобы отличить шаблоны, созданные в этом разделе, от текстовых шаблонов, созданных позже в этой главе. Скомпилируйте и запустите проект, и вы увидите следующий вывод:

```
<h1>Template Value: {Kayak Watersports 279}</h1>
<h1>Extras Template Value: {Kayak Watersports 279}</h1>
```

Загрузка нескольких шаблонов позволяет определить содержимое в нескольких файлах, чтобы один шаблон мог полагаться на содержимое, сгенерированное из другого, что я продемонстрирую в разделе «Определение блоков шаблона» далее в этой главе.

Перечисление загруженных шаблонов

Может быть полезно перечислить загруженные шаблоны, особенно при использовании функции `ParseGlob`, чтобы убедиться, что все ожидаемые файлы обнаружены. В листинге 23-11 используется метод `Templates` для получения списка шаблонов и метод `Name` для получения имени каждого из них.

```
package main

import (
    "html/template"
    //"os"
)

func main() {
    allTemplates, err := template.ParseGlob("templates/*.html")
    if (err == nil) {
        for _, t := range allTemplates.Templates() {
            Printfln("Template name: %v", t.Name())
        }
    } else {
        Printfln("Error: %v %v", err.Error())
    }
}
```

Листинг 23-11 Перечисление загруженных шаблонов в файле main.go в папке htmltext

Шаблон, переданный функции ParseGlob, выбирает все файлы с расширением html в папке templates. Скомпилируйте и запустите проект, и вы увидите список загруженных шаблонов:

```
Template name: extras.html
Template name: template.html
```

Поиск определенного шаблона

Альтернативой указанию имени является использование метода Lookup для выбора шаблона, который полезен, когда вы хотите передать шаблон в качестве аргумента функции, как показано в листинге 23-12.

```
package main

import (
    "html/template"
    "os"
)

func Exec(t *template.Template) error {
    return t.Execute(os.Stdout, &Kayak)
}

func main() {
    allTemplates, err := template.ParseGlob("templates/*.html")
    if (err == nil) {
        selectedTemplated := allTemplates.Lookup("template.html")
        err = Exec(selectedTemplated)
    }
    if (err != nil) {
        Printfln("Error: %v %v", err.Error())
    }
}
```

Листинг 23-12 Поиск шаблона в файле main.go в папке htmltext

В этом примере метод Lookup используется для загрузки шаблона из файла template.txt и использования его в качестве аргумента

функции Exec, которая выполняет шаблон с использованием стандартного вывода. Скомпилируйте и запустите проект, и вы увидите следующий вывод:

```
<h1>Template Value: {Kayak Watersports 279}</h1>
```

Понимание действий шаблона

Шаблоны Go поддерживают широкий спектр действий, которые можно использовать для создания содержимого из данных, передаваемых в метод Execute или ExecuteTemplate. Для быстрого ознакомления в таблице 23-5 приведены действия шаблона, наиболее полезные из которых демонстрируются в следующих разделах.

Таблица 23-5 Действия шаблона

Действие	Описание
{{ value }} {{ expr }}	Это действие вставляет значение данных или результат выражения в шаблон. Точка используется для ссылки на значение данных, переданное функции Execute или ExecuteTemplate. Подробнее см. в разделе «Вставка значений данных».
{{ value.fieldname }}	Это действие вставляет значение поля структуры. Подробнее см. в разделе «Вставка значений данных».
{{ value.method arg }}	Это действие вызывает метод и вставляет результат в выходные данные шаблона. Скобки не используются, а аргументы разделяются пробелами. Подробнее см. в разделе «Вставка значений данных».
{{ func arg }}	Это действие вызывает функцию и вставляет результат в выходные данные. Существуют встроенные функции для общих задач, таких как форматирование значений данных, и могут быть определены пользовательские функции, как описано в разделе «Определение функций шаблона».
{{ expr | value.method }} {{ expr | func	Выражения можно объединять в цепочку с помощью вертикальной черты, чтобы результат первого выражения использовался в качестве последнего аргумента во втором выражении.
{{ range value }} ... {{ end }}	Это действие выполняет итерацию по указанному срезу и добавляет содержимое между ключевым словом range и end для каждого элемента. Действия во вложенном содержимом выполняются с текущим элементом, доступным через точку. Дополнительные сведения см. в разделе «Использование срезов в шаблонах».

Действие	Описание
`{{ range value }} ... {{ else }} ... {{ end }}`	Это действие похоже на комбинацию range/end, но определяет раздел вложенного содержимого, который используется, если срез не содержит элементов. This action is similar to the range/end combination but defines a section of nested content that is used if the slice contains no elements.
`{{ if expr }} ... {{ end }}`	Это действие оценивает выражение и выполняет содержимое вложенного шаблона, если результат `true`, как показано в разделе «Условное выполнение содержимого шаблона». Это действие можно использовать с необязательными предложениями `else` и `else if`.
`{{ with expr }} ... {{ end }}`	Это действие оценивает выражение и выполняет содержимое вложенного шаблона, если результат не равен `nil` или пустой строке. Это действие можно использовать с дополнительными предложениями.
`{{ define "name" }} ... {{ end }}`	Это действие определяет шаблон с указанным именем
`{{ template "name" expr }}`	Это действие выполняет шаблон с указанным именем и данными и вставляет результат в выходные данные.
`{{ block "name" expr }} ... {{ end }}`	Это действие определяет шаблон с указанным именем и вызывает его с указанными данными. Обычно это используется для определения шаблона, который можно заменить шаблоном, загруженным из другого файла, как показано в разделе «Определение блоков шаблона».

Вставка значений данных

Самая простая задача в шаблоне — вставить значение в выходные данные, сгенерированные шаблоном, что делается путем создания действия, содержащего выражение, которое создает значение, которое вы хотите вставить. В таблице 23-6 описаны основные шаблонные выражения, наиболее полезные из которых демонстрируются в следующих разделах.

Таблица 23-6 Шаблонные выражения для вставки значений в шаблоны

Выражение	Описание
`.`	Это выражение вставляет значение, переданное методу `Execute` или `ExecuteTemplate`, в выходные данные шаблона.
`.Field`	Это выражение вставляет значение указанного поля в выходные данные шаблона.
`.Method`	Это выражение вызывает указанный метод без аргументов и вставляет результат в выходные данные шаблона.

Выражение	Описание
`.Method arg`	Это выражение вызывает указанный метод с указанным аргументом и вставляет результат в выходные данные шаблона.
`call .Field arg`	Это выражение вызывает поле функции структуры, используя указанные аргументы, разделенные пробелами. Результат функции вставляется в вывод шаблона.

В предыдущем разделе я использовал только точку, что приводит к вставке строкового представления значения данных, используемого для выполнения шаблона. Шаблоны в большинстве реальных проектов включают значения для конкретных полей или результаты вызова методов, как показано в листинге 23-13.

```
<h1>Template Value: {{ . }}</h1>
<h1>Name: {{ .Name }}</h1>
<h1>Category: {{ .Category }}</h1>
<h1>Price: {{ .Price }}</h1>
<h1>Tax: {{ .AddTax }}</h1>
<h1>Discount Price: {{ .ApplyDiscount 10 }}</h1>
```

Листинг 23-13 Вставка значений данных в файл template.html в папку templates

Новые действия содержат выражения, записывающие значения полей Name, `Category` и `Price`, а также результаты вызова методов `AddTax` и `ApplyDiscount`. Синтаксис доступа к полям во многом похож на код Go, но способ вызова методов и функций достаточно отличается, поэтому легко сделать ошибку. В отличие от кода Go, методы не вызываются с помощью круглых скобок, а аргументы просто указываются после имени, разделенного пробелами. Разработчик несет ответственность за то, чтобы аргументы имели тип, который может использоваться методом или функцией. Скомпилируйте и запустите проект, и вы увидите следующий вывод:

```
<h1>Template Value: {Kayak Watersports 279}</h1>
<h1>Name: Kayak</h1>
<h1>Category: Watersports</h1>
<h1>Price: 279</h1>
<h1>Tax: 334.8</h1>
<h1>Discount Price: 269</h1>
```

Понимание контекстного экранирования

Значения автоматически экранируются, чтобы сделать их безопасными для включения в код HTML, CSS и JavaScript, с соответствующими правилами экранирования, применяемыми в зависимости от контекста. Например, строковое значение, такое как `"It was a <big> boat"`, используемое в качестве текстового содержимого элемента HTML, будет вставлено в шаблон как «`"It was a <big> boat"`», но как «`It was a \u003cbig\u003e boat`» при использовании в качестве строкового литерала в коде JavaScript. Полную информацию об экранировании значений можно найти по адресу https://golang.org/pkg/html/template.

Форматирование значений данных

Шаблоны поддерживают встроенные функции для общих задач, включая форматирование значений данных, которые вставляются в выходные данные, как описано в таблице 23-7. Дополнительные встроенные функции описаны в последующих разделах.

Таблица 23-7 Встроенные функции шаблонов для форматирования данных

Функция	Описание
`print`	Это псевдоним функции `fmt.Sprint`.
`printf`	Это псевдоним функции `fmt.Sprintf`.
`println`	Это псевдоним функции `fmt.Sprintln`.
`html`	Эта функция кодирует значение для безопасного включения в документ HTML.
`js`	Эта функция кодирует значение для безопасного включения в документ JavaScript.
`urlquery`	Эта функция кодирует значение для использования в строке запроса URL.

Эти функции вызываются путем указания их имени, за которым следует список аргументов, разделенных пробелами. В листинге 23-14 я использовал функцию `printf` для форматирования некоторых полей данных, включенных в вывод шаблона.

```
<h1>Template Value: {{ . }}</h1>
<h1>Name: {{ .Name }}</h1>
<h1>Category: {{ .Category }}</h1>
```

```
<h1>Price: {{ printf "$%.2f" .Price }}</h1>
<h1>Tax: {{ printf "$%.2f" .AddTax }}</h1>
<h1>Discount Price: {{ .ApplyDiscount 10 }}</h1>
```

Листинг 23-14 Использование функции форматирования в файле template.html в папке templates

Использование функции printf позволяет мне форматировать два значения данных как суммы в долларах, создавая следующий вывод, когда проект компилируется и выполняется:

```
<h1>Extras Template Value: {Kayak Watersports 279}</h1>
<h1>Name: Kayak</h1>
<h1>Category: Watersports</h1>
<h1>Price: $279.00</h1>
<h1>Tax: $334.80</h1>
<h1>Discount Price: 269</h1>
```

Цепочки и скобки шаблонных выражений

Цепочка выражений создает конвейер для значений, который позволяет использовать выходные данные одного метода или функции в качестве входных данных для другого. В листинге 23-15 создается конвейер, объединяющий результат метода ApplyDiscount, чтобы его можно было использовать в качестве аргумента функции printf.

```
<h1>Template Value: {{ . }}</h1>
<h1>Name: {{ .Name }}</h1>
<h1>Category: {{ .Category }}</h1>
<h1>Price: {{ printf "$%.2f" .Price }}</h1>
<h1>Tax: {{ printf "$%.2f" .AddTax }}</h1>
<h1>Discount Price: {{ .ApplyDiscount 10 | printf "$%.2f" }}
</h1>
```

Листинг 23-15 Цепочки выражений в файле template.html в папке templates

Выражения объединяются в цепочку с помощью вертикальной черты (символ |), в результате чего результат одного выражения используется в качестве последнего аргумента следующего выражения. В листинге 23-15 результат вызова метода ApplyDiscount используется в качестве последнего аргумента для вызова встроенной

функции `printf`. Скомпилируйте и выполните проект, и вы увидите отформатированное значение в выводе, созданном шаблоном:

```
<h1>Extras Template Value: {Kayak Watersports 279}</h1>
<h1>Name: Kayak</h1>
<h1>Category: Watersports</h1>
<h1>Price: $279.00</h1>
<h1>Tax: $334.80</h1>
<h1>Discount Price: $269.00</h1>
```

Цепочка может использоваться только для последнего аргумента, предоставленного функции. Альтернативный подход, который можно использовать для установки других аргументов функции, заключается в использовании круглых скобок, как показано в листинге 23-16.

```
<h1>Template Value: {{ . }}</h1>
<h1>Name: {{ .Name }}</h1>
<h1>Category: {{ .Category }}</h1>
<h1>Price: {{ printf "$%.2f" .Price }}</h1>
<h1>Tax: {{ printf "$%.2f" .AddTax }}</h1>
<h1>Discount Price: {{ printf "$%.2f" (.ApplyDiscount 10) }}
</h1>
```

Листинг 23-16 Использование скобок в файле template.html в папке templates

Вызывается метод `ApplyDiscount`, и результат используется в качестве аргумента функции `printf`. Шаблон в листинге 23-16 выдает тот же результат, что и в листинге 23-15.

Обрезка пробелов

По умолчанию содержимое шаблона отображается точно так, как оно определено в файле, включая любые пробелы между действиями. HTML не чувствителен к пробелам между элементами, но пробелы могут вызвать проблемы с текстовым содержимым и значениями атрибутов, особенно если вы хотите структурировать содержимое шаблона, чтобы его было легко читать, как показано в листинге 23-17.

```
<h1>
    Name: {{ .Name }}, Category: {{ .Category }}, Price,
        {{ printf "$%.2f" .Price }}
</h1>
```

Листинг 23-17 Структурирование содержимого шаблона в файле template.html в папке templates

Я добавил новые строки и отступы, чтобы соответствовать содержимому на печатной странице и отделить содержимое элемента от его тегов. Пробел включается в вывод, когда проект компилируется и выполняется:

```
<h1>
    Name: Kayak, Category: Watersports, Price,
        $279.00
</h1>
```

Знак минус можно использовать для обрезки пробелов, применяемых непосредственно после или перед фигурными скобками, открывающими или закрывающими действие. В листинге 23-18 я использовал эту функцию для обрезки пробелов, представленных в листинге 23-17.

```
<h1>
    Name: {{ .Name }}, Category: {{ .Category }}, Price,
        {{- printf "$%.2f" .Price -}}
</h1>
```

Листинг 23-18 Обрезка пробелов в файле template.html в папке templates

Знак минус должен быть отделен от остального выражения действия пробелом. Эффект заключается в удалении всех пробелов до или после действия, что можно увидеть, скомпилировав и выполнив проект, который выдает следующий результат:

```
<h1>
    Name: Kayak, Category: Watersports, Price,$279.00</h1>
```

Пробел вокруг конечного действия был удален, но после открывающего тега `h1` по-прежнему есть символ новой строки, потому что обрезка пробелов применяется только к действиям. Если этот пробел нельзя удалить из шаблона, то действие, которое вставляет в вывод пустую строку, может использоваться только для обрезки пробела, как показано в листинге 23-19.

```
<h1>
    {{- "" -}} Name: {{ .Name }}, Category: {{ .Category }},
Price,
        {{- printf "$%.2f" .Price -}}
</h1>
```

Листинг 23-19 Обрезка дополнительных пробелов в файле template.html в папке templates

Новое действие не вводит никаких новых выходных данных и действует только для обрезки окружающего пробела, что можно увидеть, скомпилировав и выполнив проект:

```
<h1>Name: Kayak, Category: Watersports, Price,$279.00</h1>
```

Даже с этой функцией может быть сложно контролировать пробелы при написании простых для понимания шаблонов, как вы увидите в последующих примерах. Если важна конкретная структура документа, вам придется принять шаблоны, которые сложнее читать и поддерживать. Если удобочитаемость и ремонтопригодность являются приоритетом, вы должны принять дополнительные пробелы в выводе, создаваемом шаблоном.

Использование срезов в шаблонах

Действия шаблона можно использовать для создания контента для срезов, как показано в листинге 23-20, который заменяет весь шаблон.

```
{{ range . -}}
    <h1>Name: {{ .Name }}, Category: {{ .Category }}, Price,
        {{- printf "$%.2f" .Price }}</h1>
{{ end }}
```

Листинг 23-20 Обработка среза в файле template.html в папке templates

Выражение `range` повторяет указанные данные, и я использовал точку в листинге 23-20, чтобы выбрать значение данных, используемое для выполнения шаблона, который я вскоре настрою. Содержимое шаблона между выражением `range` и выражением `end` будет повторяться для каждого значения в срезе с текущим значением, назначенным точке, чтобы его можно было использовать во вложенных действиях. Эффект в листинге 23-20 заключается в том, что поля `Name`,

Category и Price вставляются в выходные данные для каждого значения в срезе, перечисляемом выражением range.

Примечание
Ключевое слово range также можно использовать для перечисления карт, как описано в разделе «Определение переменных шаблона» далее в этой главе.

В листинге 23-21 обновляется код, выполняющий шаблон, для использования среза вместо одного значения Product.

```
package main

import (
    "html/template"
    "os"
)

func Exec(t *template.Template) error {
    return t.Execute(os.Stdout, Products)
}

func main() {
                                allTemplates,        err        :=
template.ParseGlob("templates/*.html")
    if (err == nil) {
                                         selectedTemplated     :=
allTemplates.Lookup("template.html")
        err = Exec(selectedTemplated)
    }
    if (err != nil) {
        Printfln("Error: %v %v", err.Error())
    }
}
```

Листинг 23-21 Использование среза для выполнения шаблона в файле main.go в папке htmltext

Скомпилируйте и выполните код, и вы увидите следующий вывод:

```
<h1>Name: Kayak, Category: Watersports, Price,$279.00</h1>
```

```
<h1>Name:      Lifejacket,      Category:      Watersports,
Price,$49.95</h1>
<h1>Name: Soccer Ball, Category: Soccer, Price,$19.50</h1>
<h1>Name: Corner Flags, Category: Soccer, Price,$34.95</h1>
<h1>Name: Stadium, Category: Soccer, Price,$79500.00</h1>
<h1>Name: Thinking Cap, Category: Chess, Price,$16.00</h1>
<h1>Name: Unsteady Chair, Category: Chess, Price,$75.00</h1>
<h1>Name:     Bling-Bling     King,     Category:     Chess,
Price,$1200.00</h1>
```

Обратите внимание, что я применил знак минус к действию, содержащему выражение `range` в листинге 23-20. Я хотел, чтобы содержимое шаблона в `range` и `end` действиях было визуально различимым, помещая его в новую строку и добавляя отступ, но это привело бы к дополнительным разрывам строк и пробелам в выводе. Помещение знака минус в конце выражения `range` обрезает все начальные пробелы из вложенного содержимого. Я не добавлял знак «минус» к `end` действию, что позволяет сохранить завершающие символы новой строки, так что вывод для каждого элемента в срезе отображается на отдельной строке.

Использование встроенных функций среза

Текстовые шаблоны Go поддерживают встроенные функции, описанные в таблице 23-8, для работы со срезами.

Таблица 23-8 Встроенные функции шаблона для срезов

Функция	Описание
`slice`	Эта функция создает новый срез. Его аргументами являются исходный срез, начальный индекс и конечный индекс.
`index`	Эта функция возвращает элемент по указанному индексу.
`len`	Эта функция возвращает длину указанного среза.

В листинге 23-22 встроенные функции используются для сообщения размера среза, получения элемента по определенному индексу и создания нового среза.

```
<h1>There are {{ len . }} products in the source data.</h1>
<h1>First product: {{ index . 0 }}</h1>
{{ range slice . 3 5 -}}
```

```
    <h1>Name: {{ .Name }}, Category: {{ .Category }}, Price,
        {{- printf "$%.2f" .Price }}</h1>
{{ end }}
```

Листинг 23-22 Использование встроенных функций в файле template.html в папке templates

Скомпилируйте и запустите проект, и вы увидите следующий вывод:

```
<h1>There are 8 products in the source data.</h1>
<h1>First product: {Kayak Watersports 279}</h1>
<h1>Name: Corner Flags, Category: Soccer, Price,$34.95</h1>
<h1>Name: Stadium, Category: Soccer, Price,$79500.00</h1>
```

Условное выполнение содержимого шаблона

Действия можно использовать для условной вставки контента в вывод на основе оценки их выражений, как показано в листинге 23-23.

```
<h1>There are {{ len . }} products in the source data.</h1>
<h1>First product: {{ index . 0 }}</h1>
{{ range . -}}
    {{ if lt .Price 100.00 -}}
          <h1>Name: {{ .Name }}, Category: {{ .Category }},
Price,
            {{- printf "$%.2f" .Price }}</h1>
    {{ end -}}
{{ end }}
```

Листинг 23-23 Использование условного действия в файле template.html в папке templates

За ключевым словом `if` следует выражение, определяющее, выполняется ли содержимое вложенного шаблона. Чтобы упростить написание выражений для этих действий, шаблоны поддерживают функции, описанные в таблице 23-9.

Таблица 23-9 Условные функции шаблона

Функция	Описание
`eq arg1 arg2`	Эта функция возвращает `true`, если `arg1 == arg2`.
`ne arg1 arg2`	Эта функция возвращает значение `true`, если `arg1 != arg2`.

Функция	Описание
lt arg1 arg2	Эта функция возвращает значение true, если arg1 < arg2.
le arg1 arg2	Эта функция возвращает значение true, если arg1 <= arg2.
gt arg1 arg2	Эта функция возвращает значение true, если arg1 > arg2.
ge arg1 arg2	Эта функция возвращает значение true, если arg1 >= arg2.
and arg1 arg2	Эта функция возвращает значение true, если оба параметра arg1 и arg2 являются true.
not arg1	Эта функция возвращает true, если arg1 является false, и false, если true.

Синтаксис этих функций соответствует остальным функциям шаблона, что неудобно, пока вы к нему не привыкнете. В листинге 23-23 я использовал это выражение:

```
...
{{ if lt .Price 100.00 -}}
...
```

Ключевое слово if указывает условное действие, функция lt выполняет сравнение меньшего, а остальные аргументы указывают поле Price текущего значения в выражении range и литеральное значение 100.00. Функции сравнения, описанные в таблице 23-9, не имеют сложного подхода к работе с типами данных, а это означает, что я должен указать литеральное значение как 100.00, чтобы оно обрабатывалось как float64 и не могло полагаться на то, как Go работает с нетипизированными константами.

Действие range перечисляет значения в срезе Product и выполняет вложенное действие if. Действие if выполнит свое вложенное содержимое только в том случае, если значение поля Price для текущего элемента меньше 100. Скомпилируйте и выполните проект, и вы увидите следующий вывод:

```
<h1>There are 8 products in the source data.</h1>
<h1>First product: {Kayak Watersports 279}</h1>
<h1>Name:      Lifejacket,      Category:      Watersports,
Price,$49.95</h1>
```

```
            <h1>Name:   Soccer   Ball,   Category:   Soccer,
Price,$19.50</h1>
            <h1>Name:   Corner   Flags,   Category:   Soccer,
Price,$34.95</h1>
            <h1>Name:   Thinking   Cap,   Category:   Chess,
Price,$16.00</h1>
            <h1>Name:  Unsteady  Chair,  Category:   Chess,
Price,$75.00</h1>
```

Несмотря на использование знака минус для обрезки пробелов, вывод имеет странный формат из-за того, как я структурировал шаблон. Как отмечалось ранее, существует компромисс между структурированием шаблонов, чтобы их было легко читать, и управлением пробелами в выводе. В этой главе я сосредоточился на том, чтобы сделать шаблоны простыми для понимания, в результате чего выходные данные примеров имеют неуклюжий формат.

Использование дополнительных условных действий

Действие `if` можно использовать с необязательными ключевыми словами `else` и `else if`, как показано в листинге 23-24, что позволяет использовать резервное содержимое, которое будет выполняться, когда выражение `if` ложно, или выполняться только тогда, когда второе выражение истинно.

```
<h1>There are {{ len . }} products in the source data.</h1>
<h1>First product: {{ index . 0 }}</h1>
{{ range . -}}
    {{ if lt .Price 100.00 -}}
          <h1>Name: {{ .Name }}, Category: {{ .Category }},
Price,
            {{- printf "$%.2f" .Price }}</h1>
    {{ else if gt .Price 1500.00 -}}
        <h1>Expensive Product {{ .Name }} ({{ printf "$%.2f"
.Price}})</h1>
    {{ else -}}
        <h1>Midrange Product: {{ .Name }} ({{ printf "$%.2f"
.Price}})</h1>
    {{ end -}}
{{ end }}
```

Листинг 23-24 Использование необязательных ключевых слов в файле template.html в папке templates

Скомпилируйте и выполните проект, и вы увидите, что действия `if`, `else if` и `else` дают следующий результат:

```
<h1>There are 8 products in the source data.</h1>
<h1>First product: {Kayak Watersports 279}</h1>
<h1>Midrange Product: Kayak ($279.00)</h1>
            <h1>Name: Lifejacket, Category: Watersports, Price,$49.95</h1>
            <h1>Name: Soccer Ball, Category: Soccer, Price,$19.50</h1>
            <h1>Name: Corner Flags, Category: Soccer, Price,$34.95</h1>
    <h1>Expensive Product Stadium ($79500.00)</h1>
            <h1>Name: Thinking Cap, Category: Chess, Price,$16.00</h1>
            <h1>Name: Unsteady Chair, Category: Chess, Price,$75.00</h1>
    <h1>Midrange Product: Bling-Bling King ($1200.00)</h1>
```

Создание именованных вложенных шаблонов

Действие `define` используется для создания вложенного шаблона, который может выполняться по имени, что позволяет определить содержимое один раз и повторно использовать с действием шаблона, как показано в листинге 23-25.

```
{{ define "currency" }}{{ printf "$%.2f" . }}{{ end }}

{{ define "basicProduct" -}}
    Name: {{ .Name }}, Category: {{ .Category }}, Price,
        {{- template "currency" .Price }}
{{- end }}

{{ define "expensiveProduct" -}}
     Expensive Product {{ .Name }} ({{ template "currency" .Price }})
{{- end }}

<h1>There are {{ len . }} products in the source data.</h1>
<h1>First product: {{ index . 0 }}</h1>
{{ range . -}}
    {{ if lt .Price 100.00 -}}
```

```
        <h1>{{ template "basicProduct" . }}</h1>
    {{ else if gt .Price 1500.00 -}}
        <h1>{{ template "expensiveProduct" . }}</h1>
    {{ else -}}
        <h1>Midrange Product: {{ .Name }} ({{ printf "$%.2f"
.Price}})</h1>
    {{ end -}}
{{ end }}
```

Листинг 23-25 Определение и использование вложенного шаблона в файле template.html в папке templates

За ключевым словом define следует имя шаблона в кавычках, а шаблон завершается ключевым словом end. Ключевое слово template используется для выполнения именованного шаблона с указанием имени шаблона и значения данных:

```
...
{{- template "currency" .Price }}
...
```

Это действие выполняет шаблон с именем currency и использует значение поля Price в качестве значения данных, доступ к которому осуществляется в именованном шаблоне с использованием точки:

```
...
{{ define "currency" }}{{ printf "$%.2f" . }}{{ end }}
...
```

Именованный шаблон может вызывать другие именованные шаблоны, как показано в листинге 23-25, при этом шаблоны basicProduct и expensiveProduct выполняют шаблон currency.

Вложенные именованные шаблоны могут усугубить проблемы с пробелами, потому что пробелы вокруг шаблонов, которые я добавил в листинге 23-25 для ясности, включаются в выходные данные основного шаблона. Один из способов решить эту проблему — определить именованные шаблоны в отдельном файле, но эту проблему также можно решить, используя только именованные шаблоны даже для основной части вывода, как показано в листинге 23-26.

```
{{ define "currency" }}{{ printf "$%.2f" . }}{{ end }}

{{ define "basicProduct" -}}
    Name: {{ .Name }}, Category: {{ .Category }}, Price,
        {{- template "currency" .Price }}
{{- end }}

{{ define "expensiveProduct" -}}
     Expensive Product {{ .Name }} ({{ template "currency"
.Price }})
{{- end }}

{{ define "mainTemplate" -}}
     <h1>There are {{ len . }} products in the source data.
</h1>
    <h1>First product: {{ index . 0 }}</h1>
    {{ range . -}}
        {{ if lt .Price 100.00 -}}
            <h1>{{ template "basicProduct" . }}</h1>
        {{ else if gt .Price 1500.00 -}}
            <h1>{{ template "expensiveProduct" . }}</h1>
        {{ else -}}
               <h1>Midrange Product: {{ .Name }} ({{ printf
"$%.2f" .Price}})</h1>
        {{ end -}}
    {{ end }}
{{- end}}
```

Листинг 23-26 Добавление именованного шаблона в файл template.html в папке templates

Использование ключевых слов `define` и `end` для основного содержимого шаблона исключает пробелы, используемые для разделения других именованных шаблонов. В листинге 23-27 я завершаю изменение, используя имя при выборе шаблона для выполнения.

```
package main

import (
    "html/template"
    "os"
)
```

```
func Exec(t *template.Template) error {
    return t.Execute(os.Stdout, Products)
}

func main() {
                              allTemplates,      err       :=
template.ParseGlob("templates/*.html")
    if (err == nil) {
                                       selectedTemplated     :=
allTemplates.Lookup("mainTemplate")
        err = Exec(selectedTemplated)
    }
    if (err != nil) {
        Printfln("Error: %v %v", err.Error())
    }
}
```

Листинг 23-27 Выбор именованного шаблона в файле main.go в папке htmltext

Любой из названных шаблонов можно выполнить напрямую, но я выбрал `mainTemplate`, который выдает следующий результат при компиляции и выполнении проекта:

```
<h1>There are 8 products in the source data.</h1>
    <h1>First product: {Kayak Watersports 279}</h1>
    <h1>Midrange Product: Kayak ($279.00)</h1>
              <h1>Name:  Lifejacket,  Category:  Watersports,
Price,$49.95</h1>
                <h1>Name:  Soccer  Ball,  Category:  Soccer,
Price,$19.50</h1>
               <h1>Name:  Corner  Flags,  Category:  Soccer,
Price,$34.95</h1>
        <h1>Expensive Product Stadium ($79500.00)</h1>
                <h1>Name:  Thinking  Cap,  Category:  Chess,
Price,$16.00</h1>
               <h1>Name:  Unsteady  Chair,  Category:  Chess,
Price,$75.00</h1>
          <h1>Midrange  Product:  Bling-Bling  King  ($1200.00)
</h1>
```

Определение блоков шаблона

Блоки шаблона используются для определения шаблона с содержимым по умолчанию, которое может быть переопределено в другом файле шаблона, что требует загрузки и одновременного выполнения нескольких шаблонов. Это часто используется для общего содержимого, такого как макет, как показано в листинге 23-28.

```
{{ define "mainTemplate" -}}
    <h1>This is the layout header</h1>
    {{ block "body" . }}
          <h2>There are {{ len . }} products in the source data.</h2>
    {{ end }}
    <h1>This is the layout footer</h1>
{{ end }}
```

Листинг 23-28 Определение блока в файле template.html в папке templates

Действие `block` используется для присвоения имени шаблону, но, в отличие от действия `define`, шаблон будет включен в вывод без необходимости использования действия `template`, что можно увидеть, скомпилировав и выполнив проект (я отформатировал вывод для удаления пробела):

```
<h1>This is the layout header</h1>
    <h2>There are 8 products in the source data.</h2>
<h1>This is the layout footer</h1>
```

При отдельном использовании выходные данные файла шаблона включают содержимое блока. Но это содержимое может быть переопределено другим файлом шаблона. Добавьте файл с именем `list.html` в папку `templates` с содержимым, показанным в листинге 23-29.

```
{{ define "body" }}
    {{ range . }}
        <h2>Product: {{ .Name }} ({{ printf "$%.2f" .Price}})</h2>
    {{ end -}}
{{ end }}
```

Листинг 23-29 Содержимое файла list.html в папке templates

Чтобы использовать эту функцию, файлы шаблонов должны быть загружены по порядку, как показано в листинге 23-30.

```
package main

import (
    "html/template"
    "os"
)

func Exec(t *template.Template) error {
    return t.Execute(os.Stdout, Products)
}

func main() {

    allTemplates, err := template.ParseFiles("templates/template.html",
        "templates/list.html")
    if (err == nil) {
        selectedTemplated := allTemplates.Lookup("mainTemplate")
        err = Exec(selectedTemplated)
    }
    if (err != nil) {
        Printfln("Error: %v %v", err.Error())
    }
}
```

Листинг 23-30 Загрузка шаблонов в файл main.go в папку htmltext

Шаблоны должны быть загружены таким образом, чтобы файл, содержащий действие блокировки, загружался перед файлом, содержащим действие `define`, переопределяющее шаблон. Когда шаблоны загружаются, шаблон, определенный в файле `list.html`, переопределяет шаблон с именем `body`, так что содержимое файла `list.html` заменяет содержимое файла `template.html`. Скомпилируйте и выполните проект, и вы увидите следующий вывод, который я отформатировал, чтобы удалить пробелы:

```
<h1>This is the layout header</h1>
    <h2>Product: Kayak ($279.00)</h2>
```

```
    <h2>Product: Lifejacket ($49.95)</h2>
    <h2>Product: Soccer Ball ($19.50)</h2>
    <h2>Product: Corner Flags ($34.95)</h2>
    <h2>Product: Stadium ($79500.00)</h2>
    <h2>Product: Thinking Cap ($16.00)</h2>
    <h2>Product: Unsteady Chair ($75.00)</h2>
    <h2>Product: Bling-Bling King ($1200.00)</h2>
<h1>This is the layout footer</h1>
```

Определение функций шаблона

Встроенные функции шаблона, описанные в предыдущих разделах, могут быть дополнены пользовательскими функциями, специфичными для `Template`, то есть они определены и настроены в коде. В листинге 23-31 показан процесс настройки пользовательской функции..

```
package main

import (
    "html/template"
    "os"
)

func GetCategories(products []Product) (categories []string)
{
    catMap := map[string]string {}
    for _, p := range products {
        if (catMap[p.Category] == "") {
            catMap[p.Category] = p.Category
            categories = append(categories, p.Category)
        }
    }
    return
}

func Exec(t *template.Template) error {
    return t.Execute(os.Stdout, Products)
}

func main() {
    allTemplates := template.New("allTemplates")
    allTemplates.Funcs(map[string]interface{} {
        "getCats": GetCategories,
```

```
    })
    allTemplates, err := allTemplates.ParseGlob("templates/*.html")

    if (err == nil) {
        selectedTemplated := allTemplates.Lookup("mainTemplate")
        err = Exec(selectedTemplated)
    }
    if (err != nil) {
        Printfln("Error: %v %v", err.Error())
    }
}
```

Листинг 23-31 Определение пользовательской функции в файле main.go в папке htmltext

Функция GetCategories получает срез Product и возвращает набор уникальных значений Category. Чтобы настроить функцию GetCategories таким образом, чтобы ее можно было использовать в Template, вызывается метод Funcs, передающий карту имен функциям, например:

```
...
allTemplates.Funcs(map[string]interface{} {
    "getCats": GetCategories,
})
...
```

Карта в листинге 23-31 указывает, что функция GetCategories будет вызываться с использованием имени getCats. Метод Funcs должен вызываться перед анализом файлов шаблонов, что означает создание Template с использованием функции New, которая затем позволяет зарегистрировать пользовательские функции до вызова метода ParseFiles или ParseGlob:

```
...
allTemplates := template.New("allTemplates")
allTemplates.Funcs(map[string]interface{} {
    "getCats": GetCategories,
})
allTemplates, err := allTemplates.ParseGlob("templates/*.html")
```

```
...
```

В шаблонах пользовательские функции могут вызываться с использованием того же синтаксиса, что и встроенные функции, как показано в листинге 23-32.

```
{{ define "mainTemplate" -}}
    <h1>There are {{ len . }} products in the source data.
</h1>
    {{ range getCats .  -}}
        <h1>Category: {{ . }}</h1>
    {{ end }}
{{- end }}
```

Листинг 23-32 Использование пользовательской функции в файле template.html в папке templates

Ключевое слово `range` используется для перечисления категорий, возвращаемых пользовательской функцией, которые включаются в выходные данные шаблона. Скомпилируйте и выполните проект, и вы увидите следующий вывод, который я отформатировал, чтобы удалить пробелы:

```
<h1>There are 8 products in the source data.</h1>
<h1>Category: Watersports</h1>
<h1>Category: Soccer</h1>
<h1>Category: Chess</h1>
```

Отключение кодирования результата функции

Результаты, полученные функциями, кодируются для безопасного включения в документ HTML, что может представлять проблему для функций, генерирующих фрагменты HTML, JavaScript или CSS, как показано в листинге 23-33.

```
...
func GetCategories(products []Product) (categories []string)
{
    catMap := map[string]string {}
    for _, p := range products {
        if (catMap[p.Category] == "") {
            catMap[p.Category] = p.Category
```

```
                    categories = append(categories, "
<b>p.Category</b>")
        }
    }
    return
}
...
```

Листинг 23-33 Создание фрагмента HTML в файле main.go в папке htmltext

Функция GetCategories была изменена таким образом, чтобы она создавала срез, содержащий строки HTML. Механизм шаблонов кодирует эти значения, которые отображаются в выводе компиляции и выполнения проекта:

```
<h1>There are 8 products in the source data.</h1>
<h1>Category: &lt;b&gt;p.Category&lt;/b&gt;</h1>
<h1>Category: &lt;b&gt;p.Category&lt;/b&gt;</h1>
<h1>Category: &lt;b&gt;p.Category&lt;/b&gt;</h1>
```

Это хорошая практика, но она вызывает проблемы, когда функции используются для создания содержимого, которое должно быть включено в шаблон без кодирования. Для таких ситуаций пакет html/template определяет набор псевдонимов строкового типа, которые используются для обозначения того, что результат функции требует специальной обработки, как описано в таблице 23-10.

Таблица 23-10 Псевдонимы типов, используемые для обозначения типов контента

Функция	Описание
CSS	Этот тип обозначает содержимое CSS.
HTML	Этот тип обозначает фрагмент HTML.
HTMLAttr	Этот тип обозначает значение, которое будет использоваться в качестве значения атрибута HTML.
JS	Этот тип обозначает фрагмент кода JavaScript.
JSStr	Этот тип обозначает значение, которое должно отображаться между кавычками в выражении JavaScript.
Srcset	Этот тип обозначает значение, которое можно использовать в атрибуте srcset элемента img.
URL	Этот тип обозначает URL.

Чтобы предотвратить обычную обработку содержимого, функция, создающая содержимое, использует один из типов, показанных в таблице 23-10, как показано в листинге 23-34.

```
...
func     GetCategories(products     []Product)     (categories
[]template.HTML) {
    catMap := map[string]string {}
    for _, p := range products {
        if (catMap[p.Category] == "") {
            catMap[p.Category] = p.Category
                         categories   =  append(categories,   "
<b>p.Category</b>")
        }
    }
    return
}
...
```

Листинг 23-34 Возврат HTML-контента в файл main.go в папку htmltext

Это изменение сообщает системе шаблонов, что результаты функции `GetCategories` представляют собой HTML, что приводит к следующему результату при компиляции и выполнении проекта:

```
<h1>There are 8 products in the source data.</h1>
<h1>Category: <b>p.Category</b></h1>
<h1>Category: <b>p.Category</b></h1>
<h1>Category: <b>p.Category</b></h1>
```

Предоставление доступа к функциям стандартной библиотеки

Шаблонные функции также можно использовать для предоставления доступа к функциям, предоставляемым стандартной библиотекой, как показано в листинге 23-35.

```
package main

import (
    "html/template"
    "os"
```

```
    "strings"
)

func GetCategories(products []Product) (categories []string)
{
    catMap := map[string]string {}
    for _, p := range products {
        if (catMap[p.Category] == "") {
            catMap[p.Category] = p.Category
            categories = append(categories, p.Category)
        }
    }
    return
}

func Exec(t *template.Template) error {
    return t.Execute(os.Stdout, Products)
}

func main() {
    allTemplates := template.New("allTemplates")
    allTemplates.Funcs(map[string]interface{} {
        "getCats": GetCategories,
        "lower": strings.ToLower,
    })
                             allTemplates,       err       :=
allTemplates.ParseGlob("templates/*.html")

    if (err == nil) {
                                      selectedTemplated     :=
allTemplates.Lookup("mainTemplate")
        err = Exec(selectedTemplated)
    }
    if (err != nil) {
        Printfln("Error: %v %v", err.Error())
    }
}
```

Листинг 23-35 Добавление сопоставления функций в файл main.go в папке htmltext

Новое сопоставление обеспечивает доступ к функции `ToLower`, которая переводит строки в нижний регистр, как описано в главе 16.

Доступ к этой функции можно получить внутри шаблона, используя имя `lower`, как показано в листинге 23-36.

```
{{ define "mainTemplate" -}}
     <h1>There are {{ len . }} products in the source data.
</h1>
    {{ range getCats .  -}}
        <h1>Category: {{ lower . }}</h1>
    {{ end }}
{{- end }}
```

Листинг 23-36 Использование функции шаблона в файле template.html в папке templates

Скомпилируйте и запустите проект, и вы увидите следующий вывод:

```
<h1>There are 8 products in the source data.</h1>
<h1>Category: watersports</h1>
<h1>Category: soccer</h1>
<h1>Category: chess</h1>
```

Определение переменных шаблона

Действия могут определять переменные в своих выражениях, доступ к которым можно получить из содержимого встроенного шаблона, как показано в листинге 23-37. Эта функция полезна, когда вам нужно создать значение для оценки в выражении и вам нужно такое же значение во вложенном содержимом.

```
{{ define "mainTemplate" -}}
    {{ $length := len . }}
    <h1>There are {{ $length }} products in the source data.
</h1>
    {{ range getCats .  -}}
        <h1>Category: {{ lower . }}</h1>
    {{ end }}
{{- end }}
```

Листинг 23-37 Определение и использование переменной шаблона в файле template.html в папке templates

Имена переменных шаблона начинаются с символа `$` и создаются с использованием короткого синтаксиса объявления переменных.

Первое действие создает переменную с именем `length`, которая используется в следующем действии. Скомпилируйте и запустите проект, и вы увидите следующий вывод:

```
<h1>There are 8 products in the source data.</h1>
    <h1>Category: watersports</h1>
    <h1>Category: soccer</h1>
    <h1>Category: chess</h1>
```

В листинге 23-38 показан более сложный пример определения и использования переменной шаблона.

```
{{ define "mainTemplate" -}}
     <h1>There are {{ len . }} products in the source data.
</h1>
    {{- range getCats .  -}}
        {{ if ne ($char := slice (lower .) 0 1) "s"  }}
            <h1>{{$char}}: {{.}}</h1>
        {{- end }}
    {{- end }}
{{- end }}
```

Листинг 23-38 Определение и использование переменной шаблона в файле template.html в папке templates

В этом примере действие `if` использует функции `slice` и `lower` для получения первого символа текущей категории и присваивает его переменной с именем `$char` перед использованием символа для выражения `if`. Доступ к переменной `$char` осуществляется во вложенном содержимом шаблона, что позволяет избежать дублирования использования функций `slice` и `lower`. Скомпилируйте и запустите проект, и вы увидите следующий вывод:

```
<h1>There are 8 products in the source data.</h1>
            <h1>w: Watersports</h1>
            <h1>c: Chess</h1>
```

Использование переменных шаблона в действиях диапазона

Переменные также можно использовать с действием range, что позволяет использовать карты в шаблонах. В листинге 23-39 я обновил

код Go, который выполняет шаблон, чтобы передать карту методу Execute.

```
...
func Exec(t *template.Template) error {
    productMap := map[string]Product {}
    for _, p := range Products {
        productMap[p.Name] = p
    }
    return t.Execute(os.Stdout, &productMap)
}
...
```

Листинг 23-39 Использование карты в файле main.go в папке htmltext

Листинг 23-40 обновляет шаблон для перечисления содержимого карты с использованием переменных шаблона.

```
{{ define "mainTemplate" -}}
    {{ range $key, $value := . -}}
        <h1>{{ $key }}: {{ printf "$%.2f" $value.Price }}
</h1>
    {{ end }}
{{- end }}
```

Листинг 23-40 Перечисление карты в файле template.html в папке templates

Синтаксис неудобен, ключевое слово range, переменные и оператор присваивания появляются в необычном порядке, но в результате ключи и значения в карте можно использовать в шаблоне. Скомпилируйте и запустите проект, и вы увидите следующий вывод:

```
<h1>Bling-Bling King: $1200.00</h1>
    <h1>Corner Flags: $34.95</h1>
    <h1>Kayak: $279.00</h1>
    <h1>Lifejacket: $49.95</h1>
    <h1>Soccer Ball: $19.50</h1>
    <h1>Stadium: $79500.00</h1>
    <h1>Thinking Cap: $16.00</h1>
    <h1>Unsteady Chair: $75.00</h1>
```

Создание текстовых шаблонов

Пакет `html/template` основывается на функциях пакета `text/template`, которые можно использовать непосредственно для выполнения текстовых шаблонов. HTML — это, конечно же, текст, и разница в том, что пакет `text/template` автоматически не экранирует содержимое. Во всем остальном использование текстового шаблона аналогично использованию HTML-шаблона. Добавьте файл с именем `template.txt` в папку `templates` с содержимым, показанным в листинге 23-41.

```
{{ define "mainTemplate" -}}
    {{ range $key, $value := . -}}
        {{ $key }}: {{ printf "$%.2f" $value.Price }}
    {{ end }}
{{- end }}
```

Листинг 23-41 Содержимое файла template.txt в папке templates

Этот шаблон аналогичен шаблону в листинге 23-40, за исключением того, что он не содержит элемента `h1`. Действия шаблона, выражения, переменные и обрезка пробелов одинаковы. И, как показано в листинге 23-42, даже имена функций, используемых для загрузки и выполнения шаблонов, одинаковы, просто доступ к ним осуществляется через другой пакет.

```
package main

import (
    "text/template"
    "os"
    "strings"
)

func GetCategories(products []Product) (categories []string)
{
    catMap := map[string]string {}
    for _, p := range products {
        if (catMap[p.Category] == "") {
            catMap[p.Category] = p.Category
            categories = append(categories, p.Category)
```

```
        }
    }
    return
}

func Exec(t *template.Template) error {
    productMap := map[string]Product {}
    for _, p := range Products {
        productMap[p.Name] = p
    }
    return t.Execute(os.Stdout, &productMap)
}

func main() {
    allTemplates := template.New("allTemplates")
    allTemplates.Funcs(map[string]interface{} {
        "getCats": GetCategories,
        "lower": strings.ToLower,
    })
                                    allTemplates,       err        :=
allTemplates.ParseGlob("templates/*.txt")

    if (err == nil) {
                                            selectedTemplated    :=
allTemplates.Lookup("mainTemplate")
        err = Exec(selectedTemplated)
    }
    if (err != nil) {
        Printfln("Error: %v %v", err.Error())
    }
}
```

Листинг 23-42 Загрузка и выполнение текстового шаблона в файле main.go в папке htmltext

Помимо изменения оператора `import` и выбора файлов с расширением `txt`, процесс загрузки и выполнения текстового шаблона остается прежним. Скомпилируйте и запустите проект, и вы увидите следующий вывод:

```
Bling-Bling King: $1200.00
    Corner Flags: $34.95
    Kayak: $279.00
```

```
Lifejacket: $49.95
Soccer Ball: $19.50
Stadium: $79500.00
Thinking Cap: $16.00
Unsteady Chair: $75.00
```

Резюме

В этой главе я описал стандартную библиотеку для создания HTML и текстовых шаблонов. Шаблоны могут содержать широкий спектр действий, которые используются для включения содержимого в выходные данные. Синтаксис шаблонов может быть неудобным — и нужно позаботиться о том, чтобы отображать содержимое точно так, как этого требует механизм шаблонов, — но механизм шаблонов гибок и расширяем, и, как я покажу в третьей части, его можно легко модифицировать, чтобы изменить его поведение.

24. Создание HTTP-серверов

В этой главе я описываю поддержку стандартной библиотеки для создания серверов HTTP и обработки запросов HTTP и HTTPS. Я покажу вам, как создать сервер, и объясню различные способы обработки запросов, включая запросы форм. Таблица 24-1 помещает серверы HTTP в контекст.

Таблица 24-1 Помещение HTTP-серверов в контекст

Вопрос	Ответ
Кто они такие?	Функции, описанные в этой главе, позволяют приложениям Go легко создавать HTTP-серверы.
Почему они полезны?	HTTP является одним из наиболее широко используемых протоколов и полезен как для пользовательских приложений, так и для веб-служб.
Как это используется?	Возможности пакета `net/http` используются для создания сервера и обработки запросов.
Есть ли подводные камни или ограничения?	Эти функции хорошо продуманы и просты в использовании.
Есть ли альтернативы?	Стандартная библиотека включает поддержку других сетевых протоколов, а также для открытия и использования сетевых соединений более низкого уровня. См. `https://pkg.go.dev/net@go1.17.1` для получения подробной информации о пакете `net` и его подпакетах, таких как, например, `net/smtp`, который реализует протокол SMTP.

Таблица 24-2 суммирует содержание главы.

Таблица 24-2 Краткое содержание главы

Проблема	Решение	Листинг
Создать HTTP или HTTPS сервер	Используйте функции `ListenAndServe` или `ListenAndServeTLS`	6, 7, 11
Проверить HTTP-запрос	Используйте возможности структуры `Request`	8
Произвести ответ	Используйте интерфейс `ResponseWriter` или удобные функции	9
Обрабатывать запросы к определенным URL-адресам	Используйте встроенный маршрутизатор	10, 12
Обслуживать статический контент	Используйте функцию `FileServer` и `StripPrefix`	13–17
Используйте шаблон для создания ответа или создания ответа JSON	Запишите содержимое в `ResponseWriter`	18–20
Обработка данных формы	Используйте методы запроса	21–25
Установить или прочитать файлы cookie	Используйте методы `Cookie`, `Cookies` и `SetCookie`	26

Подготовка к этой главе

Чтобы подготовиться к этой главе, откройте новую командную строку, перейдите в удобное место и создайте каталог с именем `httpsserver`. Запустите команду, показанную в листинге 24-1, чтобы создать файл модуля.

> **Подсказка**
> Вы можете загрузить пример проекта для этой главы — и для всех остальных глав этой книги — с `https://github.com/apress/pro-go`. См. Главу 2 о том, как получить помощь, если у вас возникнут проблемы с запуском примеров.

```
go mod init httpserver
```

Листинг 24-1 Инициализация модуля

Добавьте файл с именем `printer.go` в папку `httpsserver` с содержимым, показанным в листинге 24-2.

```
package main

import "fmt"

func Printfln(template string, values ...interface{}) {
    fmt.Printf(template + "\n", values...)
}
```

Листинг 24-2 Содержимое файла print.go в папке httpsserver

Добавьте файл с именем `product.go` в папку `httpsserver` с содержимым, показанным в листинге 24-3.

```
package main

type Product struct {
    Name, Category string
    Price float64
}

var Products = []Product {
    { "Kayak", "Watersports", 279 },
    { "Lifejacket", "Watersports", 49.95 },
    { "Soccer Ball", "Soccer", 19.50 },
    { "Corner Flags", "Soccer", 34.95 },
    { "Stadium", "Soccer", 79500 },
    { "Thinking Cap", "Chess", 16 },
    { "Unsteady Chair", "Chess", 75 },
    { "Bling-Bling King", "Chess", 1200 },
}
```

Листинг 24-3 Содержимое файла product.go в папке httpsserver

Добавьте файл с именем main.go в папку httpsserver с содержимым, показанным в листинге 24-4.

```
package main

func main() {
    for _, p := range Products {
        Printfln("Product: %v, Category: %v, Price: $%.2f",
            p.Name, p.Category, p.Price)
    }
}
```

Листинг 24-4 Содержимое файла main.go в папке httpsserver

Используйте командную строку для запуска команды, показанной в листинге 24-5, в папке httpsserver.

```
go run .
```

Листинг 24-5 Запуск примера проекта

Проект будет скомпилирован и выполнен, что приведет к следующему результату:

```
Product: Kayak, Category: Watersports, Price: $279.00
Product: Lifejacket, Category: Watersports, Price: $49.95
Product: Soccer Ball, Category: Soccer, Price: $19.50
Product: Corner Flags, Category: Soccer, Price: $34.95
Product: Stadium, Category: Soccer, Price: $79500.00
Product: Thinking Cap, Category: Chess, Price: $16.00
Product: Unsteady Chair, Category: Chess, Price: $75.00
Product: Bling-Bling King, Category: Chess, Price: $1200.00
```

Создание простого HTTP-сервера

Пакет net/http упрощает создание простого HTTP-сервера, который затем можно расширить, добавив более сложные и полезные функции. В листинге 24-6 показан сервер, который отвечает на запросы простым строковым ответом.

```
package main

import (
    "net/http"
    "io"
)

type StringHandler struct {
    message string
}

func (sh StringHandler) ServeHTTP(writer http.ResponseWriter,
        request *http.Request) {
```

```
    io.WriteString(writer, sh.message)
}

func main() {
     err := http.ListenAndServe(":5000", StringHandler{ message: "Hello,
World"})
    if (err != nil) {
        Printfln("Error: %v", err.Error())
    }
}
```

Листинг 24-6 Создание простого HTTP-сервера в файле main.go в папке httpsserver

Здесь всего несколько строк кода, но их достаточно для создания HTTP-сервера, отвечающего на запросы `Hello, World`. Скомпилируйте и выполните проект, а затем используйте веб-браузер для запроса `http://localhost:5000`, что даст результат, показанный на рисунке 24-1

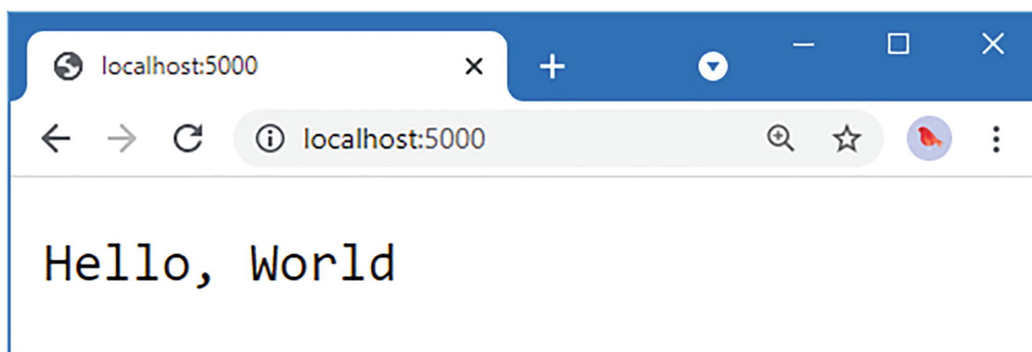

Рисунок 24-1 Ответ на HTTP-запрос

РАБОТА С ЗАПРОСАМИ РАЗРЕШЕНИЙ БРАНДМАУЭРА WINDOWS

Пользователям Windows встроенный брандмауэр может предложить разрешить доступ к сети. К сожалению, команда `go run` создает исполняемый файл по уникальному пути при каждом запуске, а это означает, что вам будет предложено предоставить доступ каждый раз, когда вы вносите изменения и выполняете код. Чтобы решить эту проблему, создайте файл с именем `buildandrun.ps1` в папке проекта со следующим содержимым:

```
$file = "./httpserver.exe"

&go build -o $file

if ($LASTEXITCODE -eq 0) {
    &$file
}
```

Этот сценарий PowerShell каждый раз компилирует проект в один и тот же файл, а затем выполняет результат, если ошибок нет, то есть вам нужно будет

предоставить доступ к брандмауэру только один раз. Скрипт выполняется запуском этой команды в папке проекта:

```
./buildandrun.ps1
```

Вы должны использовать эту команду каждый раз для сборки и выполнения проекта, чтобы убедиться, что скомпилированные выходные данные записываются в одно и то же место.

Хотя в листинге 24-6 несколько строк кода, их распаковка занимает некоторое время. Но стоит потратить время на то, чтобы понять, как был создан HTTP-сервер, потому что он многое говорит о возможностях, предоставляемых пакетом `net/http`.

Создание прослушивателя и обработчика HTTP

Пакет `net/http` предоставляет набор удобных функций, упрощающих создание HTTP-сервера без необходимости указывать слишком много деталей. Таблица 24-3 описывает удобные функции для настройки сервера.

Таблица 24-3 Удобные функции net/http

Функция	Описание
`ListenAndServe(addr, handler)`	Эта функция начинает прослушивать HTTP-запросы по указанному адресу и передает запросы указанному обработчику.
`ListenAndServeTLS(addr, cert, key, handler)`	Эта функция начинает прослушивать HTTPS-запросы. Аргументы - это адрес

Функция `ListenAndServe` начинает прослушивание HTTP-запросов по указанному сетевому адресу. Функция `ListenAndServeTLS` делает то же самое для HTTP-запросов, которые я демонстрирую в разделе «Поддержка HTTPS-запросов».

Адреса, принимаемые функциями в таблице 24-3, можно использовать для ограничения HTTP-сервера, чтобы он принимал запросы только на определенном интерфейсе или прослушивал запросы на любом интерфейсе. В листинге 24-6 используется последний подход, который заключается в указании только номера порта:

```
...
err := http.ListenAndServe(":5000", StringHandler{ message: "Hello, World"})
...
```

Имя или адрес не указаны, а номер порта следует за двоеточием, что означает, что этот оператор создает HTTP-сервер, который прослушивает запросы на порту 5000 на всех интерфейсах.

Когда приходит запрос, он передается обработчику, который отвечает за получение ответа. Обработчики должны реализовать интерфейс `Handler`, который определяет метод, описанный в таблице 24-4.

Таблица 24-4 Метод, определяемый интерфейсом обработчика

Функция	Описание
ServeHTTP(writer, request)	Этот метод вызывается для обработки HTTP-запроса. Запрос описывается значением Request, а ответ записывается с использованием ResponseWriter, оба из которых принимаются в качестве параметров.

Я опишу типы Request и ResponseWriter более подробно в следующих разделах, но интерфейс ResponseWriter определяет метод Write, необходимый для интерфейса Writer, описанный в главе 20, что означает, что я могу создать string ответ, используя функцию WriteString, определенную в пакет io:

```
...
io.WriteString(writer, sh.message)
...
```

Сложите эти функции вместе, и в результате получится HTTP-сервер, который прослушивает запросы на порту 5000 на всех интерфейсах и создает ответы, записывая строку. О таких деталях, как открытие сетевого соединения и анализ HTTP-запросов, заботятся за кулисами.

Проверка запроса

HTTP-запросы представлены структурой Request, определенной в пакете net/http. В таблице 24-5 описаны основные поля, определенные структурой Request.

Таблица 24-5 Основные поля, определяемые структурой запроса

Функция	Описание
Method	В этом поле указывается метод HTTP (GET, POST и т. д.) в виде строки. Пакет net/http определяет константы для методов HTTP, таких как MethodGet и MethodPost.
URL	Это поле возвращает запрошенный URL-адрес, выраженный в виде URL значения.
Proto	Это поле возвращает string, указывающую версию HTTP, используемую для запроса.
Host	Это поле возвращает string, содержащую запрошенный хост.
Header	Это поле возвращает значение Header, которое является псевдонимом для map[string][]string и содержит заголовки запроса. Ключи карты — это имена заголовков, а значения — срезы строк, содержащие значения заголовков.
Trailer	Это поле возвращает строку map[string], содержащую любые дополнительные заголовки, включенные в запрос после тела.
Body	Это поле возвращает ReadCloser, представляющий собой интерфейс, сочетающий метод Read интерфейса Reader с методом Close интерфейса Closer, оба из которых описаны в главе 22.

В листинге 24-7 к функции обработчика запросов добавлены операторы, которые выводят значения из основных полей Request в стандартный вывод.

```
package main

import (
    "net/http"
```

```
    "io"
)

type StringHandler struct {
    message string
}

func (sh StringHandler) ServeHTTP(writer http.ResponseWriter,
        request *http.Request) {
    Printfln("Method: %v", request.Method)
    Printfln("URL: %v", request.URL)
    Printfln("HTTP Version: %v", request.Proto)
    Printfln("Host: %v", request.Host)
    for name, val := range  request.Header {
        Printfln("Header: %v, Value: %v", name, val)
    }
    Printfln("---")
    io.WriteString(writer, sh.message)
}

func main() {
     err := http.ListenAndServe(":5000", StringHandler{ message: "Hello,
World"})
    if (err != nil) {
        Printfln("Error: %v", err.Error())
    }
}
```

Листинг 24-7 Запись полей запроса в файл main.go в папке httpsserver

Скомпилируйте и запустите проект и запросите `http://localhost:5000`. Вы увидите тот же ответ в окне браузера, что и в предыдущем примере, но на этот раз он также будет выводиться в командной строке. Точный вывод будет зависеть от вашего браузера, но вот вывод, который я получил с помощью Google Chrome:

```
Method: GET
URL: /
HTTP Version: HTTP/1.1
Host: localhost:5000
Header: Upgrade-Insecure-Requests, Value: [1]
Header: Sec-Fetch-Site, Value: [none]
Header: Sec-Fetch-Mode, Value: [navigate]
Header: Sec-Fetch-User, Value: [?1]
Header: Accept-Encoding, Value: [gzip, deflate, br]
Header: Connection, Value: [keep-alive]
Header: Cache-Control, Value: [max-age=0]
Header: User-Agent, Value: [Mozilla/5.0 (Windows NT 10.0; Win64; x64)
         AppleWebKit/537.36  (KHTML,  like  Gecko)  Chrome/91.0.4472.124
Safari/537.36]
Header:                          Accept,                          Value:
[text/html,application/xhtml+xml,application/xml;q=0.9,
```

```
    image/avif,image/webp,image/apng,*/*;q=0.8,application/signedexchange;
    v=b3;q=0.9]
Header: Sec-Fetch-Dest, Value: [document]
Header: Sec-Ch-Ua, Value: [" Not;A Brand";v="99", "Google Chrome";v="91",
    "Chromium";v="91"]
Header: Accept-Language, Value: [en-GB,en-US;q=0.9,en;q=0.8]
Header: Sec-Ch-Ua-Mobile, Value: [?0]
---
Method: GET
URL: /favicon.ico
HTTP Version: HTTP/1.1
Host: localhost:5000
Header: Sec-Fetch-Site, Value: [same-origin]
Header: Sec-Fetch-Dest, Value: [image]
Header: Referer, Value: [http://localhost:5000/]
Header: Pragma, Value: [no-cache]
Header: Cache-Control, Value: [no-cache]
Header: User-Agent, Value: [Mozilla/5.0 (Windows NT 10.0; Win64; x64)
        AppleWebKit/537.36  (KHTML,  like  Gecko)  Chrome/91.0.4472.124
Safari/537.36]
Header: Accept-Language, Value: [en-GB,en-US;q=0.9,en;q=0.8]
Header: Sec-Ch-Ua, Value: [" Not;A Brand";v="99", "Google Chrome";v="91",
    "Chromium";v="91"]
Header: Sec-Ch-Ua-Mobile, Value: [?0]
Header: Sec-Fetch-Mode, Value: [no-cors]
Header: Accept-Encoding, Value: [gzip, deflate, br]
Header: Connection, Value: [keep-alive]
Header: Accept, Value:[image/avif,image/webp,image/apng,image/svg+xml,
    image/*,*/*;q=0.8]
---
```

Браузер делает два HTTP-запроса. Первый предназначен для `/`, который является компонентом пути запрошенного URL-адреса. Второй запрос относится к `/favicon.ico`, который браузер отправляет, чтобы получить значок, отображаемый в верхней части окна или вкладки.

> **ИСПОЛЬЗОВАНИЕ КОНТЕКСТА ЗАПРОСА**
> Пакет `net/http` определяет метод `Context` для структуры `Request`, который возвращает реализацию интерфейса `context.Context`. Интерфейс `Context` используется для управления потоком запросов через приложение и описан в главе 30. В третьей части я использую функцию `Context` в пользовательской веб-платформе и интернет-магазине.

Фильтрация запросов и генерация ответов

HTTP-сервер отвечает на все запросы одинаково, что не идеально. Чтобы получить разные ответы, мне нужно проверить URL-адрес, чтобы выяснить, что запрашивается, и использовать функции, предоставляемые пакетом `net/http`, для

отправки соответствующего ответа. Наиболее полезные поля и методы, определенные структурой URL, описаны в таблице 24-6.

Таблица 24-6 Полезные поля и методы, определяемые структурой URL

Функция	Описание
`Scheme`	Это поле возвращает компонент схемы URL.
`Host`	Это поле возвращает хост-компонент URL-адреса, который может включать порт.
`RawQuery`	Это поле возвращает строку запроса из URL-адреса. Используйте метод `Query` для преобразования строки запроса в карту.
`Path`	Это поле возвращает компонент пути URL-адреса.
`Fragment`	Это поле возвращает компонент фрагмента URL без символа `#`.
`Hostname()`	Этот метод возвращает компонент имени хоста URL-адреса в виде `string`.
`Port()`	Этот метод возвращает компонент порта URL-адреса в виде `string`.
`Query()`	Этот метод возвращает строку `map[string][]string` (карту со строковыми ключами и строковыми значениями срезов), содержащую поля строки запроса.
`User()`	Этот метод возвращает информацию о пользователе, связанную с запросом, как описано в главе 30.
`String()`	Этот метод возвращает `string` представление URL-адреса.

Интерфейс `ResponseWriter` определяет методы, доступные при создании ответа. Как отмечалось ранее, этот интерфейс включает метод `Write`, так что его можно использовать в качестве Writer, но `ResponseWriter` также определяет методы, описанные в таблице 24-7. Обратите внимание, что вы должны завершить настройку заголовков перед использованием метода `Write`.

Таблица 24-7 Метод ResponseWriter

Функция	Описание
`Header()`	Этот метод возвращает `Header`, который является псевдонимом для `map[string][]string`, который можно использовать для установки заголовков ответа.
`WriteHeader(code)`	Этот метод устанавливает код состояния для ответа, заданного как `int`. Пакет `net/http` определяет константы для большинства кодов состояния.
`Write(data)`	Этот метод записывает данные в тело ответа и реализует интерфейс `Writer`.

В листинге 24-8 я обновил свою функцию обработчика запросов, чтобы она выдавала ответ 404 Not Found на запросы файлов значков.

```
package main

import (
    "net/http"
    "io"
)

type StringHandler struct {
    message string
}
```

```
func (sh StringHandler) ServeHTTP(writer http.ResponseWriter,
        request *http.Request) {
    if (request.URL.Path == "/favicon.ico") {
        Printfln("Request for icon detected - returning 404")
        writer.WriteHeader(http.StatusNotFound)
        return
    }
    Printfln("Request for %v", request.URL.Path)
    io.WriteString(writer, sh.message)
}

func main() {
     err := http.ListenAndServe(":5000", StringHandler{ message: "Hello,
World"})
    if (err != nil) {
        Printfln("Error: %v", err.Error())
    }
}
```

Листинг 24-8 Создание разностных ответов в файле main.go в папке httpsserver

Обработчик запроса проверяет поле `URL.Path`, чтобы обнаружить запросы значков, и отвечает, используя `WriteHeader`, чтобы установить ответ, используя константу `StatusNotFound` (хотя я мог бы просто указать литеральное значение `404`). Скомпилируйте и запустите проект и используйте браузер для запроса `http://localhost:5000`. Браузер получит ответ, показанный на рисунке 24-1, и вы увидите следующий вывод приложения Go в командной строке:

```
Request for /
Request for icon detected - returning 404
```

Вы можете обнаружить, что последующие запросы от браузера на `http://localhost:5000` не вызывают второй запрос на файл значка. Это потому, что браузер отмечает ответ 404 и знает, что для этого URL-адреса нет файла значка. Очистите кеш браузера и запросите `http://localhost:5000`, чтобы вернуться к исходному поведению.

Использование удобных функций ответа

Пакет `net/http` предоставляет набор удобных функций, которые можно использовать для создания стандартных ответов на HTTP-запросы, как описано в таблице 24-8.

Таблица 24-8 Удобные функции ответа

Функция	Описание
`Error(writer, message, code)`	Эта функция устанавливает для заголовка указанный код, устанавливает для заголовка `Content-Type` значение `text/plain` и записывает сообщение об ошибке в ответ. Заголовок `X-Content-Type-Options` также настроен так, чтобы браузеры не могли интерпретировать ответ как что-либо, кроме текста.

Функция	Описание
`NotFound(writer, request)`	Эта функция вызывает `Error` и указывает код ошибки 404.
`Redirect(writer, request, url, code)`	Эта функция отправляет ответ о перенаправлении на указанный URL-адрес и с указанным кодом состояния.
`ServeFile(writer, request, fileName)`	Эта функция отправляет ответ, содержащий содержимое указанного файла. Заголовок `Content-Type` устанавливается на основе имени файла, но его можно переопределить, явно установив заголовок перед вызовом функции. См. раздел «Создание статического HTTP-сервера» для примера, который обслуживает файлы.

В листинге 24-9 я использовал функцию `NotFound` для реализации простой схемы обработки URL.

```
package main

import (
    "net/http"
    "io"
)

type StringHandler struct {
    message string
}

func (sh StringHandler) ServeHTTP(writer http.ResponseWriter,
        request *http.Request) {
    Printfln("Request for %v", request.URL.Path)
    switch request.URL.Path {
        case "/favicon.ico":
            http.NotFound(writer, request)
        case "/message":
            io.WriteString(writer, sh.message)
        default:
                            http.Redirect(writer, request, "/message",
http.StatusTemporaryRedirect)
    }
}

func main() {
     err := http.ListenAndServe(":5000", StringHandler{ message: "Hello,
World"})
    if (err != nil) {
        Printfln("Error: %v", err.Error())
    }
}
```

Листинг 24-9 Использование функций удобства в файле main.go в папке httpsserver

В листинге 24-9 используется оператор `switch`, чтобы решить, как реагировать на запрос. Скомпилируйте и запустите проект и используйте браузер для запроса

http://localhost:5000/message, который даст ответ, ранее показанный на рисунке 24-1. Если браузер запрашивает файл значка, сервер возвращает ответ 404. Для всех остальных запросов браузеру отправляется перенаправление на /message.

Использование обработчика удобной маршрутизации

Процесс проверки URL-адреса и выбора ответа может привести к созданию сложного кода, который трудно читать и поддерживать. Чтобы упростить этот процесс, пакет net/http предоставляет реализацию обработчика, которая позволяет отделить сопоставление URL-адреса от создания запроса, как показано в листинге 24-10.

```
package main

import (
    "net/http"
    "io"
)

type StringHandler struct {
    message string
}

func (sh StringHandler) ServeHTTP(writer http.ResponseWriter,
        request *http.Request) {
    Printfln("Request for %v", request.URL.Path)
    io.WriteString(writer, sh.message)
}

func main() {
    http.Handle("/message", StringHandler{ "Hello, World"})
    http.Handle("/favicon.ico", http.NotFoundHandler())
                    http.Handle("/",     http.RedirectHandler("/message",
http.StatusTemporaryRedirect))

    err := http.ListenAndServe(":5000", nil)
    if (err != nil) {
        Printfln("Error: %v", err.Error())
    }
}
```

Листинг 24-10 Использование обработчика удобной маршрутизации в файле main.go в папке httpsserver

Ключом к этой функции является использование nil в качестве аргумента функции ListenAndServe, например:

```
...
err := http.ListenAndServe(":5000", nil)
...
```

Это включает обработчик по умолчанию, который направляет запросы обработчикам на основе правил, установленных с помощью функций, описанных в таблице 24-9.

Таблица 24-9 Функции net/http для создания правил маршрутизации

Функция	Описание
`Handle(pattern, handler)`	Эта функция создает правило, которое вызывает указанный метод `ServeHTTP` указанного `Handler` для запросов, соответствующих шаблону.
`HandleFunc(pattern, handlerFunc)`	Эта функция создает правило, которое вызывает указанную функцию для запросов, соответствующих шаблону. Функция вызывается с аргументами `ResponseWriter` и `Request`.

Чтобы помочь установить правила маршрутизации, пакет `net/http` предоставляет функции, описанные в таблице 24-10, которые создают реализации обработчиков, некоторые из которых обертывают функции ответа, описанные в таблице 24-7.

Таблица 24-10 The net/http Functions for Creating Request Handlers

Функция	Описание
`FileServer(root)`	Эта функция создает `Handler`, который выдает ответы с помощью функции `ServeFile`. См. раздел «Создание статического HTTP-сервера» для примера, который обслуживает файлы.
`NotFoundHandler()`	Эта функция создает `Handler`, который выдает ответы с помощью функции `NotFound`.
`RedirectHandler(url, code)`	Эта функция создает `Handler`, который выдает ответы с помощью функции `Redirect`.
`StripPrefix(prefix, handler)`	Эта функция создает `Handler`, который удаляет указанный префикс из URL-адреса запроса и передает запрос указанному `Handler`. Подробнее см. в разделе «Создание статического HTTP-сервера».
`TimeoutHandler(handler, duration, message)`	Эта функция передает запрос указанному `Handler`, но генерирует ответ об ошибке, если ответ не был получен в течение указанного времени.

Шаблоны, используемые для сопоставления запросов, выражаются в виде путей, таких как `/favicon.ico`, или в виде деревьев, заканчивающихся косой чертой, таких как `/files/`. Сначала сопоставляются самые длинные шаблоны, а корневой путь (`"/"`) соответствует любому запросу и действует как резервный маршрут.

В листинге 24-10 я использовал функцию `Handle` для настройки трех маршрутов:

```
...
http.Handle("/message", StringHandler{ "Hello, World"})
http.Handle("/favicon.ico", http.NotFoundHandler())
http.Handle("/",                          http.RedirectHandler("/message",
http.StatusTemporaryRedirect))
...
```

В результате запросы на `/message` направляются в `StringHandler`, запросы на `/favicon.ico` обрабатываются с ответом `404 Not Found`, а все остальные запросы

вызывают перенаправление на /message. Это та же конфигурация, что и в предыдущем разделе, но сопоставление между URL-адресами и обработчиками запросов осуществляется отдельно от кода, создающего ответы.

Поддержка HTTPS-запросов

Пакет net/http обеспечивает встроенную поддержку HTTPS. Для подготовки к HTTPS вам потребуется добавить в папку httpsserver два файла: файл сертификата и файл закрытого ключа.

ПОЛУЧЕНИЕ СЕРТИФИКАТОВ ДЛЯ HTTPS

Хороший способ начать работу с HTTPS — использовать самоподписанный сертификат, который можно использовать для разработки и тестирования. Если у вас еще нет самозаверяющего сертификата, вы можете создать его в Интернете с помощью таких сайтов, как https://getacert.com или https://www.selfsignedcertificate.com, оба из которых позволят вам создать самоподписанный сертификат легко и бесплатно.

Для использования HTTPS необходимы два файла, независимо от того, является ли ваш сертификат самоподписанным или нет. Первый — это файл сертификата, который обычно имеет расширение cer или cert. Второй — это файл закрытого ключа, который обычно имеет расширение файла key.

Когда вы будете готовы развернуть свое приложение, вы можете использовать настоящий сертификат. Я рекомендую https://letsencrypt.org, который предлагает бесплатные сертификаты и (относительно) прост в использовании. Я не могу помочь читателям получить и использовать сертификаты, поскольку для этого требуется контроль над доменом, для которого выдан сертификат, и доступ к закрытому ключу, который должен оставаться в секрете. Если у вас возникли проблемы по примеру, то рекомендую использовать самоподписанный сертификат.

Функция ListenAndServeTLS используется для включения HTTPS, где дополнительные аргументы указывают файлы сертификата и закрытого ключа, которые в моем проекте называются certificate.cer и certificate.key, как показано в листинге 24-11.

```
package main

import (
    "net/http"
    "io"
)

type StringHandler struct {
    message string
}

func (sh StringHandler) ServeHTTP(writer http.ResponseWriter,
        request *http.Request) {
```

```
    Printfln("Request for %v", request.URL.Path)
    io.WriteString(writer, sh.message)
}

func main() {
    http.Handle("/message", StringHandler{ "Hello, World"})
    http.Handle("/favicon.ico", http.NotFoundHandler())
                    http.Handle("/",     http.RedirectHandler("/message",
http.StatusTemporaryRedirect))

    go func () {
        err := http.ListenAndServeTLS(":5500", "certificate.cer",
            "certificate.key", nil)
        if (err != nil) {
            Printfln("HTTPS Error: %v", err.Error())
        }
    }()

    err := http.ListenAndServe(":5000", nil)
    if (err != nil) {
        Printfln("Error: %v", err.Error())
    }
}
```

Листинг 24-11 Включение HTTPS в файле main.go в папке httpsserver

Функции `ListenAndServeTLS` и `ListenAndServe` блокируются, поэтому я использовал горутину для поддержки HTTP- и HTTPS-запросов, причем HTTP обрабатывается через порт 5000, а HTTPS — через порт 5500.

Функции `ListenAndServeTLS` и `ListenAndServe` были вызваны с `nil` в качестве обработчика, что означает, что запросы HTTP и HTTPS будут обрабатываться с использованием одного и того же набора маршрутов. Скомпилируйте и запустите проект и используйте браузер для запроса `http://localhost:5000` и `https://localhost:5500`. Запросы будут обрабатываться таким же образом, как показано на рисунке 24-2. Если вы используете самоподписанный сертификат, ваш браузер предупредит вас о том, что сертификат недействителен, и вам придется принять на себя риск безопасности, прежде чем браузер отобразит содержимое.

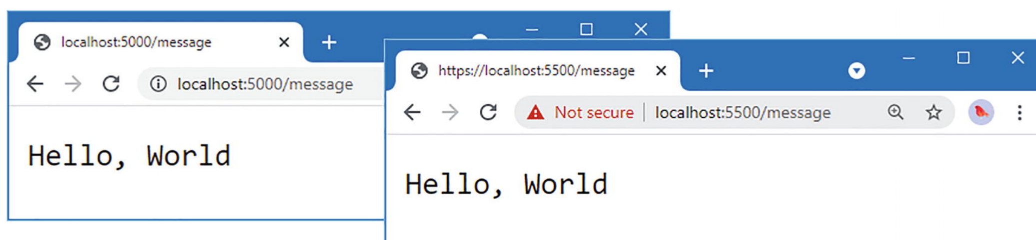

Рисунок 24-2 Поддержка HTTPS-запросов

Перенаправление HTTP-запросов на HTTPS

Общим требованием при создании веб-серверов является перенаправление HTTP-запросов на порт HTTPS. Это можно сделать, создав собственный обработчик, как показано в листинге 24-12.

```
package main

import (
    "net/http"
    "io"
    "strings"
)

type StringHandler struct {
    message string
}

func (sh StringHandler) ServeHTTP(writer http.ResponseWriter,
        request *http.Request) {
    Printfln("Request for %v", request.URL.Path)
    io.WriteString(writer, sh.message)
}

func HTTPSRedirect(writer http.ResponseWriter,
        request *http.Request) {
    host := strings.Split(request.Host, ":")[0]
    target := "https://" + host + ":5500" + request.URL.Path
    if len(request.URL.RawQuery) > 0 {
        target += "?" + request.URL.RawQuery
    }
    http.Redirect(writer, request, target, http.StatusTemporaryRedirect)
}

func main() {
    http.Handle("/message", StringHandler{ "Hello, World"})
    http.Handle("/favicon.ico", http.NotFoundHandler())
                    http.Handle("/",     http.RedirectHandler("/message",
http.StatusTemporaryRedirect))

    go func () {
        err := http.ListenAndServeTLS(":5500", "certificate.cer",
            "certificate.key", nil)
        if (err != nil) {
            Printfln("HTTPS Error: %v", err.Error())
        }
    }()

    err := http.ListenAndServe(":5000", http.HandlerFunc(HTTPSRedirect))
    if (err != nil) {
        Printfln("Error: %v", err.Error())
    }
}
```

Листинг 24-12 Перенаправление на HTTPS в файле main.go в папке httpsserver

Обработчик HTTP в листинге 24-12 перенаправляет клиента на URL-адрес HTTPS. Скомпилируйте и запустите проект и запросите `http://localhost:5000`. Ответ перенаправит браузер на службу HTTPS, создав вывод, показанный на рисунке 24-3.

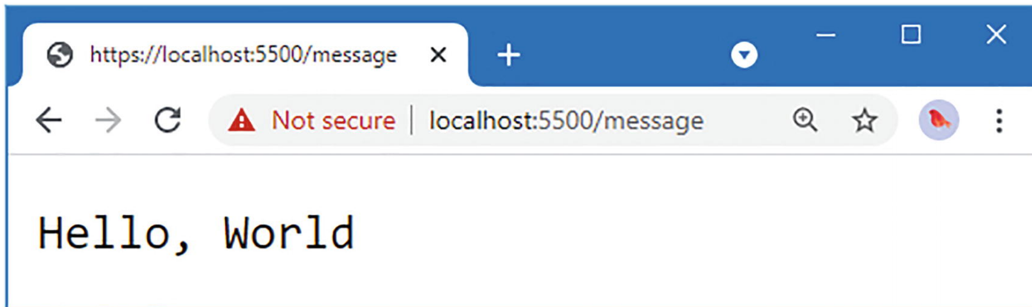

Рисунок 24-3 Использование HTTPS

Создание статического HTTP-сервера

Пакет `net/http` включает встроенную поддержку для ответа на запросы с содержимым файлов. Чтобы подготовиться к статическому HTTP-серверу, создайте папку `httpsserver/static` и добавьте в нее файл с именем `index.html` с содержимым, показанным в листинге 24-13.

> **Примечание**
> Все атрибуты класса в HTML-файлах и шаблонах в этой главе имеют стили, определенные в CSS-пакете Bootstrap, который добавляется в проект в листинге 24-15. См. `https://getbootstrap.com` для получения подробной информации о том, что делает каждый класс, и о других функциях, предоставляемых пакетом Bootstrap.

```
<!DOCTYPE html>
<html>
<head>
    <title>Pro Go</title>
    <meta name="viewport" content="width=device-width" />
    <link href="bootstrap.min.css" rel="stylesheet" />
</head>
<body>
    <div class="m-1 p-2 bg-primary text-white h2">
        Hello, World
    </div>
</body>
</html>
```

Листинг 24-13 Содержимое файла index.html в папке static

Затем добавьте файл с именем `store.html` в папку `httpsserver/static` с содержимым, показанным в листинге 24-14.

```
<!DOCTYPE html>
<html>
<head>
    <title>Pro Go</title>
    <meta name="viewport" content="width=device-width" />
    <link href="bootstrap.min.css" rel="stylesheet" />
</head>
<body>
    <div class="m-1 p-2 bg-primary text-white h2 text-center">
        Products
    </div>
    <table class="table table-sm table-bordered table-striped">
        <thead>
            <tr><th>Name</th><th>Category</th><th>Price</th></tr>
        </thead>
        <tbody>
            <tr><td>Kayak</td><td>Watersports</td><td>$279.00</td></tr>
               <tr><td>Lifejacket</td><td>Watersports</td><td>$49.95</td>
</tr>
        </tbody>
    </table>
</body>
</html>
```

Листинг 24-14 Содержимое файла store.html в папке static

Файлы HTML зависят от пакета Bootstrap CSS для стилизации содержимого HTML. Запустите команду, показанную в листинге 24-15, в папке `httpsserver`, чтобы загрузить CSS-файл Bootstrap в папку `static`. (Возможно, вам придется установить команду `curl`.)

```
curl
https://cdn.jsdelivr.net/npm/bootstrap@5.0.2/dist/css/bootstrap.min.css -
-output static/bootstrap.min.css
```

Листинг 24-15 Загрузка файла CSS

Если вы используете Windows, вы можете загрузить файл CSS с помощью команды PowerShell, показанной в листинге 24-16.

```
Invoke-WebRequest -OutFile static/bootstrap.min.css -Uri
https://cdn.jsdelivr.net/npm/bootstrap@5.0.2/dist/css/bootstrap.min.css
```

Листинг 24-16 Загрузка файла CSS (Windows)

Создание статического маршрута к файлу

Теперь, когда есть файлы HTML и CSS для работы, пришло время определить маршрут, который сделает их доступными для запросов по HTTP, как показано в

листинге 24-17.

```
...
func main() {
    http.Handle("/message", StringHandler{ "Hello, World"})
    http.Handle("/favicon.ico", http.NotFoundHandler())
                    http.Handle("/",     http.RedirectHandler("/message",
http.StatusTemporaryRedirect))

    fsHandler := http.FileServer(http.Dir("./static"))
    http.Handle("/files/", http.StripPrefix("/files", fsHandler))

    go func () {
        err := http.ListenAndServeTLS(":5500", "certificate.cer",
            "certificate.key", nil)
        if (err != nil) {
            Printfln("HTTPS Error: %v", err.Error())
        }
    }()

    err := http.ListenAndServe(":5000", http.HandlerFunc(HTTPSRedirect))
    if (err != nil) {
        Printfln("Error: %v", err.Error())
    }
}
...
```

Листинг 24-17 Определение маршрута в файле main.go в папке httpsserver

Функция FileServer создает обработчик, который будет обслуживать файлы, а каталог указывается с помощью функции Dir. (Можно обслуживать файлы напрямую, но требуется осторожность, поскольку легко разрешить запросы на выбор файлов за пределами целевой папки. Самый безопасный вариант — использовать функцию Dir, как показано в этом примере.)

Я собираюсь предоставлять содержимое в папке static с URL-адресами, начинающимися с files, так что, например, запрос /files/store.html будет обрабатываться с использованием файла static/store.html. Для этого я использовал функцию StripPrefix, которая создает обработчик, который удаляет префикс пути и передает запрос другому обработчику для обслуживания. Объединение этих обработчиков, как я сделал в листинге 24-17, означает, что я могу безопасно открывать содержимое папки static, используя префикс files.

Обратите внимание, что я указал маршрут с завершающей косой чертой, например:

```
...
http.Handle("/files/", http.StripPrefix("/files", fsHandler))
...
```

Как отмечалось ранее, встроенный маршрутизатор поддерживает пути и деревья, а для маршрутизации каталога требуется дерево, которое указывается косой чертой в конце. Скомпилируйте и выполните проект и используйте браузер для запроса `https://localhost:5500/files/store.html`, и вы получите ответ, показанный на рисунке 24-4.

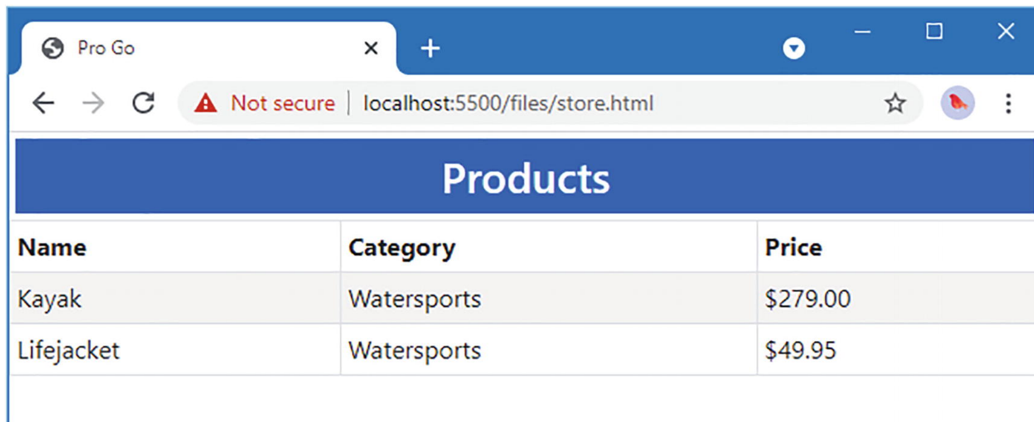

Рисунок 24-4 Обслуживание статического контента

Поддержка обслуживания файлов имеет несколько полезных функций. Во-первых, заголовок ответа Content-Type устанавливается автоматически на основе расширения файла. Во-вторых, запросы, в которых не указан файл, обрабатываются с помощью `index.html`, что можно увидеть, запросив `https://localhost:5500/files`, что дает ответ, показанный на рисунке 24-5. Наконец, если в запросе указан файл, но файл не существует, то автоматически отправляется ответ 404, также показанный на рисунке 24-5.

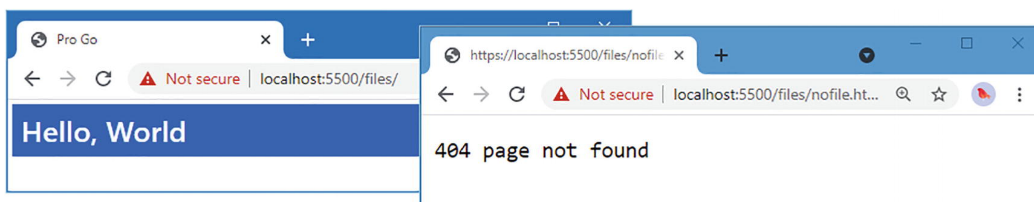

Рисунок 24-5 Запасные ответы

Использование шаблонов для генерации ответов

Встроенной поддержки использования шаблонов в качестве ответов на HTTP-запросы нет, но настроить обработчик, использующий функции, предоставляемые пакетом `html/template`, который я описал в главе 23. Для начала создайте папку `httpsserver/templates` и добавьте в нее файл с именем `products.html` с содержимым, показанным в листинге 24-18.

```
<!DOCTYPE html>
<html>
<head>
```

```
    <meta name="viewport" content="width=device-width" />
    <title>Pro Go</title>
    <link rel="stylesheet" href="/files/bootstrap.min.css" >
</head>
<body>
    <h3 class="bg-primary text-white text-center p-2 m-2">Products</h3>
    <div class="p-2">
        <table class="table table-sm table-striped table-bordered">
            <thead>
                <tr>
                    <th>Index</th><th>Name</th><th>Category</th>
                    <th class="text-end">Price</th>
                </tr>
            </thead>
            <tbody>
                {{ range $index, $product := .Data }}
                    <tr>
                        <td>{{ $index }}</td>
                        <td>{{ $product.Name }}</td>
                        <td>{{ $product.Category }}</td>
                        <td class="text-end">
                            {{ printf "$%.2f" $product.Price }}
                        </td>
                    </tr>
                {{ end }}
            </tbody>
        </table>
    </div>
</body>
</html>
```

Листинг 24-18 Содержимое файла products.html в папке templates

Затем добавьте файл с именем `dynamic.go` в папку `httpsserver` с содержимым, показанным в листинге 24-19.

```
package main

import (
    "html/template"
    "net/http"
    "strconv"
)

type Context struct {
    Request *http.Request
    Data []Product
}

var htmlTemplates *template.Template
```

```
func    HandleTemplateRequest(writer    http.ResponseWriter,    request
*http.Request) {
    path := request.URL.Path
    if (path == "") {
        path = "products.html"
    }
    t := htmlTemplates.Lookup(path)
    if (t == nil) {
        http.NotFound(writer, request)
    } else {
        err := t.Execute(writer, Context{  request, Products})
        if (err != nil) {
                                        http.Error(writer,   err.Error(),
http.StatusInternalServerError)
        }
    }
}

func init() {
    var err error
    htmlTemplates = template.New("all")
    htmlTemplates.Funcs(map[string]interface{} {
        "intVal": strconv.Atoi,
    })
    htmlTemplates, err = htmlTemplates.ParseGlob("templates/*.html")
    if (err == nil) {
        http.Handle("/templates/", http.StripPrefix("/templates/",
            http.HandlerFunc(HandleTemplateRequest)))
    } else {
        panic(err)
    }
}
```

Листинг 24-19 Содержимое файла dynamic.go в папке httpsserver

Функция инициализации загружает все шаблоны с расширением `html` в папку `templates` и устанавливает маршрут, чтобы запросы, начинающиеся с `/templates/`, обрабатывались функцией `HandleTemplateRequest`. Эта функция просматривает шаблон, возвращаясь к файлу `products.html`, если путь к файлу не указан, выполняет шаблон и записывает ответ. Скомпилируйте и выполните проект и используйте браузер для запроса `https://localhost:5500/templates`, который даст ответ, показанный на рисунке 24-6.

Products

Index	Name	Category	Price
0	Kayak	Watersports	$279.00
1	Lifejacket	Watersports	$49.95
2	Soccer Ball	Soccer	$19.50
3	Corner Flags	Soccer	$34.95
4	Stadium	Soccer	$79500.00
5	Thinking Cap	Chess	$16.00
6	Unsteady Chair	Chess	$75.00
7	Bling-Bling King	Chess	$1200.00

Рисунок 24-6 Использование шаблона HTML для генерации ответа

Примечание
Одним из ограничений показанного здесь подхода является то, что данные, передаваемые в шаблон, жестко связаны с функцией `HandleTemplateRequest`. Я демонстрирую более гибкий подход в третьей части.

ПОНИМАНИЕ ОБНАРУЖИВАНИЯ ТИПА КОНТЕНТА
Обратите внимание, что мне не нужно было устанавливать заголовок `Content-Type` при использовании шаблона для генерации ответа. При обслуживании файлов заголовок `Content-Type` устанавливается на основе расширения файла, но в данной ситуации это невозможно, поскольку я пишу контент непосредственно в `ResponseWriter`.

Если в ответе нет заголовка `Content-Type`, первые 512 байтов содержимого, записанного в `ResponseWriter`, передаются функции `DetectContentType`, которая реализует алгоритм анализа MIME, определенный `https://mimesniff.spec.whatwg.org`. Процесс анализа не может обнаружить каждый тип контента, но он хорошо справляется со стандартными веб-типами, такими как HTML, CSS и JavaScript. Функция `DetectContentType` возвращает тип MIME, который используется в качестве значения для заголовка `Content-Type`. В этом примере алгоритм прослушивания определяет, что содержимое представляет собой HTML, и устанавливает для заголовка значение `text/html`. Процесс анализа контента можно отключить, явно установив заголовок `Content-Type`.

Ответ с данными JSON

Ответы JSON широко используются в веб-сервисах, которые предоставляют доступ к данным приложения для клиентов, которые не хотят получать HTML, таких как клиенты Angular или React JavaScript. В части 3 я создаю более сложную веб-службу, но для этой главы достаточно понять, что те же функции, которые позволили мне обслуживать статический и динамический HTML-контент, можно использовать и для генерации ответов JSON. Добавьте файл с именем `json.go` в папку `httpsserver` с содержимым, показанным в листинге 24-20.

```
package main

import (
    "net/http"
    "encoding/json"
)

func HandleJsonRequest(writer http.ResponseWriter, request *http.Request)
{
    writer.Header().Set("Content-Type", "application/json")
    json.NewEncoder(writer).Encode(Products)
}

func init() {
    http.HandleFunc("/json", HandleJsonRequest)
}
```

Листинг 24-20 Содержимое файла json.go в папке httpsserver

Функция инициализации создает маршрут, а это значит, что запросы на `/json` будут обрабатываться функцией `HandleJsonRequest`. Эта функция использует функции JSON, описанные в главе 21, для кодирования среза значений `Product`, созданного в листинге 24-3. Обратите внимание, что я явно установил заголовок `Content-Type` в листинге 24-20:

```
...
writer.Header().Set("Content-Type", "application/json")
...
```

На функцию сниффинга, описанную ранее в этой главе, нельзя полагаться для идентификации содержимого JSON, и она приведет к ответам с типом содержимого `text/plain`. Многие клиенты веб-служб обрабатывают ответы как JSON независимо от заголовка `Content-Type`, но полагаться на такое поведение не рекомендуется. Скомпилируйте и запустите проект и используйте браузер для запроса `https://localhost:5500/json`. Браузер отобразит следующее содержимое JSON:

```
[{"Name":"Kayak","Category":"Watersports","Price":279},
{"Name":"Lifejacket","Category":"Watersports","Price":49.95},
{"Name":"Soccer Ball","Category":"Soccer","Price":19.5},
```

```
{"Name":"Corner Flags","Category":"Soccer","Price":34.95},
{"Name":"Stadium","Category":"Soccer","Price":79500},
{"Name":"Thinking Cap","Category":"Chess","Price":16},
{"Name":"Unsteady Chair","Category":"Chess","Price":75},
{"Name":"Bling-Bling King","Category":"Chess","Price":1200}]
```

Обработка данных формы

Пакет `net/http` обеспечивает поддержку простого получения и обработки данных форм. Добавьте файл с именем `edit.html` в папку `templates` с содержимым, показанным в листинге 24-21.

```
<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>Pro Go</title>
    <link rel="stylesheet" href="/files/bootstrap.min.css" >
</head>
<body>
    {{ $index := intVal (index (index .Request.URL.Query "index") 0) }}
    {{ if lt $index (len .Data)}}
        {{ with index .Data $index}}
                   <h3 class="bg-primary text-white text-center p-2 m-
2">Product</h3>
            <form method="POST" action="/forms/edit" class="m-2">
                <div class="form-group">
                    <label>Index</label>
                    <input name="index" value="{{$index}}"
                        class="form-control" disabled />
                     <input name="index" value="{{$index}}" type="hidden"
/>
                </div>
                <div class="form-group">
                    <label>Name</label>
                        <input name="name" value="{{.Name}}" class="form-
control"/>
                </div>
                <div class="form-group">
                    <label>Category</label>
                    <input name="category" value="{{.Category}}"
                        class="form-control"/>
                </div>
                <div class="form-group">
                    <label>Price</label>
                      <input name="price" value="{{.Price}}" class="form-
control"/>
                </div>
                <div class="mt-2">
```

```
                                  <button type="submit" class="btn btn-
primary">Save</button>
                                  <a href="/templates/" class="btn btn-
secondary">Cancel</a>
                </div>
            </form>
        {{ end }}
    {{ else }}
        <h3 class="bg-danger text-white text-center p-2">
            No Product At Specified Index
        </h3>
    {{end }}
</body>
</html>
```

Листинг 24-21 Содержимое файла edit.html в папке templates

Этот шаблон использует переменные шаблона, выражения и функции для получения строки запроса из запроса и выбора первого значения `index`, которое преобразуется в `int` и используется для извлечения значения `Product` из данных, предоставленных шаблону:

```
...
{{ $index := intVal (index (index .Request.URL.Query "index") 0) }}
{{ if lt $index (len .Data)}}
    {{ with index .Data $index}}
...
```

Эти выражения более сложны, чем мне обычно нравится видеть в шаблоне, и я покажу вам подход, который я нахожу более надежным в части 3. Однако для этой главы он позволяет мне сгенерировать HTML-форму, которая представляет `input` элементы для поля, определенные структурой `Product`, которая отправляет свои данные на URL-адрес, указанный атрибутом `action`, следующим образом:

```
...
<form method="POST" action="/forms/edit" class="m-2">
...
```

Чтение данных формы из запросов

Теперь, когда я добавил `form` в проект, я могу написать код, который получает содержащиеся в нем данные. Структура `Request` определяет поля и методы, описанные в таблице 24-11, для работы с данными формы.

Таблица 24-11 Поля данных и методы формы запроса

Функция	Описание
`Form`	Это поле возвращает строку `map[string][]string`, содержащую проанализированные данные формы и параметры строки запроса. Перед чтением этого поля необходимо вызвать метод `ParseForm`.

Функция	Описание
PostForm	Это поле похоже на Form, но исключает параметры строки запроса, поэтому в карте содержатся только данные из тела запроса. Перед чтением этого поля необходимо вызвать метод ParseForm.
MultipartForm	Это поле возвращает составную форму, представленную с помощью структуры Form, определенной в пакете mime/multipart. Перед чтением этого поля необходимо вызвать метод ParseMultipartForm.
FormValue(key)	Этот метод возвращает первое значение для указанного ключа формы и возвращает пустую строку, если значение отсутствует. Источником данных для этого метода является поле Form, а вызов метода FormValue автоматически вызывает ParseForm или ParseMultipartForm для анализа формы.
PostFormValue(key)	Этот метод возвращает первое значение для указанного ключа формы и возвращает пустую строку, если значение отсутствует. Источником данных для этого метода является поле PostForm, а вызов метода PostFormValue автоматически вызывает ParseForm или ParseMultipartForm для анализа формы.
FormFile(key)	Этот метод обеспечивает доступ к первому файлу с указанным в форме ключом. Результатами являются File и FileHeader, оба из которых определены в пакете mime/multipart, и error. Вызов этой функции приводит к вызову функций ParseForm или ParseMultipartForm для анализа формы.
ParseForm()	Этот метод анализирует форму и заполняет поля Form и PostForm. Результатом является error, которая описывает любые проблемы синтаксического анализа.
ParseMultipartForm(max)	Этот метод анализирует составную форму MIME и заполняет поле MultipartForm. Аргумент указывает максимальное количество байтов, выделяемых для данных формы, а результатом является error, описывающая любые проблемы с обработкой формы.

Методы FormValue и PostFormValue — наиболее удобный способ доступа к данным формы, если вы знаете структуру обрабатываемой формы. Добавьте файл с именем forms.go в папку httpsserver с содержимым, показанным в листинге 24-22.

```
package main

import (
    "net/http"
    "strconv"
)

func ProcessFormData(writer http.ResponseWriter, request *http.Request) {
    if (request.Method == http.MethodPost) {
        index, _ := strconv.Atoi(request.PostFormValue("index"))
        p := Product {}
        p.Name = request.PostFormValue("name")
        p.Category = request.PostFormValue("category")
          p.Price, _ = strconv.ParseFloat(request.PostFormValue("price"),
64)
        Products[index] = p
    }
                    http.Redirect(writer,     request,     "/templates",
http.StatusTemporaryRedirect)
}
```

```
func init() {
    http.HandleFunc("/forms/edit", ProcessFormData)
}
```

Листинг 24-22 Содержимое файла form.go в папке httpsserver

Функция `init` устанавливает новый маршрут, чтобы функция `ProcessFormData` обрабатывала запросы, путь к которым — `/forms/edit`. В функции `ProcessFormData` проверяется метод запроса, и данные формы в запросе используются для создания структуры `Product` и замены существующего значения данных. В реальном проекте проверка данных, представленных в форме, необходима, но в этой главе я уверен, что форма содержит достоверные данные.

Скомпилируйте и выполните проект и используйте браузер для запроса `https://localhost:5500/templates/edit.html?index=2`, который выбирает значение `Product` по индексу 2 в срезе, определенном в листинге 24-3. Измените значение поля `Category` на `Soccer/Football` и нажмите кнопку **Save**. Данные в форме будут применены, и браузер будет перенаправлен, как показано на рисунке 24-7.

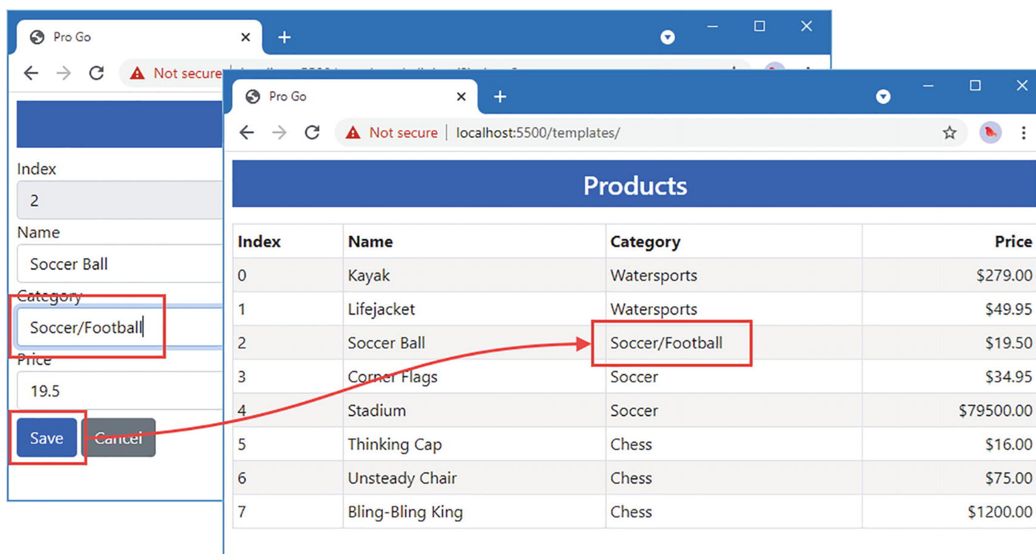

Рисунок 24-7 Обработка данных формы

Чтение составных форм

Формы, закодированные как `multipart/form-data`, чтобы обеспечить безопасную отправку двоичных данных, таких как файлы, на сервер. Чтобы создать форму, позволяющую серверу получать файл, создайте файл с именем `upload.html` в папке `static` с содержимым, показанным в листинге 24-23.

```
<!DOCTYPE html>
<html>
<head>
    <title>Pro Go</title>
    <meta name="viewport" content="width=device-width" />
```

```
    <link href="bootstrap.min.css" rel="stylesheet" />
</head>
<body>
    <div class="m-1 p-2 bg-primary text-white h2 text-center">
        Upload File
    </div>
    <form method="POST" action="/forms/upload" class="p-2"
            enctype="multipart/form-data">
        <div class="form-group">
            <label class="form-label">Name</label>
            <input class="form-control" type="text" name="name">
        </div>
        <div class="form-group">
            <label class="form-label">City</label>
            <input class="form-control" type="text" name="city">
        </div>
        <div class="form-group">
            <label class="form-label">Choose Files</label>
                <input class="form-control" type="file" name="files"
multiple>
        </div>
                <button type="submit" class="btn btn-primary mt-
2">Upload</button>
    </form>
</body>
</html>
```

Листинг 24-23 Содержимое файла upload.html в папке static

Атрибут `enctype` элемента `form` создает составную форму, а `input` элемент, тип которого — `file`, создает элемент управления формы, который позволяет пользователю выбрать файл. Атрибут `multiple` указывает браузеру разрешить пользователю выбирать несколько файлов, к которым я вскоре вернусь. Добавьте файл с именем `upload.go` в папку `httpsserver` с кодом из листинга 24-24 для получения и обработки данных формы.

```
package main

import (
    "net/http"
    "io"
    "fmt"
)

func HandleMultipartForm(writer http.ResponseWriter, request
*http.Request) {
                fmt.Fprintf(writer, "Name: %v, City: %v\n",
request.FormValue("name"),
        request.FormValue("city"))
    fmt.Fprintln(writer, "------")
    file, header, err := request.FormFile("files")
```

```
    if (err == nil) {
        defer file.Close()
            fmt.Fprintf(writer, "Name: %v, Size: %v\n", header.Filename,
header.Size)
        for k, v := range header.Header {
            fmt.Fprintf(writer, "Key: %v, Value: %v\n", k, v)
        }
        fmt.Fprintln(writer, "------")
        io.Copy(writer, file)
    } else {
        http.Error(writer, err.Error(), http.StatusInternalServerError)
    }
}

func init() {
    http.HandleFunc("/forms/upload", HandleMultipartForm)
}
```

Листинг 24-24 Содержимое файла upload.go в папке httpsserver

Методы FormValue и PostFormValue можно использовать для доступа к строковым значениям в форме, но доступ к файлу должен осуществляться с помощью метода FormFile, например:

```
...
file, header, err := request.FormFile("files")
...
```

Первым результатом метода FormFile является File, определенный в пакете mime/multipart, который представляет собой интерфейс, объединяющий интерфейсы Reader, Closer, Seeker и ReaderAt, описанные в главах 20 и 22. В результате содержимое загруженного файла можно обрабатывать как Reader с поддержкой поиска или чтения из определенного места. В этом примере я копирую содержимое загруженного файла в ResponseWriter.

Вторым результатом метода FormFile является FileHeader, также определенный в пакете mime/multipart. Эта структура определяет поля и метод, описанные в таблице 24-12.

Таблица 24-12 Поля и метод FileHeader

Функция	Описание
Name	Это поле возвращает string, содержащую имя файла.
Size	Это поле возвращает значение int64, содержащее размер файла.
Header	Это поле возвращает строку map[string][]string, которая содержит заголовки для части MIME, содержащей файл.
Open()	Этот метод возвращает File, который можно использовать для чтения содержимого, связанного с заголовком, как показано в следующем разделе.

Скомпилируйте и запустите проект и используйте браузер для запроса https://localhost:5500/files/upload.html. Введите свое имя и город, нажмите кнопку **Files** и выберите один файл (в следующем разделе я объясню, как работать с несколькими файлами). Вы можете выбрать любой файл в вашей системе, но текстовый файл — лучший выбор для простоты. Нажмите кнопку **Upload**, и форма будет опубликована. Ответ будет содержать значения имени и города, а также заголовок и содержимое файла, как показано на рисунке 24-8.

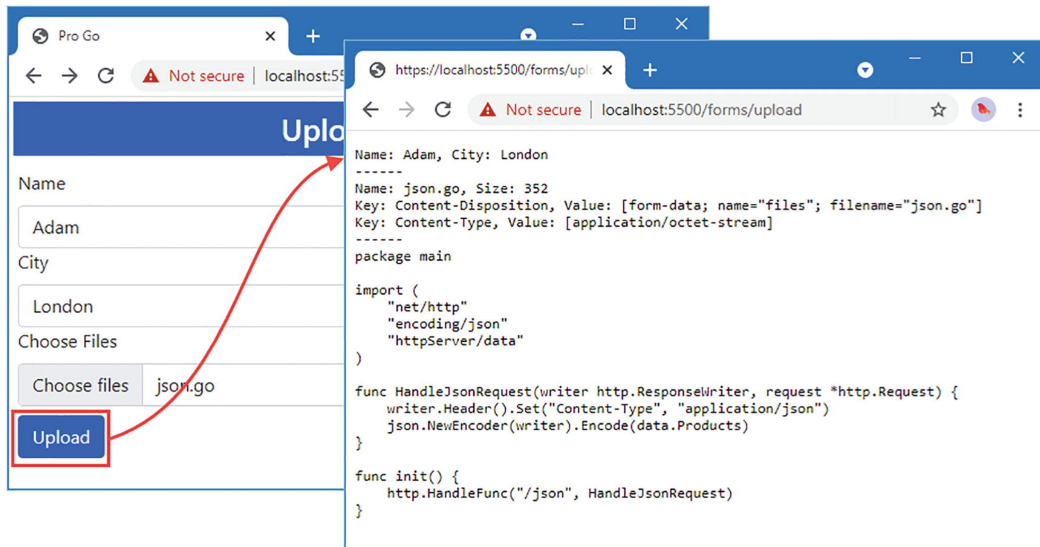

Рисунок 24-8 Обработка составной формы, содержащей файл

Получение нескольких файлов в форме

Метод FormFile возвращает только первый файл с указанным именем, а это означает, что его нельзя использовать, когда пользователю разрешено выбирать несколько файлов для одного элемента формы, как в случае с примером формы.

Поле Request.MultipartForm обеспечивает полный доступ к данным в составной форме, как показано в листинге 24-25.

```
package main

import (
    "net/http"
    "io"
    "fmt"
)

func HandleMultipartForm(writer http.ResponseWriter, request *http.Request) {
    request.ParseMultipartForm(10000000)
    fmt.Fprintf(writer, "Name: %v, City: %v\n",
        request.MultipartForm.Value["name"][0],
        request.MultipartForm.Value["city"][0])
    fmt.Fprintln(writer, "------")
```

```
    for _, header := range request.MultipartForm.File["files"] {
            fmt.Fprintf(writer, "Name: %v, Size: %v\n", header.Filename,
header.Size)
        file, err := header.Open()
        if (err == nil) {
            defer file.Close()
            fmt.Fprintln(writer, "------")
            io.Copy(writer, file)
        } else {
                                        http.Error(writer,   err.Error(),
http.StatusInternalServerError)
            return
        }
    }
}

func init() {
    http.HandleFunc("/forms/upload", HandleMultipartForm)
}
```

Листинг 24-25 Обработка нескольких файлов в файле upload.go в папке httpsserver

Вы должны убедиться, что метод `ParseMultipartForm` вызывается перед использованием поля `MultipartForm`. Поле `MultipartForm` возвращает структуру `Form`, которая определена в пакете `mime/multipart` и определяет поля, описанные в таблице 24-13.

Таблица 24-13 Поля формы

Функция	Описание
`Value`	Это поле возвращает строку `map[string][]string`, содержащую значения формы.
`File`	Это поле возвращает `map[string][]*FileHeader`, содержащий файлы.

В листинге 24-25 я использую поле `Value` для получения значений `Name` и `City` из формы. Я использую поле `File`, чтобы получить все файлы в форме с именами `files`, которые представлены значениями `FileHeader`, описанными в Таблице 24-13. Скомпилируйте и запустите проект, используйте браузер для запроса `https://localhost:5500/files/upload.html` и заполните форму. На этот раз при нажатии кнопки «Выбрать файлы» выберите два или более файлов. Отправьте форму, и вы увидите содержимое всех выбранных вами файлов, как показано на рисунке 24-9. Для этого примера предпочтительны текстовые файлы.

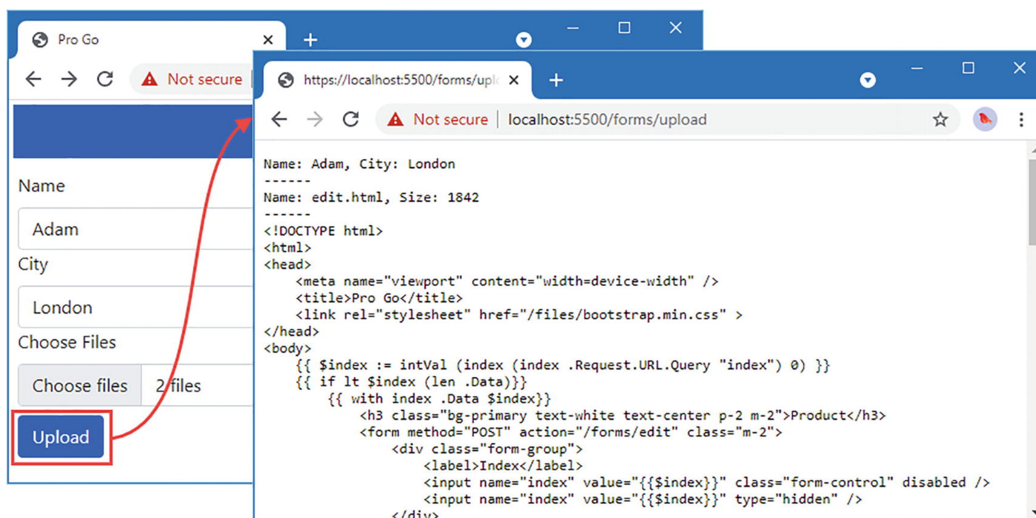

Рисунок 24-9 Обработка нескольких файлов

Чтение и настройка файлов cookie

Пакет `net/http` определяет функцию `SetCookie`, которая добавляет заголовок `Set-Cookie` в ответ, отправляемый клиенту. Для быстрого ознакомления таблица 24-14 описывает функцию `SetCookie`.

Таблица 24-14 Функция net/http для настройки файлов cookie

Функция	Описание
`SetCookie(writer, cookie)`	Эта функция добавляет заголовок `Set-Cookie` к указанному `ResponseWriter`. Файл cookie описывается с помощью указателя на структуру `Cookie`, которая описана далее.

Файлы cookie описываются с помощью структуры `Cookie`, которая определена в пакете `net/http` и определяет поля, описанные в таблице 24-15. Базовый файл cookie может быть создан только с полями `Name` и `Value`.

Таблица 24-15 Поля, определяемые структурой cookie

Функция	Описание
`Name`	Это поле представляет имя файла cookie, выраженное в виде строки.
`Value`	Это поле представляет значение файла cookie, выраженное в виде строки.
`Path`	В этом необязательном поле указывается путь к файлу cookie.
`Domain`	В этом необязательном поле указывается host/domain, для которого будет установлен файл cookie.
`Expires`	В этом поле указывается срок действия файла cookie, выраженный в виде значения `time.Time`.
`MaxAge`	В этом поле указывается количество секунд до истечения срока действия файла cookie, выраженное как `int`.
`Secure`	Когда это `bool` поле имеет значение `true`, клиент будет отправлять файл cookie только через соединения HTTPS.
`HttpOnly`	Когда это `bool` поле имеет значение `true`, клиент предотвратит доступ кода JavaScript к файлу cookie.

Функция	Описание
SameSite	В этом поле указывается политика перекрестного происхождения для файла cookie с использованием констант SameSite, которые определяют SameSiteDefaultMode, SameSiteLaxMode, SameSiteStrictMode и SameSiteNoneMode.

Структура Cookie также используется для получения набора файлов cookie, отправляемых клиентом, что делается с помощью методов Request, описанных в таблице 24-16.

Таблица 24-16 Методы запроса файлов cookie

Функция	Описание
Cookie(name)	Этот метод возвращает указатель на значение Cookie с указанным именем и error, указывающую на отсутствие соответствующего файла cookie.
Cookies()	Этот метод возвращает срез указателей Cookie.

Добавьте файл с именем cookies.go в папку httpsserver с кодом, показанным в листинге 24-26.

```
package main

import (
    "net/http"
    "fmt"
    "strconv"
)

func GetAndSetCookie(writer http.ResponseWriter, request *http.Request) {

    counterVal := 1
    counterCookie, err := request.Cookie("counter")
    if (err == nil) {
        counterVal, _ = strconv.Atoi(counterCookie.Value)
        counterVal++
    }
    http.SetCookie(writer, &http.Cookie{
        Name: "counter", Value: strconv.Itoa(counterVal),
    })

    if (len(request.Cookies()) > 0) {
        for _, c := range request.Cookies() {
            fmt.Fprintf(writer, "Cookie Name: %v, Value: %v", c.Name,
c.Value)
        }
    } else {
        fmt.Fprintln(writer, "Request contains no cookies")
    }
}

func init() {
```

```
    http.HandleFunc("/cookies", GetAndSetCookie)
}
```

Листинг 24-26 Содержимое файла cookies.go в папке httpsserver

В этом примере задается маршрут `/cookies`, для которого функция `GetAndSetCookie` устанавливает cookie с именем `counter` с начальным значением, равным нулю. Когда запрос содержит файл cookie, значение файла cookie считывается, преобразуется в `int` и увеличивается, чтобы его можно было использовать для установки нового значения файла cookie. Функция также перечисляет файлы cookie в запросе и записывает в ответ поля `Name` и `Value`.

Скомпилируйте и запустите проект и используйте браузер для запроса `https://localhost:5500/cookies`. У клиента не будет файла cookie для первоначальной отправки, но каждый раз, когда вы впоследствии повторяете запрос, значение файла cookie будет считываться и увеличиваться, как показано на рисунке 24-10.

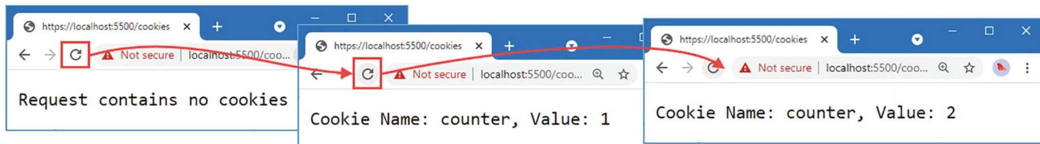

Рисунок 24-10 Чтение и настройка файлов cookie

Резюме

В этой главе я описал стандартные функции библиотеки для создания HTTP-серверов и обработки HTTP-запросов. В следующей главе я опишу дополнительные функции для создания и отправки HTTP-запросов.

25. Создание HTTP-клиентов

В этой главе я описываю стандартные функции библиотеки для создания HTTP-запросов, позволяющие приложениям использовать веб-серверы. Таблица 25-1 помещает HTTP-запросы в контекст

Таблица 25-1 Помещение HTTP-клиентов в контекст

Вопрос	Ответ
Кто они такие?	HTTP-запросы используются для получения данных с HTTP-серверов, таких как созданные в главе 24.
Почему они полезны?	HTTP является одним из наиболее широко используемых протоколов и обычно используется для предоставления доступа к содержимому, которое может быть представлено пользователю, а также к данным, которые используются программно.
Как это используется?	Возможности пакета `net/http` используются для создания и отправки запросов и обработки ответов.
Есть ли подводные камни или ограничения?	Эти функции хорошо продуманы и просты в использовании, хотя некоторые функции требуют определенной последовательности для использования.
Есть ли альтернативы?	Стандартная библиотека включает поддержку других сетевых протоколов, а также для открытия и использования сетевых соединений более низкого уровня. См. https://pkg.go.dev/net@go1.17.1 для получения подробной информации о пакете `net` и его подпакетах, таких как, например, `net/smtp`, который реализует протокол SMTP.

Таблица 25-2 суммирует содержание главы.

Таблица 25-2 Краткое содержание главы

Проблема	Решение	Листинг
Отправлять HTTP-запросы	Используйте удобные методы для определенных методов HTTP	8–12
Настройка HTTP-запросов	Используйте поля и методы, определенные структурой `Client`.	13
Создайте предварительно настроенный запрос	Используйте удобные функции `NewRequest`	14

Проблема	Решение	Листинг
Использовать куки в запросе	Используйте cookie jar	15–18
Настройка, как обрабатываются перенаправления	Используйте поле `CheckRedirect` для регистрации функции, которая вызывается для обработки перенаправления.	19–21
Отправка составных форм	Используйте пакет `mime/multipart`	22, 23

Подготовка к этой главе

Чтобы подготовиться к этой главе, откройте новую командную строку, перейдите в удобное место и создайте каталог с именем `httpclient`. Запустите команду, показанную в листинге 25-1, чтобы создать файл модуля.

> **Подсказка**
> Вы можете загрузить пример проекта для этой главы — и для всех остальных глав этой книги — с `https://github.com/apress/pro-go`. См. Главу 2 о том, как получить помощь, если у вас возникнут проблемы с запуском примеров.

```
go mod init httpclient
```

Листинг 25-1 Инициализация модуля

Добавьте файл с именем `printer.go` в папку `httpclient` с содержимым, показанным в листинге 25-2.

```
package main

import "fmt"

func Printfln(template string, values ...interface{}) {
    fmt.Printf(template + "\n", values...)
}
```

Листинг 25-2 Содержимое файла printer.go в папке httpclient

Добавьте файл с именем `product.go` в папку `httpclient` с содержимым, показанным в листинге 25-3.

```
package main

type Product struct {
    Name, Category string
    Price float64
}

var Products = []Product {
    { "Kayak", "Watersports", 279 },
     { "Lifejacket", "Watersports", 49.95 },
    { "Soccer Ball", "Soccer", 19.50 },
    { "Corner Flags", "Soccer", 34.95 },
    { "Stadium", "Soccer", 79500 },
    { "Thinking Cap", "Chess", 16 },
    { "Unsteady Chair", "Chess", 75 },
    { "Bling-Bling King", "Chess", 1200 },
}
```

Листинг 25-3 Содержимое файла product.go в папке httpclient

Добавьте файл с именем `index.html` в папку `httpclient` с содержимым, показанным в листинге 25-4.

```
<!DOCTYPE html>
<html>
<head>
    <title>Pro Go</title>
    <meta name="viewport" content="width=device-width" />
</head>
<body>
    <h1>Hello, World</div>
</body>
</html>
```

Листинг 25-4 Содержимое файла index.html в папке httpclient

Добавьте файл с именем `server.go` в папку `httpclient` с содержимым, показанным в листинге 25-5.

```
package main
```

```
import (
    "encoding/json"
    "fmt"
    "io"
    "net/http"
    "os"
)

func init() {

    http.HandleFunc("/html",
              func  (writer  http.ResponseWriter,  request
*http.Request) {
            http.ServeFile(writer, request, "./index.html")
        })
    http.HandleFunc("/json",
              func  (writer  http.ResponseWriter,  request
*http.Request) {
                      writer.Header().Set("Content-Type",
"application/json")
            json.NewEncoder(writer).Encode(Products)
        })
    http.HandleFunc("/echo",
              func  (writer  http.ResponseWriter,  request
*http.Request) {
            writer.Header().Set("Content-Type", "text/plain")
                    fmt.Fprintf(writer,  "Method:  %v\n",
request.Method)
            for header, vals := range request.Header {
                   fmt.Fprintf(writer, "Header: %v: %v\n",
header, vals)
            }
            fmt.Fprintln(writer, "----")
            data, err := io.ReadAll(request.Body)
            if (err == nil) {
                if len(data) == 0 {
                    fmt.Fprintln(writer, "No body")
                } else {
                    writer.Write(data)
                }
            } else {
                    fmt.Fprintf(os.Stdout,"Error reading body:
%v\n", err.Error())
```

```
            }
        })
}
```

Листинг 25-5 Содержимое файла server.go в папке httpclient

Функция инициализации в этом файле кода создает маршруты, которые генерируют ответы HTML и JSON. Существует также маршрут, который повторяет детали запроса в ответе.

Добавьте файл с именем `main.go` в папку `httpclient` с содержимым, показанным в листинге 25-6.

```
package main

import (
    "net/http"
)

func main() {
    Printfln("Starting HTTP Server")
    http.ListenAndServe(":5000", nil)
}
```

Листинг 25-6 Содержимое файла main.go в папке httpclient

Используйте командную строку для запуска команды, показанной в листинге 25-7, в папке `usingstrings`.

```
go run .
```

Листинг 25-7 Запуск примера проекта

Работа с запросами разрешений брандмауэра Windows

Как отмечалось в главе 24, брандмауэр Windows будет запрашивать доступ к сети каждый раз при компиляции кода. Чтобы решить эту проблему, создайте файл с именем `buildandrun.ps1` в папке проекта со следующим содержимым:

```
$file = "./httpclient.exe"

&go build -o $file

if ($LASTEXITCODE -eq 0) {
```

```
    &$file
}
```

Этот сценарий PowerShell каждый раз компилирует проект в один и тот же файл, а затем выполняет результат, если ошибок нет, то есть вам нужно будет предоставить доступ к брандмауэру только один раз. Скрипт выполняется запуском этой команды в папке проекта:

```
./buildandrun.ps1
```

Вы должны использовать эту команду каждый раз для сборки и выполнения проекта, чтобы убедиться, что скомпилированные выходные данные записываются в одно и то же место.

Код в папке `httpclient` будет скомпилирован и выполнен. Используйте веб-браузер, чтобы запросить `http://localhost:5000/html` и `http://localhost:5000/json`, которые выдают ответы, показанные на рисунок 25-1.

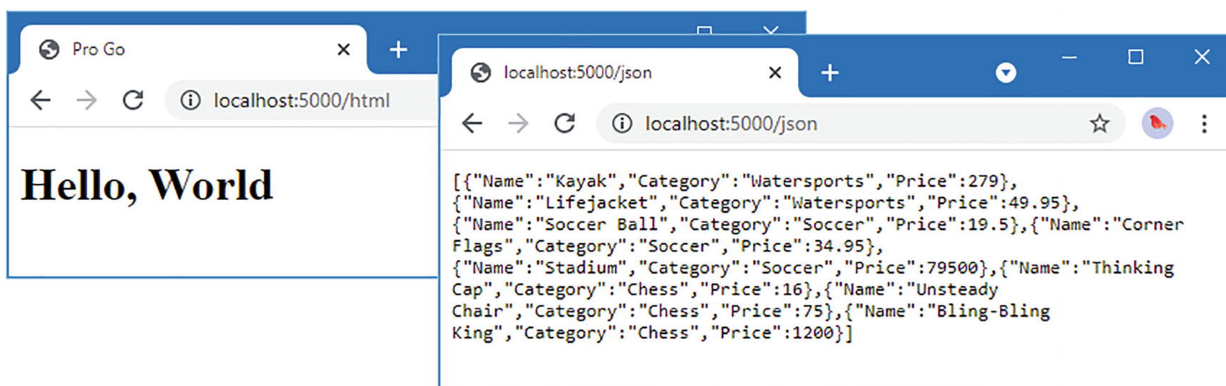

Рисунок 25-1 Запуск примера приложения

Чтобы увидеть результат эха, запросите `http://localhost:5000/echo`, что приведет к выводу, подобному рисунку 25-2, хотя вы можете увидеть разные детали в зависимости от вашей операционной системы и браузера.

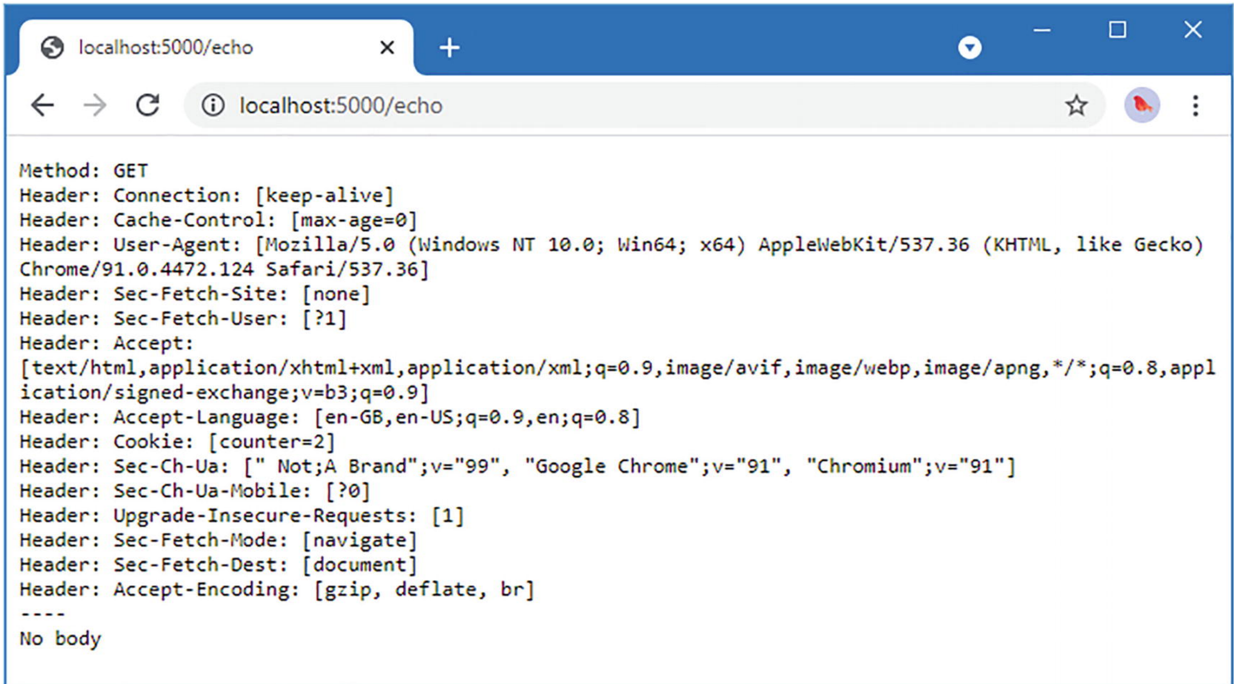

Рисунок 25-2 Повторение деталей запроса в ответе

Отправка простых HTTP-запросов

Пакет `net/http` предоставляет набор удобных функций, которые выполняют базовые HTTP-запросы. Функции названы в честь созданного ими HTTP-метода запроса, как описано в таблице 25-3.

Таблица 25-3 Удобные методы для HTTP-запросов

Функция	Описание
`Get(url)`	Эта функция отправляет запрос GET на указанный URL-адрес HTTP или HTTPS. Результатом являются ответ и `error`, сообщающая о проблемах с запросом.
`Head(url)`	Эта функция отправляет запрос HEAD на указанный URL-адрес HTTP или HTTPS. Запрос HEAD возвращает заголовки, которые были бы возвращены для запроса GET. Результатом являются `Response` и `error`, сообщающая о проблемах с запросом.
`Post(url, contentType, reader)`	Эта функция отправляет запрос POST на указанный URL-адрес HTTP или HTTPS с указанным значением заголовка `Content-Type`. Содержимое формы предоставляется указанным `Reader`. Результатом являются `Response` и `error`, сообщающая о проблемах с запросом.

Функция	Описание
PostForm(url, data)	Эта функция отправляет запрос POST на указанный URL-адрес HTTP или HTTPS с заголовком Content-Type, установленным в application/x-www-form-urlencoded. Содержимое формы предоставляется с помощью map[string][]string. Результатом являются Response и error, сообщающая о проблемах с запросом.

В листинге 25-8 метод Get используется для отправки запроса GET на сервер. Сервер запускается в горутине, чтобы предотвратить его блокировку и разрешить отправку HTTP-запроса в том же приложении. Это шаблон, который я буду использовать на протяжении всей этой главы, потому что он позволяет избежать необходимости разделять клиентские и серверные проекты. Я использую функцию time.Sleep, описанную в главе 19, чтобы убедиться, что горутина успевает запустить сервер. Возможно, вам потребуется увеличить задержку для вашей системы.

```
package main

import (
    "net/http"
    "os"
    "time"
)

func main() {
    go http.ListenAndServe(":5000", nil)
    time.Sleep(time.Second)

    response, err := http.Get("http://localhost:5000/html")
    if (err == nil) {
        response.Write(os.Stdout)
    } else {
        Printfln("Error: %v", err.Error())
    }
}
```

Листинг 25-8 Отправка запроса GET в файле main.go в папке httpclient

Аргумент функции Get — это строка, содержащая URL-адрес для запроса. Результатами являются значение Response и error, которая сообщает о любых проблемах с отправкой запроса.

Примечание

Значения error, возвращаемые функциями в таблице 25-3, сообщают о проблемах при создании и отправке запроса, но не используются, когда сервер возвращает код состояния ошибки HTTP.

Структура Response описывает ответ, отправленный сервером HTTP, и определяет поля и методы, показанные в таблице 25-4.

Таблица 25-4 Поля и методы, определяемые структурой Response

Функция	Описание
StatusCode	Это поле возвращает код состояния ответа, выраженный как int.
Status	Это поле возвращает string, содержащую описание статуса.
Proto	Это поле возвращает string, содержащую ответный HTTP-протокол.
Header	Это поле возвращает строку map[string][]string, содержащую заголовки ответа.
Body	Это поле возвращает ReadCloser, который является Reader, определяющим метод Close и обеспечивающим доступ к телу ответа.
Trailer	Это поле возвращает строку map[string][]string, содержащую трейлеры ответов.
ContentLength	Это поле возвращает значение заголовка Content-Length, преобразованное в значение int64.
TransferEncoding	Это поле возвращает набор значений заголовка Transfer-Encoding.
Close	Это логическое поле возвращает значение true, если ответ содержит заголовок Connection, для которого установлено значение close, что указывает на то, что HTTP-соединение должно быть закрыто.
Uncompressed	Это поле возвращает значение true, если сервер отправил сжатый ответ, который был распакован пакетом net/http.
Request	Это поле возвращает Request, который использовался для получения ответа. Структура Request описана в главе 24.
TLS	В этом поле содержится информация о соединении HTTPS.
Cookies()	Этот метод возвращает []*Cookie, который содержит заголовки Set-Cookie в ответе. Структура Cookie описана в главе 24.
Location	Этот метод возвращает URL-адрес из ответа заголовка Location и error, указывающую, что ответ не содержит этот заголовок.
Write(writer)	Этот метод записывает сводку ответа на указанный Writer.

Я использовал метод Write из листинга 25-8, который записывает сводку ответа. Скомпилируйте и выполните проект, и вы увидите следующий вывод, хотя и с другими значениями заголовка:

```
HTTP/1.1 200 OK
Content-Length: 182
Accept-Ranges: bytes
Content-Type: text/html; charset=utf-8
Date: Sat, 25 Sep 2021 08:23:21 GMT
Last-Modified: Sat, 25 Sep 2021 06:51:09 GMT
<!DOCTYPE html>
<html>
<head>
    <title>Pro Go</title>
    <meta name="viewport" content="width=device-width" />
</head>
<body>
    <h1>Hello, World</div>
</body>
</html>
```

Метод Write удобен, когда вы просто хотите увидеть ответ, но большинство проектов проверяют код состояния, чтобы убедиться, что запрос был успешным, а затем считывают тело ответа, как показано в листинге 25-9.

```
package main

import (
    "net/http"
    "os"
    "time"
    "io"
)

func main() {
    go http.ListenAndServe(":5000", nil)
    time.Sleep(time.Second)

    response, err := http.Get("http://localhost:5000/html")
    if (err == nil && response.StatusCode == http.StatusOK) {
        data, err := io.ReadAll(response.Body)
```

```
        if (err == nil) {
            defer response.Body.Close()
            os.Stdout.Write(data)
        }
    } else {
        Println("Error: %v, Status Code: %v", err.Error(),
response.StatusCode)
    }
}
```

Листинг 25-9 Чтение тела ответа в файле main.go в папке httpclient

Я использовал функцию ReadAll, определенную в пакете io, для чтения ответа Body в байтовый срез, который я записываю в стандартный вывод. Скомпилируйте и выполните проект, и вы увидите следующий вывод, показывающий тело ответа, отправленного HTTP-сервером:

```
<!DOCTYPE html>
<html>
<head>
    <title>Pro Go</title>
    <meta name="viewport" content="width=device-width" />
</head>
<body>
    <h1>Hello, World</div>
</body>
</html>
```

Когда ответы содержат данные, такие как JSON, их можно преобразовать в значения Go, как показано в листинге 25-10.

```
package main

import (
    "net/http"
    //"os"
    "time"
    //"io"
    "encoding/json"
)

func main() {
```

```
    go http.ListenAndServe(":5000", nil)
    time.Sleep(time.Second)

    response, err := http.Get("http://localhost:5000/json")
    if (err == nil && response.StatusCode == http.StatusOK) {
        defer response.Body.Close()
        data := []Product {}
        err = json.NewDecoder(response.Body).Decode(&data)
        if (err == nil) {
            for _, p := range data {
                Printfln("Name: %v, Price: $%.2f", p.Name,
p.Price)
            }
        } else {
            Printfln("Decode error: %v", err.Error())
        }
    } else {
         Printfln("Error: %v, Status Code: %v", err.Error(),
response.StatusCode)
    }
}
```

Листинг 25-10 Чтение и анализ данных в файле main.go в папке httpclient

Данные JSON декодируются с помощью пакета `encoding/json`, описанного в главе 21. Данные декодируются в срез `Product`, который перечисляется с помощью цикла `for`, в результате чего после компиляции и выполнения проекта выводятся следующие данные:

```
Name: Kayak, Price: $279.00
Name: Lifejacket, Price: $49.95
Name: Soccer Ball, Price: $19.50
Name: Corner Flags, Price: $34.95
Name: Stadium, Price: $79500.00
Name: Thinking Cap, Price: $16.00
Name: Unsteady Chair, Price: $75.00
Name: Bling-Bling King, Price: $1200.00
```

Отправка POST-запросов

Функции `Post` и `PostForm` используются для отправки запросов POST. Функция `PostForm` кодирует карту значений как данные формы, как показано в листинге 25-11.

```
package main

import (
    "net/http"
    "os"
    "time"
    "io"
    //"encoding/json"
)

func main() {
    go http.ListenAndServe(":5000", nil)
    time.Sleep(time.Second)

    formData := map[string][]string {
        "name":  { "Kayak "},
        "category": { "Watersports"},
        "price":  { "279"},
    }

                                response,       err        :=
http.PostForm("http://localhost:5000/echo", formData)

    if (err == nil && response.StatusCode == http.StatusOK) {
        io.Copy(os.Stdout, response.Body)
        defer response.Body.Close()
    } else {
         Printfln("Error: %v, Status Code: %v", err.Error(),
response.StatusCode)
    }
}
```

Листинг 25-11 Отправка формы в файле main.go в папке httpclient

HTML-формы поддерживают несколько значений для каждого ключа, поэтому значения на карте представляют собой срезы строк. В листинге 25-11 я отправляю только одно значение для каждого ключа в форме, но мне все равно нужно заключить это значение в фигурные скобки, чтобы создать срез. Функция `PostForm` кодирует карту, добавляет данные в тело запроса и устанавливает для заголовка `Content-Type` значение `application/x-www-form-urlencoded`. Форма отправляется на URL-адрес `/echo`, который просто отправляет обратно

запрос, полученный сервером в ответе. Скомпилируйте и запустите проект, и вы увидите следующий вывод:

```
Method: POST
Header: User-Agent: [Go-http-client/1.1]
Header: Content-Length: [42]
Header: Content-Type: [application/x-www-form-urlencoded]
Header: Accept-Encoding: [gzip]
----
category=Watersports&name=Kayak+&price=279
```

Публикация формы с помощью ридера

Функция `Post` отправляет запрос POST на сервер и создает тело запроса, считывая содержимое из `Reader`, как показано в листинге 25-12. В отличие от функции `PostForm`, данные не нужно кодировать как форму.

```
package main

import (
    "net/http"
    "os"
    "time"
    "io"
    "encoding/json"
    "strings"
)

func main() {
    go http.ListenAndServe(":5000", nil)
    time.Sleep(time.Second)

    var builder strings.Builder
    err := json.NewEncoder(&builder).Encode(Products[0])
    if (err == nil) {
                                            response,     err     :=
http.Post("http://localhost:5000/echo",
            "application/json",
            strings.NewReader(builder.String()))
                if  (err  ==  nil  &&  response.StatusCode  ==
http.StatusOK) {
            io.Copy(os.Stdout, response.Body)
```

```
            defer response.Body.Close()
        } else {
            Printfln("Error: %v", err.Error())
        }
    } else {
        Printfln("Error: %v", err.Error())
    }
}
```

Листинг 25-12 Публикация из Reader в файле main.go в папке httpclient

В этом примере первый элемент в срезе значений `Product`, определенных в листинге 25-12, кодируется как JSON, подготавливая данные, чтобы их можно было обрабатывать как `Reader`. Аргументами функции `Post` являются URL-адрес, на который отправляется запрос, значение заголовка `Content-Type` и `Reader`. Скомпилируйте и выполните проект, и вы увидите данные эхо-запроса:

```
Method: POST
Header: User-Agent: [Go-http-client/1.1]
Header: Content-Length: [54]
Header: Content-Type: [application/json]
Header: Accept-Encoding: [gzip]
----
{"Name":"Kayak","Category":"Watersports","Price":279}
```

Понимание заголовка Content-Length

Если вы изучите запросы, отправленные листингом 25-11 и листингом 25-12, вы увидите, что они включают заголовок `Content-Length`. Этот заголовок устанавливается автоматически, но включается в запросы только тогда, когда можно заранее определить, сколько данных будет включено в тело. Это делается путем проверки `Reader` для определения динамического типа. Когда данные хранятся в памяти с использованием типа `strings.Reader`, `bytes.Reader` или `bytes.Buffer`, встроенная функция `len` используется для определения объема данных, а результат используется для установки заголовка `Content-Length`.

Для всех остальных типов заголовок `Content-Type` не устанавливается, а вместо этого используется *фрагментированное кодирование*, что означает, что тело записывается блоками данных,

размер которых объявлен как часть тела запроса. Этот подход позволяет отправлять запросы без необходимости считывания всех данных из Reader, просто для определения количества байтов. Фрагментарное кодирование описано на странице https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Transfer-Encoding.

Настройка запросов HTTP-клиента

Структура Client используется, когда требуется управление HTTP-запросом, и определяет поля и методы, описанные в таблице 25-5.

Таблица 25-5 Клиентские поля и методы

Функция	Описание
Transport	Это поле используется для выбора транспорта, который будет использоваться для отправки HTTP-запроса. Пакет net/http предоставляет транспорт по умолчанию.
CheckRedirect	Это поле используется для указания пользовательской политики для обработки повторяющихся перенаправлений, как описано в разделе «Управление перенаправлениями».
Jar	Это поле возвращает файл CookieJar, который используется для управления файлами cookie, как описано в разделе «Работа с файлами cookie».
Timeout	Это поле используется для установки тайм-аута для запроса, указанного как time.Duration.
Do(request)	Этот метод отправляет указанный Request, возвращая Response и error, указывающую на проблемы с отправкой запроса.
CloseIdleConnections()	Этот метод закрывает все бездействующие HTTP-запросы, которые в настоящее время открыты и не используются.
Get(url)	Этот метод вызывается функцией Get, описанной в таблице 25-3.
Head(url)	Этот метод вызывается функцией Head, описанной в таблице 25-3.
Post(url, contentType, reader)	Этот метод вызывается функцией Post, описанной в таблице 25-3.
PostForm(url, data)	Этот метод вызывается функцией PostForm, описанной в таблице 25-3.

Пакет net/http определяет переменную DefaultClient, которая предоставляет Client по умолчанию, который можно использовать для

использования полей и методов, описанных в таблице 25-5, и именно эта переменная используется, когда используются функции, описанные в таблице 25-3.

Структура `Request`, описывающая HTTP-запрос, та же самая, которую я использовал в главе 24 для HTTP-серверов. В таблице 25-6 описаны поля и методы `Request`, наиболее полезные для клиентских запросов.

Таблица 25-6 Полезные поля и методы запроса

Функция	**Описание**
`Method`	Это строковое поле указывает метод HTTP, который будет использоваться для запроса. Пакет `net/http` определяет константы для методов HTTP, таких как `MethodGet` и `MethodPost`.
`URL`	В этом поле `URL` указывается URL-адрес, на который будет отправлен запрос. Структура URL определена в главе 24.
`Header`	Это поле используется для указания заголовков запроса. Заголовки указываются в `map[string][]string`, и поле будет `nil`, когда значение `Request` создается с использованием синтаксиса литеральной структуры.
`ContentLength`	Это поле используется для установки заголовка `Content-Length` с использованием значения `int64`.
`TransferEncoding`	Это поле используется для установки заголовка `Transfer-Encoding` с использованием среза строк.
`Body`	Это поле `ReadCloser` указывает источник тела запроса. Если у вас есть `Reader`, который не определяет метод `Close`, то можно использовать функцию `io.NopCloser` для создания `ReadCloser`, метод `Close` которого ничего не делает.

Самый простой способ создать значение URL — использовать функцию `Parse`, предоставляемую пакетом `net/url`, которая анализирует строку и описана в таблице 25-7 для быстрого ознакомления.

Таблица 25-7 Функция для анализа значений URL

Функция	**Описание**
`Parse(string)`	Этот метод анализирует `string` в URL. Результатами являются значение `URL` и `error`, указывающая на проблемы с разбором `string`.

В листинге 25-13 функции, описанные в таблицах, объединены для создания простого HTTP-запроса POST.

```
package main

import (
    "net/http"
    "os"
    "time"
    "io"
    "encoding/json"
    "strings"
    "net/url"
)

func main() {
    go http.ListenAndServe(":5000", nil)
    time.Sleep(time.Second)

    var builder strings.Builder
    err := json.NewEncoder(&builder).Encode(Products[0])
    if (err == nil) {
                                        reqURL,     err     :=
url.Parse("http://localhost:5000/echo")
        if (err == nil) {
            req := http.Request {
                Method: http.MethodPost,
                URL: reqURL,
                Header: map[string][]string {
                    "Content-Type": { "application.json" },
                },
                                                            Body:
io.NopCloser(strings.NewReader(builder.String())),
            }
            response, err := http.DefaultClient.Do(&req)
                  if  (err  ==  nil  &&  response.StatusCode  ==
http.StatusOK) {
                io.Copy(os.Stdout, response.Body)
                defer response.Body.Close()
            } else {
                Printfln("Request Error: %v", err.Error())
            }
        } else {
            Printfln("Parse Error: %v", err.Error())
        }
    } else {
```

```
        Println("Encoder Error: %v", err.Error())
    }
}
```

Листинг 25-13 Отправка запроса в файле main.go в папке httpclient

В этом листинге создается новый запрос с использованием буквального синтаксиса, а затем задаются поля `Method`, `URL` и `Body`. Метод настроен таким образом, что отправляется запрос POST, `URL` создается с помощью функции `Parse`, а поле `Body` устанавливается с помощью функции `io.NopCloser`, которая принимает `Reader` и возвращает `ReadCloser`, тип которого требуется для структуры `Request`. Полю `Header` назначается карта, определяющая заголовок `Content-Type`. Указатель на `Request` передается `Do` методу `Client`, назначенному переменной `DefaultClient`, которая отправляет запрос.

В этом примере используется URL-адрес `/echo`, установленный в начале главы, который повторяет запрос, полученный сервером, в ответе. Скомпилируйте и запустите проект, и вы увидите следующий вывод:

```
Method: POST
Header: Content-Type: [application.json]
Header: Accept-Encoding: [gzip]
Header: User-Agent: [Go-http-client/1.1]
----
{"Name":"Kayak","Category":"Watersports","Price":279}
```

Использование удобных функций для создания запроса

В предыдущем примере показано, что литеральный синтаксис структур можно использовать для создания значений `Request`, но пакет `net/http` также предоставляет удобные функции, упрощающие процесс, как описано в таблице 25-8.

Таблица 25-8 Удобные функции `net/http` для создания запросов

Функция	Описание
`NewRequest(method, url, reader)`	Эта функция создает новый `Reader`, настроенный с указанным методом, URL-адресом и телом. Функция также возвращает ошибку, указывающую на проблемы с созданием значения, включая синтаксический анализ URL-адреса, выраженного в виде строки.

Функция	Описание
NewRequestWithContext(context, method, url, reader)	Эта функция создает новый Reader, который будет отправлен в указанном контексте. Контексты описаны в главе 30.

В листинге 25-14 для создания запроса используется функция NewRequest вместо литерального синтаксиса создания Request.

```
package main

import (
    "net/http"
    "os"
    "time"
    "io"
    "encoding/json"
    "strings"
    //"net/url"
)

func main() {
    go http.ListenAndServe(":5000", nil)
    time.Sleep(time.Second)

    var builder strings.Builder
    err := json.NewEncoder(&builder).Encode(Products[0])
    if (err == nil) {
            req, err := http.NewRequest(http.MethodPost,
"http://localhost:5000/echo",
        io.NopCloser(strings.NewReader(builder.String())))
        if (err == nil) {
                req.Header["Content-Type"] = []string{
"application/json" }
            response, err := http.DefaultClient.Do(req)
                if (err == nil && response.StatusCode ==
http.StatusOK) {
                io.Copy(os.Stdout, response.Body)
                defer response.Body.Close()
            } else {
                Printfln("Request Error: %v", err.Error())
            }
        } else {
```

```
            Printfln("Request Init Error: %v", err.Error())
        }
    } else {
        Printfln("Encoder Error: %v", err.Error())
    }
}
```

Листинг 25-14 Использование функции удобства в файле main.go в папке httpclient

Результат тот же — `Request`, который можно передать методу `Client.Do`, но мне не нужно явно анализировать URL-адрес. Функция `NewRequest` инициализирует поле `Header`, поэтому я могу добавить заголовок `Content-Type` без предварительного создания карты. Скомпилируйте и выполните проект, и вы увидите детали запроса, отправленного на сервер:

```
Method: POST
Header: User-Agent: [Go-http-client/1.1]
Header: Content-Type: [application/json]
Header: Accept-Encoding: [gzip]
----
{"Name":"Kayak","Category":"Watersports","Price":279}
```

Работа с файлами cookie

`Client` отслеживает файлы cookie, которые он получает от сервера, и автоматически включает их в последующие запросы. Для подготовки добавьте файл с именем `server_cookie.go` в папку `httpclient` с содержимым, показанным в листинге 25-15.

```
package main

import (
    "net/http"
    "strconv"
    "fmt"
)

func init() {
    http.HandleFunc("/cookie",
     func (writer http.ResponseWriter, request *http.Request)
{
        counterVal := 1
```

```
        counterCookie, err := request.Cookie("counter")
        if (err == nil) {
             counterVal, _ = strconv.Atoi(counterCookie.Value)
            counterVal++
        }
        http.SetCookie(writer, &http.Cookie{
             Name: "counter", Value: strconv.Itoa(counterVal),
        })

        if (len(request.Cookies()) > 0) {
            for _, c := range request.Cookies() {
                 fmt.Fprintf(writer, "Cookie Name: %v, Value:
%v\n",
                   c.Name, c.Value)
            }
        } else {
                   fmt.Fprintln(writer, "Request contains no
cookies")
        }
    })

}
```

Листинг 25-15 Содержимое файла server_cookie.go в папке httpclient

Новый маршрут устанавливает и считывает файл cookie с именем counter, используя код из одного из примеров в главе 24. Листинг 25-16 обновляет клиентский запрос для использования нового URL-адреса.

```
package main

import (
    "net/http"
    "os"
    "time"
    "io"
    // "encoding/json"
    // "strings"
    //"net/url"
    "net/http/cookiejar"
)
```

```
func main() {
    go http.ListenAndServe(":5000", nil)
    time.Sleep(time.Second)

    jar, err := cookiejar.New(nil)
    if (err == nil) {
        http.DefaultClient.Jar = jar
    }

    for i := 0; i < 3; i++ {
        req, err := http.NewRequest(http.MethodGet,
            "http://localhost:5000/cookie", nil)
        if (err == nil) {
            response, err := http.DefaultClient.Do(req)
                  if (err == nil && response.StatusCode ==
http.StatusOK) {
                io.Copy(os.Stdout, response.Body)
                defer response.Body.Close()
            }  else {
                Printfln("Request Error: %v", err.Error())
            }
        } else {
            Printfln("Request Init Error: %v", err.Error())
        }
    }
}
```

Листинг 25-16 Изменение URL-адреса в файле main.go в папке httpclient

По умолчанию файлы cookie игнорируются значениями `Client`, что является разумной политикой, поскольку файлы cookie, установленные в одном ответе, повлияют на последующие запросы, что может привести к неожиданным результатам. Чтобы включить отслеживание файлов cookie, полю `Jar` назначается реализация интерфейса `net/http/CookieJar`, которая определяет методы, описанные в таблице 25-9.

Таблица 25-9 Методы, определяемые интерфейсом CookieJar

Функция	**Описание**
`SetCookies(url, cookies)`	Этот метод сохраняет срез `*Cookie` для указанного URL-адреса.

Функция	Описание
Cookes(url)	Этот метод возвращает срез *Cookie, содержащий файлы cookie, которые должны быть включены в запрос для указанного URL-адреса.

Пакет net/http/cookiejar содержит реализацию интерфейса CookieJar, который хранит файлы cookie в памяти. Куки-файлы создаются с помощью функции-конструктора, как описано в таблице 25-10.

Таблица 25-10 Функция конструктора Cookie Jar в пакете net/http/cookiejar

Функция	Описание
New(options)	Эта функция создает новый CookieJar, настроенный с помощью структуры Options, описанной далее. Функция также возвращает error, сообщающую о проблемах с созданием jar.

Функция New принимает структуру net/http/cookiejar/Options, которая используется для настройки cookie jar. Существует только одно поле Options, PublicSuffixList, которое используется для указания реализации интерфейса с тем же именем, который обеспечивает поддержку для предотвращения слишком широкой установки файлов cookie, что может привести к нарушению конфиденциальности. Стандартная библиотека не содержит реализации интерфейса PublicSuffixList, но она доступна по адресу https://pkg.go.dev/golang.org/x/net/publicsuffix.

В листинге 25-16 я вызвал функцию New с nil, что означает, что реализация PublicSuffixList не используется, а затем присвоил CookieJar полю Jar Client-а, назначенному переменной DefaultClient. Скомпилируйте и запустите проект, и вы увидите следующий вывод:

```
Request contains no cookies
Cookie Name: counter, Value: 1
Cookie Name: counter, Value: 2
```

Код из листинга 25-16 отправляет три HTTP-запроса. Первый запрос не содержит cookie, но сервер включает его в ответ. Этот файл cookie включается во второй и третий запросы, что позволяет серверу читать и увеличивать содержащееся в нем значение.

Обратите внимание, что мне не нужно было управлять файлами cookie в листинге 25-16. Настройка файла cookie — это все, что требуется, и `Client` автоматически отслеживает файлы cookie.

Создание отдельных клиентов и файлов cookie

Следствием использования `DefaultClient` является то, что все запросы используют одни и те же файлы cookie, что может быть полезно, тем более что файл cookie гарантирует, что каждый запрос включает только те файлы cookie, которые требуются для каждого URL-адреса.

Если вы не хотите делиться файлами cookie, вы можете создать `Client` с собственным файлом cookie, как показано в листинге 25-17.

```
package main

import (
    "net/http"
    "os"
    "time"
    "io"
    //"encoding/json"
    //"strings"
    //"net/url"
    "net/http/cookiejar"
    "fmt"
)

func main() {
    go http.ListenAndServe(":5000", nil)
    time.Sleep(time.Second)

    clients := make([]http.Client, 3)
    for index, client := range clients {
        jar, err := cookiejar.New(nil)
        if (err == nil) {
            client.Jar = jar
        }

        for i := 0; i < 3; i++ {
            req, err := http.NewRequest(http.MethodGet,
                "http://localhost:5000/cookie", nil)
            if (err == nil) {
```

```
                response, err := client.Do(req)
                    if (err == nil && response.StatusCode ==
http.StatusOK) {
                        fmt.Fprintf(os.Stdout, "Client %v: ",
index)
                        io.Copy(os.Stdout, response.Body)
                        defer response.Body.Close()
                    }  else {
                        Printfln("Request Error: %v",
err.Error())
                    }
                } else {
                    Printfln("Request Init Error: %v",
err.Error())
                }
            }
        }
}
```

Листинг 25-17 Создание отдельных клиентов в файле main.go в папке httpclient

В этом примере создаются три отдельных значения `Client`, каждое из которых имеет собственный `CookieJar`. Каждый `Client` делает три запроса, и код выдает следующий результат, когда проект компилируется и выполняется:

```
Client 0: Request contains no cookies
Client 0: Cookie Name: counter, Value: 1
Client 0: Cookie Name: counter, Value: 2
Client 1: Request contains no cookies
Client 1: Cookie Name: counter, Value: 1
Client 1: Cookie Name: counter, Value: 2
Client 2: Request contains no cookies
Client 2: Cookie Name: counter, Value: 1
Client 2: Cookie Name: counter, Value: 2
```

Если требуется несколько значений `Client`, но файлы cookie должны быть общими, можно использовать один файл `CookieJar`, как показано в листинге 25-18.

```
package main

import (
```

```
    "net/http"
    "os"
    "io"
    "time"
    //"encoding/json"
    //"strings"
    //"net/url"
    "net/http/cookiejar"
    "fmt"
)

func main() {
    go http.ListenAndServe(":5000", nil)
    time.Sleep(time.Second)

    jar, err := cookiejar.New(nil)

    clients := make([]http.Client, 3)
    for index, client := range clients {
        //jar, err := cookiejar.New(nil)
        if (err == nil) {
            client.Jar = jar
        }

        for i := 0; i < 3; i++ {
            req, err := http.NewRequest(http.MethodGet,
                "http://localhost:5000/cookie", nil)
            if (err == nil) {
                response, err := client.Do(req)
                    if (err == nil && response.StatusCode ==
http.StatusOK) {
                        fmt.Fprintf(os.Stdout, "Client %v: ",
index)
                    io.Copy(os.Stdout, response.Body)
                    defer response.Body.Close()
                }  else {
                                Printfln("Request  Error:  %v",
err.Error())
                }
            } else {
                          Printfln("Request  Init  Error:  %v",
err.Error())
            }
```

```
        }
    }
}
```

Листинг 25-18 Совместное использование CookieJar в файле main.go в папке httpclient

Файлы cookie, полученные одним `Client`, используются в последующих запросах, как показано в выводе, полученном при компиляции и выполнении проекта:

```
Client 0: Request contains no cookies
Client 0: Cookie Name: counter, Value: 1
Client 0: Cookie Name: counter, Value: 2
Client 1: Cookie Name: counter, Value: 3
Client 1: Cookie Name: counter, Value: 4
Client 1: Cookie Name: counter, Value: 5
Client 2: Cookie Name: counter, Value: 6
Client 2: Cookie Name: counter, Value: 7
Client 2: Cookie Name: counter, Value: 8
```

Управление перенаправлениями

По умолчанию `Client` перестанет выполнять перенаправления после десяти запросов, но это можно изменить, указав пользовательскую политику. Добавьте файл с именем `server_redirects.go` в папку `httpclient` с содержимым, показанным в листинге 25-19.

```
package main

import "net/http"

func init() {

    http.HandleFunc("/redirect1",
                 func  (writer  http.ResponseWriter,  request
*http.Request) {
            http.Redirect(writer, request, "/redirect2",
                http.StatusTemporaryRedirect)
        })
    http.HandleFunc("/redirect2",
                 func  (writer  http.ResponseWriter,  request
*http.Request) {
            http.Redirect(writer, request, "/redirect1",
```

```
                http.StatusTemporaryRedirect)
        })
}
```

Листинг 25-19 Содержимое файла server_redirects.go в папке httpclient

Перенаправления будут продолжаться до тех пор, пока клиент не перестанет им следовать. В листинге 25-20 создается запрос, который отправляется на URL-адрес, обрабатываемый первым маршрутом, определенным в листинге 25-19.

```
package main

import (
    "net/http"
    "os"
    "io"
    "time"
    //"encoding/json"
    //"strings"
    //"net/url"
    //"net/http/cookiejar"
    //"fmt"
)

func main() {
    go http.ListenAndServe(":5000", nil)
    time.Sleep(time.Second)

    req, err := http.NewRequest(http.MethodGet,
        "http://localhost:5000/redirect1", nil)
    if (err == nil) {
        var response *http.Response
        response, err = http.DefaultClient.Do(req)
        if (err == nil) {
            io.Copy(os.Stdout, response.Body)
        } else {
            Printfln("Request Error: %v", err.Error())
        }
    } else {
        Printfln("Error: %v", err.Error())
    }
}
```

Листинг 25-20 Отправка запроса в файле main.go в папке httpclient

Скомпилируйте и запустите проект, и вы увидите ошибку, которая останавливает `Client` после перенаправления после десяти запросов:

```
Request Error: Get "/redirect1": stopped after 10 redirects
```

Пользовательская политика определяется назначением функции полю `Client.CheckRedirect`, как показано в листинге 25-21.

```
package main

import (
    "net/http"
    "os"
    "io"
    "time"
    //"encoding/json"
    //"strings"
    "net/url"
    //"net/http/cookiejar"
    //"fmt"
)

func main() {
    go http.ListenAndServe(":5000", nil)
    time.Sleep(time.Second)

              http.DefaultClient.CheckRedirect    =    func(req
*http.Request,
        previous []*http.Request) error {
        if len(previous) == 3 {
            url, _ := url.Parse("http://localhost:5000/html")
            req.URL =   url
        }
        return nil
    }

    req, err := http.NewRequest(http.MethodGet,
        "http://localhost:5000/redirect1", nil)
    if (err == nil) {
        var response *http.Response
        response, err = http.DefaultClient.Do(req)
```

```
        if (err == nil) {
            io.Copy(os.Stdout, response.Body)
        } else {
            Println("Request Error: %v", err.Error())
        }
    } else {
        Println("Error: %v", err.Error())
    }
}
```

Листинг 25-21 Определение пользовательской политики перенаправления в файле main.go в папке httpclient

Аргументами функции являются указатель на `Request`, который должен быть выполнен, и срез `*Request`, содержащий запросы, которые привели к перенаправлению. Это означает, что срез будет содержать по крайней мере одно значение, потому что `CheckRedirect` вызывается только тогда, когда сервер возвращает ответ о перенаправлении.

Функция `CheckRedirect` может блокировать запрос, возвращая `error`, которая затем возвращается как результат метода `Do`. Или функция `CheckRedirect` может изменить запрос, который должен быть сделан, что и происходит в листинге 25-21. Когда запрос привел к трем перенаправлениям, настраиваемая политика изменяет поле `URL` таким образом, что `Request` относится к URL-адресу `/html`, настроенному ранее в этой главе, и это дает результат HTML.

В результате запрос URL-адреса `/redirect1` приведет к короткому циклу перенаправления между `/redirect2` и `/redirect1`, прежде чем политика изменит URL-адрес, выдав следующий результат:

```
<!DOCTYPE html>
<html>
<head>
    <title>Pro Go</title>
    <meta name="viewport" content="width=device-width" />
</head>
<body>
    <h1>Hello, World</div>
</body>
</html>
```

Создание составных форм

Пакет `mime/multipart` можно использовать для создания тела запроса, закодированного как `multipart/form-data`, что позволяет форме безопасно содержать двоичные данные, например содержимое файла. Добавьте файл с именем `server_forms.go` в папку `httpclient` с содержимым, показанным в листинге 25-22.

```
package main

import (
    "net/http"
    "fmt"
    "io"
)

func init() {

    http.HandleFunc("/form",
                  func  (writer  http.ResponseWriter,  request
*http.Request) {
            err := request.ParseMultipartForm(10000000)
            if (err == nil) {
                                    for  name,  vals  :=  range
request.MultipartForm.Value {
                          fmt.Fprintf(writer, "Field %v: %v\n",
name, vals)
                }
                                   for  name,  files  :=  range
request.MultipartForm.File {
                    for _, file := range files {
                          fmt.Fprintf(writer, "File %v: %v\n",
name, file.Filename)
                          if f, err := file.Open(); err == nil
{
                             defer f.Close()
                             io.Copy(writer, f)
                        }
                    }
                }
            } else {
```

```
                        fmt.Fprintf(writer, "Cannot parse form %v",
err.Error())
                }
        })
}
```

Листинг 25-22 Содержимое файла server_forms.go в папке httpclient

Новая функция-обработчик использует возможности, описанные в главе 24, для разбора составной формы и отображения содержащихся в ней полей и файлов.

Для создания формы на стороне клиента используется структура `multipart.Writer`, которая представляет собой оболочку над `io.Writer` и создается с помощью функции-конструктора, описанной в таблице 25-11.

Таблица 25-11 Функция конструктора multipart.Writer

Функция	Описание
`NewWriter(writer)`	Эта функция создает новый `multipart.Writer`, который записывает данные формы в указанный `io.Writer`.

Если у вас есть `multipart.Writer` для работы, содержимое формы можно создать с помощью методов, описанных в таблице 25-12.

Таблица 25-12 Методы multipart.Writer

Функция	Описание
`CreateFormField(fieldname)`	Этот метод создает новое поле формы с указанным именем. Результатом является `io.Writer`, который используется для записи данных поля, и `error`, сообщающая о проблемах с созданием поля.
`CreateFormFile(fieldname, filename)`	Этот метод создает новое поле файла с указанным именем поля и именем файла. Результатом является `io.Writer`, который используется для записи данных поля, и `error`, сообщающая о проблемах с созданием поля.
`FormDataContentType()`	Этот метод возвращает строку, которая используется для установки заголовка запроса `Content-Type` и включает строку, обозначающую границы между частями формы.
`Close()`	Эта функция завершает форму и записывает конечную границу, обозначающую конец данных формы.

Существуют дополнительные определения методов, которые обеспечивают детальное управление тем, как строится форма, но методы в таблице 25-12 являются наиболее полезными для большинства проектов. В листинге 25-23 эти методы используются для создания составной формы, отправляемой на сервер.

```
package main

import (
    "net/http"
    "os"
    "io"
    "time"
    //"encoding/json"
    //"strings"
    //"net/url"
    //"net/http/cookiejar"
    //"fmt"
    "mime/multipart"
    "bytes"
)

func main() {
    go http.ListenAndServe(":5000", nil)
    time.Sleep(time.Second)

    var buffer bytes.Buffer
    formWriter := multipart.NewWriter(&buffer)
    fieldWriter, err := formWriter.CreateFormField("name")
    if (err == nil) {
        io.WriteString(fieldWriter, "Alice")
    }
    fieldWriter, err = formWriter.CreateFormField("city")
    if (err == nil) {
        io.WriteString(fieldWriter, "New York")
    }
     fileWriter, err := formWriter.CreateFormFile("codeFile",
"printer.go")
    if (err == nil) {
        fileData, err := os.ReadFile("./printer.go")
        if (err == nil) {
            fileWriter.Write(fileData)
```

```
        }
    }

    formWriter.Close()

    req, err := http.NewRequest(http.MethodPost,
        "http://localhost:5000/form", &buffer)

                  req.Header["Content-Type"]    =    []string{
formWriter.FormDataContentType()}

    if (err == nil) {
        var response *http.Response
        response, err = http.DefaultClient.Do(req)
        if (err == nil) {
            io.Copy(os.Stdout, response.Body)
        } else {
            Printfln("Request Error: %v", err.Error())
        }
    } else {
        Printfln("Error: %v", err.Error())
    }
}
```

Листинг 25-23 Создание и отправка составной формы в файле main.go в папке httpclient

Для создания формы требуется определенная последовательность. Сначала вызовите функцию `NewWriter`, чтобы получить `multipart.Writer`:

```
...
var buffer bytes.Buffer
formWriter := multipart.NewWriter(&buffer)
...
```

Для использования данных формы в качестве тела HTTP-запроса требуется `Reader`, а для создания формы требуется `Writer`. Это идеальная ситуация для структуры `bytes.Buffer`, которая обеспечивает реализацию в памяти интерфейсов `Reader` и `Writer`.

После создания `multipart.Writer` методы `CreateFormField` и `CreateFormFile` используются для добавления полей и файлов в форму:

```
...
fieldWriter, err := formWriter.CreateFormField("name")
...
fileWriter, err := formWriter.CreateFormFile("codeFile",
"printer.go")
...
```

Оба этих метода возвращают Writer, который используется для записи содержимого в форму. После добавления полей и файлов следующим шагом будет установка заголовка Content-Type, используя результат метода FormDataContentType:

```
...
req.Header["Content-Type"] = []string{
formWriter.FormDataContentType()}
...
```

Результат метода включает строку, используемую для обозначения границ между частями формы. Последний шаг, о котором легко забыть, — это вызов метода Close для объекта multipart.Writer, который добавляет в форму окончательную граничную строку.

Осторожно
Не используйте ключевое слово defer при вызове метода Close; в противном случае окончательная граничная строка не будет добавлена в форму до тех пор, пока не будет отправлен запрос, что приведет к созданию формы, которую обработают не все серверы. Перед отправкой запроса важно вызвать метод Close.

Скомпилируйте и выполните проект, и вы увидите содержимое составной формы, отраженное в ответе от сервера:

```
Field city: [New York]
Field name: [Alice]
File codeFile: printer.go
package main
import "fmt"
func Printfln(template string, values ...interface{}) {
    fmt.Printf(template + "\n", values...)
}
```

Резюме

В этой главе я описываю функции стандартной библиотеки для отправки HTTP-запросов, объясняя, как использовать различные HTTP-глаголы, как отправлять формы и как решать такие проблемы, как файлы cookie. В следующей главе я покажу вам, как стандартная библиотека Go обеспечивает поддержку работы с базами данных.

26. Работа с базами данных

В этой главе я описываю стандартную библиотеку Go для работы с базами данных SQL. Эти функции обеспечивают абстрактное представление возможностей, предлагаемых базой данных, и полагаются на пакеты драйверов для реализации конкретной базы данных..

Существуют драйверы для широкого спектра баз данных, и их список можно найти по адресу `https://github.com/golang/go/wiki/sqldrivers`. Драйверы баз данных распространяются в виде пакетов Go, и большинство баз данных имеют несколько пакетов драйверов. Некоторые пакеты драйверов основаны на `cgo`, что позволяет коду Go использовать библиотеки C, а другие написаны на чистом Go.

В этой главе я использую базу данных SQLite (а также в части 3, где использую более сложную базу данных). SQLite — отличная встроенная база данных, которая поддерживает широкий спектр платформ, находится в свободном доступе и не требует установки и настройки серверного компонента. В таблиуе 26-1 представлены функции базы данных стандартной библиотеки в контексте.

Таблица 26-1 Работа с базами данных в контексте

Вопрос	Ответ
Что это?	Пакет `database/sql` предоставляет возможности для работы с базами данных SQL.
Почему это полезно?	Реляционные базы данных остаются наиболее эффективным способом хранения больших объемов структурированных данных и используются в большинстве крупных проектов.
Как это используется?	Пакеты драйверов обеспечивают поддержку определенных баз данных, а пакет `database/sql` предоставляет набор типов, позволяющих последовательно использовать базы данных.
Есть ли подводные камни или ограничения?	Эти функции не заполняют автоматически поля структуры из строк результатов.
Есть ли альтернативы?	Существуют сторонние пакеты, основанные на этих функциях, чтобы упростить или улучшить их использование.

Жалобы на базы данных

У вас может возникнуть соблазн связаться со мной, чтобы пожаловаться на выбор базы данных в этой книге. Вы, конечно, не будете одиноки, потому что выбор базы данных — одна из тем, о которых я получаю больше всего писем. Жалобы обычно предполагают, что я выбрал «неправильную» базу данных, что обычно означает «не ту базу данных, которую использует отправитель электронной почты».

Прежде чем связаться со мной, пожалуйста, учтите два момента. Во-первых, это не книга о базах данных, и подход SQLite с нулевой конфигурацией означает, что большинство читателей смогут следовать примерам, не диагностируя проблемы с установкой и конфигурацией. Во-вторых, SQLite — это выдающаяся база данных, которую многие проекты упускают из виду, поскольку она не имеет традиционного серверного компонента, хотя многие проекты не нуждаются в отдельном сервере базы данных или не выигрывают от него.

Приношу свои извинения, если вы являетесь преданным пользователем Oracle/DB2/MySQL/MariaDB и хотите иметь возможность вырезать и вставлять код подключения в свой проект. Но такой подход позволяет мне сосредоточиться на Go, а образцы кода, необходимые для выбранной вами базы данных, вы найдете в документации к выбранному вами драйверу.

Таблица 26-2 суммирует главу.

Таблица 26-2 Краткое содержание главы

Проблема	**Решение**	**Листинг**
Добавить поддержку в проект для определенного типа базы данных	Используйте команду `go get`, чтобы добавить пакет драйвера базы данных	8
Открытие и закрытие базы данных	Используйте функцию `Open` и метод `Close`	9, 10
Запросить базу данных	Используйте метод `Query` и обработайте результат `Rows` с помощью метода `Scan`	11–16, 22, 23
Запрос к базе данных для одной строки	Используйте метод `QueryRow` и обработайте результат `Row`	17
Выполнять запросы или операторы, которые не дают результатов строки	Используйте метод `Exec` и обработайте полученный `Result`	18
Обработать оператор, чтобы его можно было использовать повторно	Создайте опреатор подготовки запроса	19, 20
Выполнение нескольких запросов как единой единицы работы	Использовать транзакцию	21

Подготовка к этой главе

Чтобы подготовиться к этой главе, откройте новую командную строку, перейдите в удобное место и создайте каталог с именем `data`. Запустите команду, показанную в листинге 26-1, в папке `data`, чтобы создать файл модуля.

Подсказка
Вы можете загрузить пример проекта для этой главы — и для всех остальных глав этой книги — с `https://github.com/apress/pro-go`. См. Главу 2 о том, как получить помощь, если у вас возникнут проблемы с запуском примеров.

```
go mod init data
```

Листинг 26-1 Инициализация модуля

Добавьте файл с именем `printer.go` в папку `data` с содержимым, показанным в листинге 26-2.

```
package main

import "fmt"

func Printfln(template string, values ...interface{}) {
    fmt.Printf(template + "\n", values...)
}
```

Листинг 26-2 ТСодержимое файла printer.go в папке data

Добавьте файл с именем `main.go` в папку `data` с содержимым, показанным в листинге 26-3.

```
package main

func main() {
    Printfln("Hello, Data")
}
```

Листинг 26-3 Содержимое файла main.go в папке data

Запустите команду, показанную в листинге 26-4, в папке `data`, чтобы скомпилировать и запустить проект.

```
go run .
```

Листинг 26-4 Компиляция и выполнение проекта

Эта команда выдаст следующий вывод:

```
Hello, Data
```

Подготовка базы данных

Менеджер баз данных SQLite, используемый в этой главе, будет установлен позже, но для начала потребуется пакет инструментов для создания базы данных из файла SQL. (В третьей части я демонстрирую процесс создания базы данных из приложения.)

Чтобы определить файл SQL, который будет создавать базу данных, добавьте файл с именем `products.sql` в папку `data` с содержимым, показанным в листинге 26-5.

```
DROP TABLE IF EXISTS Categories;
```

```
DROP TABLE IF EXISTS Products;

CREATE TABLE IF NOT EXISTS Categories (
    Id INTEGER NOT NULL PRIMARY KEY,
    Name TEXT
);

CREATE TABLE IF NOT EXISTS Products (
    Id INTEGER NOT NULL PRIMARY KEY,
    Name TEXT,
    Category INTEGER,
    Price decimal(8, 2),
    CONSTRAINT CatRef FOREIGN KEY(Category) REFERENCES Categories (Id)
);

INSERT INTO Categories (Id, Name) VALUES
    (1, "Watersports"),
    (2, "Soccer");

INSERT INTO Products (Id, Name, Category, Price) VALUES
    (1, "Kayak", 1, 279),
    (2, "Lifejacket", 1, 48.95),
    (3, "Soccer Ball", 2, 19.50),
    (4, "Corner Flags", 2, 34.95);
```

Листинг 26-5 Содержимое файла products.sql в папке data

Перейдите на страницу `https://www.sqlite.org/download.html`, найдите раздел предварительно скомпилированных двоичных файлов для вашей операционной системы и загрузите пакет инструментов. Я не могу включать ссылки в эту главу, потому что URL-адреса содержат номер версии пакета, который изменится к тому времени, когда вы будете читать эту главу.

Распакуйте zip-архив и скопируйте файл `sqlite3` или `sqlite3.exe` в папку с данными. Запустите команду, показанную в листинге 26-6, в папке `data`, чтобы создать базу данных.

> **Примечание**
> Предварительно скомпилированные двоичные файлы для Linux являются 32-разрядными, что может потребовать установки некоторых дополнительных пакетов только для 64-разрядных операционных систем.

```
./sqlite3 products.db ".read products.sql"
```

Листинг 26-6 Создание базы данных

Чтобы убедиться, что база данных создана и заполнена данными, выполните команду, показанную в листинге 26-7, в папке `data`.

```
./sqlite3 products.db "select * from PRODUCTS"
```

Листинг 26-7 Тестирование базы данных

Если база данных создана правильно, вы увидите следующий вывод:

```
1|Kayak|1|279
2|Lifejacket|1|48.95
3|Soccer Ball|2|19.5
4|Corner Flags|2|34.95
```

Если вам нужно сбросить базу данных, следуя примерам из этой главы, вы можете удалить файл

```
products.db
```

и снова запустить команду из листинга 26-6.

Установка драйвера базы данных

Стандартная библиотека Go включает функции для простой и последовательной работы с базами данных, но полагается на пакеты драйверов баз данных для реализации этих функций для каждого конкретного ядра базы данных или сервера. Как уже отмечалось, в этой главе я использую SQLite, для которого имеется хороший чистый драйвер Go. Запустите команду, показанную в листинге 26-8, в папке `data`, чтобы установить пакет драйверов.

```
go get modernc.org/sqlite
```

Листинг 26-8 Установка пакета драйвера SQL

Большинство серверов баз данных настраиваются отдельно, поэтому драйвер базы данных открывает соединение с отдельным процессом. SQLite — это встроенная база данных, включенная в пакет драйверов, что означает, что дополнительная настройка не требуется.

Открытие базы данных

Стандартная библиотека предоставляет пакет `database/sql` для работы с базами данных. Функции, описанные в таблице 26-3, используются для открытия базы данных, чтобы ее можно было использовать в приложении.

Таблица 26-3 Функции database/sql для открытия базы данных

Функция	Описание
`Drivers()`	Эта функция возвращает срез строк, каждая из которых содержит имя драйвера базы данных.

Функция	Описание
`Open(driver, connectionStr)`	Эта функция открывает базу данных, используя указанный драйвер и строку подключения. Результатом является указатель на структуру DB, которая используется для взаимодействия с базой данных, и `error`, указывающая на проблемы с открытием базы данных.

Добавьте файл с именем `database.go` в папку `data` с кодом, показанным в листинге 26-9.

```
package main

import (
    "database/sql"
    _ "modernc.org/sqlite"
)

func listDrivers() {
    for _, driver := range sql.Drivers() {
        Printfln("Driver: %v", driver)
    }
}

func openDatabase() (db *sql.DB, err error) {
    db, err = sql.Open("sqlite", "products.db")
    if (err == nil) {
        Printfln("Opened database")
    }
    return
}
```

Листинг 26-9 Содержимое файла database.go в папке data

Пустой идентификатор используется для импорта пакета драйвера базы данных, который загружает драйвер и позволяет ему зарегистрироваться в качестве поставщика SQL API:

```
...
_ "modernc.org/sqlite"
...
```

Пакет импортируется только для инициализации и не используется напрямую, хотя вы можете найти драйверы, требующие начальной настройки. База данных используется через пакет `database/sql`, как показано в функциях, определенных в листинге 26-9. Функция `listDrivers` записывает доступные драйверы, хотя в этом примере проекта только один. Функция `openDatabase` использует функцию `Open`, описанную в таблице 26-3, для открытия базы данных:

```
...
```

```
db, err = sql.Open("sqlite", "products.db")
...
```

Аргументами функции Open являются имя используемого драйвера и строка подключения к базе данных, которая будет специфичной для используемого ядра базы данных. SQLite открывает базы данных, используя имя файла базы данных.

Результатом функции Open является указатель на структуру sql.DB и ошибка, сообщающая о любых проблемах с открытием базы данных. Структура DB обеспечивает доступ к базе данных, не раскрывая деталей механизма базы данных или его соединений.

Я опишу функции, которые предоставляет структура DB, в следующих разделах. Однако для начала я буду использовать только один метод, показанный в листинге 26-10.

```
package main

func main() {

    listDrivers()
    db, err := openDatabase()
    if (err == nil) {
        db.Close()
    } else {
        panic(err)
    }
}
```

Листинг 26-10 Использование структуры БД в файле main.go в папке data

Метод main вызывает функцию listDrivers для вывода имен загруженных драйверов, а затем вызывает функцию openDatabase для открытия базы данных. С базой пока ничего не делается, но вызывается метод Close. Этот метод, описанный в таблице 26-4, закрывает базу данных и предотвращает выполнение дальнейших операций.

Таблица 26-4 Метод БД для закрытия базы данных

Функция	Описание
Close()	Эта функция закрывает базу данных и предотвращает выполнение дальнейших операций.

Хотя вызов метода Close является хорошей идеей, вам нужно сделать это только после того, как вы полностью закончите работу с базой данных. Одну DB можно использовать для повторных запросов к одной и той же базе данных, а подключения к базе данных будут автоматически управляться за кулисами. Нет необходимости вызывать метод Open, чтобы получить новую DB для каждого запроса, а затем использовать Close, чтобы закрыть ее после завершения запроса.

Скомпилируйте и выполните проект, и вы увидите следующий вывод, который показывает имя драйвера базы данных и подтверждает, что база данных была открыта:

```
Driver: sqlite
Opened database
```

Выполнение операторов и запросов

Структура DB используется для выполнения операторов SQL с использованием методов, описанных в таблице 26-5, которые демонстрируются в следующих разделах.

Таблица 26-5 Методы DB для выполнения операторов SQL

Функция	Описание
Query(query, ...args)	Этот метод выполняет указанный запрос, используя необязательные аргументы-заполнители. Результаты представляют собой структуру Rows, содержащую результаты запроса, и error, указывающую на проблемы с выполнением запроса.
QueryRow(query, ...args)	Этот метод выполняет указанный запрос, используя необязательные аргументы-заполнители. Результатом является структура Row, представляющая первую строку результатов запроса. См. раздел «Выполнение запросов для отдельных строк».
Exec(query, ...args)	Этот метод выполняет операторы или запросы, которые не возвращают строки данных. Метод возвращает Result, описывающий ответ от базы данных, и error, сигнализирующую о проблемах с выполнением. См. раздел «Выполнение других запросов».

Использование контекстов с базами данных

В главе 30 я описываю пакет context и определяемый им интерфейс Context, который используется для управления запросами по мере их обработки сервером. Все важные методы, определенные в пакете database/sql, также имеют версии, которые принимают аргумент Context, что полезно, если вы хотите воспользоваться такими функциями, как тайм-ауты обработки запросов. Я не перечислил эти методы в этой главе, но я широко использую интерфейс Context, включая методы database/sql, которые принимают их в качестве аргументов, в третьей части, где я использую Go и его стандартную библиотеку для создания платформы веб-приложений и интернет-магазина.

Запрос нескольких строк

Метод Query выполняет запрос, извлекающий одну или несколько строк из базы данных. Метод Query возвращает структуру Rows, которая содержит результаты запроса и error, указывающую на проблемы. Доступ к данным строки осуществляется с помощью методов, описанных в таблице 26-6.

Таблица 26-6 Методы структуры строк

Функция	Описание
`Next()`	Этот метод переходит к следующей строке результата. Результатом является логическое значение, которое принимает значение `true`, когда есть данные для чтения, и значение `false`, когда достигнут конец данных, после чего автоматически вызывается метод `Close`.
`NextResultSet()`	Этот метод переходит к следующему набору результатов, когда в одном и том же ответе базы данных имеется несколько наборов результатов. Метод возвращает `true`, если есть другой набор строк для обработки.
`Scan(...targets)`	Этот метод присваивает значения SQL из текущей строки указанным переменным. Значения назначаются с помощью указателей, и метод возвращает `error`, указывающую, что значения не могут быть просканированы. Дополнительные сведения см. в разделе «Понимание метода сканирования».
`Close()`	Этот метод предотвращает дальнейшее перечисление результатов и используется, когда требуются не все данные. Нет необходимости вызывать этот метод, если для перехода используется метод `Next`, пока он не вернет значение `false`.

Листинг 26-11 демонстрирует простой запрос, который показывает, как используется структура `Rows`.

```
package main

import "database/sql"

func queryDatabase(db *sql.DB) {
    rows, err := db.Query("SELECT * from Products")
    if (err == nil) {
        for (rows.Next()) {
            var id, category int
            var name string
            var price float64
            rows.Scan(&id, &name, &category, &price)
            Printfln("Row: %v %v %v %v", id, name, category, price)
        }
    } else {
        Printfln("Error: %v", err)
    }
}

func main() {

    //listDrivers()
    db, err := openDatabase()
    if (err == nil) {
        queryDatabase(db)
        db.Close()
    } else {
        panic(err)
    }
}
```

Листинг 26-11 Запрос базы данных в файле main.go в папке data

Функция queryDatabase выполняет простой запрос SELECT к таблице Products с помощью метода Query, который выдает результат Rows и error. Если error равна nil, цикл for используется для перемещения по строкам результатов путем вызова метода Next, который возвращает true, если есть строка для обработки, и возвращает false, когда достигнут конец данных.

Метод Scan используется для извлечения значений из строки результата и присвоения их переменным Go, например:

```
...
rows.Scan(&id, &name, &category, &price)
...
```

Указатели на переменные передаются методу Scan в том же порядке, в котором столбцы считываются из базы данных. Необходимо позаботиться о том, чтобы переменные Go могли представлять результат SQL, который им будет присвоен. Скомпилируйте и запустите проект, и вы увидите следующие результаты:

```
Opened database
Row: 1 Kayak 1 279
Row: 2 Lifejacket 1 48.95
Row: 3 Soccer Ball 2 19.5
Row: 4 Corner Flags 2 34.95
```

Понимание метода сканирования

Метод Scan чувствителен к количеству, порядку и типам параметров, которые он получает. Если количество параметров не соответствует количеству столбцов в результатах или параметры не могут хранить значения результатов, будет возвращена ошибка, как показано в листинге 26-12.

```
...
func queryDatabase(db *sql.DB) {
    rows, err := db.Query("SELECT * from Products")
    if (err == nil) {
        for (rows.Next()) {
            var id, category int
            var name int
            var price float64
            scanErr := rows.Scan(&id, &name, &category, &price)
            if (scanErr == nil) {
                    Printfln("Row: %v %v %v %v", id, name, category,
price)
            } else {
                Printfln("Scan error: %v", scanErr)
```

```
                break
            }
        }
    } else {
        Printfln("Error: %v", err)
    }
}
...
```

Листинг 26-12 Несовпадающее сканирование в файле main.go в папке

Вызов метода `Scan` в листинге 26-12 предоставляет `int` для значения, которое хранится в базе данных как тип SQL `TEXT`. Скомпилируйте и запустите проект, и вы увидите, что метод `Scan` возвращает ошибку:

```
Scan error: sql: Scan error on column index 1, name "Name": converting
driver.Value type string ("Kayak") to a int: invalid syntax
```

Метод `Scan` не просто пропускает столбец, вызывающий проблему, и в случае возникновения проблемы никакие значения не сканируются.

Понимание того, как можно сканировать значения SQL

Наиболее распространенная проблема с методом `Scan` — это несоответствие между типом данных SQL и переменной Go, в которую он сканируется. Метод `Scan` предлагает некоторую гибкость при сопоставлении значений SQL со значениями Go. Вот краткое изложение наиболее важных правил:

- SQL строки, числовые и логические значения могут быть сопоставлены с их аналогами в Go, хотя следует соблюдать осторожность с числовыми типами, чтобы предотвратить переполнение.
- Числовые и логические типы SQL можно сканировать в строки Go.
- Строки SQL могут быть просканированы в числовые типы Go, но только если строка может быть проанализирована с использованием обычных функций Go (описанных в главе 5) и только если нет переполнения.
- Значения времени SQL можно сканировать в строки Go или значения `*time.Time`.
- Любое значение SQL можно преобразовать в указатель на пустой интерфейс (`*interface{}`), что позволяет преобразовать значение в другой тип.

Это наиболее полезные сопоставления, но подробные сведения см. в документации Go для метода `Scan`. В общем, я предпочитаю выбирать типы консервативно, и я часто просматриваю строки Go, а затем сам анализирую значение, чтобы управлять процессом преобразования. В листинге 26-13 все результирующие значения сканируются в строки.

```
package main
```

```
import "database/sql"

func queryDatabase(db *sql.DB) {
    rows, err := db.Query("SELECT * from Products")
    if (err == nil) {
        for (rows.Next()) {
            var id, category string
            var name string
            var price string
            scanErr := rows.Scan(&id, &name, &category, &price)
            if (scanErr == nil) {
                Printfln("Row: %v %v %v %v", id, name, category,
price)
            } else {
                Printfln("Scan error: %v", scanErr)
                break
            }

        }
    } else {
        Printfln("Error: %v", err)
    }
}

func main() {

    //listDrivers()
    db, err := openDatabase()
    if (err == nil) {
        queryDatabase(db)
        db.Close()
    } else {
        panic(err)
    }
}
```

Листинг 26-13 Сканирование в строки в файле main.go в папке data

Такой подход гарантирует, что вы получите результаты SQL в приложении Go, хотя и требует дополнительной работы по анализу значений для их использования. Скомпилируйте и запустите проект, и вы увидите следующий вывод:

```
Row: 1 Kayak 1 279
Row: 2 Lifejacket 1 48.95
Row: 3 Soccer Ball 2 19.5
Row: 4 Corner Flags 2 34.95
```

Сканирование значений в структуру

Метод Scan работает только с отдельными полями, что означает отсутствие поддержки автоматического заполнения полей структуры. Вместо этого вы должны указать указатели на отдельные поля, для которых результаты содержат значения, как показано в листинге 26-14.

> **Примечание**
> В конце этой главы я продемонстрирую использование пакета Go reflect для динамического сканирования строк в структуры. Подробнее см. в разделе «Использование рефлексии для сканирования данных в структуру».

```
package main

import "database/sql"

type Product struct {
    Id int
    Name string
    Category int
    Price float64
}

func queryDatabase(db *sql.DB) []Product {
    products := []Product {}
    rows, err := db.Query("SELECT * from Products")
    if (err == nil) {
        for (rows.Next()) {
            p := Product{}
                scanErr := rows.Scan(&p.Id, &p.Name, &p.Category,
&p.Price)
            if (scanErr == nil) {
                products = append(products, p)
            } else {
                Printfln("Scan error: %v", scanErr)
                break
            }
        }
    } else {
        Printfln("Error: %v", err)
    }
    return products
}

func main() {
    db, err := openDatabase()
    if (err == nil) {
        products := queryDatabase(db)
        for i, p := range products {
```

```
            Println("#%v: %v", i, p)
        }
        db.Close()
    } else {
        panic(err)
    }
}
```

Листинг 26-14 Сканирование в структуру в файле main.go в папке data

В этом примере сканируются те же данные результата, но для создания среза Product. Скомпилируйте и запустите проект, и вы получите следующий вывод:

```
Opened database
#0: {1 Kayak 1 279}
#1: {2 Lifejacket 1 48.95}
#2: {3 Soccer Ball 2 19.5}
#3: {4 Corner Flags 2 34.95}
```

Этот подход может быть многословным и дублирующим, если у вас есть много типов результатов для анализа, но его преимущество состоит в том, что он прост и предсказуем, и его можно легко адаптировать для отражения сложности результатов. Например, в листинге 26-15 запрос, отправляемый в базу данных, изменяется таким образом, что он включает данные из таблицы Categories.

```
package main

import "database/sql"

type Category struct {
    Id int
    Name string
}

type Product struct {
    Id int
    Name string
    Category
    Price float64
}

func queryDatabase(db *sql.DB) []Product {
    products := []Product {}
    rows, err := db.Query(`
        SELECT Products.Id, Products.Name, Products.Price,
                Categories.Id as Cat_Id, Categories.Name as CatName
                FROM Products, Categories
        WHERE Products.Category = Categories.Id`)
    if (err == nil) {
```

```
        for (rows.Next()) {
            p := Product{}
            scanErr := rows.Scan(&p.Id, &p.Name, &p.Price,
                &p.Category.Id, &p.Category.Name)
            if (scanErr == nil) {
                products = append(products, p)
            } else {
                Printfln("Scan error: %v", scanErr)
                break
            }
        }
    } else {
        Printfln("Error: %v", err)
    }
    return products
}

func main() {
    db, err := openDatabase()
    if (err == nil) {
        products := queryDatabase(db)
        for i, p := range products {
            Printfln("#%v: %v", i, p)
        }
        db.Close()
    } else {
        panic(err)
    }
}
```

Листинг 26-15 Сканирование более сложных результатов в файле main.go в папке data

Результаты SQL-запроса в этом примере включают данные из таблицы `Categories`, которые сканируются во вложенное поле структуры. Скомпилируйте и выполните проект, и вы увидите следующий вывод, включающий данные из двух таблиц:

```
Opened database
#0: {1 Kayak {1 Watersports} 279}
#1: {2 Lifejacket {1 Watersports} 48.95}
#2: {3 Soccer Ball {2 Soccer} 19.5}
#3: {4 Corner Flags {2 Soccer} 34.95}
```

Выполнение операторов с заполнителями

Необязательные аргументы метода `Query` — это значения заполнителей в строке запроса, что позволяет использовать одну строку для разных запросов, как показано в листинге 26-16.

```
package main

import "database/sql"

type Category struct {
    Id int
    Name string
}

type Product struct {
    Id int
    Name string
    Category
    Price float64
}

func queryDatabase(db *sql.DB, categoryName string) []Product {
    products := []Product {}
    rows, err := db.Query(`
        SELECT Products.Id, Products.Name, Products.Price,
                Categories.Id as Cat_Id, Categories.Name as CatName
                FROM Products, Categories
        WHERE Products.Category = Categories.Id
            AND Categories.Name = ?`, categoryName)
    if (err == nil) {
        for (rows.Next()) {
            p := Product{}
            scanErr := rows.Scan(&p.Id, &p.Name, &p.Price,
                &p.Category.Id, &p.Category.Name)
            if (scanErr == nil) {
                products = append(products, p)
            } else {
                Printfln("Scan error: %v", scanErr)
                break
            }
        }
    } else {
        Printfln("Error: %v", err)
    }
    return products
}

func main() {
    db, err := openDatabase()
    if (err == nil) {
        for _, cat := range []string { "Soccer", "Watersports"} {
            Printfln("--- %v Results ---", cat)
            products := queryDatabase(db, cat)
            for i, p := range products {
```

```
                Printfln("#%v: %v %v %v", i, p.Name, p.Category.Name,
p.Price)
            }
        }
        db.Close()
    } else {
        panic(err)
    }
}
```

Листинг 26-16 Использование заполнителей запросов в файле main.go в папке data

Строка SQL-запроса в этом примере содержит вопросительный знак (символ ?), обозначающий заполнитель. Это позволяет избежать необходимости создавать строки для каждого запроса и обеспечивает правильное экранирование значений. Скомпилируйте и выполните проект, и вы увидите следующий вывод, показывающий, как функция queryDatabase вызывает метод Query с разными значениями заполнителя:

```
Opened database
--- Soccer Results ---
#0: Soccer Ball Soccer 19.5
#1: Corner Flags Soccer 34.95
--- Watersports Results ---
#0: Kayak Watersports 279
#1: Lifejacket Watersports 48.95
```

Выполнение запросов для отдельных строк

Метод QueryRow выполняет запрос, который должен вернуть одну строку, что позволяет избежать необходимости перечисления результатов, как показано в листинге 26-17.

```
package main

import "database/sql"

type Category struct {
    Id int
    Name string
}

type Product struct {
    Id int
    Name string
    Category
    Price float64
}
```

```
func queryDatabase(db *sql.DB, id int) (p Product) {
    row := db.QueryRow(`
        SELECT Products.Id, Products.Name, Products.Price,
                Categories.Id as Cat_Id, Categories.Name as CatName
                FROM Products, Categories
        WHERE Products.Category = Categories.Id
            AND Products.Id = ?`, id)
    if (row.Err() == nil) {
        scanErr := row.Scan(&p.Id, &p.Name, &p.Price,
                &p.Category.Id, &p.Category.Name)
        if (scanErr != nil) {
            Printfln("Scan error: %v", scanErr)
        }
    } else {
        Printfln("Row error: %v", row.Err().Error())
    }
    return
}

func main() {
    db, err := openDatabase()
    if (err == nil) {
        for _, id := range []int { 1, 3, 10 } {
            p := queryDatabase(db, id)
            Printfln("Product: %v", p)
        }
        db.Close()
    } else {
        panic(err)
    }
}
```

Листинг 26-17 Запрос одной строки в файле main.go в папке data

Метод QueryRow возвращает структуру Row, которая представляет результат одной строки и определяет методы, описанные в таблице 26-7.

Таблица 26-7 Методы, определяемые структурой строк

Функция	Описание
Scan(...targets)	Этот метод присваивает значения SQL из текущей строки указанным переменным. Значения назначаются с помощью указателей, и метод возвращает error, указывающую, когда значения не могут быть просканированы или в результате нет строк. Если в ответе несколько строк, все строки, кроме первой, будут отброшены.
Err()	Этот метод возвращает ошибку, указывающую на проблемы с выполнением запроса.

Row является единственным результатом метода QueryRow, и его метод Err возвращает ошибки при выполнении запроса. Метод Scan будет сканировать только первую строку результатов и вернет error, если в результатах нет строк.

Скомпилируйте и выполните проект, и вы увидите следующие результаты, в том числе ошибку, выдаваемую методом Scan, когда в результатах нет строк:

```
Opened database
Product: {1 Kayak {1 Watersports} 279}
Product: {3 Soccer Ball {2 Soccer} 19.5}
Scan error: sql: no rows in result set
Product: {0  {0 } 0}
```

Выполнение других запросов

Метод Exec используется для выполнения инструкций, которые не создают строки. Результатом метода Exec является значение Result, определяющее методы, описанные в таблице 26-8, и error, указывающая на проблемы с выполнением оператора.

Таблица 26-8 Методы результатов

Функция	Описание
RowsAffected()	Этот метод возвращает количество строк, затронутых инструкцией, выраженное как int64. Этот метод также возвращает error, которая используется, когда есть проблемы с анализом ответа или когда база данных не поддерживает эту функцию.
LastInsertId()	Этот метод возвращает значение int64, представляющее значение, сгенерированное базой данных при выполнении инструкции, которая обычно является автоматически сгенерированным ключом. Этот метод также возвращает error, которая используется, когда значение, возвращаемое базой данных, не может быть преобразовано в Go int.

В листинге 26-18 показано использование метода Exec для вставки новой строки в таблицу Products.

```
package main

import "database/sql"

type Category struct {
    Id int
    Name string
}

type Product struct {
    Id int
    Name string
    Category
    Price float64
}

func queryDatabase(db *sql.DB, id int) (p Product) {
    row := db.QueryRow(`
        SELECT Products.Id, Products.Name, Products.Price,
```

```
                Categories.Id as Cat_Id, Categories.Name as CatName
                FROM Products, Categories
            WHERE Products.Category = Categories.Id
                AND Products.Id = ?`, id)
    if (row.Err() == nil) {
        scanErr := row.Scan(&p.Id, &p.Name, &p.Price,
                &p.Category.Id, &p.Category.Name)
        if (scanErr != nil) {
            Printfln("Scan error: %v", scanErr)
        }
    } else {
        Printfln("Row error: %v", row.Err().Error())
    }
    return
}

func insertRow(db *sql.DB, p *Product) (id int64) {
    res, err := db.Exec(`
        INSERT INTO Products (Name, Category, Price)
        VALUES (?, ?, ?)`, p.Name, p.Category.Id, p.Price)
    if (err == nil) {
        id, err = res.LastInsertId()
        if (err != nil) {
            Printfln("Result error: %v", err.Error())
        }
    } else {
        Printfln("Exec error: %v", err.Error())
    }
    return
}

func main() {
    db, err := openDatabase()
    if (err == nil) {
        newProduct := Product { Name: "Stadium", Category:
            Category{ Id: 2}, Price: 79500 }
        newID := insertRow(db, &newProduct)
        p := queryDatabase(db, int(newID))
        Printfln("New Product: %v", p)
        db.Close()
    } else {
        panic(err)
    }
}
```

Листинг 26-18 Вставка строки в файл main.go в папку data

Метод `Exec` поддерживает заполнители, а оператор в листинге 26-18 вставляет новую строку в таблицу `Products`, используя поля из структуры `Product`. Метод

`Result.LastInsertId` вызывается для получения значения ключа, назначенного новой строке базой данных, которое затем используется для запроса вновь добавленной строки. Скомпилируйте и запустите проект, и вы увидите следующий вывод:

```
Opened database
New Product: {5 Stadium {2 Soccer} 79500}
```

Вы увидите разные результаты, если будете выполнять проект повторно, поскольку каждой новой строке будет присвоено новое значение первичного ключа.

Использование подготовленных операторов

Структура `DB` обеспечивает поддержку создания подготовленных операторов, которые затем можно использовать для выполнения подготовленного SQL. Таблица 26-9 описывает метод `DB` для создания подготовленных операторов. Подготовленные операторы представлены структурой `Stmt`, которая определяет методы, описанные в таблице 26-10.

Таблица 26-9 Метод DB для создания подготовленных операторов

Функция	Описание
`Prepare(query)`	Этот метод создает подготовленный оператор для указанного запроса. Результатом является структура `Stmt` и `error`, указывающая на проблемы с подготовкой инструкции.

> **Примечание**
> Необычность пакета `database/sql` заключается в том, что многие методы, описанные в таблице 26-5, также создают подготовленные операторы, которые отбрасываются после одного запроса.

Таблица 26-10 Методы, определяемые структурой Stmt

Функция	Описание
`Query(...vals)`	Этот метод выполняет подготовленный оператор с необязательными значениями заполнителей. Результатом являются структура `Rows` и `error`. Этот метод эквивалентен методу `DB.Query`.
`QueryRow(...vals)`	Этот метод выполняет подготовленный оператор с необязательными значениями заполнителей. Результатом являются структура `Row` и `error`. Этот метод эквивалентен методу `DB.QueryRow`.
`Exec(...vals)`	Этот метод выполняет подготовленный оператор с необязательными значениями заполнителей. Результатами являются `Result` и `error`. Этот метод эквивалентен методу `DB.Exec`.
`Close()`	Этот метод закрывает оператор. Операторы не могут быть выполнены после их закрытия.

В листинге 26-19 показано создание подготовленных операторов.

```
package main

import (
    "database/sql"
    _ "modernc.org/sqlite"
)

func listDrivers() {
    for _, driver := range sql.Drivers() {
        Printfln("Driver: %v", driver)
    }
}

var insertNewCategory *sql.Stmt
var changeProductCategory *sql.Stmt

func openDatabase() (db *sql.DB, err error) {
    db, err = sql.Open("sqlite", "products.db")
    if (err == nil) {
        Printfln("Opened database")
            insertNewCategory, _ = db.Prepare("INSERT INTO Categories
(Name) VALUES (?)")
        changeProductCategory, _ =
             db.Prepare("UPDATE Products SET Category = ? WHERE Id =
?")
    }
    return
}
```

Листинг 26-19 Использование подготовленных отчетов в файле database.go в папке data

Подготовленные операторы создаются после открытия базы данных и действительны только до тех пор, пока не будет вызван метод `DB.Close`. В листинге 26-20 подготовленные операторы используются для добавления новой категории в базу данных и присвоения ей продукта.

```
package main

import "database/sql"

type Category struct {
    Id int
    Name string
}

type Product struct {
    Id int
```

```
    Name string
    Category
    Price float64
}

func queryDatabase(db *sql.DB, id int) (p Product) {
    row := db.QueryRow(`
        SELECT Products.Id, Products.Name, Products.Price,
                Categories.Id as Cat_Id, Categories.Name as CatName
                FROM Products, Categories
        WHERE Products.Category = Categories.Id
            AND Products.Id = ?`, id)
    if (row.Err() == nil) {
        scanErr := row.Scan(&p.Id, &p.Name, &p.Price,
                &p.Category.Id, &p.Category.Name)
        if (scanErr != nil) {
            Printfln("Scan error: %v", scanErr)
        }
    } else {
        Printfln("Row error: %v", row.Err().Error())
    }
    return
}

func insertAndUseCategory(name string, productIDs ...int) {
    result, err := insertNewCategory.Exec(name)
    if (err == nil) {
        newID, _ := result.LastInsertId()
        for _, id := range productIDs {
            changeProductCategory.Exec(int(newID), id)
        }
    } else {
        Printfln("Prepared statement error: %v", err)
    }
}

func main() {
    db, err := openDatabase()
    if (err == nil) {
        insertAndUseCategory("Misc Products", 2)
        p := queryDatabase(db, 2)
        Printfln("Product: %v", p)
        db.Close()
    } else {
        panic(err)
    }
}
```

Листинг 26-20 Использование подготовленных операторов в файле main.go в папке data

Функция `insertAndUseCategory` использует подготовленные операторы. Скомпилируйте и выполните проект, и вы увидите следующий вывод, отражающий добавление категории `Misc Products`:

```
Opened database
Product: {2 Lifejacket {3 Misc Products} 48.95}
```

Использование транзакций

Транзакции позволяют выполнять несколько операторов, чтобы все они применялись к базе данных или ни один из них. Структура `DB` определяет описанный в таблице 26-11 метод создания новой транзакции.

Таблица 26-11 Метод DB для создания транзакции

Функция	Описание
`Begin()`	Этот метод запускает новую транзакцию. Результатом является указатель на значение `Tx` и `error`, указывающая на проблемы с созданием транзакции.

Транзакции представлены структурой `Tx`, которая определяет методы, описанные в таблице 26-12.

Таблица 26-12 Методы, определяемые структурой Tx

Функция	Описание
`Query(query, ...args)`	Этот метод эквивалентен методу `DB.Query`, описанному в таблице 26-5, но запрос выполняется в рамках транзакции.
`QueryRow(query, ...args)`	Этот метод эквивалентен методу `DB.QueryRow`, описанному в таблице 26-5, но запрос выполняется в рамках транзакции.
`Exec(query, ...args)`	Этот метод эквивалентен методу `DB.Exec`, описанному в таблице 26-5, но query/statement выполняется в рамках транзакции.
`Prepare(query)`	Этот метод эквивалентен методу `DB.Query`, описанному в таблице 26-9, но созданный им подготовленный оператор выполняется в рамках транзакции.
`Stmt(statement)`	Этот метод принимает подготовленный оператор, созданный за пределами области транзакции, и возвращает тот, который выполняется в рамках транзакции.
`Commit()`	Этот метод фиксирует ожидающие изменения в базе данных, возвращая `error`, указывающую на проблемы с применением изменений.
`Rollback()`	Этот метод прерывает транзакцию, так что ожидающие изменения отбрасываются. Этот метод возвращает `error`, указывающую на проблемы с прерыванием транзакции.

Функция `insertAndUseCategory`, определенная в предыдущем разделе, является хорошим — хотя и простым — кандидатом на транзакцию, поскольку есть две связанные операции. В листинге 26-21 представлена транзакция, которая откатывается, если нет продуктов, соответствующих указанным идентификаторам.

```
package main

import "database/sql"

// ...statements omitted for brevity...

func insertAndUseCategory(db *sql.DB, name string, productIDs ...int)
(err error) {
    tx, err := db.Begin()
    updatedFailed := false
    if (err == nil) {
        catResult, err := tx.Stmt(insertNewCategory).Exec(name)
        if (err == nil) {
            newID, _ := catResult.LastInsertId()
            preparedStatement := tx.Stmt(changeProductCategory)
            for _, id := range productIDs {
                changeResult, err := preparedStatement.Exec(newID, id)
                if (err == nil) {
                    changes, _ := changeResult.RowsAffected()
                    if (changes == 0) {
                        updatedFailed = true
                        break
                    }
                }
            }
        }
    }
    if (err != nil || updatedFailed) {
        Printfln("Aborting transaction %v", err)
        tx.Rollback()
    } else {
        tx.Commit()
    }
    return
}

func main() {
    db, err := openDatabase()
    if (err == nil) {
        insertAndUseCategory(db, "Category_1", 2)
        p := queryDatabase(db, 2)
        Printfln("Product: %v", p)
        insertAndUseCategory(db, "Category_2", 100)
        db.Close()
    } else {
        panic(err)
    }
}
```

Листинг 26-21 Использование транзакции в файле main.go в папке data

Первый вызов insertAndUseCategory завершится успешно, и изменения будут применены к базе данных. Второй вызов insertAndUseCategory завершится ошибкой, что означает, что транзакция завершена, а категория, созданная первым оператором, не применяется к базе данных. Скомпилируйте и запустите проект, и вы увидите следующий вывод:

```
Opened database
Product: {2 Lifejacket {4 Category_1} 48.95}
Aborting transaction <nil>
```

Вы можете увидеть немного другие результаты, особенно если вы снова запустите этот пример, потому что вновь созданной строке базы данных категорий будет присвоен уникальный идентификатор, который будет включен в выходные данные.

Использование рефлексии для сканирования данных в структуру

Рефлексия — это функция, которая позволяет проверять и использовать типы и значения во время выполнения. Рефлексия — это продвинутая и сложная функция, подробно описанная в главах 27–29. Я не буду описывать рефлексию в этой главе, но существуют методы, определенные структурой Rows, которые полезны при использовании рефлексии для обработки ответа базы данных, как описано в таблице 26-13. Возможно, вы захотите вернуться к этому примеру после прочтения глав, посвященных рефлексии.

Таблица 26-13 Методы строк, используемые с рефлексией

Функция	Описание
Columns()	Этот метод возвращает фрагмент строк, содержащих имена столбцов результатов и error, которая используется при закрытии результатов.
ColumnTypes()	Этот метод возвращает срез *ColumnType, который описывает типы данных результирующих столбцов. Этот метод также возвращает error, которая используется, когда результаты закрыты.

Понимание недостатков рефлексии

Рефлексия — это продвинутая функция, о которой можно судить по трем главам, которые мне потребуются, чтобы описать, как она используется. Этот пример просто показывает, что возможно с информацией, предоставляемой пакетом database/sql. Для простоты в этом примере заданы фиксированные ожидания относительно структуры строк результатов.

Указание отдельных полей, как показано в листинге 26-14, — это самый простой и надежный подход к просмотру структур. Если вы настроены на динамическое сканирование структур, рассмотрите возможность

использования одного из хорошо протестированных сторонних пакетов, например SQLX (https://github.com/jmoiron/sqlx).

Эти методы описывают структуру строк, возвращаемых из базы данных. Метод Columns возвращает фрагмент строки, содержащий имена столбцов результатов. Метод ColumnTypes возвращает срез указателей на структуру ColumnType, которая определяет методы, описанные в таблице 26-14.

Таблица 26-14 Методы ColumnType

Функция	Описание
Name()	Этот метод возвращает имя столбца, указанное в результатах, выраженное в виде строки.
DatabaseTypeName()	Этот метод возвращает имя типа столбца в базе данных, выраженное в виде строки..
Nullable()	Этот метод возвращает два bool результата. Первый результат true, если тип базы данных может быть null. Второй результат true, если драйвер поддерживает нулевые значения.
DecimalSize()	Этот метод возвращает сведения о размере десятичных значений. Результатом является int64, указывающий точность, int64, указывающий масштаб, и bool, который имеет значение true для десятичных типов и false для других типов.
Length()	Этот метод возвращает длину для типов баз данных, которые могут иметь переменную длину. Результатом является int64, указывающий длину, и bool значение, которое имеет значение true для типов, определяющих длину, и значение false для других типов.
ScanType()	Этот метод возвращает reflect.Type, указывающий тип Go, который будет использоваться при сканировании этого столбца с помощью метода Rows.Scan. См. главы 27–29 для получения подробной информации об использовании пакета reflect.

В листинге 26-22 используется метод Columns для сопоставления имен столбцов в данных результатов с полями структуры и используется метод ColumnType.ScanType, чтобы гарантировать, что типы результатов могут быть безопасно назначены совпавшему полю структуры.

Осторожно
Как уже отмечалось, этот пример основан на функциях, описанных в последующих главах. Вам следует прочитать главы 27–29 и вернуться к этому примеру, как только вы поймете, как работает рефлексия в Go.

```
package main

import (
    "database/sql"
    _ "modernc.org/sqlite"
    "reflect"
    "strings"
```

```
)

func listDrivers() {
    for _, driver := range sql.Drivers() {
        Printfln("Driver: %v", driver)
    }
}

var insertNewCategory *sql.Stmt
var changeProductCategory *sql.Stmt

func openDatabase() (db *sql.DB, err error) {
    db, err = sql.Open("sqlite", "products.db")
    if (err == nil) {
        Printfln("Opened database")
            insertNewCategory, _ = db.Prepare("INSERT INTO Categories
(Name) VALUES (?)")
        changeProductCategory, _ =
            db.Prepare("UPDATE Products SET Category = ? WHERE Id = ?")
    }
    return
}

func scanIntoStruct(rows *sql.Rows, target interface{}) (results
interface{},
        err error) {
    targetVal := reflect.ValueOf(target)
    if (targetVal.Kind() == reflect.Ptr) {
        targetVal = targetVal.Elem()
    }
    if (targetVal.Kind() != reflect.Struct) {
        return
    }
    colNames, _ := rows.Columns()
    colTypes, _ := rows.ColumnTypes()
    references := []interface{} {}
    fieldVal := reflect.Value{}
    var placeholder interface{}

    for i, colName := range colNames {
        colNameParts := strings.Split(colName, ".")
        fieldVal = targetVal.FieldByName(colNameParts[0])
        if (fieldVal.IsValid() && fieldVal.Kind() == reflect.Struct &&
            len(colNameParts) > 1 ) {
            var namePart string
            for _, namePart = range colNameParts[1:] {
                compFunction := matchColName(namePart)
                fieldVal = fieldVal.FieldByNameFunc(compFunction)
            }
```

```
        }

        if (!fieldVal.IsValid() ||
                !colTypes[i].ScanType().ConvertibleTo(fieldVal.Type()))
{
            references = append(references, &placeholder)
                } else if (fieldVal.Kind() != reflect.Ptr &&
fieldVal.CanAddr()) {
            fieldVal = fieldVal.Addr()
            references = append(references, fieldVal.Interface())
        }
    }

    resultSlice := reflect.MakeSlice(reflect.SliceOf(targetVal.Type()),
0, 10)
    for rows.Next() {
        err = rows.Scan(references...)
        if (err == nil) {
            resultSlice = reflect.Append(resultSlice, targetVal)
        } else {
            break
        }
    }
    results = resultSlice.Interface()
    return
}

func matchColName(colName string) func(string) bool {
    return func(fieldName string) bool {
        return strings.EqualFold(colName, fieldName)
    }
}
```

Листинг 26-22 Сканирование структур с рефлексией в файле database.go в папке the data

Функция scanIntoStruct принимает значение Rows и цель, в которую будут сканироваться значения. Функции рефлексии Go используются для поиска поля в структуре с тем же именем, совпадающим независимо от регистра. Для полей вложенной структуры имя столбца должно соответствовать имени поля, разделенному точками, чтобы, например, поле Category.Name сканировалось из результирующего столбца с именем category.name.

Создается срез указателей для работы метода Scan, а отсканированные значения структуры добавляются к срезу, который используется для получения результатов метода. Если ни одно поле структуры не соответствует столбцу результатов, используется фиктивное значение, так как метод Scan ожидает полный набор указателей для сканирования данных. В листинге 26-23 новая функция используется для просмотра результатов запроса.

```
package main

import "database/sql"

type Category struct {
    Id int
    Name string
}

type Product struct {
    Id int
    Name string
    Category
    Price float64
}

func queryDatabase(db *sql.DB) (products []Product, err error) {
        rows,  err  :=  db.Query(`SELECT  Products.Id,  Products.Name,
Products.Price,
                  Categories.Id  as  "Category.Id",  Categories.Name  as
"Category.Name"
            FROM Products, Categories
            WHERE Products.Category = Categories.Id`)
    if (err != nil) {
        return
    } else {
        results, err := scanIntoStruct(rows, &Product{})
        if err == nil {
            products = (results).([]Product)
        } else {
            Printfln("Scanning error: %v", err)
        }
    }
    return
}

func main() {
    db, err := openDatabase()
    if (err == nil) {
        products, _ := queryDatabase(db)
        for _, p := range products {
            Printfln("Product: %v", p)
        }
        db.Close()
    } else {
        panic(err)
    }
}
```

Листинг 26-23 Сканирование результатов запроса в файле main.go в папке data

База данных запрашивается, указывая имена столбцов, которые будут сопоставляться с полями, определенными в структурах `Product` и `Category`. Как я объясню в главе 27, результаты, полученные в результате размышления, требуют утверждения, чтобы сузить их тип.

Эффект этого примера заключается в том, что сканирование выполняется динамически на основе сопоставления столбцов результатов с именами и типами полей структуры. Скомпилируйте и запустите проект, и вы увидите следующий вывод:

```
Opened database
Product: {1 Kayak {1 Watersports} 279}
Product: {2 Lifejacket {4 Category_1} 48.95}
Product: {3 Soccer Ball {2 Soccer} 19.5}
Product: {4 Corner Flags {2 Soccer} 34.95}
Product: {5 Stadium {2 Soccer} 79500}
```

Резюме

В этой главе я описал поддержку стандартной библиотеки Go для работы с базами данных SQL, которые просты, но хорошо продуманы и просты в использовании. В следующей главе я начну процесс описания функций рефлексии Go, которые позволяют определять типы и использовать их во время выполнения.

27. Использование рефлексии

В этой главе я описываю поддержку *рефлексии* (отражения) в Go, которая позволяет приложению работать с типами, неизвестными при компиляции проекта, что полезно, например, для создания API, которые будут использоваться другими проектами. Вы можете увидеть широкое использование рефлексии в третьей части, где я создаю пользовательскую структуру веб-приложения. В этой ситуации код в структуре приложения ничего не знает о типах данных, которые будут определены приложениями, для которых он используется, и должен использовать рефлексию для получения информации об этих типах и для работы со значениями, созданными из них.

Рефлексию следует использовать с осторожностью. Поскольку используемые типы данных неизвестны, обычные меры безопасности, применяемые компилятором, не могут быть использованы, и ответственность за проверку и безопасное использование типов лежит на программисте. Код рефлексии имеет тенденцию быть многословным и трудным для чтения, и при написании кода рефлексии легко сделать ошибочные предположения, которые не проявляются как ошибки, пока они не будут использованы с реальными типами данных, что часто происходит, когда код оказывается в руках разработчиков. Ошибки в коде рефлексии обычно вызывают панику.

Код, использующий рефлексию, работает медленнее, чем обычный код Go, хотя в большинстве проектов это не будет проблемой. Если у вас нет особых требований к производительности, вы обнаружите, что весь код Go работает с приемлемой скоростью, независимо от того, использует ли он рефлексию или нет. Есть некоторые задачи программирования на Go, которые можно выполнить только с помощью рефлексии, а рефлексия используется во всей стандартной библиотеке.

Это не означает, что вы должны спешить с использованием рефлексии — ее сложно использовать и легко ошибиться, — но бывают случаи, когда ее нельзя избежать, и как только вы поймете, как она работает, осторожное применение функций отражения Go может

привести к созданию гибкого и адаптируемого кода, как вы увидите в третьей части. Таблица 27-1 помещает рефлексию в контекст.

Таблица 27-1 Рефлексия в контексте

Вопрос	**Ответ**
Что это?	Рефлексия позволяет проверять типы и значения во время выполнения, даже если эти типы не были определены во время компиляции.
Почему это полезно?	Рефлексия полезна при написании кода, основанного на типах, которые будут определены в будущем, например, при написании API, который будет использоваться в других проектах.
Как это используется?	Пакет `reflect` предоставляет функции, позволяющие отображать типы и значения, чтобы их можно было использовать без явного знания используемых типов данных.
Есть ли подводные камни или ограничения?	Рефлексия сложна и требует пристального внимания к деталям. Легко делать предположения о типах данных, которые не создают проблем, пока код не будет использован в других проектах.
Есть ли альтернативы?	Рефлексия требуется только тогда, когда типы неизвестны при компиляции проекта. Стандартные возможности языка Go следует использовать, когда типы известны заранее.

Таблица 27-2 суммирует главу.

Таблица 27-2 Краткое содержание главы

Проблема	**Решение**	**Листинг**
Получить отраженные типы и значения	Используйте функции `TypeOf` и `ValueOf`	8
Проверить отраженный тип	Используйте методы, определенные интерфейсом `Type`	9
Проверить отраженное значение	Используйте методы, определенные структурой `Value`	10
Определить отраженный тип	Проверьте его вид и, при необходимости, тип элемента	11, 12
Получить базовый тип	Используйте метод `Interface`	13
Установка отраженного значения	Используйте методы `Set*`	14–16
Сравнить отраженные значения	Используйте метод `Comparable` и оператор сравнения Go или функцию `DeepEqual`	17–19
Преобразование отраженного значения в другой тип	Используйте методы `ConvertibleTo` и `Convert`	20, 21

Проблема	Решение	Листинг
Создать новое отраженное значение	Используйте тип New для базовых типов и один из методов Make* для других типов	22

Подготовка к этой главе

Чтобы подготовиться к этой главе, откройте новую командную строку, перейдите в удобное место и создайте каталог с именем reflection. Запустите команду, показанную в листинге 27-1, в папке reflection, чтобы создать файл модуля.

> **Подсказка**
> Вы можете загрузить пример проекта для этой главы — и для всех остальных глав этой книги — с https://github.com/apress/pro-go. См. Главу 2 о том, как получить помощь, если у вас возникнут проблемы с запуском примеров.

```
go mod init reflection
```

Листинг 27-1 Инициализация модуля

Добавьте файл с именем printer.go в папку reflection с содержимым, показанным в листинге 27-2.

```
package main

import "fmt"

func Printfln(template string, values ...interface{}) {
    fmt.Printf(template + "\n", values...)
}
```

Листинг 27-2 Содержимое файла printer.go в папке reflection

Добавьте файл с именем types.go в папку reflection с содержимым, показанным в листинге 27-3.

```
package main

type Product struct {
```

```
    Name, Category string
    Price float64
}

type Customer struct {
    Name, City string
}
```

Листинг 27-3 Содержимое файла types.go в папке reflection

Добавьте файл с именем `main.go` в папку `reflection` с содержимым, показанным в листинге 27-4.

```
package main

func printDetails(values ...Product) {
    for _, elem := range values {
         Printfln("Product: Name: %v, Category: %v, Price: 
%v",
           elem.Name, elem.Category, elem.Price)
    }
}

func main() {

    product := Product {
        Name: "Kayak", Category: "Watersports", Price: 279,
    }
    printDetails(product)
}
```

Листинг 27-4 Содержимое файла main.go в папке reflection

Используйте командную строку для запуска команды, показанной в листинге 27-5, в папке `usingstrings`.

```
go run .
```

Листинг 27-5 Запуск примера проекта

Код в проекте будет скомпилирован и выполнен, что даст следующие результаты:

```
Product: Name: Kayak, Category: Watersports, Price: 279
```

Понимание необходимости рефлексии

Система типов Go строго соблюдается, что означает, что вы не можете использовать значение одного типа при проверке другого типа. В листинге 27-6 создается значение Customer и передается функции printDetails, которая определяет вариативный параметр Product.

```
package main

func printDetails(values ...Product) {
    for _, elem := range values {
          Printfln("Product: Name: %v, Category: %v, Price: %v",
           elem.Name, elem.Category, elem.Price)
    }
}

func main() {

    product := Product {
        Name: "Kayak", Category: "Watersports", Price: 279,
    }
    customer := Customer { Name: "Alice", City: "New York" }
    printDetails(product, customer)
}
```

Листинг 27-6 Смешивание типов в файле main.go в папке reflection

Этот код не будет компилироваться, потому что он нарушает правила типов Go. При компиляции проекта вы увидите следующую ошибку:

```
.\main.go:16:17: cannot use customer (type Customer) as type Product in argument to printDetails
```

В главе 11 я представил интерфейсы, которые позволяют определять общие характеристики с помощью методов, которые можно вызывать независимо от типа, реализующего интерфейс. В главе 11 также представлен пустой интерфейс, который можно использовать для принятия любого типа, как показано в листинге 27-7.

```
package main

func printDetails(values ...interface{}) {
    for _, elem := range values {
        switch val := elem.(type) {
            case Product:
                Println("Product: Name: %v, Category: %v,
Price: %v",
                    val.Name, val.Category, val.Price)
            case Customer:
                    Println("Customer: Name: %v, City: %v",
val.Name, val.City)
        }

    }
}

func main() {

    product := Product {
        Name: "Kayak", Category: "Watersports", Price: 279,
    }
    customer := Customer { Name: "Alice", City: "New York" }
    printDetails(product, customer)
}
```

Листинг 27-7 Использование пустого интерфейса в файле main.go в папке reflection

Пустой интерфейс позволяет функции `printDetails` получать любой тип, но не позволяет получить доступ к определенным функциям, поскольку интерфейс не определяет методы. Утверждение типа требуется для сужения пустого интерфейса до определенного типа, который затем позволяет обрабатывать каждое значение. Скомпилируйте и выполните код, и вы получите следующий вывод:

```
Product: Name: Kayak, Category: Watersports, Price: 279
Customer: Name: Alice, City: New York
```

Ограничение этого подхода состоит в том, что функция `printDetails` может обрабатывать только заранее известные типы. Каждый раз, когда я добавляю тип в проект, мне приходится расширять функцию `printDetails` для обработки этого типа.

Многие проекты будут иметь дело с достаточно небольшим набором типов, так что это не будет проблемой, или смогут определять интерфейсы с помощью методов, обеспечивающих доступ к общим функциям. Рефлексия решает эту проблему для тех проектов, для которых это не так, либо потому, что приходится иметь дело с большим количеством типов, либо потому, что интерфейсы и методы не могут быть написаны.

Использование рефлексии

Пакет `reflect` предоставляет функции отражения Go, а ключевые функции называются `TypeOf` и `ValueOf`, обе из которых описаны в таблице 27-3 для быстрого ознакомления.

Таблица 27-3 Ключевые функции рефлексии

Функция	Описание
`TypeOf(val)`	Эта функция возвращает значение, реализующее интерфейс `Type`, описывающий тип указанного значения.
`ValueOf(val)`	Эта функция возвращает структуру `Value`, которая позволяет проверять указанное значение и управлять им.

За функциями `TypeOf` и `ValueOf` и их результатами стоит много деталей, и легко упустить из виду, почему отражение может быть полезным. Прежде чем перейти к деталям, в листинге 27-8 функция `printDetails` пересматривается, чтобы использовать пакет `reflect`, чтобы он мог обрабатывать любой тип, демонстрируя базовый шаблон, необходимый для применения отражения.

```
package main

import (
    "reflect"
    "strings"
    "fmt"
)

func printDetails(values ...interface{}) {
    for _, elem := range values {
        fieldDetails := []string {}
```

```
        elemType := reflect.TypeOf(elem)
        elemValue := reflect.ValueOf(elem)
        if elemType.Kind() == reflect.Struct {
            for i := 0; i < elemType.NumField(); i++ {
                fieldName := elemType.Field(i).Name
                fieldVal := elemValue.Field(i)
                fieldDetails = append(fieldDetails,
                    fmt.Sprintf("%v: %v", fieldName, fieldVal
))
            }
                        Printfln("%v:  %v",  elemType.Name(),
strings.Join(fieldDetails, ", "))
        } else {
            Printfln("%v: %v", elemType.Name(), elemValue)
        }
    }
}

type Payment struct {
    Currency string
    Amount float64
}

func main() {

    product := Product {
        Name: "Kayak", Category: "Watersports", Price: 279,
    }
    customer := Customer { Name: "Alice", City: "New York" }
    payment := Payment { Currency: "USD", Amount: 100.50 }
    printDetails(product, customer, payment, 10, true)
}
```

Листинг 27-8 Использование Reflection в файле main.go в папке reflection

Код, использующий рефлексию, может быть многословным, но после знакомства с основами становится легко следовать основному шаблону. Важно помнить, что есть два аспекта отражения, которые работают вместе: отраженный тип и отраженное значение.

Отраженный тип дает вам доступ к деталям типа Go, не зная заранее, что это такое. Вы можете исследовать отраженный тип, изучая

его детали и характеристики с помощью методов, определенных интерфейсом Type.

Отраженное значение позволяет вам работать с конкретным значением, которое вам было предоставлено. Вы не можете просто прочитать поле структуры или вызвать метод, например, как в обычном коде, когда вы не знаете, с каким типом имеете дело.

Использование отраженного типа и отраженного значения приводит к многословию кода. Например, если вы знаете, что имеете дело со структурой Product, вы можете просто прочитать поле Name и получить строковый результат. Если вы не знаете, какой тип используется, вы должны использовать отраженный тип, чтобы установить, имеете ли вы дело со структурой и имеет ли она поле Name. Как только вы определили, что такое поле есть, вы используете отраженное значение, чтобы прочитать это поле и получить его значение.

Рефлексия может сбивать с толку, поэтому я пройдусь по операторам в листинге 27-8 и кратко опишу эффект, который оказывает каждое из них, что обеспечит некоторый контекст для последующего подробного описания пакета reflect.

Функция printDetails определяет переменный параметр, используя пустой интерфейс, который перечисляется с помощью ключевого слова range:

```
...
func printDetails(values ...interface{}) {
    for _, elem := range values {
...
```

Как уже отмечалось, пустой интерфейс позволяет функции принимать любой тип данных, но не позволяет получить доступ к функциям какого-либо конкретного типа. Пакет reflect используется для получения отраженного типа и отраженного значения каждого полученного значения:

```
...
elemType := reflect.TypeOf(elem)
elemValue := reflect.ValueOf(elem)
...
```

Функция TypeOf возвращает отраженный тип, который описывается интерфейсом Type. Функция ValueOf возвращает отраженное значение, которое представлено интерфейсом Value.

Следующим шагом является определение типа обрабатываемого типа, что делается путем вызова метода Type.Kind:

```
...
if elemType.Kind() == reflect.Struct {
...
```

Пакет reflect определяет константы, идентифицирующие различные типы типов в Go, которые я описываю в таблице 27-5. В этом операторе оператор if используется для определения того, является ли отраженный тип структурой. Если это структура, то используется цикл for с методом NumField, который возвращает количество полей, определяемых структурой:

```
...
for i := 0; i < elemType.NumField(); i++ {
...
```

В цикле for получаются имя и значение поля:

```
...
fieldName := elemType.Field(i).Name
fieldVal := elemValue.Field(i)
...
```

Вызов метода Field для отраженного типа возвращает StructField, который описывает одно поле, включая поле Name. Вызов метода Field для отраженного значения возвращает структуру Value, которая представляет значение поля.

Имя и значение поля добавляются к срезу строк, который является частью вывода. Пакет fmt используется для создания строкового представления значения поля:

```
...
fieldDetails = append(fieldDetails, fmt.Sprintf("%v: %v",
fieldName, fieldVal ))
...
```

После того, как все поля структуры обработаны, выписывается строка, содержащая имя отраженного типа, которое получается с помощью метода Name, и для каждого поля получается подробная информация:

```
...
Printfln("%v:                  %v",                  elemType.Name(),
strings.Join(fieldDetails, ", "))
...
```

Если отраженный тип не является структурой, то выводится более простое сообщение, содержащее имя отраженного типа и значение, форматирование которого обрабатывается пакетом fmt:

```
...
Printfln("%v: %v", elemType.Name(), elemValue)
...
```

Новый код позволяет функции printDetails получать данные любого типа, включая вновь определенную структуру Payment и встроенные типы, такие как значения int и bool. Скомпилируйте и запустите проект, и вы увидите следующий вывод:

```
Product: Name: Kayak, Category: Watersports, Price: 279
Customer: Name: Alice, City: New York
Payment: Currency: USD, Amount: 100.5
int: 10
bool: true
```

Использование основных функций типа

Интерфейс Type предоставляет основные сведения о типе с помощью методов, описанных в таблице 27-4. Существуют специальные методы для работы с определенными типами, такими как массивы, которые описаны в следующих разделах, но именно эти методы предоставляют основные сведения обо всех типах.

Таблица 27-4 Основные методы, определяемые интерфейсом Type

Функция	Описание
Name()	Этот метод возвращает имя типа.

Функция	Описание
PkgPath()	Этот метод возвращает путь пакета для типа. Пустая строка возвращается для встроенных типов, таких как int и bool.
Kind()	Этот метод возвращает вид типа, используя значение, которое соответствует одному из постоянных значений, определенных пакетом reflect, как описано в таблице 27-5.
String()	Этот метод возвращает строковое представление имени типа, включая имя пакета.
Comparable()	Этот метод возвращает значение true, если значения этого типа можно сравнить с помощью стандартного оператора сравнения, как описано в разделе «Сравнение значений».
AssignableTo(type)	Этот метод возвращает значение true, если значения этого типа могут быть присвоены переменным или полям указанного отраженного типа.

Пакет reflect определяет тип с именем Kind, который является псевдонимом для uint и используется для серии констант, описывающих разные типы типов, как описано в таблице 27-5.

Таблица 27-5 Kind константы

Функция	Описание
Bool	Это значение обозначает bool значение
Int, Int8, Int16, Int32, Int64	Эти значения обозначают различные размеры целочисленных типов
Uint, Uint8, Uint16, Uint32, Uint64	Эти значения обозначают различные размеры целочисленных типов без знака
Float32, Float64	Эти значения обозначают различные размеры типов с плавающей запятой
String	Это значение обозначает строку
Struct	Это значение обозначает структуру
Array	Это значение обозначает массив
Slice	Это значение обозначает срез
Map	Это значение обозначает карту
Chan	Это значение обозначает канал
Func	Это значение определяет функцию
Interface	Это значение обозначает интерфейс
Ptr	Это значение обозначает указатель

Функция	Описание
Uintptr	Это значение обозначает небезопасный указатель, который не описан в этой книге

Листинг 27-9 упрощает пример для отображения сведений из отраженного типа каждого из значений, полученных функцией printDetails.

```
package main

import (
    "reflect"
    // "strings"
    // "fmt"
)

func getTypePath(t reflect.Type) (path string) {
    path = t.PkgPath()
    if (path == "") {
        path = "(built-in)"
    }
    return
}

func printDetails(values ...interface{}) {
    for _, elem := range values {
        elemType := reflect.TypeOf(elem)
        Printfln("Name: %v, PkgPath: %v, Kind: %v",
                     elemType.Name(),  getTypePath(elemType),
elemType.Kind())
    }
}

type Payment struct {
    Currency string
    Amount float64
}

func main() {

    product := Product {
        Name: "Kayak", Category: "Watersports", Price: 279,
```

```
    }
    customer := Customer { Name: "Alice", City: "New York" }
    payment := Payment { Currency: "USD", Amount: 100.50 }
    printDetails(product, customer, payment, 10, true)
}
```

Листинг 27-9 Печать сведений о типе в файле main.go в папке reflection

Я добавил функцию, которая заменяет пустые имена пакетов, чтобы встроенные типы были более понятными. Скомпилируйте и запустите проект, и вы увидите следующий вывод:

```
Name: Product, PkgPath: main, Kind: struct
Name: Customer, PkgPath: main, Kind: struct
Name: Payment, PkgPath: main, Kind: struct
Name: int, PkgPath: (built-in), Kind: int
Name: bool, PkgPath: (built-in), Kind: bool
```

Многие функции отражения, характерные для одного типа типа, такие как массивы, например, вызовут панику, если они будут вызваны для других типов, что делает метод `Kind` особенно важным при использовании отражения.

Использование базовых возможностей Value

Для каждой группы признаков отраженного типа существуют соответствующие признаки для отраженных значений. Структура `Value` определяет методы, описанные в таблице 27-6, которые обеспечивают доступ к основным функциям отражения, включая доступ к базовому значению.

Таблица 27-6 Основные методы, определяемые структурой Value

Функция	Описание
`Kind()`	Этот метод возвращает вид типа значения, используя одно из значений из таблицы 27-5.
`Type()`	Этот метод возвращает `Type` для `Value`.
`IsNil()`	Этот метод возвращает `true`, если значение равно нулю. Этот метод вызовет панику, если базовое значение не является функцией, интерфейсом, указателем, срезом или каналом.
`IsZero()`	Этот метод возвращает значение `true`, если базовое значение является нулевым значением для своего типа.

Функция	Описание
Bool()	Этот метод возвращает базовое bool значение. Метод вызывает панику, если Kind базового значения не является Bool.
Bytes()	Этот метод возвращает базовое значение []byte. Метод вызывает панику, если базовое значение не является байтовым срезом. Я демонстрирую, как определить тип слайса в разделе «Идентификация байтовых срезов».
Int()	Этот метод возвращает базовое значение в виде int64. Метод вызывает панику, если Kind базового значения не является Int, Int8, Int16, Int32 или Int64.
Uint()	Этот метод возвращает базовое значение в виде uint64. Метод вызывает панику, если Kind базового значения не является Uint, Uint8, Uint16, Uint32 или Uint64.
Float()	Этот метод возвращает базовое значение в виде float64. Метод вызывает панику, если Kind базового значения не равен Float32 или Float64.
String()	Этот метод возвращает базовое значение в виде строки, если значение Kind — String. Для других значений Kind этот метод возвращает строку <T Value>, где T — базовый тип, например <int Value>.
Elem()	Этот метод возвращает Value, на которое указывает указатель. Этот метод также можно использовать с интерфейсами, как описано в главе 29. Этот метод вызывает панику, если Kind базового значения не равен Ptr.
IsValid()	Этот метод возвращает false, если Value является нулевым значением, созданным как Value{}, а не полученным, например, с помощью ValueOf. Этот метод не относится к отраженным значениям, которые являются нулевым значением их отраженного типа. Если этот метод возвращает false, то все остальные методы Value будут паниковать.

При использовании методов, возвращающих базовое значение, важно проверять результат Kind, чтобы избежать паники. В листинге 27-10 показаны некоторые методы, описанные в таблице.

```
package main

import (
    "reflect"
    // "strings"
    // "fmt"
)

func printDetails(values ...interface{}) {
    for _, elem := range values {
        elemValue := reflect.ValueOf(elem)
        switch elemValue.Kind() {
            case reflect.Bool:
```

```
                var val bool = elemValue.Bool()
                Printfln("Bool: %v", val)
            case reflect.Int:
                var val int64 = elemValue.Int()
                Printfln("Int: %v", val)
            case reflect.Float32, reflect.Float64:
                var val float64 = elemValue.Float()
                Printfln("Float: %v", val)
            case reflect.String:
                var val string = elemValue.String()
                Printfln("String: %v", val)
            case reflect.Ptr:
                var val reflect.Value = elemValue.Elem()
                if (val.Kind() == reflect.Int) {
                    Printfln("Pointer to Int: %v", val.Int())
                }
            default:
                Printfln("Other: %v", elemValue.String())
        }
    }
}

func main() {

    product := Product {
        Name: "Kayak", Category: "Watersports", Price: 279,
    }
    number := 100
    printDetails(true, 10, 23.30, "Alice", &number, product)
}
```

Листинг 27-10 Использование методов основных значений в файле main.go в папке reflection

В этом примере используется оператор `switch` с результатом метода `Kind` для определения типа значения и вызывается соответствующий метод для получения базового значения. Скомпилируйте и запустите проект, и вы увидите следующий вывод:

```
Bool: true
Int: 10
Float: 23.3
String: Alice
```

```
Pointer to Int: 100
Other: <main.Product Value>
```

Метод `String` ведет себя иначе, чем другие методы, и не вызывает паники при вызове значения, не являющегося строкой. Вместо этого метод возвращает строку, подобную этой:

```
...
Other: <main.Product Value>
...
```

Это не типичное использование метода `String`, встречающееся в других местах стандартной библиотеки Go, где этот метод обычно возвращает строковое представление значения. При использовании отражения вы можете либо использовать методы, описанные в следующих разделах, либо положиться на пакет формата, который использует те же методы, чтобы создать для вас строковые представления значений.

Определение типов

Обратите внимание, что при работе с указателями в листинге 27-10 требуется два шага. На первом этапе используется метод `Kind` для определения значения `Ptr`, а на втором этапе используется метод `Elem` для получения `Value`, представляющего данные, на которые ссылается указатель:

```
...
case reflect.Ptr:
    var val reflect.Value = elemValue.Elem()
    if (val.Kind() == reflect.Int) {
        Printfln("Pointer to Int: %v", val.Int())
}
...
```

Первый шаг говорит мне, что я имею дело с указателем, а второй шаг говорит мне, что он указывает на значение типа `int`. Этот процесс можно упростить, выполнив сравнение отраженных типов. Если два значения имеют одинаковый тип данных Go, то оператор сравнения

вернет true при применении к результатам функции reflect.TypeOf, как показано в листинге 27-11.

```
package main

import (
    "reflect"
    // "strings"
    // "fmt"
)

var intPtrType = reflect.TypeOf((*int)(nil))

func printDetails(values ...interface{}) {
    for _, elem := range values {
        elemValue := reflect.ValueOf(elem)
        elemType := reflect.TypeOf(elem)
        if (elemType == intPtrType) {
                            Println("Pointer to Int: %v",
elemValue.Elem().Int())
        } else {
            switch elemValue.Kind() {
                case reflect.Bool:
                    var val bool = elemValue.Bool()
                    Println("Bool: %v", val)
                case reflect.Int:
                    var val int64 = elemValue.Int()
                    Println("Int: %v", val)
                case reflect.Float32, reflect.Float64:
                    var val float64 = elemValue.Float()
                    Println("Float: %v", val)
                case reflect.String:
                    var val string = elemValue.String()
                    Println("String: %v", val)
                // case reflect.Ptr:
                          //        var val reflect.Value =
elemValue.Elem()
                //     if (val.Kind() == reflect.Int) {
                  //          Println("Pointer to Int: %v",
val.Int())
                //     }
                default:
```

```
                        Printfln("Other: %v", elemValue.String())
                }
            }
        }
}

func main() {

    product := Product {
        Name: "Kayak", Category: "Watersports", Price: 279,
    }
    number := 100
    printDetails(true, 10, 23.30, "Alice", &number, product)
}
```

Листинг 27-11 Сравнение типов в файле main.go в папке reflection

Этот метод начинается со значения `nil` и преобразует его в указатель на значение `int`, которое затем передается в функцию `TypeOf` для получения `Type`, который можно использовать в сравнениях:

```
...
var intPtrType = reflect.TypeOf((*int)(nil))
...
```

Круглые скобки, необходимые для выполнения этой операции, затрудняют ее чтение, но такой подход позволяет избежать необходимости определять переменную только для того, чтобы получить ее `Type`. `Type` можно использовать с обычным оператором сравнения Go:

```
...
if (elemType == intPtrType) {
    Printfln("Pointer to Int: %v", elemValue.Elem().Int())
} else {
...
```

Сравнение таких типов может быть проще, чем проверка значения `Kind` как для типа указателя, так и для значения, на которое он

указывает. Скомпилируйте и выполните код, и вы увидите следующий вывод:

```
Bool: true
Int: 10
Float: 23.3
String: Alice
Pointer to Int: 100
Other: <main.Product Value>
```

Идентификация байтовых срезов

Использование оператора сравнения также является хорошим способом обеспечения безопасности метода Bytes. Метод Bytes вызовет панику, если он будет вызван для любого типа, отличного от среза байтов, но метод Kind указывает только срезы, а не их содержимое. В листинге 27-12 определяется переменная типа для байтовых срезов и используется с оператором сравнения, чтобы определить, когда безопасно вызывать метод Bytes.

```
package main

import (
    "reflect"
    // "strings"
    // "fmt"
)

var intPtrType = reflect.TypeOf((*int)(nil))
var byteSliceType = reflect.TypeOf([]byte(nil))

func printDetails(values ...interface{}) {
    for _, elem := range values {
        elemValue := reflect.ValueOf(elem)
        elemType := reflect.TypeOf(elem)
        if (elemType == intPtrType) {
                            Printfln("Pointer to Int: %v",
elemValue.Elem().Int())
        } else if (elemType == byteSliceType) {
            Printfln("Byte slice: %v", elemValue.Bytes())
        } else {
            switch elemValue.Kind() {
```

```
                case reflect.Bool:
                    var val bool = elemValue.Bool()
                    Printfln("Bool: %v", val)
                case reflect.Int:
                    var val int64 = elemValue.Int()
                    Printfln("Int: %v", val)
                case reflect.Float32, reflect.Float64:
                    var val float64 = elemValue.Float()
                    Printfln("Float: %v", val)
                case reflect.String:
                    var val string = elemValue.String()
                    Printfln("String: %v", val)
                default:
                    Printfln("Other: %v", elemValue.String())
            }
        }
    }
}

func main() {

    product := Product {
        Name: "Kayak", Category: "Watersports", Price: 279,
    }
    number := 100
    slice := []byte("Alice")
    printDetails(true, 10, 23.30, "Alice", &number, product,
slice)
}
```

Листинг 27-12 Определение срезов байтов в файле main.go в папке reflection

Скомпилируйте и выполните проект, и вы увидите следующий вывод, который включает обнаружение байтового среза:

```
Bool: true
Int: 10
Float: 23.3
String: Alice
Pointer to Int: 100
Other: <main.Product Value>
Byte slice: [65 108 105 99 101]
```

}

Получение базовых значений

Структура Value определяет методы, описанные в таблице 27-7, для получения базового значения.

Таблица 27-7 Методы Value для получения базового значения

Функция	Описание
Interface()	Этот метод возвращает базовое значение, используя пустой интерфейс. Этот метод вызывает панику, если он используется для неэкспортированных полей структуры.
CanInterface()	Этот метод возвращает значение true, если метод Interface можно использовать без паники.

Метод Interface позволяет выйти из рефлексии и получить значение, которое можно использовать в обычном коде Go, как показано в листинге 27-13.

```
package main

import (
    "reflect"
    // "strings"
    // "fmt"
)

func selectValue(data interface{}, index int) (result interface{}) {
    dataVal := reflect.ValueOf(data)
    if (dataVal.Kind() == reflect.Slice) {
        result = dataVal.Index(index).Interface()
    }
    return
}

func main() {

    names := []string {"Alice", "Bob", "Charlie"}
    val := selectValue(names, 1).(string)
    Printfln("Selected: %v", val)
}
```

Листинг 27-13 Получение базового значения в файле main.go в папке reflection

Функция selectValue выбирает значение из среза, не зная типа элемента среза. Значение извлекается из среза с помощью метода Index, описанного в главе 28. Для этой главы важно то, что метод Index возвращает Value, которое полезно только для кода, использующего отражение. Метод Interface используется для получения значения, которое можно использовать в качестве результата функции:

```
...
result = dataVal.Index(index).Interface()
...
```

Одним из недостатков использования отражения является способ обработки результатов функций и методов. Если тип результата не фиксирован, то вызывающий функцию или метод должен взять на себя ответственность за преобразование результата в определенный тип, что и делает этот оператор в листинге 27-13:

```
...
val := selectValue(names, 1).(string)
...
```

Результат функции selectValue будет иметь тот же тип, что и элементы среза, но в Go нет способа выразить это, поэтому функция использует в качестве результата пустой интерфейс, а также почему метод Interface возвращает пустой интерфейс.

Проблема в том, что вызывающий код требует понимания того, как работает функция, чтобы обработать результат. Когда поведение функции изменяется, это изменение должно быть отражено во всем коде, который вызывает функцию, что требует уровня усердия, который часто трудно поддерживать.

Это не идеально — и это одна из причин, по которой рефлексию следует использовать с осторожностью. Скомпилируйте и запустите проект, и вы увидите следующий вывод:

```
Selected: Bob
```

Установка Value с использованием рефлексии

Структура Value определяет методы, которые позволяют устанавливать значения с помощью рефлексии, как описано в таблице 27-8.

Таблица 27-8 Методы Value для установки значений

Функция	Описание
CanSet()	Этот метод возвращает true, если значение может быть установлено, и false в противном случае.
SetBool(val)	Этот метод устанавливает базовое значение в указанное логическое значение.
SetBytes(slice)	Этот метод устанавливает базовое значение для указанного байтового среза.
SetFloat(val)	Этот метод устанавливает базовое значение в указанное значение float64.
SetInt(val)	Этот метод устанавливает базовое значение в указанное значение int64.
SetUint(val)	Этот метод устанавливает базовое значение для указанного uint64.
SetString(val)	Этот метод устанавливает базовое значение в указанную строку.
Set(val)	Этот метод устанавливает базовое значение в базовое значение указанного Value.

Методы Set в таблице 27-8 вызовут панику, если результат метода CanSet окажется false или если они используются для установки значения, не относящегося к ожидаемому типу. В листинге 27-14 показана проблема, которую решает метод CanSet.

```
package main

import (
    "reflect"
    "strings"
    // "fmt"
)

func incrementOrUpper(values ...interface{}) {
    for _, elem := range values {
        elemValue := reflect.ValueOf(elem)
        if (elemValue.CanSet()) {
            switch (elemValue.Kind()) {
                case reflect.Int:
```

```
                    elemValue.SetInt(elemValue.Int() + 1)
                case reflect.String:
                        elemValue.SetString(strings.ToUpper(
elemValue.String()))
            }
            Printfln("Modified Value: %v", elemValue)
        } else {
             Printfln("Cannot set %v: %v", elemValue.Kind(),
elemValue)
        }
    }
}

func main() {

    name := "Alice"
    price := 279
    city := "London"

    incrementOrUpper(name, price, city)
     for _, val := range []interface{} { name, price, city
}  {
        Printfln("Value: %v", val)
    }
}
```

Листинг 27-14 Создание неустанавливаемых значений в файле main.go в папке reflection

Функция incrementOrUpper увеличивает значения int и преобразует строковые значения в upper регистр. Скомпилируйте и выполните код, и вы получите следующий вывод, показывающий, что ни одно из значений, полученных функцией incrementOrUpper, не может быть установлено:

```
Cannot set string: Alice
Cannot set int: 279
Cannot set string: London
Value: Alice
Value: 279
Value: London
```

Метод CanSet вызывает путаницу, но помните, что значения копируются при использовании в качестве аргументов функций и методов. Когда значения передаются в incrementOrUpper, они копируются:

```
...
incrementOrUpper(name, price, city)
...
```

Это предотвращает изменение значений, поскольку значения копируются для использования внутри функции. В листинге 27-15 проблема решается с помощью указателей.

```
package main

import (
    "reflect"
    "strings"
    // "fmt"
)

func incrementOrUpper(values ...interface{}) {
    for _, elem := range values {
        elemValue := reflect.ValueOf(elem)
        if (elemValue.Kind() == reflect.Ptr) {
            elemValue = elemValue.Elem()
        }
        if (elemValue.CanSet()) {
            switch (elemValue.Kind()) {
                case reflect.Int:
                    elemValue.SetInt(elemValue.Int() + 1)
                case reflect.String:
                         elemValue.SetString(strings.ToUpper(
elemValue.String()))
            }
            Printfln("Modified Value: %v", elemValue)
        } else {
              Printfln("Cannot set %v: %v", elemValue.Kind(),
elemValue)
        }
    }
}
```

```
func main() {

    name := "Alice"
    price := 279
    city := "London"

    incrementOrUpper(&name, &price, &city)
     for _, val := range []interface{} { name, price, city
}  {
        Println("Value: %v", val)
    }
}
```

Листинг 27-15 Установка значений в файле main.go в папке reflection

Таким образом, как и обычный код, отражение может изменить значение только в том случае, если есть доступ к исходному хранилищу. В листинге 27-15 указатели используются для вызова функции `incrementOrUpper`, и для этого требуется изменить код рефлексии для обнаружения указателей и, когда они будут найдены, использовать метод `Elem` для отслеживания указателя до его значения. Скомпилируйте и запустите проект, и вы увидите следующий вывод:

```
Modified Value: ALICE
Modified Value: 280
Modified Value: LONDON
Value: ALICE
Value: 280
Value: LONDON
```

Установка одного Value с помощью другого

Метод `Set` позволяет установить одно Value с помощью другого, что может быть удобным способом изменения значения значением, полученным путем рефлексии, как показано в листинге 27-16.

```
package main

import (
    "reflect"
    //"strings"
    // "fmt"
```

```
)

func setAll(src interface{}, targets ...interface{}) {
    srcVal := reflect.ValueOf(src)
    for _, target := range  targets {
        targetVal := reflect.ValueOf(target)
        if (targetVal.Kind() == reflect.Ptr &&
                targetVal.Elem().Type() == srcVal.Type() &&
                targetVal.Elem().CanSet()) {
            targetVal.Elem().Set(srcVal)
        }
    }
}

func main() {

    name := "Alice"
    price := 279
    city := "London"

    setAll("New String", &name, &price, &city)
    setAll(10, &name, &price, &city)
     for _, val := range []interface{} { name, price, city
}  {
        Printfln("Value: %v", val)
    }
}
```

Листинг 27-16 Установка одного Value другим в файле main.go в папке reflection

Функция `setAll` использует цикл `for` для обработки своего вариативного параметра и ищет значения, которые являются указателями на значения того же типа, что и параметр `src`. Когда соответствующий указатель найден, значение, на которое он ссылается, изменяется с помощью метода `Set`. Большая часть кода в функции `setAll` отвечает за проверку того, что значения совместимы и могут быть установлены, но в результате использование `string` в качестве первого аргумента устанавливает все последующие `string` аргументы, а использование `int` устанавливает все последующие значения int. Скомпилируйте и выполните код, и вы получите следующий вывод:

```
Value: New String
Value: 10
Value: New String
```

Сравнение Value

Не все типы данных можно сравнивать с помощью оператора сравнения Go, что позволяет легко вызвать панику в коде отражения, как показано в листинге 27-17.

```
package main

import (
    "reflect"
    //"strings"
    // "fmt"
)

func contains(slice interface{}, target interface{}) (found
bool) {
    sliceVal := reflect.ValueOf(slice)
    if (sliceVal.Kind() == reflect.Slice) {
        for i := 0; i < sliceVal.Len(); i++ {
            if sliceVal.Index(i).Interface() == target {
                found = true
            }
        }
    }
    return
}

func main() {

    // name := "Alice"
    // price := 279
    city := "London"

    citiesSlice := []string { "Paris", "Rome", "London"}
    Printfln("Found #1: %v", contains(citiesSlice, city))

    sliceOfSlices := [][]string {
        citiesSlice,  { "First", "Second", "Third"}}
```

```
        Printfln("Found #2: %v", contains(sliceOfSlices,
citiesSlice))
}
```

Листинг 27-17 Сравнение Value в файле main.go в папке reflection

Функция contains принимает срез и возвращает true, если он содержит указанное значение. Срез перечисляется с помощью методов Len и Index, которые описаны в главе 28, но для этого раздела важно следующее утверждение:

```
...
if sliceVal.Index(i).Interface() == target {
...
```

Этот оператор применяет оператор сравнения к значению по определенному индексу в срезе и к целевому значению. Но, поскольку функция contains принимает любые типы, приложение будет паниковать, если функция получит типы, которые нельзя сравнивать. Скомпилируйте и запустите проект, и вы увидите следующий вывод:

```
Found #1: true
panic: runtime error: comparing uncomparable type []string
goroutine 1 [running]:
main.contains(0x243640, 0xc000114078, 0x243f00, 0xc000153f60,
0xc000153f40)
        C:/reflection/main.go:13 +0x1a5
main.main()
        C:/reflection/main.go:33 +0x2e5
exit status 2
```

Функция main делает два вызова функции contains в листинге 27-17. Первый вызов работает, потому что срез содержит строковые значения, которые можно использовать с оператором сравнения. Второй вызов завершается ошибкой, так как слайс содержит другие слайсы, к которым нельзя применить оператор сравнения. Чтобы избежать этой проблемы, интерфейс Type определяет метод, описанный в таблице 27-9.

Таблица 27-9 Метод Type для определения возможности сравнения типов

Функция	Описание
`Comparable()`	Этот метод возвращает `true`, если отраженный тип можно использовать с оператором сравнения Go, и `false` в противном случае.

В листинге 27-18 показано использование метода `Comparable` во избежание выполнения сравнений, которые могут вызвать панику.

```
...
func contains(slice interface{}, target interface{}) (found
bool) {
    sliceVal := reflect.ValueOf(slice)
    targetType := reflect.TypeOf(target)
    if (sliceVal.Kind() == reflect.Slice &&
            sliceVal.Type().Elem().Comparable() &&
            targetType.Comparable()) {
        for i := 0; i < sliceVal.Len(); i++ {
            if sliceVal.Index(i).Interface() == target {
                found = true
            }
        }
    }
    return
}
...
```

Листинг 27-18 Безопасное Value значений в файле main.go в папке reflection

Эти изменения гарантируют, что оператор сравнения применяется только к значениям, типы которых сопоставимы. Скомпилируйте и запустите проект, и вы увидите следующий вывод:

```
Found #1: true
Found #2: false
```

Использование удобной функции сравнения

Пакет `reflect` определяет функцию, которая представляет собой альтернативу стандартному оператору сравнения Go, как описано в таблице 27-10.

Таблица 27-10 Функция пакета reflect для сравнения значений

Функция	Описание

Функция	Описание
DeepEqual(val, val)	Эта функция сравнивает любые два значения и возвращает true, если они совпадают

Функция DeepEqual не паникует и выполняет дополнительные сравнения, которые невозможны при использовании оператора ==. Все правила сравнения для этой функции перечислены по адресу https://pkg.go.dev/reflect@go1.17.1#DeepEqual, но в целом функция DeepEqual выполняет сравнение, рекурсивно проверяя все поля или элементы значения. Одним из наиболее полезных аспектов этого типа сравнения является то, что срезы равны, если все их значения равны, что устраняет одно из наиболее часто встречающихся ограничений стандартного оператора сравнения, как показано в листинге 27-19.

```
package main

import (
    "reflect"
    //"strings"
    // "fmt"
)

func contains(slice interface{}, target interface{}) (found
bool) {
    sliceVal := reflect.ValueOf(slice)
    if (sliceVal.Kind() == reflect.Slice) {
        for i := 0; i < sliceVal.Len(); i++ {
                                                              if
reflect.DeepEqual(sliceVal.Index(i).Interface(), target) {
                found = true
            }
        }
    }
    return
}

func main() {

    // name := "Alice"
    // price := 279
```

```
    city := "London"

    citiesSlice := []string { "Paris", "Rome", "London"}
    Printfln("Found #1: %v", contains(citiesSlice, city))

    sliceOfSlices := [][]string {
        citiesSlice,  { "First", "Second", "Third"}}

        Printfln("Found  #2:  %v",  contains(sliceOfSlices,
citiesSlice))
}
```

Листинг 27-19 Выполнение сравнения в файле main.go в папке reflection

Это упрощение функции `contains` не требует проверки сопоставимости типов и дает следующий результат при компиляции и выполнении проекта:

```
Found #1: true
Found #2: true
```

В этом примере можно сравнить срезы, чтобы оба вызова функции `contains` давали `true` результаты.

Преобразование значений

Как объяснялось в третьей части, Go поддерживает преобразование типов, позволяя значениям, определенным как один тип, быть представленным с использованием другого типа. Интерфейс `Type` определяет метод, описанный в таблице 27-11, для определения возможности преобразования отраженного типа.

Таблица 27-11 Метод Type для оценки преобразования типов

Функция	**Описание**
`ConvertibleTo(type)`	Этот метод возвращает значение `true`, если тип, для которого вызывается метод, может быть преобразован в указанный `Type`.

Метод, определенный интерфейсом `Type`, позволяет проверять типы на конвертируемость. Таблица 27-12 описывает метод, определенный структурой `Value`, которая выполняет преобразование.

Таблица 27-12 Метод Value для преобразования типов

Функция	Описание
`Convert(type)`	Этот метод выполняет преобразование типа и возвращает `Value` с новым типом и исходным значением.

В листинге 27-20 показано простое преобразование типов, выполненное с использованием отражения.

```
package main

import (
    "reflect"
    //"strings"
    // "fmt"
)

func convert(src, target interface{}) (result interface{},
assigned bool) {
    srcVal := reflect.ValueOf(src)
    targetVal := reflect.ValueOf(target)
    if (srcVal.Type().ConvertibleTo(targetVal.Type())) {
        result = srcVal.Convert(targetVal.Type()).Interface()
        assigned = true
    } else {
        result = src
    }
    return
}

func main() {

    name := "Alice"
    price := 279
    //city := "London"

    newVal, ok := convert(price, 100.00)
    Printfln("Converted %v: %v, %T", ok, newVal, newVal)
    newVal, ok = convert(name, 100.00)
    Printfln("Converted %v: %v, %T", ok, newVal, newVal)
}
```

Листинг 27-20 Выполнение преобразования типа в файле main.go в папке reflection Folder

Функция convert пытается преобразовать одно значение в тип другого значения, что она и делает с помощью методов ConvertibleTo и Convert. Первый вызов функции convert пытается преобразовать значение int в float64, что удается, а второй вызов пытается преобразовать string в float64, что не удается. Скомпилируйте и запустите проект, и вы увидите следующий вывод:

```
Converted true: 279, float64
Converted false: Alice, string
```

Преобразование числовых типов

Структура Value определяет методы, показанные в таблице 27-13, для проверки того, вызовет ли значение переполнение при выражении в целевом типе. Эти методы полезны при преобразовании из одного числового типа в другой.

Таблица 27-13 Методы Value для проверки переполнения

Функция	Описание
OverflowFloat(val)	Этот метод возвращает значение true, если указанное значение float64 вызовет переполнение при преобразовании в тип Value, для которого вызывается метод. Этот метод вызовет панику, если только метод Value.Kind не вернет Float32 или Float64.
OverflowInt(val)	Этот метод возвращает значение true, если указанное значение int64 вызовет переполнение при преобразовании в тип Value, для которого вызывается метод. Этот метод вызовет панику, если только метод Value.Kind не вернет один из целочисленных типов со знаком.
OverflowUint(val)	Этот метод возвращает значение true, если указанное значение uint64 вызовет переполнение при преобразовании в тип Value, для которого вызывается метод. Этот метод вызовет панику, если только метод Value.Kind не вернет один из целочисленных типов без знака.

Как объяснялось в главе 5, числовые значения в Go переносятся при переполнении. Методы, описанные в таблице 27-13, можно использовать для определения того, когда преобразование вызовет переполнение, как показано в листинге 27-21, что может привести к неожиданному результату.

```
package main

import (
	"reflect"
	//"strings"
	// "fmt"
)

func IsInt(v reflect.Value) bool {
	switch v.Kind() {
		case reflect.Int, reflect.Int8, reflect.Int16,
reflect.Int32, reflect.Int64:
		return true
	}
	return false
}

func IsFloat(v reflect.Value) bool {
	switch v.Kind() {
	case reflect.Float32, reflect.Float64:
		return true
	}
	return false
}

func convert(src, target interface{}) (result interface{},
assigned bool) {
	srcVal := reflect.ValueOf(src)
	targetVal := reflect.ValueOf(target)
	if (srcVal.Type().ConvertibleTo(targetVal.Type())) {
		if (IsInt(targetVal) && IsInt(srcVal)) &&
				targetVal.OverflowInt(srcVal.Int()) {
			Printfln("Int overflow")
			return src, false
		} else if (IsFloat(targetVal) && IsFloat(srcVal) &&
				targetVal.OverflowFloat(srcVal.Float())) {
			Printfln("Float overflow")
			return src, false
		}
		result = srcVal.Convert(targetVal.Type()).Interface()
		assigned = true
	} else {
		result = src
```

```
    }
    return
}

func main() {

    name := "Alice"
    price := 279
    //city := "London"

    newVal, ok := convert(price, 100.00)
    Printfln("Converted %v: %v, %T", ok, newVal, newVal)
    newVal, ok = convert(name, 100.00)
    Printfln("Converted %v: %v, %T", ok, newVal, newVal)

    newVal, ok = convert(5000, int8(100))
    Printfln("Converted %v: %v, %T", ok, newVal, newVal)
}
```

Листинг 27-21 Предотвращение переполнения в файле main.go в папке reflection

Новый код в листинге 27-21 добавляет защиту от переполнения при преобразовании из одного целочисленного типа в другой и из одного значения с плавающей запятой в другое. Скомпилируйте и запустите проект, и вы увидите следующий вывод:

```
Converted true: 279, float64
Converted false: Alice, string
Int overflow
Converted false: 5000, int
```

Последний вызов функции `convert` в листинге 27-21 пытается преобразовать значение `5000` в `int8`, что вызовет переполнение. Метод `OverflowInt` возвращает значение `true`, поэтому преобразование не выполняется.

Создание новых значений

Пакет `reflect` определяет функции для создания новых значений, которые описаны в таблице 27-14. Я продемонстрирую функции,

характерные для определенных структур данных, таких как срезы и карты, в последующих главах.

Таблица 27-14 Функции для создания новых значений

Функция	Описание
New(type)	Эта функция создает Value, указывающее на значение указанного типа, инициализированное нулевым значением типа.
Zero(type)	Эта функция создает Value, представляющее нулевое значение указанного типа.
MakeMap(type)	Эта функция создает новую карту, как описано в главе 28.
MakeMapWithSize(type, size)	Эта функция создает новую карту заданного размера, как описано в главе 28.
MakeSlice(type, capacity)	Эта функция создает новый срез, как описано в главе 28.
MakeFunc(type, args, results)	Эта функция создает новую функцию с указанными аргументами и результатами, как описано в главе 29.
MakeChan(type, buffer)	Эта функция создает новый канал с указанным размером буфера, как описано в главе 29.

Следует соблюдать осторожность с функцией New, поскольку она возвращает указатель на новое значение указанного типа, а это означает, что легко создать указатель на указатель. В листинге 27-22 функция New используется для создания временного значения в функции, которая меняет местами свои параметры.

```
package main

import (
    "reflect"
    //"strings"
    // "fmt"
)

func swap(first interface{}, second interface{}) {
        firstValue, secondValue := reflect.ValueOf(first),
reflect.ValueOf(second)
    if firstValue.Type() == secondValue.Type() &&
            firstValue.Kind() == reflect.Ptr &&
                                 firstValue.Elem().CanSet() &&
secondValue.Elem().CanSet() {
```

```
        temp :=  reflect.New(firstValue.Elem().Type())
        temp.Elem().Set(firstValue.Elem())
        firstValue.Elem().Set(secondValue.Elem())
        secondValue.Elem().Set(temp.Elem())
    }
}

func main() {

    name := "Alice"
    price := 279
    city := "London"

    swap(&name, &city)
     for _, val := range []interface{} { name, price, city
} {
        Printfln("Value: %v", val)
    }
}
```

Листинг 27-22 Создание значения в файле main.go в папке reflection

Для выполнения свопа требуется новое значение, которое создается с помощью функции New:

```
...
temp :=  reflect.New(firstValue.Elem().Type())
...
```

Type, передаваемый функции New, получается из результата Elem для одного из значений параметра, что позволяет избежать создания указателя на указатель. Метод Set используется для установки временного значения и выполнения свопа. Скомпилируйте и выполните проект, и вы получите следующий вывод, показывающий, что значения переменных name и city поменялись местами:

```
Value: London
Value: 279
Value: Alice
```

Резюме

В этой главе я представил основные функции отражения в Go и продемонстрировал их использование. Я объяснил, как получить отраженные типы и значения, как определить тип отраженного типа, как установить отраженное значение и как использовать удобные функции, предоставляемые пакетом `reflect`. В следующей главе я продолжу описывать отражение и покажу вам, как работать с указателями, срезами, картами и структурами.

28. Использование рефлексии, часть 2

В дополнение к основным функциям, описанным в предыдущей главе, пакет `reflect` предоставляет дополнительные возможности, полезные при работе с определенными типами, такими как карты или структуры. В следующих разделах я опишу эти функции и продемонстрирую их использование. Некоторые из описанных методов и функций используются более чем с одним типом, и я перечислил их несколько раз для быстрого ознакомления. Таблица 28-1 суммирует содержание главы.

Таблица 28-1 Краткое содержание главы

Проблема	**Решение**	**Листинг**
Создать или следовать типу указателя	Используйте методы `PtrTo` и `Elem`	3
Создание или отслеживание значения указателя	Используйте методы `Addr` и `Elem`	4
Проверить или создать срез	Используйте методы `Type` и `Value` для срезов	5–8
Создание, копирование и добавление к срезу	Используйте функции `reflect` для срезов	9
Проверить или создать карту	Используйте методы `Type` и `Value` для карт	10–14
Проверить или создать структуру	Используйте функции `reflect` для структур	15–17, 19–21
Проверить теги структуры	Используйте методы, определенные `StructTag`	18

Подготовка к этой главе

В этой главе я продолжаю использовать проект отражения, созданный в главе 27. Чтобы подготовиться к этой главе, добавьте тип, показанный в листинге 28-1, в файл `types.go` в папке `reflection`.

> **Подсказка**
> Вы можете загрузить пример проекта для этой главы — и для всех остальных глав этой книги — с `https://github.com/apress/pro-go`. См. Главу 2 о том, как получить помощь, если у вас возникнут проблемы с запуском примеров.

```
package main

type Product struct {
    Name, Category string
    Price float64
}

type Customer struct {
    Name, City string
}

type Purchase struct {
    Customer
    Product
    Total float64
    taxRate float64
}
```

Листинг 28-1 Определение типа в файле types.go в папке reflection

Запустите команду, показанную в листинге 28-2, в папке `reflection`, чтобы скомпилировать и выполнить проект.

```
go run .
```

Листинг 28-2 Компиляция и выполнение проекта

Эта команда выдаст следующий вывод:

```
Value: London
Value: 279
Value: Alice
```

Работа с указателями

Пакет `reflect` предоставляет функцию и метод, показанные в таблице 28-2, для работы с типами указателей.

Таблица 28-2 Функция пакета reflect и метод для указателейй

Функция	Описание
`PtrTo(type)`	Эта функция возвращает `Type`, который является указателем на `Type`, полученный в качестве аргумента.
`Elem()`	Этот метод, который вызывается для типа указателя, возвращает базовый `Type`. Этот метод вызывает панику при использовании для типов, не являющихся указателями.

Функция PtrTo создает тип указателя, а метод Elem возвращает тип, на который указывает указатель, как показано в листинге 28-3.

```
package main

import (
    "reflect"
    //"strings"
    // "fmt"
)

func createPointerType(t reflect.Type) reflect.Type {
    return reflect.PtrTo(t)
}

func followPointerType(t reflect.Type) reflect.Type {
    if t.Kind() == reflect.Ptr {
        return t.Elem()
    }
    return t
}

func main() {

    name := "Alice"

    t := reflect.TypeOf(name)
    Printfln("Original Type: %v", t)
    pt := createPointerType(t)
    Printfln("Pointer Type: %v", pt)
    Printfln("Follow pointer type: %v", followPointerType(pt))
}
```

Листинг 28-3 Работа с типами указателей в файле main.go в папке reflection

Функция PtrTo экспортируется из пакета reflect. Его можно вызывать для любого типа, включая типы указателей, и в результате получается тип, указывающий на исходный тип, так что string тип создает тип *string, а *string — **string.

Elem, который является методом, определенным интерфейсом Type, может использоваться только для типов указателей, поэтому функция followPointerType в листинге 28-3 проверяет результат метода Kind перед вызовом метода Elem. Скомпилируйте и запустите проект, и вы увидите следующий вывод:

```
Original Type: string
Pointer Type: *string
Follow pointer type: string
```

Работа со значениями указателя

Структура Value определяет методы, показанные в таблице 28-3, для работы со значениями указателя, в отличие от типов, описанных в предыдущем разделе.

Таблица 28-3 Методы Value для работы с типами указателей

Функция	Описание
Addr()	Этот метод возвращает Value, которое является указателем на Value, для которого он вызывается. Этот метод вызывает панику, если метод CanAddr возвращает значение false.
CanAddr()	Этот метод возвращает значение true, если значение можно использовать с методом Addr.
Elem()	Этот метод следует за указателем и возвращает его Value. Этот метод вызывает панику, если он вызывается для значения, не являющегося указателем.

Метод Elem используется для отслеживания указателя для получения его базового значения, как показано в листинге 28-4. Другие методы наиболее полезны при работе с полями структуры, как описано в разделе «Установка значений поля структуры».

```
package main

import (
    "reflect"
    "strings"
    // "fmt"
)

var stringPtrType = reflect.TypeOf((*string)(nil))

func transformString(val interface{}) {
    elemValue := reflect.ValueOf(val)
    if (elemValue.Type() == stringPtrType) {
        upperStr := strings.ToUpper(elemValue.Elem().String())
        if (elemValue.Elem().CanSet()) {
            elemValue.Elem().SetString(upperStr)
        }
    }
}
```

```
func main() {

    name := "Alice"

    transformString(&name)
    Printfln("Follow pointer value: %v", name)
}
```

Листинг 28-4 Следование указателю в файле main.go в папке reflection

Функция `transformString` идентифицирует `*string` значения и использует метод `Elem` для получения `string` значения, чтобы его можно было передать в функцию `strings.ToUpper`. Скомпилируйте и запустите проект, и вы получите следующий вывод:

```
Follow pointer value: ALICE
```

Работа с типами массивов и срезов

Структура `Type` определяет методы, которые можно использовать для проверки типов массивов и срезов, описанных в таблице 28-4.

Таблица 28-4 Методы Type для массивов и срезов

Функция	Описание
`Elem()`	Этот метод возвращает `Type` для элементов массива или среза.
`Len()`	Этот метод возвращает длину для типа массива. Этот метод вызовет панику, если будет вызван для других типов, включая срезы.

В дополнение к этим методам пакет `reflect` предоставляет функции, описанные в таблице 28-5, для создания типов массивов и срезов.

Таблица 28-5 Функции reflect для создания массивов и типов срезов

Функция	Описание
`ArrayOf(len, type)`	Эта функция возвращает `Type`, описывающий массив с указанным размером и типом элемента.
`SliceOf(type)`	Эта функция возвращает `Type`, описывающий массив с указанным типом элемента.

В листинге 28-5 используется метод `Elem` для проверки типа массивов и срезов.

```
package main
```

```
import (
    "reflect"
    //"strings"
    // "fmt"
)

func checkElemType(val interface{}, arrOrSlice interface{}) bool {
    elemType := reflect.TypeOf(val)
    arrOrSliceType := reflect.TypeOf(arrOrSlice)
    return (arrOrSliceType.Kind() == reflect.Array ||
        arrOrSliceType.Kind() == reflect.Slice) &&
         arrOrSliceType.Elem() == elemType
}

func main() {

    name := "Alice"
    city := "London"
    hobby := "Running"

    slice := []string { name, city, hobby }
    array := [3]string { name, city, hobby}

      Printfln("Slice (string): %v",  checkElemType("testString",
slice))
      Printfln("Array (string): %v",  checkElemType("testString",
array))
    Printfln("Array (int): %v",  checkElemType(10, array))
}
```

Листинг 28-5 Проверка типов массивов и срезов в файле main.go в папке reflection

checkElemType использует метод Kind для идентификации массивов и срезов и использует метод Elem для получения Type элементов. Они сравниваются с типом первого параметра, чтобы увидеть, можно ли добавить значение в качестве элемента. Скомпилируйте и запустите проект, и вы увидите следующий результат:

```
Slice (string): true
Array (string): true
Array (int): false
```

Работа со значениями массива и среза

Интерфейс Value определяет методы, описанные в таблице 28-6, для работы со значениями массива и среза.

Таблица 28-6 Value методы работы с массивами и срезами

Функция	Описание
Index(index)	Этот метод возвращает Value, представляющее элемент по указанному индексу.
Len()	Этот метод возвращает массив или длину среза.
Cap()	Этот метод возвращает емкость массива или среза.
SetLen()	Этот метод устанавливает длину среза. Его нельзя использовать в массивах.
SetCap()	Этот метод устанавливает емкость среза. Его нельзя использовать в массивах.
Slice(lo, hi)	Этот метод создает новый срез с указанными нижним и верхним значениями.
Slice3(lo, hi, max)	Этот метод создает новый срез с указанными минимальными, высокими и максимальными значениями.

Метод Index возвращает Value, которое можно использовать с методом Set, описанным в главе 27, для изменения значения в срезе или массиве, как показано в листинге 28-6.

```
package main

import (
    "reflect"
    //"strings"
    // "fmt"
)

func setValue(arrayOrSlice interface{}, index int, replacement
interface{}) {
    arrayOrSliceVal := reflect.ValueOf(arrayOrSlice)
    replacementVal := reflect.ValueOf(replacement)
    if (arrayOrSliceVal.Kind() == reflect.Slice) {
        elemVal := arrayOrSliceVal.Index(index)
        if (elemVal.CanSet()) {
            elemVal.Set(replacementVal)
        }
    } else if (arrayOrSliceVal.Kind() == reflect.Ptr &&
        arrayOrSliceVal.Elem().Kind() == reflect.Array &&
        arrayOrSliceVal.Elem().CanSet()) {
            arrayOrSliceVal.Elem().Index(index).Set(replacementVal)
    }
}

func main() {
```

```
    name := "Alice"
    city := "London"
    hobby := "Running"

    slice := []string { name, city, hobby }
    array := [3]string { name, city, hobby}

    Printfln("Original slice: %v", slice)
    newCity := "Paris"
    setValue(slice, 1, newCity)
    Printfln("Modified slice: %v", slice)

    Printfln("Original slice: %v", array)
    newCity = "Rome"
    setValue(&array, 1, newCity)
    Printfln("Modified slice: %v", array)
}
```

Листинг 28-6 Изменение элемента среза в файле main.go в папке reflection

Функция setValue изменяет значение элемента в срезе или массиве, но каждый тип должен обрабатываться по-разному. Со срезами проще всего работать, и их можно передавать как значения, например:

```
...
setValue(slice, 1, newCity)
...
```

Как я объяснял в главе 7, срезы являются ссылками и не копируются, когда используются в качестве аргументов функции. В листинге 28-6 метод setValue использует метод Kind для обнаружения среза, использует метод Index для получения Value элемента в указанном месте и использует метод Set для изменения значения элемента. Массивы должны передаваться как указатели, например:

```
...
setValue(&array, 1, newCity)
...
```

Если вы не используете указатель, то вы не сможете установить новые значения, а метод CanSet вернет false. Метод Kind используется для обнаружения указателя, а метод Elem используется для подтверждения того, что он указывает на массив:

```
...
```

```
} else if (arrayOrSliceVal.Kind() == reflect.Ptr &&
    arrayOrSliceVal.Elem().Kind() == reflect.Array &&
    arrayOrSliceVal.Elem().CanSet()) {
...
```

Чтобы установить значение элемента, за указателем следует метод `Elem` для получения отраженного `Value`, метод `Index` используется для получения `Value` элемента по указанному индексу, а метод `Set` используется для присвоения нового значения:

```
...
arrayOrSliceVal.Elem().Index(index).Set(replacementVal)
...
```

Общий эффект заключается в том, что функция `setValue` может манипулировать срезами и массивами, не зная, какие конкретные типы используются. Скомпилируйте и запустите проект, и вы увидите следующий вывод:

```
Original slice: [Alice London Running]
Modified slice: [Alice Paris Running]
Original slice: [Alice London Running]
Modified slice: [Alice Rome Running]
```

Перечисление срезов и массивов

Метод `Len` можно использовать для установки предела в цикле `for` для перечисления элементов в массиве или срезе, как показано в листинге 28-7.

```
package main

import (
    "reflect"
    //"strings"
    // "fmt"
)

func enumerateStrings(arrayOrSlice interface{}) {
    arrayOrSliceVal := reflect.ValueOf(arrayOrSlice)
    if (arrayOrSliceVal.Kind() == reflect.Array ||
            arrayOrSliceVal.Kind() == reflect.Slice) &&
            arrayOrSliceVal.Type().Elem().Kind() == reflect.String
{
        for i := 0; i < arrayOrSliceVal.Len(); i++ {
```

```
                         Printfln("Element:  %v,  Value:  %v",  i,
arrayOrSliceVal.Index(i).String())
        }
    }
}

func main() {

    name := "Alice"
    city := "London"
    hobby := "Running"

    slice := []string { name, city, hobby }
    array := [3]string { name, city, hobby}

    enumerateStrings(slice)
    enumerateStrings(array)
}
```

Листинг 28-7 Перечисление массивов и срезов в файле main.go в папке reflection

Функция `enumerateStrings` проверяет результат `Kind`, чтобы убедиться, что он имеет дело с массивом или срезом строк. Легко запутаться в том, какой метод `Elem` используется в этом процессе, потому что `Type` и `Value` определяют методы `Kind` и `Elem`. Методы `Kind` выполняют ту же задачу, но вызов метода `Elem` для среза или массива `Value` вызывает панику, а вызов метода `Elem` для среза или массива `Type` возвращает `Type` элементов:

```
...
arrayOrSliceVal.Type().Elem().Kind() == reflect.String {
...
```

Как только функция подтвердит, что имеет дело с массивом или срезом строк, используется цикл `for` с пределом, установленным результатом метода `Len`:

```
...
for i := 0; i < arrayOrSliceVal.Len(); i++ {
...
```

Метод `Index` используется в цикле `for` для получения элемента с текущим индексом, а его значение получается с помощью метода `String`:

```
...
```

```
Printfln("Element:        %v,      Value:       %v",       i,
arrayOrSliceVal.Index(i).String())
...
```

Обратите внимание, что на массив не нужно ссылаться с помощью указателя при перечислении его содержимого. Это требование только при внесении изменений. Скомпилируйте и выполните проект, и вы увидите следующий вывод, который представляет собой перечисление среза и массива:

```
Element: 0, Value: Alice
Element: 1, Value: London
Element: 2, Value: Running
Element: 0, Value: Alice
Element: 1, Value: London
Element: 2, Value: Running
```

Создание новых срезов из существующих срезов

Метод `Slice` используется для создания одного среза из другого, как показано в листинге 28-8.

```
package main

import (
    "reflect"
    //"strings"
    // "fmt"
)

func    findAndSplit(slice    interface{},    target    interface{})
interface{} {
    sliceVal := reflect.ValueOf(slice)
    targetType := reflect.TypeOf(target)
    if (sliceVal.Kind() == reflect.Slice && sliceVal.Type().Elem()
== targetType) {
        for i := 0; i < sliceVal.Len(); i++ {
            if sliceVal.Index(i).Interface() == target {
                return sliceVal.Slice(0, i +1)
            }
        }
    }
    return slice
}

func main() {
```

```
    name := "Alice"
    city := "London"
    hobby := "Running"

    slice := []string { name, city, hobby }
    //array := [3]string { name, city, hobby}
    Printfln("Strings: %v", findAndSplit(slice, "London"))

    numbers := []int {1, 3, 4, 5, 7}
    Printfln("Numbers: %v", findAndSplit(numbers, 4))
}
```

Листинг 28-8 Создание нового среза в файле main.go в папке reflection

Функция findAndSplit перечисляет срез, ища указанный элемент, что выполняется с использованием метода Interface, который позволяет сравнивать элементы среза без необходимости иметь дело с конкретными типами. Как только целевой элемент найден, метод Slice используется для создания и возврата нового среза. Скомпилируйте и запустите проект, и вы увидите следующий вывод:

```
Strings: [Alice London]
Numbers: [1 3 4]
```

Создание, копирование и добавление элементов в срезы

В пакете reflect определены функции, описанные в таблице 28-7, которые позволяют копировать значения и добавлять их к срезам без необходимости работы с базовыми типами.

Таблица 28-7 Функции добавления элементов к срезам

Функция	Описание
MakeSlice(type, len, cap)	Эта функция создает Value, отражающее новый срез, используя Type для обозначения типа элемента с заданной длиной и емкостью.
Append(sliceVal, ...val)	Эта функция добавляет к указанному срезу одно или несколько значений, все из которых выражаются с помощью интерфейса Value. Результатом является измененный срез. Функция вызывает панику, когда используется для любого типа, отличного от среза, или если типы значений не соответствуют типу элемента среза.
AppendSlice(sliceVal, sliceVal)	Эта функция добавляет один срез к другому. Функция паникует, если либо Value не представляет срез, либо если типы срезов несовместимы.
Copy(dst, src)	Эта функция копирует элементы из среза или массива, отраженного src Value, в срез или массив, отраженный dst Value. Элементы копируются до тех пор, пока целевой срез не будет заполнен или пока не будут скопированы все исходные элементы. Источник и место назначения должны иметь один и тот же тип элемента.

Эти функции принимают аргументы Type или Value, которые могут противоречить интуиции и требуют подготовки. Функция MakeSlice принимает аргумент Type, указывающий тип среза, и возвращает Value, отражающее новый срез. Другой оператор функций для аргументов Value, как показано в листинге 28-9.

```
package main

import (
    "reflect"
    //"strings"
    //"fmt"
)

func pickValues(slice interface{}, indices ...int) interface{} {
    sliceVal := reflect.ValueOf(slice)
    if (sliceVal.Kind() == reflect.Slice) {
        newSlice := reflect.MakeSlice(sliceVal.Type(), 0, 10)
        for _, index := range indices {
                        newSlice  =  reflect.Append(newSlice,
sliceVal.Index(index))
        }
        return newSlice
    }
    return nil
}

func main() {

    name := "Alice"
    city := "London"
    hobby := "Running"

       slice  :=  []string  {  name,  city,  hobby,  "Bob",  "Paris",
"Soccer" }
    picked := pickValues(slice, 0, 3, 5)
    Printfln("Picked values: %v", picked)
}
```

Листинг 28-9 Создание нового среза в файле main.go в папке reflection

Функция pickValues создает новый срез, используя Type, отраженный от существующего среза, и использует функцию Append для добавления значений в новый срез. Скомпилируйте и запустите проект, и вы увидите следующий вывод:

```
Picked values: [Alice Bob Soccer]
```

Работа с типами карт

Структура `Type` определяет методы, которые можно использовать для проверки типов карт, описанных в таблице 28-8.

Таблица 28-8 Методы Type для карт

Функция	Описание
`Key()`	Этот метод возвращает `Type` для ключей карты.
`Elem()`	Этот метод возвращает `Type` для значений карты.

В дополнение к этим методам пакет `reflect` предоставляет функцию, описанную в таблице 28-9, для создания типов карт.

Таблица 28-9 Функции reflect для создания типов карт

Функция	Описание
`MapOf(keyType, valType)`	Эта функция возвращает новый `Type`, который отражает тип карты с указанными типами ключа и значения, оба из которых описаны с использованием `Type`.

В листинге 28-10 определена функция, которая получает карту и сообщает о ее типах.

> **Примечание**
> Описать отражение для карт сложно, поскольку термин *значение* используется для обозначения пар ключ-значение, содержащихся в карте, а также отраженных значений, представленных интерфейсом `Value`. Я старался быть последовательным, но вы можете обнаружить, что вам придется прочитать некоторые части этого раздела несколько раз, прежде чем они обретут смысл.

```
package main

import (
    "reflect"
    //"strings"
    //"fmt"
)

func describeMap(m interface{}) {
    mapType := reflect.TypeOf(m)
```

```
    if (mapType.Kind() == reflect.Map) {
           Printfln("Key type: %v, Val type: %v", mapType.Key(),
mapType.Elem())
    } else {
        Printfln("Not a map")
    }
}

func main() {

    pricesMap := map[string]float64 {
        "Kayak": 279, "Lifejacket": 48.95, "Soccer Ball": 19.50,
    }
    describeMap(pricesMap)
}
```

Листинг 28-10 Работа с типом карты в файле main.go в папке reflection

Метод `Kind` используется для подтверждения того, что функция `descriptionMap` получила карту, а методы `Key` и `Elem` используются для записи типов ключа и значения. Скомпилируйте и запустите проект, и вы увидите следующий вывод:

```
Key type: string, Val type: float64
```

Работа со значениями карты

Интерфейс `Value` определяет методы, описанные в таблице 28-10, для работы со значениями карты.

Таблица 28-10 Методы Value для работы с картами

Функция	Описание
`MapKeys()`	Этот метод возвращает значение `[]Value`, содержащее ключи карты.
`MapIndex(key)`	Этот метод возвращает `Value`, соответствующее указанному ключу, которое также выражается как `Value`. Нулевое значение возвращается, если карта не содержит указанного ключа, что можно обнаружить, вызвав метод `IsValid`, который вернет `false`, как описано в главе 27.
`MapRange()`	Этот метод возвращает `*MapIter`, который позволяет повторять содержимое карты, как описано после таблицы.
`SetMapIndex(key, val)`	Этот метод устанавливает указанный ключ и значение, оба из которых выражаются с использованием интерфейса `Value`.
`Len()`	Этот метод возвращает количество пар ключ-значение, содержащихся в карте.

Пакет reflect предоставляет два разных способа перечисления содержимого карты. Первый заключается в использовании метода MapKeys для получения среза, содержащего отраженные значения ключей, и получения каждого отраженного значения карты с помощью метода MapIndex, как показано в листинге 28-11.

```
package main

import (
    "reflect"
    //"strings"
    //"fmt"
)

func printMapContents(m interface{}) {
    mapValue := reflect.ValueOf(m)
    if (mapValue.Kind() == reflect.Map) {
        for _, keyVal := range mapValue.MapKeys() {
            reflectedVal := mapValue.MapIndex(keyVal)
                    Printfln("Map Key: %v, Value: %v", keyVal, 
reflectedVal)
        }
    } else {
        Printfln("Not a map")
    }
}

func main() {

    pricesMap := map[string]float64 {
        "Kayak": 279, "Lifejacket": 48.95, "Soccer Ball": 19.50,
    }
    printMapContents(pricesMap)
}
```

Листинг 28-11 Итерация содержимого карты в файле main.go в папке reflection

Тот же эффект может быть достигнут с помощью метода MapRange, который возвращает указатель на значение MapIter, которое определяет методы, описанные в таблице 28-11.

Таблица 28-11 Методы, определенные структурой MapIter

Функция	Описание

Функция	Описание
Next()	Этот метод переходит к следующей паре ключ-значение на карте. Результатом этого метода является логическое значение, указывающее, есть ли еще пары ключ-значение для чтения. Этот метод должен вызываться перед методом Key или Value.
Key()	Этот метод возвращает Value, представляющее ключ карты в текущей позиции.
Value()	Этот метод возвращает Value, представляющее значение карты в текущей позиции.

Структура MapIter обеспечивает основанный на курсоре подход к перечислению карт, где метод Next перемещается по содержимому карты, а методы Key и Value обеспечивают доступ к ключу и значению в текущей позиции. Результат метода Next указывает, есть ли оставшиеся значения для чтения, что делает его удобным для использования с циклом for, как показано в листинге 28-12.

```
package main

import (
    "reflect"
    //"strings"
    //"fmt"
)

func printMapContents(m interface{}) {
    mapValue := reflect.ValueOf(m)
    if (mapValue.Kind() == reflect.Map) {
        iter := mapValue.MapRange()
        for iter.Next() {
                 Printfln("Map Key: %v, Value: %v", iter.Key(),
iter.Value())
        }
    } else {
        Printfln("Not a map")
    }
}

func main() {

    pricesMap := map[string]float64 {
        "Kayak": 279, "Lifejacket": 48.95, "Soccer Ball": 19.50,
    }
    printMapContents(pricesMap)
}
```

Листинг 28-12 Использование MapIter в файле main.go в папке reflection

Важно вызывать метод Next перед вызовом методов Key и Value и избегать вызова этих методов, когда метод Next возвращает значение false. Листинг 28-11 и Листинг 28-12 производят следующий вывод при компиляции и выполнении:

```
Map Key: Kayak, Value: 279
Map Key: Lifejacket, Value: 48.95
Map Key: Soccer Ball, Value: 19.5
```

Установка и удаление значений карты

Метод SetMapIndex используется для добавления, изменения или удаления пар ключ-значение на карте. В листинге 28-13 определены функции для изменения карты.

```
package main

import (
    "reflect"
    //"strings"
    //"fmt"
)

func setMap(m interface{}, key interface{}, val interface{}) {
    mapValue := reflect.ValueOf(m)
    keyValue := reflect.ValueOf(key)
    valValue := reflect.ValueOf(val)
    if (mapValue.Kind() == reflect.Map &&
        mapValue.Type().Key() == keyValue.Type() &&
        mapValue.Type().Elem() == valValue.Type()) {
            mapValue.SetMapIndex(keyValue, valValue)
    } else {
        Printfln("Not a map or mismatched types")
    }
}

func removeFromMap(m interface{}, key interface{}) {
    mapValue := reflect.ValueOf(m)
    keyValue := reflect.ValueOf(key)
    if (mapValue.Kind() == reflect.Map &&
        mapValue.Type().Key() == keyValue.Type()) {
            mapValue.SetMapIndex(keyValue, reflect.Value{})
    }
}

func main() {
```

```
    pricesMap := map[string]float64 {
        "Kayak": 279, "Lifejacket": 48.95, "Soccer Ball": 19.50,
    }
    setMap(pricesMap, "Kayak", 100.00)
    setMap(pricesMap, "Hat", 10.00)
    removeFromMap(pricesMap, "Lifejacket")
    for k, v := range pricesMap {
        Printfln("Key: %v, Value: %v", k, v)
    }
}
```

Листинг 28-13 Изменение карты в файле main.go в папке reflection

Как отмечалось в главе 7, карты не копируются, когда они используются в качестве аргументов, поэтому указатель не требуется для изменения содержимого карты. Функция `setMap` проверяет полученные значения, чтобы подтвердить, что она получила карту и что параметры ключа и значения имеют ожидаемые типы, прежде чем устанавливать значение с помощью метода `SetMapIndex`.

Метод SetMapIndex удалит ключ из карты, если аргумент значения является нулевым значением для типа значения карты. Это проблема при работе со встроенными типами, такими как `int` и `float64`, где нулевое значение является допустимой записью карты. Чтобы `SetMapIndex` не устанавливал значения в ноль, функция `removeFromMap` создает экземпляр структуры `Value`, например:

```
...
mapValue.SetMapIndex(keyValue, reflect.Value{})
...
```

Это удобный трюк, который гарантирует, что значение `float64` будет удалено с карты. Скомпилируйте и запустите проект, и вы увидите следующий вывод:

```
Key: Kayak, Value: 100
Key: Soccer Ball, Value: 19.5
Key: Hat, Value: 10
```

Создание новых карт

Пакет `reflect` определяет функции, описанные в таблице 28-12, для создания новых карт с использованием отраженных типов.

Таблица 28-12 Функции для создания карт

Функция	Описание
MakeMap(type)	Эта функция возвращает Value, которое отражает карту, созданную с указанным Type.
MakeMapWithSize(type, size)	Эта функция возвращает Value, которое отражает карту, созданную с указанным Type и размером.

При создании карты можно использовать функцию MapOf, описанную в таблице 28-9, для создания значения Type, как показано в листинге 28-14.

```
package main

import (
    "reflect"
    "strings"
    //"fmt"
)

func createMap(slice interface{}, op func(interface{})
interface{}) interface{} {
    sliceVal := reflect.ValueOf(slice)
    if (sliceVal.Kind() == reflect.Slice) {
            mapType := reflect.MapOf(sliceVal.Type().Elem(),
sliceVal.Type().Elem())
        mapVal := reflect.MakeMap(mapType)
        for i := 0; i < sliceVal.Len(); i++ {
            elemVal := sliceVal.Index(i)
                    mapVal.SetMapIndex(elemVal,
reflect.ValueOf(op(elemVal.Interface())))
        }
        return mapVal.Interface()
    }
    return nil
}

func main() {

    names := []string { "Alice", "Bob", "Charlie"}
    reverse := func(val interface{}) interface{} {
        if str, ok := val.(string); ok {
            return strings.ToUpper(str)
        }
        return val
    }

    namesMap := createMap(names, reverse).(map[string]string)
```

```
    for k, v := range namesMap {
        Printfln("Key: %v, Value:%v", k, v)
    }
}
```

Листинг 28-14 Создание карты в файле main.go в папке reflection

Функция createMap принимает срез значений и функцию. Срез перечисляется, и функция вызывается для каждого элемента с исходными и преобразованными значениями, используемыми для заполнения карты, которая возвращается в качестве результата функции.

Вызывающий код должен выполнить утверждение для результата кода createMap, чтобы сузить конкретный тип карты (в данном примере — map[string]string). Функция преобразования в этом примере должна быть написана так, чтобы принимать и возвращать пустой интерфейс, чтобы его можно было использовать функцией createMap. Я объясню, как использовать отражение для улучшения обработки функций в главе 29. Скомпилируйте и выполните проект, и вы увидите следующий вывод:

```
Key: Alice, Value:ALICE
Key: Bob, Value:BOB
Key: Charlie, Value:CHARLIE
```

Работа с типами структур

Структура Type определяет методы, которые можно использовать для проверки типов структур, описанных в таблице 28-13.

Таблица 28-13 Методы Type для структур

Функция	Описание
NumField()	Этот метод возвращает количество полей, определенных типом структуры.
Field(index)	Этот метод возвращает поле по указанному индексу, представленному StructField.
FieldByIndex(indices)	Этот метод принимает срез int, который используется для поиска вложенного поля, представленного StructField.
FieldByName(name)	Этот метод возвращает поле с указанным именем, которое представлено StructField. Результатом является StructField, представляющий поле, и bool значение, указывающее, было ли найдено совпадение.
FieldByNameFunc(func)	Этот метод передает имя каждого поля, включая вложенные поля, в указанную функцию и возвращает первое поле, для которого функция возвращает значение true. Результатом является StructField, представляющий поле, и bool значение, указывающее, было ли найдено совпадение.

Пакет reflect представляет отраженные поля со структурой StructField, которая определяет поля, описанные в таблице 28-14.

Таблица 28-14 Поля StructField

Функция	Описание
Name	В этом поле хранится имя отраженного поля.
PkgPath	Это поле возвращает имя пакета, которое используется для определения того, было ли поле экспортировано. Для экспортируемых отраженных полей это поле возвращает пустую строку. Для отраженных полей, которые не были экспортированы, это поле возвращает имя пакета, который является единственным пакетом, в котором можно использовать это поле.
Type	Это поле возвращает отраженный тип отраженного поля, описанный с помощью Type.
Tag	Это поле возвращает тег структуры, связанный с отраженным полем, как описано в разделе «Проверка тегов структуры».
Index	Это поле возвращает int срез, обозначающий индекс поля, используемый методом FieldByIndex, описанным в таблице 28-13.
Anonymous	Это поле возвращает значение true, если отраженное поле встроено, и значение false в противном случае.

В листинге 28-15 используются методы и поля, описанные в таблицах 28-13 и 28-14, для проверки типа структуры.

```
package main

import (
    "reflect"
    // "strings"
    // "fmt"
)

func inspectStructs(structs ...interface{}) {
    for _, s := range structs {
        structType := reflect.TypeOf(s)
        if (structType.Kind() == reflect.Struct) {
            inspectStructType(structType)
        }
    }
}

func inspectStructType(structType reflect.Type) {
    Printfln("--- Struct Type: %v", structType)
    for i := 0; i < structType.NumField(); i++ {
        field := structType.Field(i)
        Printfln("Field %v: Name: %v, Type: %v, Exported: %v",
```

```
            field.Index, field.Name, field.Type, field.PkgPath ==
"")
    }
    Println("--- End Struct Type: %v", structType)
}

func main() {
    inspectStructs( Purchase{} )
}
```

Листинг 28-15 Проверка типа структуры в файле main.go в папке reflection

Функция `inspectStructs` определяет переменный параметр, через который она получает значения. Функция `TypeOf` используется для получения отраженного типа, а метод `Kind` используется для подтверждения того, что каждый тип является структурой. Отраженный `Type` передается функции `inspectStructType`, в которой метод `NumField` используется в цикле `for`, что позволяет перечислять поля структур с помощью метода `Field`. Скомпилируйте и выполните проект, и вы увидите детали типа структуры `Purchase`:

```
--- Struct Type: main.Purchase
Field [0]: Name: Customer, Type: main.Customer, Exported: true
Field [1]: Name: Product, Type: main.Product, Exported: true
Field [2]: Name: Total, Type: float64, Exported: true
Field [3]: Name: taxRate, Type: float64, Exported: false
--- End Struct Type: main.Purchase
```

Обработка вложенных полей

Выходные данные листинга 28-15 включают поле `StructField.Index`, которое используется для определения положения каждого поля, определенного типом структуры, например:

```
...
Field [2]: Name: Total, Type: float64, Exported: true
...
```

Поле `Total` имеет индекс 2. Индекс полей определяется порядком, в котором они определены в исходном коде, а это означает, что изменение порядка полей приведет к изменению их индекса при отражении типа структуры.

Идентификация полей становится более сложной, когда проверяются поля вложенных структур, как показано в листинге 28-16.

```
package main

import (
    "reflect"
    // "strings"
    // "fmt"
)

func inspectStructs(structs ...interface{}) {
    for _, s := range structs {
        structType := reflect.TypeOf(s)
        if (structType.Kind() == reflect.Struct) {
            inspectStructType([]int {}, structType)
        }
    }
}

func inspectStructType(baseIndex []int, structType reflect.Type) {
    Printfln("--- Struct Type: %v", structType)
    for i := 0; i < structType.NumField(); i++ {
        fieldIndex := append(baseIndex, i)
        field := structType.Field(i)
        Printfln("Field %v: Name: %v, Type: %v, Exported: %v",
             fieldIndex, field.Name, field.Type, field.PkgPath ==
"")
        if (field.Type.Kind() == reflect.Struct) {
            field := structType.FieldByIndex(fieldIndex)
            inspectStructType(fieldIndex, field.Type)
        }
    }
    Printfln("--- End Struct Type: %v", structType)
}

func main() {
    inspectStructs( Purchase{} )
}
```

Листинг 28-16 Проверка полей вложенной структуры в файле main.go в папке reflection

Новый код обнаруживает поля структур и обрабатывает их, рекурсивно вызывая функцию `inspectStructType`

Подсказка
Тот же подход можно использовать для проверки полей, являющихся указателями на типы структур, с использованием метода `Type.Elem` для получения типа, на который указывает указатель.

Скомпилируйте и выполните проект, и вы увидите следующий вывод, к которому я добавил отступы, чтобы сделать отношения между полями более очевидными:

```
--- Struct Type: main.Purchase
Field [0]: Name: Customer, Type: main.Customer, Exported: true
  --- Struct Type: main.Customer
  Field [0 0]: Name: Name, Type: string, Exported: true
  Field [0 1]: Name: City, Type: string, Exported: true
 --- End Struct Type: main.Customer
Field [1]: Name: Product, Type: main.Product, Exported: true
  --- Struct Type: main.Product
  Field [1 0]: Name: Name, Type: string, Exported: true
  Field [1 1]: Name: Category, Type: string, Exported: true
  Field [1 2]: Name: Price, Type: float64, Exported: true
 --- End Struct Type: main.Product
Field [2]: Name: Total, Type: float64, Exported: true
Field [3]: Name: taxRate, Type: float64, Exported: false
--- End Struct Type: main.Purchase
```

Вы можете видеть, что исследование типа структуры `Purchase` теперь включает вложенные поля `Product` и `Customer` и отображает поля, определенные этими вложенными типами. Вы заметите, что выходные данные идентифицируют каждое поле по его индексу в типе, который его определяет, и родительском типе, например:

```
...
Field [1 2]: Name: Price, Type: float64, Exported: true
...
```

Поле `Price` находится в индексе 2 в окружающей его структуре `Product`, который находится в индексе 1 во внешней структуре `Purchase`.

Существует несоответствие в том, как вложенные поля структуры обрабатываются пакетом `reflect`. Метод `FieldByIndex` используется для поиска вложенных полей, чтобы я мог запросить поле напрямую, если мне известна последовательность индексов, чтобы я мог напрямую получить поле `Price`, передав метод `FieldByIndex []int {1, 2}`. Проблема заключается в том, что `StructField`, возвращаемый методом `FieldByIndex`, имеет поле `Index`, которое возвращает только один элемент, отражающий только индекс в окружающей структуре.

Это означает, что результат метода `FieldByIndex` не может быть легко использован для последующих вызовов того же метода, и именно по этой

причине мне нужно отслеживать индексы, используя собственный срез `int`, и использовать его в качестве аргумента метода `FieldByIndex` в листинге 28-16:

```
...
fieldIndex := append(baseIndex, i)
...
field := structType.FieldByIndex(fieldIndex)
...
```

Эта проблема делает изучение типа структуры немного неудобным, но ее легко обойти, если вы знаете, что это происходит, и большинство проектов не будут пытаться пройти дерево полей таким образом.

Поиск поля по имени

Проблема, описанная в предыдущем разделе, не влияет на метод `FieldByName`, который выполняет поиск поля с определенным именем и правильно устанавливает поле `Index` возвращаемого `StructField`, как показано в листинге 28-17.

```
package main

import (
    "reflect"
    //"strings"
    //"fmt"
)

func describeField(s interface{}, fieldName string) {
    structType := reflect.TypeOf(s)
    field, found := structType.FieldByName(fieldName)
    if (found) {
        Printfln("Found: %v, Type: %v, Index: %v",
            field.Name, field.Type, field.Index)
        index := field.Index
        for len(index) > 1 {
            index = index[0: len(index) -1]
            field = structType.FieldByIndex(index)
            Printfln("Parent : %v, Type: %v, Index: %v",
                field.Name, field.Type, field.Index)
        }
        Printfln("Top-Level Type: %v" , structType)
    } else {
        Printfln("Field %v not found", fieldName)
```

```
    }
}

func main() {
    describeField( Purchase{}, "Price" )
}
```

Листинг 28-17 Поиск поля структуры по имени в файле main.go в папке reflection

Функция `descriptionField` использует метод `FieldByName`, который находит первое поле с указанным именем и возвращает `StructField` с правильно установленным полем `Index`. Цикл `for` используется для резервного копирования иерархии типов, по очереди проверяя каждого родителя. Скомпилируйте и запустите проект, и вы увидите следующий результат:

```
Found: Price, Type: float64, Index: [1 2]
Parent : Product, Type: main.Product, Index: [1]
Top-Level Type: main.Purchase
```

Обратите внимание, что я должен использовать значение `Index` из `StructField`, возвращаемое методом `FieldByName`, потому что работа над иерархией с использованием метода `FieldByIndex` приводит к проблеме, описанной в предыдущем разделе.

Проверка тегов структуры

Поле `StructField.Tag` предоставляет сведения о структурном теге, связанном с полем. Теги структур можно проверять только с помощью отражения, что ограничивает их использование, и большинство проектов будут использовать теги только при определении структур, чтобы указать направление другим пакетам, как показано в главе 21 для работы с данными JSON.

Поле `Tag` возвращает значение `StructTag`, которое является псевдонимом для строки. Теги структуры представляют собой, по сути, строку с закодированными парами ключ-значение, и причина создания типа псевдонима `StructTag` заключается в том, чтобы позволить определять методы, описанные в таблице 28-15.

Таблица 28-15 Методы, определяемые типом StructTag

Функция	**Описание**
`Get(key)`	Этот метод возвращает строку, содержащую значение для указанного ключа, или пустую строку, если значение не было определено.

Функция	Описание
Lookup(key)	Этот метод возвращает строку, содержащую значение для указанного ключа, или пустую строку, если значение не было определено, и bool значение, которое имеет значение true, если значение было определено, и значение false в противном случае.

Методы в таблице 28-15 аналогичны, а разница в том, что метод Lookup различает ключи, для которых не было определено значение, и ключи, которые были определены с пустой строкой в качестве значения. В листинге 28-18 определяется структура с тегами и демонстрируется использование этих методов.

```
package main

import (
    "reflect"
    //"strings"
    //"fmt"
)

func inspectTags(s interface{}, tagName string) {
    structType := reflect.TypeOf(s)
    for i := 0; i < structType.NumField(); i++ {
        field := structType.Field(i)
        tag := field.Tag
        valGet := tag.Get(tagName)
        valLookup, ok := tag.Lookup(tagName)
          Printfln("Field: %v, Tag %v: %v", field.Name, tagName,
valGet)
        Printfln("Field: %v, Tag %v: %v, Set: %v",
            field.Name, tagName, valLookup, ok)
    }
}

type Person struct {
    Name string `alias:"id"`
    City string `alias:""`
    Country string
}

func main() {
    inspectTags(Person{}, "alias")
}
```

Листинг 28-18 Проверка тегов структуры в файле main.go в папке reflection

Функция inspectTags перечисляет поля, определенные типом структуры, и использует методы Get и Lookup для получения указанного тега. Функция применяется к типу Person, который определяет тег alias для некоторых своих полей. Скомпилируйте и запустите проект, и вы увидите следующий вывод:

```
Field: Name, Tag alias: id
Field: Name, Tag alias: id, Set: true
Field: City, Tag alias:
Field: City, Tag alias: , Set: true
Field: Country, Tag alias:
Field: Country, Tag alias: , Set: false
```

Дополнительный результат, возвращаемый методом Lookup, позволяет различать поле City, в котором тег alias определен как пустая строка, и поле Country, которое вообще не имеет тега alias.

Создание типов структур

Пакет reflect предоставляет функцию, описанную в таблице 28-16, для создания типов структур. Это не та функция, которая часто требуется, потому что результатом является тип, который можно использовать только с отражением.

Таблица 28-16 Функция отражения для создания типов структур

Функция	Описание
StructOf(fields)	Эта функция создает новый тип структуры, используя указанный срез StructField для определения полей. Можно указать только экспортированные поля.

В листинге 28-19 создается тип структуры, а затем проверяются его теги.

```
package main

import (
    "reflect"
    //"strings"
    //"fmt"
)

func inspectTags(s interface{}, tagName string) {
    structType := reflect.TypeOf(s)
    for i := 0; i < structType.NumField(); i++ {
        field := structType.Field(i)
```

```
        tag := field.Tag
        valGet := tag.Get(tagName)
        valLookup, ok := tag.Lookup(tagName)
          Printfln("Field: %v, Tag %v: %v", field.Name, tagName,
valGet)
        Printfln("Field: %v, Tag %v: %v, Set: %v",
            field.Name, tagName, valLookup, ok)
    }
}

func main() {

    stringType := reflect.TypeOf("this is a string")

    structType := reflect.StructOf([] reflect.StructField {
        { Name: "Name", Type: stringType, Tag: `alias:"id"` },
        { Name: "City", Type: stringType,Tag: `alias:""`},
        { Name: "Country", Type: stringType },
    })

    inspectTags(reflect.New(structType), "alias")
}
```

Листинг 28-19 Создание типа структуры в файле main.go в папке reflection

В этом примере создается структура с теми же характеристиками, что и структура `Person` в предыдущем разделе, с полями `Name`, `City` и `Country`. Поля описываются путем создания значений `StructField`, которые являются обычными структурами Go. Функция `New` используется для создания нового значения из структуры, которое передается функции `inspectTags`. Скомпилируйте и запустите проект, и вы получите следующий вывод:

```
Field: typ, Tag alias:
Field: typ, Tag alias: , Set: false
Field: ptr, Tag alias:
Field: ptr, Tag alias: , Set: false
Field: flag, Tag alias:
Field: flag, Tag alias: , Set: false
```

Работа со структурными значениями

Интерфейс `Value` определяет методы, описанные в таблице 28-17, для работы со значениями структуры.

Таблица 28-17 Методы Value для работы со структурами

Функция	Описание
NumField()	Этот метод возвращает количество полей, определяемое типом значения структуры.
Field(index)	Этот метод возвращает Value, которое отражает поле по указанному индексу.
FieldByIndex(indices)	Этот метод возвращает Value, отражающее вложенное поле по указанным индексам.
FieldByName(name)	Этот метод возвращает Value, которое отражает первое поле, расположенное с указанным именем.
FieldByNameFunc(func)	Этот метод передает имя каждого поля, включая вложенные поля, в указанную функцию и возвращает Value, отражающее первое поле, для которого функция возвращает значение true, и bool значение, указывающее, было ли найдено совпадение.

Методы в таблице 28-17 соответствуют описанным в предыдущем разделе для работы с типами структур. Как только вы поймете состав типа структуры, вы сможете получить Value для каждого интересующего вас поля и применить основные функции отражения, как показано в листинге 28-20.

```
package main

import (
    "reflect"
    //"strings"
    //"fmt"
)

func getFieldValues(s interface{}) {
    structValue := reflect.ValueOf(s)
    if structValue.Kind() == reflect.Struct {
        for i := 0; i < structValue.NumField(); i++ {
            fieldType := structValue.Type().Field(i)
            fieldVal := structValue.Field(i)
            Printfln("Name: %v, Type: %v, Value: %v",
                fieldType.Name, fieldType.Type, fieldVal)
        }
    } else {
        Printfln("Not a struct")
    }
}

func main() {
     product := Product{ Name: "Kayak", Category: "Watersports",
Price: 279 }
    customer := Customer{ Name: "Acme", City: "Chicago" }
```

```
    purchase := Purchase { Customer: customer, Product: product,
Total: 279,
        taxRate: 10 }

    getFieldValues(purchase)
}
```

Листинг 28-20 Чтение значений поля структуры в файле main.go в папке reflection

Функция getFieldValues перечисляет поля, определенные структурой, и записывает сведения о типе и значении поля. Скомпилируйте и запустите проект, и вы увидите следующий вывод:

```
Name: Customer, Type: main.Customer, Value: {Acme Chicago}
Name: Product, Type: main.Product, Value: {Kayak Watersports 279}
Name: Total, Type: float64, Value: 279
Name: taxRate, Type: float64, Value: 10
```

Установка значений поля структуры

Как только вы получили Value для поля структуры, это поле можно изменить так же, как и любое другое отраженное значение, как показано в листинге 28-21.

```
package main

import (
    "reflect"
    //"strings"
    //"fmt"
)

func setFieldValue(s interface{}, newVals map[string]interface{})
{
    structValue := reflect.ValueOf(s)
    if (structValue.Kind() == reflect.Ptr &&
            structValue.Elem().Kind() == reflect.Struct) {
        for name, newValue := range newVals {
            fieldVal := structValue.Elem().FieldByName(name)
            if (fieldVal.CanSet()) {
                fieldVal.Set(reflect.ValueOf(newValue))
            } else if (fieldVal.CanAddr()) {
                ptr := fieldVal.Addr()
                if (ptr.CanSet()) {
                    ptr.Set(reflect.ValueOf(newValue))
                } else {
                    Printfln("Cannot set field via pointer")
```

```
                }
            } else {
                Printfln("Cannot set field")
            }
        }
    } else {
        Printfln("Not a pointer to a struct")
    }
}

func getFieldValues(s interface{}) {
    structValue := reflect.ValueOf(s)
    if structValue.Kind() == reflect.Struct {
        for i := 0; i < structValue.NumField(); i++ {
            fieldType := structValue.Type().Field(i)
            fieldVal := structValue.Field(i)
            Printfln("Name: %v, Type: %v, Value: %v",
                fieldType.Name, fieldType.Type, fieldVal)
        }
    } else {
        Printfln("Not a struct")
    }
}

func main() {
     product := Product{ Name: "Kayak", Category: "Watersports",
Price: 279 }
    customer := Customer{ Name: "Acme", City: "Chicago" }
     purchase := Purchase { Customer: customer, Product: product,
Total: 279,
        taxRate: 10 }

    setFieldValue(&purchase, map[string]interface{} {
        "City": "London", "Category": "Boats", "Total": 100.50,
    })

    getFieldValues(purchase)
}
```

Листинг 28-21 Настройка поля структуры в файле main.go в папке reflection

Как и в случае с другими типами данных, отражение можно использовать только для изменения значений через указатель на структуру. Метод `Elem` используется для отслеживания указателя, так что `Value`, отражающее поле, может быть получено с помощью одного из методов, описанных в таблице

28-17. Метод CanSet используется для определения возможности установки поля.

Для полей, которые не являются вложенными структурами, требуется дополнительный шаг, который заключается в создании указателя на значение поля с помощью метода Addr, например:

```
...
} else if (fieldVal.CanAddr()) {
    ptr := fieldVal.Addr()
    if (ptr.CanSet()) {
        ptr.Set(reflect.ValueOf(newValue))
...
```

Без этого дополнительного шага нельзя изменить значение невложенных полей. Изменения в листинге 28-21 изменяют значения полей City, Category и Total, что приводит к следующему результату при компиляции и выполнении проекта:

```
Name: Customer, Type: main.Customer, Value: {Acme London}
Name: Product, Type: main.Product, Value: {Kayak Boats 279}
Name: Total, Type: float64, Value: 100.5
Name: taxRate, Type: float64, Value: 10
```

Обратите внимание, что я использую метод CanSet даже после того, как вызвал метод Addr для создания значения указателя в листинге >28-21. Отражение нельзя использовать для установки неэкспортированных полей структуры, поэтому мне нужно выполнить дополнительную проверку, чтобы избежать паники, пытаясь установить поле, которое никогда не может быть установлено. (На самом деле, есть некоторые обходные пути для установки неэкспортируемых полей, но они неприятны, и я не рекомендую их использовать. Поиск в Интернете даст вам необходимую информацию, если вы решили установить неэкспортируемые поля.)

Резюме

В этой главе я продолжил описывать функции отражения в Go, объясняя, как они используются с указателями, массивами, срезами, картами и структурами. В следующей главе я завершаю описание этой важной, но сложной функции.

29. Использование отражения, часть 3

В этой главе я завершаю описание поддержки отражения в Go, которое я начал в главе 27 и продолжил в главе 28. В этой главе я объясню, как отражение используется для функций, методов, интерфейсов и каналов. Таблица 29-1 суммирует содержание главы.

Таблица 29-1 Краткое содержание главы

Проблема	Решение	Листинг
Проверить и вызывать отраженные функции	Используйте методы `Type` и `Value` для функций	5–7
Создание новых функций	Используйте функции `FuncOf` и `MakeFunc`.	8, 9
Проверить и вызвать отраженные методы	Используйте методы `Type` и `Value` для методов	10–12
Проверить отраженные интерфейсы	Используйте методы `Type` и `Value` для интерфейсов	13–15
Проверить и использовать отраженные каналы	Используйте методы `Type` и `Value` для каналов	16–19

Подготовка к этой главе

В этой главе я продолжаю использовать проект `reflection` из главы 28. Чтобы подготовиться к этой главе, добавьте в проект `reflection` файл с именем `interfaces.go` с содержимым, показанным в листинге 29-1.

> **Подсказка**
> Вы можете загрузить пример проекта для этой главы — и для всех остальных глав этой книги — с `https://github.com/apress/pro-go`. См. Главу 2 о том, как получить помощь, если у вас возникнут проблемы с запуском примеров.

```
package main
```

```
import "fmt"

type NamedItem interface {
    GetName() string
    unexportedMethod()
}

type CurrencyItem interface {
    GetAmount() string
    currencyName() string
}

func (p *Product) GetName() string {
    return p.Name
}

func (c *Customer) GetName() string {
    return c.Name
}

func (p *Product) GetAmount() string {
    return fmt.Sprintf("$%.2f", p.Price)
}

func (p *Product) currencyName() string {
    return "USD"
}

func (p *Product) unexportedMethod() {}
```

Листинг 29-1 Содержимое файла interfaces.go в папке reflection

Добавьте файл с именем `functions.go` в папку отражения с содержимым, показанным в листинге 29-2.

```
package main

func Find(slice []string, vals... string) (matches bool) {
    for _, s1 := range slice {
        for _, s2 := range vals {
            if s1 == s2 {
                matches = true
                return
```

```
            }
        }
    }
    return
}
```

Листинг 29-2 Содержимое файла functions.go в папке reflection

Добавьте файл с именем `method.go` в папку отражения с содержимым, показанным в листинге 29-3.

```
package main

func (p Purchase) calcTax(taxRate float64) float64 {
    return p.Price * taxRate
}

func (p Purchase) GetTotal() float64 {
    return p.Price + p.calcTax(.20)
}
```

Листинг 29-3 Содержимое файла method.go в папке reflection

Запустите команду, показанную в листинге 29-4, в папке `reflection`, чтобы скомпилировать и выполнить проект.

```
go run .
```

Листинг 29-4 Компиляция и выполнение проекта

Эта команда производит следующий вывод:

```
Name: Customer, Type: main.Customer, Value: {Acme London}
Name: Product, Type: main.Product, Value: {Kayak Boats 279}
Name: Total, Type: float64, Value: 100.5
Name: taxRate, Type: float64, Value: 10
```

Работа с типами функций

Как объяснялось в главе 9, функции в Go являются типами, и, как вы могли ожидать, функции можно исследовать и использовать с отражением. Структура `Type` определяет методы, которые можно использовать для проверки типов функций, описанных в таблице 29-2.

Таблица 29-2 Методы Type для работы с функциями

Функция	Описание
NumIn()	Этот метод возвращает количество параметров, определенных функцией.
In(index)	Этот метод возвращает Type, который отражает параметр по указанному индексу.
IsVariadic()	Этот метод возвращает значение true, если последний параметр является вариативным.
NumOut()	Этот метод возвращает количество результатов, определенных функцией.
Out(index)	Этот метод возвращает Type, который отражает результат по указанному индексу.

В листинге 29-5 отражение используется для описания функции.

```
package main

import (
    "reflect"
    //"strings"
    //"fmt"
)

func inspectFuncType(f interface{}) {
    funcType := reflect.TypeOf(f)
    if (funcType.Kind() == reflect.Func) {
        Printfln("Function parameters: %v", funcType.NumIn())
        for i := 0 ; i < funcType.NumIn(); i++ {
            paramType := funcType.In(i)
            if (i < funcType.NumIn() -1) {
                      Printfln("Parameter #%v, Type: %v", i,
paramType)
            } else {
                 Printfln("Parameter #%v, Type: %v, Variadic:
%v", i, paramType,
                    funcType.IsVariadic())
            }
        }
        Printfln("Function results: %v", funcType.NumOut())
        for i := 0 ; i < funcType.NumOut(); i++ {
            resultType := funcType.Out(i)
            Printfln("Result #%v, Type: %v", i, resultType)
        }
```

```
    }
}

func main() {
    inspectFuncType(Find)
}
```

Листинг 29-5 Отражение функции в файле main.go в папке reflection

Функция `inspectFuncType` использует методы, описанные в таблице 29-2, для проверки типа функции, сообщая о ее параметрах и результатах. Скомпилируйте и выполните проект, и вы увидите следующий вывод, описывающий функцию `Find`, определенную в листинге 29-2:

```
Parameter #0, Type: []string
Parameter #1, Type: []string, Variadic: true
Function results: 1
Result #0, Type: bool
```

Выходные данные показывают, что функция `Find` имеет два параметра, последний из которых является переменным, и один результат.

Работа со значениями функций

Интерфейс `Value` определяет описанный в таблице 29-3 метод вызова функций.

Таблица 29-3 Метод Value для вызова функций

Функция	Описание
`Call(params)`	Эта функция вызывает отраженную функцию, используя `[]Value` в качестве параметров. Результатом является значение `[]Value`, содержащее результаты функции. Значения, предоставляемые в качестве параметров, должны соответствовать значениям, определенным функцией.

Метод `Call` вызывает функцию и возвращает срез, содержащий результаты. Параметры для функции задаются с помощью среза `Value`, а метод `Call` автоматически обнаруживает переменные параметры.

Результаты возвращаются в виде другого среза Value, как показано в листинге 29-6.

```
package main

import (
    "reflect"
    //"strings"
    //"fmt"
)

func invokeFunction(f interface{}, params ...interface{}) {
    paramVals := []reflect.Value {}
    for _, p := range params {
        paramVals = append(paramVals, reflect.ValueOf(p))
    }
    funcVal := reflect.ValueOf(f)
    if (funcVal.Kind() == reflect.Func) {
        results := funcVal.Call(paramVals)
        for i, r := range results {
            Printfln("Result #%v: %v", i, r)
        }
    }
}

func main() {
    names := []string { "Alice", "Bob", "Charlie" }
    invokeFunction(Find, names, "London", "Bob")
}
```

Листинг 29-6 Вызов функции в файле main.go в папке reflection

Скомпилируйте и запустите проект, и вы увидите следующий вывод:

```
Result #0: true
```

Вызов функции таким образом не является обычным требованием, потому что вызывающий код мог просто вызвать функцию напрямую, но этот пример делает использование метода Call понятным и подчеркивает, что параметры и результаты выражаются с помощью срезов Value. В листинге 29-7 приведен более реалистичный пример.

```
package main

import (
    "reflect"
    "strings"
    //"fmt"
)

func mapSlice(slice interface{}, mapper interface{}) (mapped
[]interface{}) {
    sliceVal := reflect.ValueOf(slice)
    mapperVal := reflect.ValueOf(mapper)
    mapped = []interface{} {}
     if sliceVal.Kind() == reflect.Slice && mapperVal.Kind()
== reflect.Func &&
            mapperVal.Type().NumIn() == 1 &&
             mapperVal.Type().In(0) == sliceVal.Type().Elem()
{
        for i := 0; i < sliceVal.Len(); i++ {
                    result  :=  mapperVal.Call([]reflect.Value
{sliceVal.Index(i)})
            for _, r := range result {
                mapped = append(mapped, r.Interface())
            }
        }
    }
    return
}

func main() {
    names := []string { "Alice", "Bob", "Charlie" }
    results := mapSlice(names, strings.ToUpper)
    Printfln("Results: %v", results)
}
```

Листинг 29-7 Вызов функции для элементов среза в файле main.go в папке reflection

Функция `mapSlice` принимает срез и функцию, передает каждый элемент среза в функцию и возвращает результаты. Может возникнуть соблазн описать параметры функции, чтобы указать количество параметров, например:

```
...
```

```
mapper func(interface{}) interface{}
...
```

Проблема с этим подходом заключается в том, что он ограничивает функции, которые можно использовать, теми, которые определены с параметрами и результатами, которые являются пустым интерфейсом. Вместо этого укажите всю функцию как одно пустое значение интерфейса, например:

```
...
func mapSlice(slice interface{}, mapper interface{}) (mapped
[]interface{}) {
...
```

Это позволяет использовать любую функцию, но требует проверки функции, чтобы убедиться, что ее можно использовать по назначению:

```
...
if sliceVal.Kind() == reflect.Slice && mapperVal.Kind() ==
reflect.Func &&
    mapperVal.Type().NumIn() == 1 &&
    mapperVal.Type().In(0) == sliceVal.Type().Elem() {
...
```

Эти проверки гарантируют, что функция определяет один параметр и что тип параметра соответствует типу элемента среза. Скомпилируйте и запустите проект, и вы увидите следующие результаты:

```
Results: [ALICE BOB CHARLIE]
```

Создание и вызов новых типов функций и значений

Пакет `reflect` определяет функции, описанные в таблице 29-4, для создания новых типов функций и значений.

Таблица 29-4 Функция reflect для создания новых типов функций и значений функций

Функция	Описание

Функция	Описание
FuncOf(params, results, variadic)	Эта функция создает новый Type, который отражает тип функции с указанными параметрами и результатами. Последний аргумент указывает, имеет ли тип функции переменный параметр. Параметры и результаты указаны как срезы Type.
MakeFunc(type, fn)	Эта функция возвращает Value, отражающее новую функцию, являющуюся оболочкой функции fn. Функция должна принимать срез Value в качестве единственного параметра и возвращать срез Value в качестве единственного результата.

Одним из применений функции FuncOf является создание сигнатуры типа и ее использование для проверки сигнатуры значения функции, заменяющей проверки, выполненные в предыдущем разделе, как показано в листинге 29-8.

```
package main

import (
    "reflect"
    "strings"
    //"fmt"
)

func mapSlice(slice interface{}, mapper interface{}) (mapped
[]interface{}) {
    sliceVal := reflect.ValueOf(slice)
    mapperVal := reflect.ValueOf(mapper)
    mapped = []interface{} {}

     if sliceVal.Kind() == reflect.Slice && mapperVal.Kind()
== reflect.Func {
        paramTypes := []reflect.Type { sliceVal.Type().Elem()
}
        resultTypes := []reflect.Type {}
        for i := 0; i < mapperVal.Type().NumOut(); i++ {
                             resultTypes = append(resultTypes,
mapperVal.Type().Out(i))
        }
        expectedFuncType := reflect.FuncOf(paramTypes,
            resultTypes, mapperVal.Type().IsVariadic())
        if (mapperVal.Type() == expectedFuncType) {
            for i := 0; i < sliceVal.Len(); i++ {
```

```
                    result := mapperVal.Call([]reflect.Value
{sliceVal.Index(i)})
                for _, r := range result {
                    mapped = append(mapped, r.Interface())
                }
            }
        } else {
            Println("Function type not as expected")
        }
    }
    return
}

func main() {
    names := []string { "Alice", "Bob", "Charlie" }
    results := mapSlice(names, strings.ToUpper)
    Printfln("Results: %v", results)
}
```

Листинг 29-8 Создание типа функции в файле main.go в папке reflection Folder

Этот подход не менее многословен, не в последнюю очередь потому, что я хочу принимать функции с тем же типом параметра, что и тип элемента среза, но с любым типом результата. Получить тип элемента среза несложно, но мне нужно проделать некоторую работу, чтобы создать срез `Type`, отражающий результаты функции сопоставления, чтобы убедиться, что я создаю тип, который будет корректно сравниваться. Скомпилируйте и запустите проект, и вы увидите следующий вывод:

```
Results: [ALICE BOB CHARLIE]
```

Функция `FuncOf` дополняется функцией `MakeFunc`, которая создает новые функции, используя тип функции в качестве шаблона. В листинге 29-9 показано использование функции `MakeFunc` для создания повторно используемой функции отображения типов.

```
package main

import (
    "reflect"
    "strings"
```

```
    "fmt"
)

func makeMapperFunc(mapper interface{}) interface{} {
    mapVal := reflect.ValueOf(mapper)
    if mapVal.Kind() == reflect.Func && mapVal.Type().NumIn()
== 1 &&
            mapVal.Type().NumOut() == 1  {
        inType := reflect.SliceOf( mapVal.Type().In(0))
        inTypeSlice := []reflect.Type { inType }
        outType := reflect.SliceOf( mapVal.Type().Out(0))
        outTypeSlice := []reflect.Type { outType }
        funcType := reflect.FuncOf(inTypeSlice, outTypeSlice,
false)
        funcVal := reflect.MakeFunc(funcType,
                      func  (params  []reflect.Value)  (results
[]reflect.Value) {
            srcSliceVal := params[0]
               resultsSliceVal  :=  reflect.MakeSlice(outType,
srcSliceVal.Len(), 10)
            for i := 0; i < srcSliceVal.Len(); i++ {
                         r  :=  mapVal.Call([]reflect.Value  {
srcSliceVal.Index(i)})
                resultsSliceVal.Index(i).Set(r[0])
            }
            results = []reflect.Value { resultsSliceVal }
            return
        })
        return funcVal.Interface()
    }
    Printfln("Unexpected types")
    return nil
}

func main() {

      lowerStringMapper  :=  makeMapperFunc(strings.ToLower).
(func([]string)[]string)
    names := []string { "Alice", "Bob", "Charlie" }
    results := lowerStringMapper(names)
    Printfln("Lowercase Results: %v", results)
```

```
    incrementFloatMapper := makeMapperFunc(func (val float64)
float64 {
        return val + 1
    }).(func([]float64)[]float64)
    prices := []float64 { 279, 48.95, 19.50}
    floatResults := incrementFloatMapper(prices)
    Printfln("Increment Results: %v", floatResults)

     floatToStringMapper := makeMapperFunc(func (val float64)
string {
        return fmt.Sprintf("$%.2f", val)
    }).(func([]float64)[]string)
                        Printfln("Price     Results:     %v",
floatToStringMapper(prices))
}
```

Листинг 29-9 Создание функции в файле main.go в папке reflection

Функция `makeMapperFunc` демонстрирует, насколько гибким может быть рефлексия, но также показывает, насколько многословным и плотным она может быть. Лучший способ понять эту функцию — сосредоточиться на входах и выходах. `makeMapperFunc` принимает функцию, которая преобразует одно значение в другое, с такой сигнатурой:

```
...
func mapper(int) string
...
```

Эта гипотетическая функция получает значение типа `int` и возвращает `string` результат. `makeMapperFunc` использует типы этой функции для создания функции, которая будет выражена следующим образом в обычном коде Go:

```
...
func useMapper(slice []int) []string {
    results := []string {}
    for _, val := range slice {
        results = append(results, mapper(val))
    }
    return results
}
```

```
...
```

Функция useMapper представляет собой оболочку для функции mapper. Функции mapper и useMapper легко определить в обычном коде Go, но они специфичны для одного набора типов. makeMapperFunc использует отражение, поэтому может принимать любую функцию сопоставления и генерировать соответствующую оболочку, которую затем можно использовать со стандартными функциями безопасности типа Go.

Первым шагом является определение типов функции отображения:

```
...
inType := reflect.SliceOf( mapVal.Type().In(0))
inTypeSlice := []reflect.Type { inType }
outType := reflect.SliceOf( mapVal.Type().Out(0))
outTypeSlice := []reflect.Type { outType }
...
```

Затем эти типы используются для создания типа функции для оболочки:

```
...
funcType := reflect.FuncOf(inTypeSlice, outTypeSlice, false)
...
```

Получив тип функции, я могу использовать его для создания функции-оболочки с помощью функции MakeFunc:

```
...
funcVal := reflect.MakeFunc(funcType,
    func (params []reflect.Value) (results []reflect.Value) {
...
```

Функция MakeFunc принимает Type, описывающий функцию, и функцию, которую будет вызывать новая функция. В листинге 29-9 функция перечисляет элементы в срезе, вызывает функцию сопоставления для каждого из них и создает срез результатов.

Результатом является функция, безопасная для типов, хотя она требует утверждения типа:

```
...
lowerStringMapper    :=    makeMapperFunc(strings.ToLower).
(func([]string)[]string)
...
```

Функция makeMapperFunc получает функцию strings.ToLower и создает функцию, которая принимает срез строки и возвращает срез строк. Другие вызовы makeMapperFunc создают функции, которые преобразуют значения float64 в другие значения float64 и преобразуют значения float64 в строки денежного формата. Скомпилируйте и запустите проект, и вы увидите следующий вывод:

```
Lowercase Results: [alice bob charlie]
Increment Results: [280 49.95 20.5]
Price Results: [$279.00 $48.95 $19.50]
```

Работа с методами

Структура Type определяет методы, описанные в таблице 29-5, для проверки методов, определенных структурой.

Таблица 29-5 Методы Type для работы с методами

Функция	**Описание**
NumMethod()	Этот метод возвращает количество экспортированных методов, определенных для отражаемого типа структуры.
Method(index)	Этот метод возвращает отраженный метод по указанному индексу, представленному структурой Method.
MethodByName(name)	Этот метод возвращает отраженный метод с указанным именем. Результатами являются структура Method и bool значение, указывающее, существует ли метод с указанным именем.

> **Примечание**
> Рефлексия не поддерживает создание новых методов. Ее можно использовать только для проверки и вызова существующих методов.

Методы представлены структурой Method, которая определяет поля, описанные в таблице 29-6.

Таблица 29-6 Поля, определяемые структурой Method

Функция	Описание
`Name`	Это поле возвращает имя метода в виде строки.
`PkgPath`	Это поле используется с интерфейсами, как описано в разделе «Работа с интерфейсами», а не с методами, доступ к которым осуществляется через тип структуры. Поле возвращает `string`, содержащую путь к пакету. Пустая строка используется для экспортируемых полей и будет содержать имя пакета структуры для неэкспортированных полей.
`Type`	Это поле возвращает `Type`, описывающий тип функции метода.
`Func`	Это поле возвращает `Value`, которое отражает значение функции метода. При вызове метода первым аргументом должна быть структура, для которой вызывается метод, как показано в разделе «Вызов методов».
`Index`	Это поле возвращает `int`, указывающее индекс метода для использования с методом `Method`, описанным в таблице 29-5.

> **Примечание**
> При проверке структур методы, которые продвигаются из встроенных полей, включаются в результаты, полученные методами, описанными в этом разделе.

Интерфейс `Value` также определяет методы для работы с отраженными методами, как описано в таблице 29-7.

Таблица 29-7 Метод Value для работы с методами

Функция	Описание
`NumMethod()`	Этот метод возвращает количество экспортированных методов, определенных для отражаемого типа структуры. Он вызывает метод `Type.NumMethod`.
`Method(index)`	Этот метод возвращает `Value`, которое отражает функцию метода по указанному индексу. Получатель не указывается в качестве первого аргумента при вызове функции, как показано в разделе «Вызов методов».
`MethodByName(name)`	Этот метод возвращает `Value`, которое отражает функцию метода с указанным именем. Получатель не указывается в качестве первого аргумента при вызове функции, как показано в разделе «Вызов методов».

Методы в таблице 29-7 — это удобные функции, которые обеспечивают доступ к тем же базовым функциям, что и методы в

таблице 29-5, хотя существуют различия в том, как методы вызываются, как описано в следующем разделе.

В листинге 29-10 определяется функция, описывающая методы, определенные структурой, с использованием методов, предоставленных структурой Type.

```
package main

import (
    "reflect"
    //"strings"
    //"fmt"
)

func inspectMethods(s interface{}) {
    sType := reflect.TypeOf(s)
     if sType.Kind() == reflect.Struct || (sType.Kind() ==
reflect.Ptr &&
            sType.Elem().Kind() == reflect.Struct) {
                Println("Type: %v, Methods: %v", sType,
sType.NumMethod())
        for i := 0; i < sType.NumMethod(); i++ {
            method := sType.Method(i)
            Println("Method name: %v, Type: %v",
                method.Name, method.Type)
        }
    }
}

func main() {

    inspectMethods(Purchase{})
    inspectMethods(&Purchase{})
}
```

Листинг 29-10 Описание методов в файле main.go в папке reflection

Go упрощает вызов методов, позволяя вызывать методы, определенные для структуры, через указатель на структуру и наоборот. Однако при использовании отражения для проверки типов результаты не столь согласуются, что вы можете увидеть в выводе, когда проект компилируется и выполняется:

```
Type: main.Purchase, Methods: 1
Method name: GetTotal, Type: func(main.Purchase) float64
Type: *main.Purchase, Methods: 2
Method name: GetAmount, Type: func(*main.Purchase) string
Method name: GetTotal, Type: func(*main.Purchase) float64
```

Когда для типа Purchase используется отражение, перечисляются только методы, определенные для Product. Но когда отражение используется для типа *Purchase, перечисляются методы, определенные для Product и *Product. Обратите внимание, что через отражение доступны только экспортированные методы — неэкспортированные методы нельзя проверить или вызвать.

Вызов методов

Структура Method определяет поле Func, которое возвращает Value, которое можно использовать для вызова метода, используя тот же подход, описанный ранее в этой главе, как показано в листинге 29-11.

```
package main

import (
    "reflect"
    //"strings"
    //"fmt"
)

func executeFirstVoidMethod(s interface{}) {
    sVal := reflect.ValueOf(s)
    for i := 0; i < sVal.NumMethod(); i++ {
        method := sVal.Type().Method(i)
        if method.Type.NumIn() == 1 {
            results := method.Func.Call([]reflect.Value{ sVal
})
            Printfln("Type: %v, Method: %v, Results: %v",
                sVal.Type(), method.Name, results)
            break
        } else {
              Printfln("Skipping method %v %v", method.Name,
method.Type.NumIn())
        }
    }
```

```
}

func main() {
    executeFirstVoidMethod(&Product { Name: "Kayak", Price:
279})
}
```

Листинг 29-11 Вызов метода в файле main.go в папке reflection

Функция `executeFirstVoidMethod` перечисляет методы, определенные типом параметра, и вызывает первый метод, определяющий один параметр. При вызове метода через поле `Method.Func` первым аргументом должен быть получатель, то есть значение структуры, для которой будет вызываться метод:

```
...
results := method.Func.Call([]reflect.Value{ sVal })
...
```

Это означает, что при поиске метода с одним параметром выбирается метод, не принимающий аргументов, что можно увидеть в результатах, полученных при компиляции и выполнении проекта:

```
Type: *main.Product, Method: GetAmount, Results: [$279.00]
```

Метод `executeFirstVoidMethod` выбрал метод `GetAmount`. Получатель не указывается, когда метод вызывается через интерфейс `Value`, как показано в листинге 29-12.

```
package main

import (
    "reflect"
    //"strings"
    //"fmt"
)

func executeFirstVoidMethod(s interface{}) {
    sVal := reflect.ValueOf(s)
    for i := 0; i < sVal.NumMethod(); i++ {
        method := sVal.Method(i)
        if method.Type().NumIn() == 0 {
```

```
            results := method.Call([]reflect.Value{})
            Println("Type: %v, Method: %v, Results: %v",
                    sVal.Type(), sVal.Type().Method(i).Name,
results)
            break
        } else {
            Println("Skipping method %v %v",
                                sVal.Type().Method(i).Name,
method.Type().NumIn())
        }
    }
}

func main() {
     executeFirstVoidMethod(&Product { Name: "Kayak", Price:
279})
}
```

Листинг 29-12 Вызов метода через значение в файле main.go в папке reflection

Чтобы найти метод, который я могу вызвать без дополнительных аргументов, я должен искать нулевые параметры, так как получатель явно не указан. Вместо этого получатель определяется из `Value`, для которого вызывается метод `Call`:

```
...
results := method.Call([]reflect.Value{})
...
```

Этот пример выдает тот же результат, что и код в листинге 29-11.

Работы с интерфейсами

Структура `Type` определяет методы, которые можно использовать для проверки типов интерфейсов, описанных в таблице 29-8. Большинство этих методов также можно применять к структурам, как показано в предыдущем разделе, но поведение немного отличается.

Таблица 29-8 Методы Type Methods для интерфейсов

Функция	Описание

Функция	Описание
Implements(type)	Этот метод возвращает значение true, если отраженное значение реализует указанный интерфейс, который также представлен Value.
Elem()	Этот метод возвращает Value, которое отражает значение, содержащееся в интерфейсе.
NumMethod()	Этот метод возвращает количество экспортированных методов, определенных для отражаемого типа структуры.
Method(index)	Этот метод возвращает отраженный метод по указанному индексу, представленному структурой Method.
MethodByName(name)	Этот метод возвращает отраженный метод с указанным именем. Результатами являются структура Method и bool значение, указывающее, существует ли метод с указанным именем.

Следует соблюдать осторожность при использовании отражения для интерфейсов, поскольку пакет reflect всегда начинается со значения и будет пытаться работать с базовым типом этого значения. Самый простой способ решить эту проблему — преобразовать значение nil, как показано в листинге 29-13.

```
package main

import (
    "reflect"
    //"strings"
    //"fmt"
)

func    checkImplementation(check    interface{},    targets
...interface{}) {
    checkType := reflect.TypeOf(check)
    if (checkType.Kind() == reflect.Ptr &&
            checkType.Elem().Kind() == reflect.Interface) {
        checkType := checkType.Elem()
        for _, target := range targets {
            targetType := reflect.TypeOf(target)
            Printfln("Type %v implements %v: %v",
                                        targetType,   checkType,
targetType.Implements(checkType))
        }
    }
}
```

```
func main() {
    currencyItemType := (*CurrencyItem)(nil)
           checkImplementation(currencyItemType,   Product{},
&Product{}, &Purchase{})
}
```

Листинг 29-13 Отражение интерфейса в файле main.go в папке reflection

Чтобы указать интерфейс, который я хочу проверить, я конвертирую nil в указатель интерфейса, например:

```
...
currencyItemType := (*CurrencyItem)(nil)
...
```

Это необходимо сделать с помощью указателя, который затем следует в функции checkImplementation с помощью метода Elem, чтобы получить Type, отражающий интерфейс, которым в этом примере является CurrencyItem:

```
...
if (checkType.Kind() == reflect.Ptr &&
        checkType.Elem().Kind() == reflect.Interface) {
    checkType := checkType.Elem()
...
```

После этого легко проверить, реализует ли тип интерфейс, используя метод Implements. Скомпилируйте и запустите проект, и вы увидите следующий вывод:

```
Type main.Product implements main.CurrencyItem: false
Type *main.Product implements main.CurrencyItem: true
Type *main.Purchase implements main.CurrencyItem: true
```

Вывод показывает, что структура Product не реализует интерфейс, а *Product реализует, потому что *Product — это тип получателя, используемый для реализации методов, необходимых для CurrencyItem. Тип *Purchase также реализует интерфейс, поскольку он имеет вложенные поля структуры, определяющие необходимые методы.

Получение базовых значений из интерфейсов

Хотя рефлексия обычно создает конкретные типы, бывают случаи, когда необходимо использовать метод `Elem` для перехода от интерфейса к типу, который его реализует, как показано в листинге 29-14.

```
package main

import (
    "reflect"
    //"strings"
    //"fmt"
)

type Wrapper struct {
    NamedItem
}

func getUnderlying(item Wrapper, fieldName string) {
    itemVal := reflect.ValueOf(item)
    fieldVal := itemVal.FieldByName(fieldName)
    Printfln("Field Type: %v", fieldVal.Type())
    if (fieldVal.Kind() == reflect.Interface) {
                              Printfln("Underlying   Type:   %v",
fieldVal.Elem().Type())
    }
}

func main() {
              getUnderlying(Wrapper{NamedItem:   &Product{}},
"NamedItem")
}
```

Листинг 29-14 Получение базовых значений интерфейса в файле main.go в папке reflection

Тип `Wrapper` определяет вложенное поле `NamedItem`. Функция `getUnderlying` использует рефлексию для получения поля и записывает тип поля и базовый тип, полученный с помощью метода `Elem`. Скомпилируйте и запустите проект, и вы увидите следующие результаты:

```
Field Type: main.NamedItem
```

```
Underlying Type: *main.Product
```

Тип поля — это интерфейс NamedItem, но метод Elem показывает, что базовое значение, присвоенное полю NamedItem, — это *Product.

Изучение методов интерфейса

Методы NumMethod, Method и MethodByName можно использовать для интерфейсных типов, но результаты включают неэкспортированные методы, чего нельзя сказать о непосредственном исследовании типа структуры, как показано в листинге 29-15.

```
package main

import (
    "reflect"
    //"strings"
    //"fmt"
)

type Wrapper struct {
    NamedItem
}

func getUnderlying(item Wrapper, fieldName string) {
    itemVal := reflect.ValueOf(item)
    fieldVal := itemVal.FieldByName(fieldName)
    Printfln("Field Type: %v", fieldVal.Type())
    for i := 0; i < fieldVal.Type().NumMethod(); i++ {
        method := fieldVal.Type().Method(i)
        Printfln("Interface Method: %v, Exported: %v",
            method.Name, method.PkgPath == "")
    }
    Printfln("--------")
    if (fieldVal.Kind() == reflect.Interface) {
                         Printfln("Underlying   Type:   %v",
fieldVal.Elem().Type())
         for i := 0; i < fieldVal.Elem().Type().NumMethod();
i++ {
            method := fieldVal.Elem().Type().Method(i)
            Printfln("Underlying Method: %v", method.Name)
        }
```

```
    }
}

func main() {
            getUnderlying(Wrapper{NamedItem:   &Product{}},
"NamedItem")
}
```

Листинг 29-15 Изучение методов интерфейса в файле main.go в папке reflection Folder

Изменения записывают детали методов, полученных из интерфейса и базовых типов. Скомпилируйте и запустите проект, и вы увидите следующий вывод:

```
Field Type: main.NamedItem
Interface Method: GetName, Exported: true
Interface Method: unexportedMethod, Exported: false
--------
Underlying Type: *main.Product
Underlying Method: GetAmount
Underlying Method: GetName
```

Список методов для интерфейса `NamedItem` включает `unexportedMethod`, которого нет в списке для `*Product`. Существуют дополнительные методы, определенные для `*Product` помимо тех, которые требуются для интерфейса, поэтому метод `GetAmount` отображается в выходных данных.

Методы можно вызывать через интерфейс, но перед использованием метода `Call` необходимо убедиться, что они экспортированы. Если вы попытаетесь вызвать неэкспортированный метод, `Call` вызовет панику.

Работа с типами каналов

Структура `Type` определяет методы, которые можно использовать для проверки типов каналов, описанных в таблице 29-9.

Таблица 29-9 Методы Type для каналов

Функция	Описание

Функция	Описание
ChanDir()	Этот метод возвращает значение ChanDir, которое описывает направление канала, используя одно из значений, показанных в таблице 29-10.
Elem()	Этот метод возвращает Type, который отражает тип, переносимый каналом.

Результат ChanDir, возвращаемый методом ChanDir, указывает направление канала, которое можно сравнить с одной из констант пакета reflect, описанных в таблице 29-10.

Таблица 29-10 Значения ChanDir

Функция	Описание
RecvDir	Это значение указывает, что канал можно использовать для приема данных. При выражении в виде строки это значение возвращает <-chan.
SendDir	Это значение указывает, что канал можно использовать для отправки данных. При выражении в виде строки это значение возвращает chan<-.
BothDir	Это значение указывает, что канал можно использовать для отправки и получения данных. При выражении в виде строки это значение возвращает chan.

В листинге 29-16 показано использование методов из таблицы 29-9 для проверки типа канала.

```
package main

import (
    "reflect"
    //"strings"
    //"fmt"
)

func inspectChannel(channel interface{}) {
    channelType := reflect.TypeOf(channel)
    if (channelType.Kind() == reflect.Chan) {
        Printfln("Type %v, Direction: %v",
            channelType.Elem(), channelType.ChanDir())
    }
}

func main() {
    var c chan<- string
    inspectChannel(c)
```

```
}
```

Листинг 29-16 Проверка типа канала в файле main.go в папке reflection

Канал, рассмотренный в этом примере, предназначен только для отправки и выдает следующий вывод, когда проект компилируется и выполняется:

```
Type string, Direction: chan<-
```

Работа со значениями канала

Интерфейс `Value` определяет описанные в таблице 29-11 методы работы с каналами.

Таблица 29-11 Метод Value для каналов

Функция	Описание
`Send(val)`	Этот метод отправляет значение, отраженное аргументом `Value` в канале. Этот метод блокируется до тех пор, пока значение не будет отправлено.
`Recv()`	Этот метод получает значение из канала, которое возвращается как Value для рефлексии. Этот метод также возвращает `bool` значение, которое указывает, было ли получено значение, и будет `false`, если канал закрылся. Этот метод блокируется до тех пор, пока не будет получено значение или канал не будет закрыт.
`TrySend(val)`	Этот метод отправляет указанное значение, но не блокируется. Логический результат указывает, было ли отправлено значение.
`TryRecv()`	Этот метод пытается получить значение из канала, но не блокируется. Результатом является `Value`, отражающее полученное значение, и `bool` значение, указывающее, было ли получено значение.
`Close()`	Этот метод закрывает канал.

В листинге 29-17 определена функция, которая получает канал и срез, содержащий значения, которые будут отправлены по каналу.

```
package main

import (
    "reflect"
    //"strings"
    //"fmt"
)
```

```
func sendOverChannel(channel interface{}, data interface{}) {
    channelVal := reflect.ValueOf(channel)
    dataVal := reflect.ValueOf(data)
    if (channelVal.Kind() == reflect.Chan &&
            dataVal.Kind() == reflect.Slice &&
                                    channelVal.Type().Elem()   ==
dataVal.Type().Elem()) {
        for i := 0; i < dataVal.Len(); i++ {
            val := dataVal.Index(i)
            channelVal.Send(val)
        }
        channelVal.Close()
    } else {
                    Printfln("Unexpected   types:   %v,   %v",
channelVal.Type(), dataVal.Type())
    }
}

func main() {

    values := []string { "Alice", "Bob", "Charlie", "Dora"}
    channel := make(chan string)

    go sendOverChannel(channel, values)
    for {
        if val, open := <- channel; open {
            Printfln("Received value: %v", val)
        } else {
            break
        }
    }
}
```

Листинг 29-17 Использование канала в файле main.go в папке reflection Folder

`SendOverChannel` проверяет типы, которые он получает, перечисляет значения в срезе и отправляет каждое из них по каналу. После отправки всех значений канал закрывается. Скомпилируйте и запустите проект, и вы увидите следующий вывод:

```
Received value: Alice
Received value: Bob
Received value: Charlie
```

```
Received value: Dora
```

Создание новых типов и значений каналов

Пакет `reflect` определяет функции, описанные в таблице 29-12, для создания новых типов и значений каналов.

Таблица 29-12 Функции пакета reflect для создания типов и значений каналов

Функция	Описание
`ChanOf(dir, type)`	Эта функция возвращает `Type`, который отражает канал с указанным направлением и типом данных, которые выражаются с помощью `ChanDir` и `Value`.
`MakeChan(type, buffer)`	Эта функция возвращает `Value`, отражающее новый канал, созданный с использованием указанного `Type` и размера буфера `int`.

В листинге `29-18` определена функция, которая принимает срез и использует его для создания канала, который затем используется для отправки элементов среза.

```
package main

import (
    "reflect"
    //"strings"
    //"fmt"
)

func createChannelAndSend(data interface{}) interface{} {
    dataVal := reflect.ValueOf(data)
          channelType   :=   reflect.ChanOf(reflect.BothDir,
dataVal.Type().Elem())
    channel := reflect.MakeChan(channelType, 1)
    go func() {
        for i := 0; i < dataVal.Len(); i++ {
            channel.Send(dataVal.Index(i))
        }
        channel.Close()
    }()
    return channel.Interface()
}
```

```
func main() {

    values := []string { "Alice", "Bob", "Charlie", "Dora"}
    channel := createChannelAndSend(values).(chan string)

    for {
        if val, open := <- channel; open {
            Printfln("Received value: %v", val)
        } else {
            break
        }
    }
}
```

Листинг 29-18 Создание канала в файле main.go в папке reflection

Функция `createChannelAndSend` использует тип элемента среза для создания типа канала, который затем используется для создания канала. Горутина используется для отправки элементов среза в канал, и канал возвращается как результат функции. Скомпилируйте и запустите проект, и вы увидите следующий вывод:

```
Received value: Alice
Received value: Bob
Received value: Charlie
Received value: Dora
```

Выбор из нескольких каналов

Функция выбора канала, описанная в главе 14, может использоваться в коде отражения с использованием функции `Select`, определенной пакетом `reflect`, которая описана в таблице 29-13 для быстрого ознакомления.

Таблица 29-13 Функция пакета reflect для выбора каналов

Функция	Описание
`Select(cases)`	Эта функция принимает срез `SelectCase`, где каждый элемент описывает набор операций отправки или получения. Результатами являются `int`-индекс выполненного `SelectCase`, полученное `Value` (если выбранный случай был операцией чтения) и `bool` значение, указывающее, было ли прочитано значение или канал был заблокирован или закрыт.

Структура SelectCase используется для представления одного оператора case с использованием полей, описанных в таблице 29-14.

Таблица 29-14 Поля структуры SelectCase

Функция	Описание
Chan	Этому полю присваивается Value, отражающее канал.
Dir	Этому полю присваивается значение SelectDir, которое определяет тип операции канала для данного случая.
Send	Этому полю присваивается Value, отражающее значение, которое будет отправлено по каналу для операций отправки.

Тип SelectDir является псевдонимом для int, а пакет reflect определяет константы, описанные в таблице 29-15, для указания типа выбора.

Таблица 29-15 Константы SelectDir

Функция	Описание
SelectSend	Эта константа обозначает операцию отправки значения по каналу.
SelectRecv	Эта константа обозначает операцию получения значения из канала.
SelectDefault	Эта константа обозначает предложение по умолчанию для выбора.

Определение операторов select с использованием отражения является подробным, но результаты могут быть гибкими и принимать более широкий диапазон типов, чем обычный код Go. В листинге 29-19 используется функция Select для чтения значений из нескольких каналов.

```
package main

import (
    "reflect"
    //"strings"
    //"fmt"
)

func createChannelAndSend(data interface{}) interface{} {
    dataVal := reflect.ValueOf(data)
```

```
            channelType   :=   reflect.ChanOf(reflect.BothDir,
dataVal.Type().Elem())
    channel := reflect.MakeChan(channelType, 1)
    go func() {
        for i := 0; i < dataVal.Len(); i++ {
            channel.Send(dataVal.Index(i))
        }
        channel.Close()
    }()
    return channel.Interface()
}

func readChannels(channels ...interface{}) {
    channelsVal := reflect.ValueOf(channels)
    cases := []reflect.SelectCase {}
    for i := 0; i < channelsVal.Len(); i++ {
        cases = append(cases, reflect.SelectCase{
            Chan: channelsVal.Index(i).Elem(),
            Dir: reflect.SelectRecv,
        })
    }
    for {
        caseIndex, val, ok := reflect.Select(cases)
        if (ok) {
                 Printfln("Value  read:  %v,  Type:  %v",  val,
val.Type())
        } else {
            if len(cases) == 1 {
                Printfln("All channels closed.")
                return
            }
                         cases   =   append(cases[:caseIndex],
cases[caseIndex+1:]... )
        }
    }
}

func main() {

    values := []string { "Alice", "Bob", "Charlie", "Dora"}
    channel := createChannelAndSend(values).(chan string)

    cities := []string { "London", "Rome", "Paris"}
```

```
    cityChannel := createChannelAndSend(cities).(chan string)

    prices := []float64 { 279, 48.95, 19.50}
        priceChannel  :=  createChannelAndSend(prices).(chan
float64)

    readChannels(channel, cityChannel, priceChannel)
}
```

Листинг 29-19 Использование функции выбора в файле main.go в папке reflection

В этом примере создаются три канала с помощью функции `createChannelAndSend` и передаются функции `readChannels`, которая использует функцию `Select` для чтения значений, пока все каналы не будут закрыты. Чтобы гарантировать, что чтение выполняется только на открытых каналах, значения `SelecCase` удаляются из среза, переданного функции `Select`, когда канал, который они представляют, закрывается. Скомпилируйте и запустите проект, и вы увидите следующий вывод:

```
Value read: London, Type: string
Value read: Alice, Type: string
Value read: Rome, Type: string
Value read: Bob, Type: string
Value read: Paris, Type: string
Value read: Charlie, Type: string
Value read: 279, Type: float64
Value read: Dora, Type: string
Value read: 48.95, Type: float64
Value read: 19.5, Type: float64
All channels closed.
```

Вы можете увидеть значения, отображаемые в другом порядке, потому что горутины используются для отправки значений по каналам.

Резюме

В этой главе я описал функции отражения для работы с функциями, методами, интерфейсами и каналами, завершив описание функций отражения Go, начатое в главе 27 и продолженное в главе 28. В

следующей главе я опишу функции стандартной библиотеки для координации горутин.

30. Координация горутин

В этой главе я описываю пакеты стандартной библиотеки Go с функциями, которые используются для координации горутин. В таблице 30-1 описаны функции, описанные в этой главе в контексте.

Таблица 30-1 Помещение функций для координации горутин в контекст

Вопрос	**Ответ**
Кто они такие?	Эти функции полезны, когда приложение использует несколько горутин.
Почему они полезны?	Использование горутин может быть сложным, когда они совместно используют данные или когда горутина используется для обработки запроса между несколькими компонентами API на сервере.
Как они используются?	Пакет `sync` предоставляет типы и функции для управления горутинами, включая обеспечение монопольного доступа к данным. Пакет `context` предоставляет функции, которые используются для поддержки обработки запроса сервером, что обычно выполняется с помощью горутины.
Есть ли подводные камни или ограничения?	Это расширенные функции, и их следует использовать с осторожностью.
Есть ли альтернативы?	Не всем приложениям требуются эти функции, особенно если они используют горутины, которые не обмениваются данными.

Таблица 30-2 суммирует главу.

Таблица 30-2 Краткое содержание главы

Проблема	**Решение**	**Листинг**
Ожидание завершения одной или нескольких горутин.	Используйте группу ожидания	5, 6
Предотвращение одновременного доступа к данным нескольких горутин	Использовать взаимное исключение	7–10
Подождите, пока произойдет событие	Используйте условие	11, 12
Убедиться, что функция выполняется один раз	Используйте структуру `Once`	13

Проблема	Решение	Листинг
Предоставить контекст для запросов, обрабатываемых через границы API на серверах.	Использовать контекст	14–17

Подготовка к этой главе

Чтобы подготовиться к этой главе, откройте новую командную строку, перейдите в удобное место и создайте каталог с именем `coordination`. Запустите команду, показанную в листинге 30-1, в папке `coordination`, чтобы создать файл модуля.

> **Подсказка**
> Вы можете загрузить пример проекта для этой главы — и для всех остальных глав этой книги — с `https://github.com/apress/pro-go`. См. Главу 2 о том, как получить помощь, если у вас возникнут проблемы с запуском примеров.

```
go mod init coordination
```

Листинг 30-1 Инициализация модуля

Добавьте файл с именем `printer.go` в папку `coordination` с содержимым, показанным в листинге 30-2.

```
package main

import "fmt"

func Printfln(template string, values ...interface{}) {
    fmt.Printf(template + "\n", values...)
}
```

Листинг 30-2 Содержимое файла printer.go в папке coordination

Добавьте файл с именем `main.go` в папку `coordination` с содержимым, показанным в листинге 30-3.

```
package main

func doSum(count int, val *int)  {
```

```
    for i := 0; i < count; i++ {
        *val++
    }
}

func main() {
    counter := 0
    doSum(5000, &counter)
    Printfln("Total: %v", counter)
}
```

Листинг 30-3 Содержимое файла main.go в папке coordination

Запустите команду, показанную в листинге 30-4, в папке `coordination`, чтобы скомпилировать и выполнить проект.

```
go run .
```

Листинг 30-4 Компиляция и выполнение проекта

Эта команда выдаст следующий вывод:

```
Total: 5000
```

Использование групп ожидания

Распространенной проблемой является обеспечение того, чтобы `main` функция не завершилась до завершения запускаемых ею горутин, после чего программа завершается. По крайней мере, для меня это обычно происходит, когда горутина вводится в существующий код, как показано в листинге 30-5.

```
package main

func doSum(count int, val *int)  {
    for i := 0; i < count; i++ {
        *val++
    }
}

func main() {
    counter := 0
```

```
    go doSum(5000, &counter)
    Println("Total: %v", counter)
}
```

Листинг 30-5 Представляем горутину в файле main.go в папке coordination

Горутины настолько просты в создании, что легко забыть об их влиянии. В этом случае выполнение main функции продолжается параллельно с горутиной, что означает, что последний оператор в main функции выполняется до того, как горутина завершит выполнение функции doSum, производя следующий вывод, когда проект компилируется и выполняется:

```
Total: 0
```

Пакет sync предоставляет структуру WaitGroup, которую можно использовать для ожидания завершения одной или нескольких горутин с помощью методов, описанных в таблице 30-3.

Таблица 30-3 Методы, определяемые структурой WaitGroup

Функция	Описание
Add(num)	Этот метод увеличивает количество горутин, которые ожидает WaitGroup, на указанное int.
Done()	Этот метод уменьшает количество горутин, которые ожидает WaitGroup, на одну.
Wait()	Этот метод блокируется до тех пор, пока метод Done не будет вызван один раз для общего числа горутин, указанного вызовами метода Add.

WaitGroup действует как счетчик. При создании горутин вызывается метод Add для указания количества запущенных горутин, который увеличивает счетчик, после чего вызывается метод Wait, который блокирует. По завершении каждой горутины вызывается метод Done, который уменьшает значение счетчика. Когда счетчик равен нулю, метод Wait прекращает блокировку, завершая процесс ожидания. Листинг 30-6 добавляет к примеру WaitGroup.

```
package main

import (
    "sync"
)
```

```
var waitGroup = sync.WaitGroup{}

func doSum(count int, val *int)  {
    for i := 0; i < count; i++ {
        *val++
    }
    waitGroup.Done()
}

func main() {
    counter := 0

    waitGroup.Add(1)
    go doSum(5000, &counter)
    waitGroup.Wait()
    Printfln("Total: %v", counter)
}
```

Листинг 30-6 Использование группы ожидания в файле main.go в папке coordination

WaitGroup вызовет панику, если значение счетчика станет отрицательным, поэтому важно вызвать метод Add перед запуском горутины, чтобы предотвратить ранний вызов метода Done. Также важно убедиться, что общее количество значений, переданных методу Add, равно числу вызовов метода Done. Если вызовов Done слишком мало, то метод Wait будет заблокирован навсегда, но если метод Done вызывается слишком много раз, то WaitGroup вызовет панику. В примере есть только одна горутина, но если вы скомпилируете и запустите проект, вы увидите, что она предотвращает преждевременное завершение main функции и выдает следующий вывод:

```
Total: 5000
```

Как избежать ловушки копирования

Важно не копировать значения WaitGroup, потому что это означает, что горутины будут вызывать Done и Wait для разных значений, что обычно означает взаимоблокировку приложения. Если вы хотите передать WaitGroup в качестве аргумента функции, это означает, что вам нужно использовать указатель, например:

```
package main

import (
    "sync"
)

        func  doSum(count  int,  val  *int,  waitGroup  *
sync.WaitGroup)  {
    for i := 0; i < count; i++ {
        *val++
    }
    waitGroup.Done()
}

func main() {
    counter := 0

    waitGroup := sync.WaitGroup{}

    waitGroup.Add(1)
    go doSum(5000, &counter, &waitGroup)
    waitGroup.Wait()
    Printfln("Total: %v", counter)
}
```

Это относится ко всем структурам, описанным в этом разделе. Как правило, координация требует, чтобы все горутины использовали одно и то же значение структуры.

Использование взаимного исключения

Если несколько горутин обращаются к одним и тем же данным, возможно, что две горутины будут обращаться к этим данным одновременно и приведут к неожиданным результатам. В качестве простой демонстрации в листинге 30-7 увеличено количество горутин, используемых в примере.

```
package main

import (
```

```
    "sync"
    "time"
)

var waitGroup = sync.WaitGroup{}

func doSum(count int, val *int)  {
    time.Sleep(time.Second)
    for i := 0; i < count; i++ {
        *val++
    }
    waitGroup.Done()
}

func main() {
    counter := 0

    numRoutines := 3
    waitGroup.Add(numRoutines)
    for i := 0; i < numRoutines; i++ {
        go doSum(5000, &counter)
    }
    waitGroup.Wait()
    Printfln("Total: %v", counter)
}
```

Листинг 30-7 Использование дополнительных горутин в файле main.go в папке coordination

Этот листинг увеличивает количество горутин, которые выполняют функцию `doSum`, и все они обращаются к одной и той же переменной в одно и то же время. (Вызов функции `time.Sleep` обеспечивает одновременный запуск всех горутин, что помогает подчеркнуть проблему, рассмотренную в этом разделе, но не то, что вам следует делать в реальных проектах.) Скомпилируйте и выполните проект, и вы увидите следующий вывод:

```
Total: 12129
```

Вы можете увидеть другой результат, и повторный запуск проекта может каждый раз давать разные результаты. Вы можете получить правильный результат — 15 000, поскольку есть три горутины, каждая из которых выполняет 5 000 операций, — но на моей машине такое

случается редко. Это поведение может отличаться в разных операционных системах. В моем простом тестировании я постоянно сталкивался с проблемами в Windows, в то время как Linux работал чаще.

Проблема в том, что оператор приращения не является атомарным, а это означает, что для его выполнения требуется несколько шагов: переменная `counter` читается, увеличивается и записывается. Это упрощение, но проблема в том, что эти шаги выполняются горутинами параллельно и начинают перекрываться, как показано на рисунке 30-1.

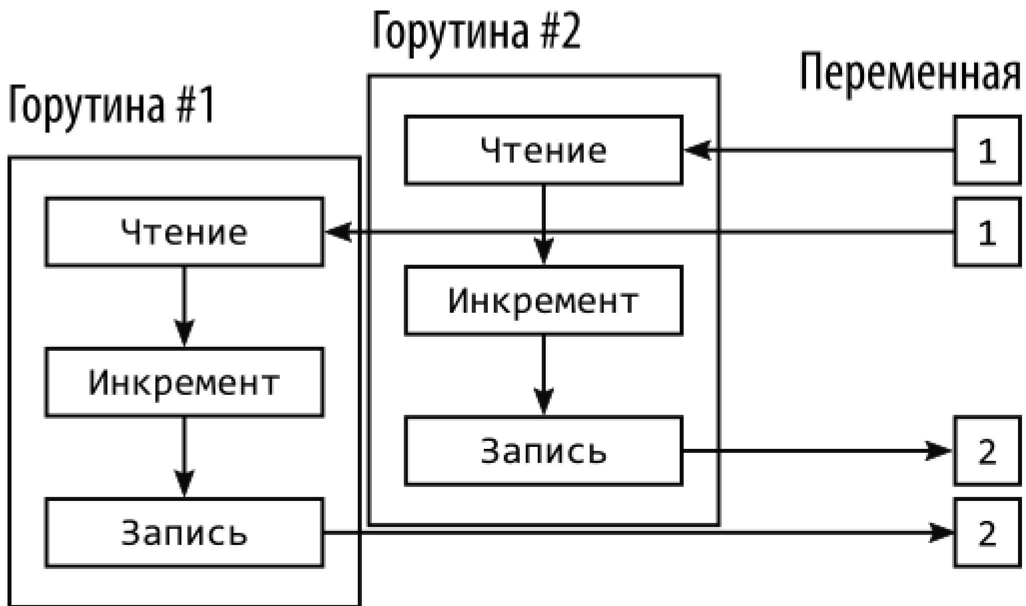

Рисунок 30-1 Несколько горутин обращаются к одной и той же переменной

Вторая горутина считывает значение до того, как первая горутина сможет его обновить, что означает, что обе горутины пытаются увеличить одно и то же значение. В результате обе горутины выдают один и тот же результат и записывают одно и то же значение. Это только одна из потенциальных проблем, которые может вызвать совместное использование данных между горутинами, но все такие проблемы возникают из-за того, что операции требуют времени для выполнения, в течение которого другие горутины также пытаются работать с данными.

Одним из способов решения этой проблемы является *взаимное исключение*, которое гарантирует, что горутина имеет эксклюзивный

доступ к требуемым данным, и предотвращает доступ к этим данным другим горутинам.

Взаимное исключение — это как взять книгу в библиотеке. Только один человек может взять книгу в любой момент времени, а все остальные люди, которым нужна эта книга, должны ждать, пока первый человек не закончит, после чего книгу может взять кто-то другой.

Пакет `sync` обеспечивает взаимное исключение с помощью структуры `Mutex`, которая определяет методы, описанные в таблице 30-4.

Таблица 30-4 Методы, определяемые структурой Mutex

Функция	Описание
`Lock()`	Этот метод блокирует `Mutex`. Если `Mutex` уже заблокирован, этот метод блокируется до тех пор, пока он не будет разблокирован.
`Unlock()`	Этот метод разблокирует `Mutex`.

В листинге 30-8 используется `Mutex` для решения проблемы с примером.

Примечание
Стандартная библиотека включает пакет `sync/atomic`, определяющий функции для низкоуровневых операций, таких как приращение целого числа, атомарным образом, что означает, что они не подвержены проблемам, показанным на рисунке 30-1. Я не описывал эти функции, потому что их сложно правильно использовать, а также потому, что команда разработчиков Go рекомендует вместо этого использовать функции, описанные в этой главе.

```
package main

import (
    "sync"
    "time"
)
```

```
var waitGroup = sync.WaitGroup{}
var mutex = sync.Mutex{}

func doSum(count int, val *int)  {
    time.Sleep(time.Second)
    for i := 0; i < count; i++ {
        mutex.Lock()
        *val++
        mutex.Unlock()
    }
    waitGroup.Done()
}

func main() {
    counter := 0

    numRoutines := 3
    waitGroup.Add(numRoutines)
    for i := 0; i < numRoutines; i++ {
        go doSum(5000, &counter)
    }
    waitGroup.Wait()
    Printfln("Total: %v", counter)
}
```

Листинг 30-8 Использование мьютекса в файле main.go в папке coordination

`Mutex` разблокируется при его создании, а это означает, что первая горутина, вызывающая метод `Lock`, не будет блокироваться и сможет увеличивать переменную `counter`. Говорят, что горутина *получила блокировку*. Любая другая горутина, вызывающая метод `Lock`, будет блокироваться до тех пор, пока не будет вызван метод `Unlock`, известный как снятие блокировки, после чего другая горутина сможет получить блокировку и продолжить доступ к переменной `counter`. В результате только одна горутина одновременно может увеличивать переменную, как показано на рисунке 30-2.

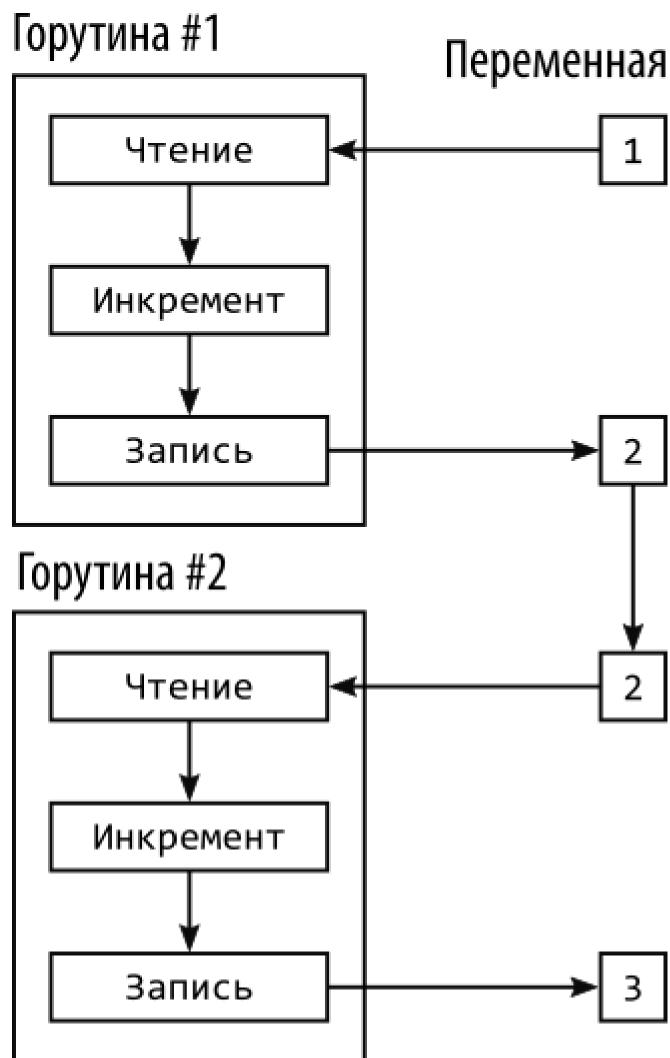

Рисунок 30-2 Использование взаимного исключения

Скомпилируйте и выполните проект, и вы увидите следующий вывод, показывающий, что горутины смогли правильно увеличить переменную `counter`:

```
Total: 15000
```

Следует соблюдать осторожность при использовании взаимного исключения, и важно продумать влияние того, как используется мьютекс. Например, в листинге 30-8 я блокировал и разблокировал мьютекс каждый раз, когда переменная увеличивалась. Использование мьютекса оказывает влияние, и альтернативный подход состоит в том, чтобы заблокировать мьютекс перед выполнением цикла `for`, как показано в листинге 30-9.

```
...
func doSum(count int, val *int)  {
    time.Sleep(time.Second)
    mutex.Lock()
    for i := 0; i < count; i++ {
        *val++
    }
    mutex.Unlock()
    waitGroup.Done()
}
...
```

Листинг 30-9 Выполнение меньшего количества операций с мьютексом в файле main.go в папке coordination

Это более разумный подход для такого простого примера, но обычно ситуация оказывается более сложной, и блокировка больших участков кода может сделать приложения менее отзывчивыми и снизить общую производительность. Мой совет — начать блокировать только операторы, которые обращаются к общим данным.

> **Как избежать ловушек с мьютексом**
> Лучший подход к использованию взаимного исключения — быть осторожным и консервативным. Вы должны убедиться, что весь код, который обращается к общим данным, использует один и тот же `Mutex`, и каждый вызов метода `Lock` должен быть сбалансирован вызовом метода `Unlock`. Может возникнуть соблазн попытаться создать умные усовершенствования или оптимизации, но это может привести к снижению производительности или взаимоблокировкам приложений.

Использование мьютекса чтения-записи

`Mutex` рассматривает все горутины как равные и позволяет только одной горутине получить блокировку. Структура `RWMutex` более гибкая и поддерживает две категории горутин: чтения и записи. Любое количество читателей может получить блокировку одновременно, или один писатель может получить блокировку. Идея состоит в том, что читатели заботятся только о конфликтах с писателями и могут без

труда работать одновременно с другими читателями. Структура RWMutex определяет методы, описанные в таблице 30-5.

Таблица 30-5 Методы, определенные RWMutex

Функция	Описание
RLock()	Этот метод пытается получить блокировку чтения и будет блокироваться до тех пор, пока она не будет получена.
RUnlock()	Этот метод снимает блокировку чтения.
Lock()	Этот метод пытается получить блокировку записи и будет блокироваться, пока она не будет получена.
Unlock()	Этот метод снимает блокировку записи.
RLocker()	Этот метод возвращает указатель на Locker для получения и снятия блокировки чтения, как описано в разделе «Использование условий для координации горутин».

RWMutex не так сложен, как может показаться. Вот правила, которым следует RWMutex:

- Если RWMutex разблокирован, то блокировку может получить читатель (вызвав метод RLock) или писатель (вызвав метод Lock).
- Если блокировка получена читателем, другие читатели также могут получить блокировку, вызвав метод RLock, который не будет блокироваться. Метод Lock будет блокироваться до тех пор, пока все считыватели не снимут блокировку, вызвав метод RUnlock.
- Если блокировка получена модулем записи, то оба метода RLock и Lock будут заблокированы, чтобы предотвратить получение блокировки другими горутинами до тех пор, пока не будет вызван метод Unlock.
- Если блокировка получена модулем чтения, а модуль записи вызывает метод Lock, оба метода Lock и RLock будут блокироваться до тех пор, пока не будет вызван метод Unlock. Это предотвращает постоянную блокировку мьютекса читателями, не давая шанса писателям получить блокировку записи.

В листинге 30-10 показано использование RWMutex.

```
package main

import (
```

```
    "sync"
    "time"
    "math"
    "math/rand"
)

var waitGroup = sync.WaitGroup{}
var rwmutex = sync.RWMutex{}

var squares = map[int]int {}

func calculateSquares(max, iterations int) {
    for i := 0; i < iterations; i++ {
        val := rand.Intn(max)
        rwmutex.RLock();
        square, ok := squares[val]
        rwmutex.RUnlock()
        if (ok) {
            Printfln("Cached value: %v = %v", val, square)
        } else {
            rwmutex.Lock()
            if _, ok := squares[val]; !ok {
                squares[val] = int(math.Pow(float64(val), 2))
                      Printfln("Added value: %v = %v", val,
squares[val])
            }
            rwmutex.Unlock()
        }
    }
    waitGroup.Done()
}

func main() {
    rand.Seed(time.Now().UnixNano())
    //counter := 0
    numRoutines := 3
    waitGroup.Add(numRoutines)
    for i := 0; i < numRoutines; i++ {
        go calculateSquares(10, 5)
    }
    waitGroup.Wait()
    Printfln("Cached values: %v", len(squares))
}
```

Листинг 30-10 Использование RWMutex в файле main.go в папке coordination

Функция calculateSquares получает блокировку чтения, чтобы проверить, содержит ли карта случайно выбранный ключ. Если карта содержит ключ, связанное значение считывается, и блокировка чтения снимается. Если карта не содержит ключа, то устанавливается блокировка записи, к карте добавляется значение для ключа, а затем блокировка записи снимается.

Использование RWMutex означает, что когда одна горутина имеет блокировку чтения, другие подпрограммы также могут получить блокировку и выполнить чтение. Чтение данных не вызывает проблем с параллелизмом, если только они не изменяются одновременно. Если горутина вызывает метод Lock, она не сможет получить блокировку записи до тех пор, пока блокировка чтения не будет снята всеми горутинами, которые ее получили.

Обратите внимание, что горутины освобождают блокировку чтения перед получением блокировки записи в листинге 30-10. RWMutex не поддерживает обновление с блокировки чтения до блокировки записи, с которой вы, возможно, сталкивались в других языках, и вы должны снять блокировку чтения перед вызовом метода Lock, чтобы избежать взаимоблокировки. Между освобождением блокировки чтения и получением блокировки записи может быть задержка, в течение которой другие горутины могут получить блокировку записи и внести изменения, поэтому важно убедиться, что состояние данных не изменилось после блокировки записи. приобрел вот так:

```
...
rwmutex.Lock()
if _, ok := squares[val]; !ok {
    squares[val] = int(math.Pow(float64(val), 2))
...
```

Скомпилируйте и выполните проект, и вы увидите вывод, аналогичный следующему, хотя конкретные результаты определяются случайно выбранными ключами:

```
Added value: 6 = 36
Added value: 2 = 4
Added value: 7 = 49
```

```
Cached value: 7 = 49
Added value: 8 = 64
Cached value: 6 = 36
Added value: 1 = 1
Cached value: 1 = 1
Added value: 3 = 9
Cached value: 8 = 64
Cached value: 8 = 64
Cached value: 1 = 1
Cached value: 1 = 1
Added value: 5 = 25
Cached values: 7
```

Использование условий для координации горутин

Горутины в предыдущем примере используют одни и те же данные, но в остальном они независимы друг от друга. Когда горутинам требуется координация, например ожидание какого-либо события, можно использовать структуру `Cond`. Пакет `sync` предоставляет функцию, описанную в таблице 30-6, для создания значений структуры `Cond`.

Таблица 30-6 Функция sync для создания значений Cond

Функция	Описание
`NewCond(*locker)`	Эта функция создает `Cond`, используя указатель на указанный `Locker`.

Аргументом функции `NewCond` является `Locker`, представляющий собой интерфейс, определяющий методы, описанные в таблице 30-7.

Таблица 30-7 Методы, определяемые интерфейсом Locker

Функция	Описание
`Lock()`	Этот метод получает блокировку, управляемую `Locker`.
`Unlock()`	Этот метод снимает блокировку, управляемую `Locker`.

Структуры `Mutex` и `RWMutex` определяют метод, требуемый интерфейсом `Locker`. В случае `RWMutex` методы `Lock` и `Unlock` работают с блокировкой записи, а метод `RLocker` можно использовать

для получения Locker, который работает с блокировкой чтения. В таблице 30-8 описаны поля и методы, определенные структурой Cond.

Таблица 30-8 Поле и методы, определяемые структурой Cond

Функция	Описание
L	Это поле возвращает Locker, который был передан функции NewCond и используется для получения блокировки.
Wait()	Этот метод снимает блокировку и приостанавливает горутину.
Signal()	Этот метод пробуждает одну ожидающую горутину.
Broadcast()	Этот метод пробуждает все ожидающие горутины.

В листинге 30-11 показано использование Cond для уведомления ожидающих горутин о событии.

```
package main

import (
    "sync"
    "time"
    "math"
    "math/rand"
)

var waitGroup = sync.WaitGroup{}
var rwmutex = sync.RWMutex{}
var readyCond = sync.NewCond(rwmutex.RLocker())

var squares = map[int]int {}

func generateSquares(max int) {
    rwmutex.Lock()
    Printfln("Generating data...")
    for val := 0; val < max; val++ {
        squares[val] = int(math.Pow(float64(val), 2))
    }
    rwmutex.Unlock()
    Printfln("Broadcasting condition")
    readyCond.Broadcast()
    waitGroup.Done()
}
```

```
func readSquares(id, max, iterations int) {
    readyCond.L.Lock()
    for len(squares) == 0 {
        readyCond.Wait()
    }
    for i := 0; i < iterations; i++ {
        key := rand.Intn(max)
            Printfln("#%v  Read  value:  %v  =  %v",  id,  key,
squares[key])
        time.Sleep(time.Millisecond * 100)
    }
    readyCond.L.Unlock()
    waitGroup.Done()
}

func main() {
    rand.Seed(time.Now().UnixNano())
    numRoutines := 2
    waitGroup.Add(numRoutines)
    for i := 0; i < numRoutines; i++ {
        go readSquares(i, 10, 5)
    }

    waitGroup.Add(1)
    go generateSquares(10)
    waitGroup.Wait()
    Printfln("Cached values: %v", len(squares))
}
```

Листинг 30-11 Использование Cond в файле main.go в папке coordination

Этот пример требует координации между горутинами, чего было бы трудно достичь без Cond. Одна горутина отвечает за заполнение карты значениями данных, которые затем считываются другими горутинами. Читатели требуют уведомления о том, что генерация данных завершена, прежде чем они запустятся.

Читатели ждут, получая блокировку Cond и вызывая метод Wait, например:

```
...
readyCond.L.Lock()
for len(squares) == 0 {
```

```
    readyCond.Wait()
}
...
```

Вызов метода `Wait` приостанавливает горутину и освобождает блокировку, чтобы ее можно было получить. Вызов метода `Wait` обычно выполняется внутри цикла `for`, который проверяет выполнение условия, которого ожидает горутина, просто для того, чтобы убедиться, что данные находятся в ожидаемом состоянии.

Нет необходимости снова получать блокировку, когда метод `Wait` разблокируется, и горутина может либо снова вызвать метод `Wait`, либо получить доступ к общим данным. Когда закончите с общими данными, блокировка должна быть снята:

```
...
readyCond.L.Unlock()
...
```

Горутина, генерирующая данные, получает блокировку записи с помощью `RWMutex`, изменяет данные, снимает блокировку записи, а затем вызывает метод `Cond.Broadcast`, который пробуждает все ожидающие горутины. Скомпилируйте и выполните проект, и вы увидите результат, аналогичный следующему, с учетом выбранных случайных значений ключа:

```
Generating data...
Broadcasting condition
#0 Read value: 4 = 16
#1 Read value: 1 = 1
#1 Read value: 5 = 25
#0 Read value: 6 = 36
#0 Read value: 2 = 4
#1 Read value: 2 = 4
#1 Read value: 6 = 36
#0 Read value: 6 = 36
#0 Read value: 6 = 36
#1 Read value: 8 = 64
Cached values: 10
```

Вызов функции time.Sleep в функции readSquares замедляет процесс чтения данных, так что обе горутины чтения обрабатывают данные одновременно, что вы можете видеть по чередованию первого числа в выходных строках. Поскольку эти горутины получают блокировку чтения RWMutex, они одновременно получают блокировку и могут читать данные. В листинге 30-12 изменяется тип блокировки, используемой Cond.

```
...
var waitGroup = sync.WaitGroup{}
var rwmutex = sync.RWMutex{}
var readyCond = sync.NewCond(&rwmutex)
...
```

Листинг 30-12 Изменение типа блокировки в файле main.go в папке coordination

Это изменение означает, что все горутины используют блокировку записи, а это означает, что только одна горутина сможет получить блокировку. Скомпилируйте и выполните проект, и вы увидите, что вывод больше не чередуется:

```
Generating data...
Broadcasting condition
#0 Read value: 5 = 25
#0 Read value: 8 = 64
#0 Read value: 9 = 81
#0 Read value: 0 = 0
#0 Read value: 4 = 16
#1 Read value: 7 = 49
#1 Read value: 8 = 64
#1 Read value: 5 = 25
#1 Read value: 8 = 64
#1 Read value: 5 = 25
Cached values: 10
```

Обеспечение однократного выполнения функции

Альтернативный подход к предыдущему примеру заключается в обеспечении однократного выполнения функции generateSquares с

использованием структуры `sync.Once`. Структура `Once` определяет один метод, описанный в таблице 30-9.

Таблица 30-9 Метод Once

Функция	Описание
`Do(func)`	Этот метод выполняет указанную функцию, но только если она еще не была выполнена.

В листинге 30-13 показано использование структуры `Once`.

```
package main

import (
    "sync"
    "time"
    "math"
    "math/rand"
)

var waitGroup = sync.WaitGroup{}
//var rwmutex = sync.RWMutex{}
//var readyCond = sync.NewCond(rwmutex.RLocker())
var once = sync.Once{}

var squares = map[int]int {}

func generateSquares(max int) {
    //rwmutex.Lock()
    Printfln("Generating data...")
    for val := 0; val < max; val++ {
        squares[val] = int(math.Pow(float64(val), 2))
    }
    // rwmutex.Unlock()
    // Printfln("Broadcasting condition")
    // readyCond.Broadcast()
    // waitGroup.Done()
}

func readSquares(id, max, iterations int) {
    once.Do(func () {
        generateSquares(max)
```

```
    })
    // readyCond.L.Lock()
    // for len(squares) == 0 {
    //     readyCond.Wait()
    // }
    for i := 0; i < iterations; i++ {
        key := rand.Intn(max)
             Printfln("#%v  Read  value:  %v  =  %v",  id,  key,
squares[key])
        time.Sleep(time.Millisecond * 100)
    }
    //readyCond.L.Unlock()
    waitGroup.Done()
}

func main() {
    rand.Seed(time.Now().UnixNano())
    numRoutines := 2
    waitGroup.Add(numRoutines)
    for i := 0; i < numRoutines; i++ {
        go readSquares(i, 10, 5)
    }
    // waitGroup.Add(1)
    // go generateSquares(10)
    waitGroup.Wait()
    Printfln("Cached values: %v", len(squares))
}
```

Листинг 30-13 Выполнение функции один раз в файле main.go в папке coordination

Использование структуры `Once` упрощает пример, поскольку метод `Do` блокируется до тех пор, пока функция, которую он получает, не будет выполнена, после чего он возвращается без повторного выполнения функции. Поскольку единственные изменения общих данных в этом примере вносятся функцией `generateSquares`, использование метода `Do` для выполнения этой функции гарантирует безопасное внесение изменений. Не весь код так хорошо соответствует модели `Once`, но в этом примере я могу удалить `RWMutex` и `Cond`. Скомпилируйте и запустите проект, и вы увидите вывод, подобный следующему:

```
Generating data...
```

```
#1 Read value: 0 = 0
#0 Read value: 0 = 0
#0 Read value: 4 = 16
#1 Read value: 9 = 81
#1 Read value: 2 = 4
#0 Read value: 9 = 81
#0 Read value: 8 = 64
#1 Read value: 3 = 9
#1 Read value: 7 = 49
#0 Read value: 3 = 9
Cached values: 10
```

Использование контекстов

Go упрощает создание серверных приложений, которые получают запросы от имени клиентов и обрабатывают их в собственной горутине. Пакет `context` предоставляет интерфейс `Context`, упрощающий управление запросами с помощью методов, описанных в таблице 30-10.

Таблица 30-10 Методы, определяемые интерфейсом Context

Функция	Описание
`Value(key)`	Этот метод возвращает значение, связанное с указанным ключом.
`Done()`	Этот метод возвращает канал, который можно использовать для получения уведомления об отмене.
`Deadline()`	Этот метод возвращает `time.Time`, представляющий крайний срок для запроса, и логическое значение, которое будет `false`, если крайний срок не указан.
`Err()`	Этот метод возвращает `error`, указывающую, почему канал `Done` получил сигнал. Пакет `context` определяет две переменные, которые можно использовать для сравнения ошибок: `Canceled` указывает, что запрос был отменен, а `DeadlineExeeded` указывает, что срок истек.

Пакет `context` предоставляет функции, описанные в таблице 30-11, для создания значений контекста.

Таблица 30-11 Функции пакета context для создания значений контекста

Функция	Описание
`Background()`	Этот метод возвращает `Context` по умолчанию, из которого получаются другие контексты.

Функция	Описание
`WithCancel(ctx)`	Этот метод возвращает контекст и функцию отмены, как описано в разделе «Отмена запроса».
`WithDeadline(ctx, time)`	Этот метод возвращает контекст с крайним сроком, который выражается с помощью значения `time.Time`, как описано в разделе «Установка крайнего срока».
`WithTimeout(ctx, duration)`	Этот метод возвращает контекст с крайним сроком, который выражается с помощью значения `time.Duration`, как описано в разделе «Установка крайнего срока».
`WithValue(ctx, key, val)`	Этот метод возвращает контекст, содержащий указанную пару ключ-значение, как описано в разделе «Предоставление данных запроса».

Чтобы подготовиться к этому разделу, в листинге 30-14 определяется функция, имитирующая обработку запросов.

```
package main

import (
    "sync"
    "time"
    // "math"
    // "math/rand"
)

func processRequest(wg *sync.WaitGroup, count int) {
    total := 0
    for i := 0; i < count; i++ {
        Printfln("Processing request: %v", total)
        total++
        time.Sleep(time.Millisecond * 250)
    }
    Printfln("Request processed...%v", total)
    wg.Done()
}

func main() {
    waitGroup := sync.WaitGroup {}
    waitGroup.Add(1)
    Printfln("Request dispatched...")
    go processRequest(&waitGroup, 10)
    waitGroup.Wait()
}
```

Листинг 30-14 Симуляция обработки запросов в файле main.go в папке coordination

Функция `processRequest` имитирует обработку запроса путем увеличения счетчика с вызовом функции `time.Sleep` для замедления всего процесса. Функция `main` использует горутину для вызова функции `processRequest` вместо запроса, поступающего от клиента. (См. часть 3 для примера, который обрабатывает фактические запросы. Этот раздел как раз о том, как работают контексты.) Скомпилируйте и выполните проект, и вы увидите следующий вывод:

```
Request dispatched...
Processing request: 0
Processing request: 1
Processing request: 2
Processing request: 3
Processing request: 4
Processing request: 5
Processing request: 6
Processing request: 7
Processing request: 8
Processing request: 9
Request processed...10
```

Отмена запроса

Первое использование `Context` — уведомление кода, обрабатывающего запрос, об отмене запроса, как показано в листинге 30-15.

```
package main

import (
    "sync"
    "time"
    // "math"
    // "math/rand"
    "context"
)

func processRequest(ctx context.Context, wg *sync.WaitGroup,
count int) {
    total := 0
```

```
    for i := 0; i < count; i++ {
        select {
            case <- ctx.Done():
                 Printfln("Stopping processing - request
cancelled")
                goto end
            default:
                Printfln("Processing request: %v", total)
                total++
                time.Sleep(time.Millisecond * 250)
        }
    }
    Printfln("Request processed...%v", total)
    end:
    wg.Done()
}

func main() {
    waitGroup := sync.WaitGroup {}
    waitGroup.Add(1)
    Printfln("Request dispatched...")
    ctx, cancel := context.WithCancel(context.Background())
    go processRequest(ctx, &waitGroup, 10)

    time.Sleep(time.Second)
    Printfln("Canceling request")
    cancel()

    waitGroup.Wait()
}
```

Листинг 30-15 Отмена запроса в файле main.go в папке coordination

Функция `Background` возвращает `Context` по умолчанию, который не делает ничего полезного, но предоставляет отправную точку для получения новых значений `Context` с помощью других функций, описанных в таблице 30-11. Функция `WithCancel` возвращает контекст, который можно отменить, и функцию, которая вызывается для выполнения отмены:

```
...
ctx, cancel := context.WithCancel(context.Background())
```

```
go processRequest(ctx, &waitGroup, 10)
...
```

Полученный контекст передается функции `processRequest`. Функция `main` вызывает функцию `time.Sleep`, чтобы дать функции `processRequest` изменение для выполнения некоторой работы, а затем вызывает функцию отмены:

```
...
time.Sleep(time.Second)
Printfln("Canceling request")
cancel()
...
```

Вызов функции отмены отправляет сообщение в канал, возвращаемый контекстным методом `Done`, который отслеживается с помощью оператора `select`:

```
...
case <- ctx.Done():
    Printfln("Stopping processing - request cancelled")
    goto end
default:
    Printfln("Processing request: %v", total)
    total++
    time.Sleep(time.Millisecond * 250)
}
...
```

Канал `Done` блокируется, если запрос не был отменен, поэтому будет выполнено предложение `default`, позволяющее обработать запрос. Канал проверяется после каждой единицы работы, и оператор `goto` используется для выхода из цикла обработки, чтобы можно было передать сигнал `WaitGroup` и завершить функцию. Скомпилируйте и выполните проект, и вы увидите, что имитируемая обработка запроса завершается досрочно, как показано ниже:

```
Request dispatched...
Processing request: 0
Processing request: 1
Processing request: 2
```

```
Processing request: 3
Canceling request
Stopping processing - request cancelled
```

Установка крайнего срока

Контексты могут создаваться с дедлайном, по истечении которого по каналу Done отправляется сигнал, как если бы запрос был отменен. Абсолютное время можно указать с помощью функции WithDeadline, которая принимает значение time.Time, или, как показано в листинге 30-16, функция WithTimeout принимает time.Duration, указывающее крайний срок относительно текущего времени. Метод Context.Deadline можно использовать для проверки крайнего срока во время обработки запроса.

```
package main

import (
    "sync"
    "time"
    // "math"
    // "math/rand"
    "context"
)

func processRequest(ctx context.Context, wg *sync.WaitGroup,
count int) {
    total := 0
    for i := 0; i < count; i++ {
        select {
            case <- ctx.Done():
                if (ctx.Err() == context.Canceled) {
                     Printfln("Stopping processing - request
cancelled")
                } else {
                    Printfln("Stopping processing - deadline
reached")
                }
                goto end
            default:
                Printfln("Processing request: %v", total)
                total++
```

```
                time.Sleep(time.Millisecond * 250)
            }
        }
        Printfln("Request processed...%v", total)
        end:
        wg.Done()
}

func main() {
    waitGroup := sync.WaitGroup {}
    waitGroup.Add(1)
    Printfln("Request dispatched...")
        ctx, _ := context.WithTimeout(context.Background(),
time.Second * 2)
    go processRequest(ctx, &waitGroup, 10)

    // time.Sleep(time.Second)
    // Printfln("Canceling request")
    // cancel()

    waitGroup.Wait()
}
```

Листинг 30-16 Указание крайнего срока в файле main.go в папке coordination

Функции `WithDeadline` и `WithTimeout` возвращают производный контекст и функцию отмены, которая позволяет отменить запрос до истечения крайнего срока. В этом примере количество времени, требуемое функцией `processRequest`, превышает крайний срок, что означает, что канал `Done` прекратит обработку. Скомпилируйте и запустите проект, и вы увидите вывод, подобный следующему:

```
Request dispatched...
Processing request: 0
Processing request: 1
Processing request: 2
Processing request: 3
Processing request: 4
Processing request: 5
Processing request: 6
Processing request: 7
Stopping processing - deadline reached
```

Предоставление данных запроса

Функция `WithValue` создает производный `Context` с парой ключ-значение, которую можно прочитать во время обработки запроса, как показано в листинге 30-17..

```
package main

import (
    "sync"
    "time"
    // "math"
    // "math/rand"
    "context"
)

const (
    countKey   = iota
    sleepPeriodKey
)

func processRequest(ctx context.Context, wg *sync.WaitGroup)
{
    total := 0
    count := ctx.Value(countKey).(int)
    sleepPeriod := ctx.Value(sleepPeriodKey).(time.Duration)
    for i := 0; i < count; i++ {
        select {
            case <- ctx.Done():
                if (ctx.Err() == context.Canceled) {
                      Printfln("Stopping processing - request
cancelled")
                } else {
                     Printfln("Stopping processing - deadline
reached")
                }
                goto end
            default:
                Printfln("Processing request: %v", total)
                total++
                time.Sleep(sleepPeriod)
        }
    }
```

```
    Printfln("Request processed...%v", total)
    end:
    wg.Done()
}

func main() {
    waitGroup := sync.WaitGroup {}
    waitGroup.Add(1)
    Printfln("Request dispatched...")
    ctx, _ := context.WithTimeout(context.Background(),
time.Second * 2)
    ctx = context.WithValue(ctx, countKey, 4)
    ctx = context.WithValue(ctx, sleepPeriodKey,
time.Millisecond * 250)
    go processRequest(ctx, &waitGroup)

    // time.Sleep(time.Second)
    // Printfln("Canceling request")
    // cancel()

    waitGroup.Wait()
}
```

Листинг 30-17 Использование данных запроса в файле main.go в папке coordination

Функция `WithValue` принимает только одну пару ключ-значение, но функции в таблице 30-11 можно вызывать многократно, чтобы создать требуемую комбинацию функций. В листинге 30-17 функция `WithTimeout` используется для получения `Context` с крайним сроком, а производный `Context` используется в качестве аргумента функции `WithValue` для добавления двух пар ключ-значение. Доступ к этим данным осуществляется через метод `Value`, что означает, что функциям обработки запросов не нужно определять параметры для всех требуемых значений данных. Скомпилируйте и запустите проект, и вы увидите следующий вывод:

```
Request dispatched...
Processing request: 0
Processing request: 1
Processing request: 2
Processing request: 3
Request processed...4
```

Резюме

В этой главе я описал стандартные библиотечные функции для координации горутин, в том числе использование групп ожидания, которые позволяют одной горутине ждать завершения других, и взаимное исключение, которое не позволяет горутинам изменять одни и те же данные одновременно. Я также описал функцию `Context`, которая позволяет серверу более последовательно обрабатывать запросы. Это функция, которую я неоднократно использую в части 3 этой книги, в которой я создаю пользовательскую среду веб-приложений и интернет-магазин, который ее использует. В следующей главе я опишу поддержку стандартной библиотеки для модульного тестирования.

31. Модульное тестирование, бенчмаркинг и ведение журнала

В этой главе я заканчиваю описание наиболее полезных пакетов стандартных библиотек с модульным тестированием, бенчмаркингом и ведением журнала. Функции ведения журнала хороши, хотя и немного примитивны, и существует множество сторонних пакетов для направления сообщений журнала в разные места назначения. Функции тестирования и бенчмаркинга интегрированы в команду `go`, но, как я уже объяснял, я не в восторге ни от одной из этих функций. Таблица 31-1 суммирует содержание главы.

Таблица 31-1 Краткое содержание главы

Проблема	Решение	Листинг
Создать модульный тест	Добавьте файл, имя которого заканчивается на `_test.go`, определите функцию, имя которой начинается с `Test`, за которым следует заглавная буква, и используйте функции, предоставляемые пакетом `testing`.	4, 6, 7, 10, 11
Запустить модульный тест	Используйте команду `go test`	5, 8, 9
Создать бенчмарк	Определите функцию, имя которой начинается с `Benchmark`, за которым следует заглавная буква.	12, 14, 15
Запустить бенчмарк	Используйте команду `go test` с аргументом `-bench`.	13
Данные журнала	Используйте функции, предоставляемые пакетом `log`	16, 17

Подготовка к этой главе

Чтобы подготовиться к этой главе, откройте новую командную строку, перейдите в удобное место и создайте каталог с именем `tests`. Запустите команду, показанную в листинге 31-1, в папке `tests`, чтобы создать файл модуля.

> **Подсказка**
> Вы можете загрузить пример проекта для этой главы — и для всех остальных глав этой книги — с `https://github.com/apress/pro-go`. См. Главу 2 о том, как получить помощь, если у вас возникнут проблемы с запуском примеров.

```
go mod init tests
```

Листинг 31-1 Инициализация модуля

Добавьте файл с именем `main.go` в папку `tests` с содержимым, показанным в листинге 31-2.

```
package main

import (
    "sort"
    "fmt"
)

func sortAndTotal(vals []int) (sorted []int, total int) {
    sorted = make([]int, len(vals))
    copy(sorted, vals)
    sort.Ints(sorted)
    for _, val := range sorted {
        total += val
        total++
    }
    return
}

func main() {
    nums := []int { 100, 20, 1, 7, 84 }
    sorted, total := sortAndTotal(nums)
    fmt.Println("Sorted Data:", sorted)
    fmt.Println("Total:", total)
}
```

Листинг 31-2 Содержимое файла main.go в папке tests

Функция `sortAndTotal` содержит преднамеренную ошибку, которая поможет продемонстрировать функции тестирования в

следующем разделе. Запустите команду, показанную в листинге 31-3, в папке tests, чтобы скомпилировать и выполнить проект.

```
go run .
```

Листинг 31-3 Компиляция и запуск проекта

Эта команда производит следующий вывод:

```
Sorted Data: [1 7 20 84 100]
Total: 217
```

Использование тестирования

Модульные тесты определяются в файлах, имя которых заканчивается на _test.go. Чтобы создать простой тест, добавьте файл с именем simple_test.go в папку tests, содержимое которого показано в листинге 31-4.

```
package main

import "testing"

func TestSum(t *testing.T) {
    testValues := []int{ 10, 20, 30 }
    _, sum := sortAndTotal(testValues)
    expected := 60
    if (sum != expected) {
        t.Fatalf("Expected %v, Got %v", expected, sum)
    }
}
```

Листинг 31-4 Содержимое файла simple_test.go в папке tests

Стандартная библиотека Go обеспечивает поддержку написания модульных тестов через testing пакет. Модульные тесты выражаются в виде функций, имя которых начинается с Test, за которым следует термин, начинающийся с заглавной буквы, например TestSum. (Заглавная буква важна, потому что инструменты тестирования не распознают имя функции, такое как Testsum, в качестве модульного теста.)

РЕШЕНИЕ ИСПОЛЬЗОВАТЬ ИНСТРУМЕНТЫ ТЕСТИРОВАНИЯ

Мне нравится идея интегрированного тестирования, но я обнаружил, что не очень часто использую функции тестирования Go, а если и использую, то не по назначению.

Мне нравится модульное тестирование, но я пишу тесты только тогда, когда пытаюсь разобраться в коде со сложными проблемами, или когда я пишу функцию, которую, как я знаю, будет сложно реализовать правильно. Возможно, я просто так думаю о тестах или привык к классическому шаблону инструментов тестирования «упорядочить/действовать/утвердить», но есть что-то в инструментах тестирования Go, что мне не нравится.

В итоге я использую тесты, чтобы создавать простые точки входа в определенные пакеты, чтобы убедиться, что они работают правильно. Но даже в этом случае я просто создаю один тест, который использую для создания экземпляров типов в пакете, что позволяет мне получить доступ к полям, функциям и методам, которые они определяют, без необходимости изменять мою `main` функцию. Код в этих тестах всегда представляет собой неряшливый беспорядок, и я использую операторы `println` для вывода вместо методов, описанных в таблице 31-2. Убедившись, что код работает, я удаляю тестовый файл.

Я вполне готов признать, что это моя ошибка, но у меня нет никакого энтузиазма по поводу инструментов тестирования Go. Это не значит, что вы не найдете их полезными, возможно, потому, что вы более прилежный тестер, чем я. Но если вы обнаружите, что функции, описанные в этом разделе, не мотивируют вас писать тесты, знайте, что вы не одиноки.

Функции модульного тестирования получают указатель на структуру `T`, которая определяет методы управления тестами и создания отчетов о результатах тестов. Тесты Go не полагаются на утверждения и пишутся с использованием обычных операторов кода. Все, о чем заботятся инструменты тестирования, — это то, не проходит ли тест, о чем сообщается с помощью методов, описанных в таблице 31-2.

Таблица 31-2 Т методы для отчет о результатах тестирования

Функция	Описание
`Log(...vals)`	Этот метод записывает указанные значения в журнал ошибок теста.
`Logf(template, ...vals)`	Этот метод использует указанный шаблон и значения для записи сообщения в журнал ошибок теста.
`Fail()`	Вызов этого метода помечает тест как не пройденный, но продолжает выполнение теста.
`FailNow()`	Вызов этого метода помечает тест как не пройденный и прекращает его выполнение.
`Failed()`	Этот метод возвращает `true`, если тест не пройден.
`Error(...errs)`	Вызов этого метода эквивалентен вызову метода `Log`, за которым следует метод `Fail`.
`Errorf(template, ...vals)`	Вызов этого метода эквивалентен вызову метода `Logf`, за которым следует метод `Fail`.
`Fatal(...vals)`	Вызов этого метода эквивалентен вызову метода `Log`, за которым следует метод `FailNow`.
`Fatalf(template, ...vals)`	Вызов этого метода эквивалентен вызову метода `Logf`, за которым следует метод `FailNow`.

Тест в листинге 31-4 вызывал функцию `sumAndTotal` с набором значений и сравнивал результат с ожидаемым, используя стандартный оператор сравнения Go. Если результат не равен ожидаемому значению, вызывается метод `Fatalf`, который сообщает о сбое теста и останавливает выполнение всех оставшихся операторов в модульном тесте (хотя в этом примере оставшихся операторов нет).

Понимание доступа к тестовым пакетам

В тестовом файле в листинге 31-4 используется ключевое слово `package` для указания `main` пакета. Поскольку тесты написаны на стандартном Go, это означает, что тесты в этом файле имеют доступ ко всем функциям, определенным в `main` пакете, включая те, которые не экспортируются за пределы пакета.

Если вы хотите написать тесты, которые имеют доступ только к экспортированным функциям, вы можете использовать оператор `package` для указания пакета `main_test`. Суффикс `_test` не вызывает проблем с компилятором и позволяет писать тесты, которые имеют доступ только к экспортированным функциям из тестируемого пакета.

Запуск модульных тестов

Чтобы обнаружить и запустить модульные тесты в проекте, запустите команду, показанную в листинге 31-5, в папке `tests`.

> **НАПИСАНИЕ МАКЕТОВ ДЛЯ МОДУЛЬНЫХ ТЕСТОВ**
>
> Единственный способ создать фиктивные реализации для модульных тестов — это создать реализации интерфейса, которые позволяют определять пользовательские методы, дающие результаты, необходимые для теста. Если вы хотите использовать макеты для своих модульных тестов, вам следует написать свои API, чтобы они принимали типы интерфейса.
>
> Но даже несмотря на то, что использование макетов ограничено интерфейсами, обычно можно создавать структурные значения, полям которых присваиваются определенные значения, которые вы можете проверить. Иногда это может быть немного неловко, но большинство функций и методов так или иначе можно протестировать, даже если требуется некоторое упорство, чтобы разобраться в деталях.

```
go test
```

Листинг 31-5 Выполнение модульных тестов

Как уже отмечалось, в коде, определенном в листинге 31-2, есть ошибка, которая приводит к сбою модульного теста:

```
tests > go test
--- FAIL: TestSum (0.00s)
    simple_test.go:10: Expected 60, Got 63
FAIL
exit status 1
FAIL    tests   0.090s
```

Выходные данные тестов сообщают об ошибке, а также об общем результате выполнения теста. В листинге 31-6 исправлена ошибка в функции `sortAndTotal`.

```
...
func sortAndTotal(vals []int) (sorted []int, total int) {
```

```
    sorted = make([]int, len(vals))
    copy(sorted, vals)
    sort.Ints(sorted)
    for _, val := range sorted {
        total += val
        //total++
    }
    return
}
...
```

Листинг 31-6 Исправление ошибки в файле main.go в папке tests

Сохраните изменения и запустите команду `go test`, и вывод покажет, что тест пройден:

```
PASS
ok      tests   0.102s
```

Тестовый файл может содержать несколько тестов, которые будут обнаружены и выполнены автоматически. В листинге 31-7 добавлена вторая тестовая функция в файл `simple_test.go`.

```
package main

import (
    "testing"
    "sort"
)

func TestSum(t *testing.T) {
    testValues := []int{ 10, 20, 30 }
    _, sum := sortAndTotal(testValues)
    expected := 60
    if (sum != expected) {
        t.Fatalf("Expected %v, Got %v", expected, sum)
    }
}

func TestSort(t *testing.T) {
    testValues := []int{ 1, 279, 48, 12, 3}
    sorted, _ := sortAndTotal(testValues)
    if (!sort.IntsAreSorted(sorted)) {
```

```
        t.Fatalf("Unsorted data %v", sorted)
    }
}
```

Листинг 31-7 Определение теста в файле simple_test.go в папке tests

Тест `TestSort` проверяет, что функция `sortAndTotal` сортирует данные. Обратите внимание, что я могу полагаться на функции, предоставляемые стандартной библиотекой Go в модульных тестах, и использовать функцию `sort.IntsAreSorted` для выполнения теста. Запустите команду `go test`, и вы увидите следующий результат:

```
ok      tests   0.087s
```

Команда `go test` по умолчанию не сообщает никаких подробностей, но дополнительную информацию можно получить, выполнив команду, показанную в листинге 31-8, в папке `tests`.

```
go test  -v
```

Листинг 31-8 Выполнение подробных тестов

Аргумент `-v` включает подробный режим, который сообщает о каждом из тестов:

```
=== RUN   TestSum
--- PASS: TestSum (0.00s)
=== RUN   TestSort
--- PASS: TestSort (0.00s)
PASS
ok      tests   0.164s
```

Запуск определенных тестов

Команду `go test` можно использовать для запуска тестов, выбранных по имени. Запустите команду, показанную в листинге 31-9, в папке `tests`.

```
go test -v -run "um"
```

Листинг 31-9 Выбор тестов в файле main.go в папке tests

Тесты выбираются с помощью регулярного выражения, и команда в листинге 31-9 выбирает тесты, имя функции которых содержит um (нет необходимости включать часть Test в имя функции). Единственным тестом, имя которого соответствует выражению, является TestSum, и команда выводит следующий результат:

```
=== RUN   TestSum
--- PASS: TestSum (0.00s)
PASS
ok      tests   0.123s
```

Управление выполнением теста

Структура T также предоставляет набор методов для управления выполнением тестов, как описано в таблице 31-3.

Таблица 31-3 Т методы для управления выполнением теста

Функция	Описание
Run(name, func)	Вызов этого метода выполняет указанную функцию как подтест. Метод блокируется, пока тест выполняется в собственной горутине, и возвращает bool значение, указывающее, успешно ли прошел тест.
SkipNow()	Вызов этого метода останавливает выполнение теста и помечает его как пропущенный.
Skip(...args)	Этот метод эквивалентен вызову метода Log, за которым следует метод SkipNow.
Skipf(template, ...args)	Этот метод эквивалентен вызову метода Logf, за которым следует метод SkipNow.
Skipped()	Этот метод возвращает true, если тест был пропущен.

Метод Run используется для выполнения подтеста, что является удобным способом запуска серии связанных тестов из одной функции, как показано в листинге 31-10.

```
package main

import (
    "testing"
    "sort"
    "fmt"
)
```

```
func TestSum(t *testing.T) {
    testValues := []int{ 10, 20, 30 }
    _, sum := sortAndTotal(testValues)
    expected := 60
    if (sum != expected) {
        t.Fatalf("Expected %v, Got %v", expected, sum)
    }
}

func TestSort(t *testing.T) {
    slices := [][]int {
        { 1, 279, 48, 12, 3 },
        { -10, 0, -10 },
        { 1, 2, 3, 4, 5, 6, 7 },
        { 1 },
    }
    for index, data := range slices {
            t.Run(fmt.Sprintf("Sort #%v", index), func(subT
*testing.T) {
            sorted, _ := sortAndTotal(data)
            if (!sort.IntsAreSorted(sorted)) {
                subT.Fatalf("Unsorted data %v", sorted)
            }
        })
    }
}
```

Листинг 31-10 Запуск подтестов в файле simple_test.go в папке tests

Аргументами метода `Run` являются имя теста и функция, которая принимает структуру `T` и выполняет тест. В листинге 31-10 метод `Run` используется для проверки правильности сортировки набора различных срезов `int`. Используйте команду `go test -v` для запуска тестов с подробным выводом, и вы увидите следующий вывод:

```
=== RUN   TestSum
--- PASS: TestSum (0.00s)
=== RUN   TestSort
=== RUN   TestSort/Sort_#0
=== RUN   TestSort/Sort_#1
=== RUN   TestSort/Sort_#2
=== RUN   TestSort/Sort_#3
--- PASS: TestSort (0.00s)
```

```
    --- PASS: TestSort/Sort_#0 (0.00s)
    --- PASS: TestSort/Sort_#1 (0.00s)
    --- PASS: TestSort/Sort_#2 (0.00s)
    --- PASS: TestSort/Sort_#3 (0.00s)
PASS
ok      tests   0.112s
```

Пропуск тестов

Тесты можно пропустить, используя методы, описанные в таблице 31-3, что может быть полезно, когда сбой одного теста означает, что нет смысла выполнять связанные тесты, как показано в листинге 31-11.

```
package main

import (
    "testing"
    "sort"
    "fmt"
)

type SumTest struct {
    testValues []int
    expectedResult int
}

func TestSum(t *testing.T) {
    testVals := []SumTest {
                      { testValues: []int{10, 20, 30},
expectedResult: 10},
                  { testValues: []int{ -10, 0, -10 },
expectedResult: -20},
                  { testValues: []int{ -10, 0, -10 },
expectedResult: -20},
    }
    for index, testVal := range testVals {
            t.Run(fmt.Sprintf("Sum #%v", index), func(subT
*testing.T) {
            if (t.Failed()) {
                subT.SkipNow()
            }
            _, sum := sortAndTotal(testVal.testValues)
            if (sum != testVal.expectedResult) {
```

```
                        subT.Fatalf("Expected  %v,  Got  %v",
testVal.expectedResult, sum)
            }
        })
    }
}

func TestSort(t *testing.T) {
    slices := [][]int {
        { 1, 279, 48, 12, 3 },
        { -10, 0, -10 },
        { 1, 2, 3, 4, 5, 6, 7 },
        { 1 },
    }
    for index, data := range slices {
            t.Run(fmt.Sprintf("Sort  #%v",  index),  func(subT
*testing.T) {
            sorted, _ := sortAndTotal(data)
            if (!sort.IntsAreSorted(sorted)) {
                subT.Fatalf("Unsorted data %v", sorted)
            }
        })
    }
}
```

Листинг 31-11 Пропуск тестов в файле simple_test.go в папке tests

Функция `TestSum` была переписана для запуска подтестов. При использовании подтестов, если какой-либо отдельный тест дает сбой, то и общий тест также не проходит. В листинге 31-11 я полагаюсь на это поведение, вызывая метод `Failed` в структуре `T` для общего теста и используя метод `SkipNow` для пропуска подтестов после сбоя. Ожидаемый результат, определенный для первого подтеста, выполненного `TestSum`, неверен и приводит к сбою теста, что приводит к следующему выводу при использовании команды `go test -v`:

```
=== RUN   TestSum
=== RUN   TestSum/Sum_#0
    simple_test.go:27: Expected 10, Got 60
=== RUN   TestSum/Sum_#1
=== RUN   TestSum/Sum_#2
```

```
--- FAIL: TestSum (0.00s)
    --- FAIL: TestSum/Sum_#0 (0.00s)
    --- SKIP: TestSum/Sum_#1 (0.00s)
    --- SKIP: TestSum/Sum_#2 (0.00s)
=== RUN   TestSort
=== RUN   TestSort/Sort_#0
=== RUN   TestSort/Sort_#1
=== RUN   TestSort/Sort_#2
=== RUN   TestSort/Sort_#3
--- PASS: TestSort (0.00s)
    --- PASS: TestSort/Sort_#0 (0.00s)
    --- PASS: TestSort/Sort_#1 (0.00s)
    --- PASS: TestSort/Sort_#2 (0.00s)
    --- PASS: TestSort/Sort_#3 (0.00s)
FAIL
exit status 1
FAIL    tests   0.138s
```

Код бенчмаркинга

Функции, имя которых начинается с `Benchmark`, за которым следует термин, начинающийся с прописной буквы, например `Sort`, являются эталонами, выполнение которых рассчитано по времени. Функции эталона получают указатель на структуру `testing.B`, которая определяет поле, описанное в таблице 31-4.

Таблица 31-4 Поле, определяемое структурой B

Функция	Описание
`N`	В этом поле `int` указывается, сколько раз функция эталонного теста должна выполнять код, подлежащий измерению.

Значение `N` используется в цикле `for` в функции эталонного теста для повторения кода, производительность которого измеряется. Инструменты эталонного тестирования могут повторно вызывать функцию эталонного тестирования, используя различные значения `N`, чтобы установить стабильное измерение. Добавьте файл с именем `Benchmark_test.go` в папку `tests` с содержимым, показанным в листинге 31-12.

РЕШЕНИЕ, КОГДА ПРОВОДИТЬ БЕНЧМАРК

Код настройки производительности похож на настройку производительности автомобиля: это может быть весело, обычно дорого и почти каждый раз создает больше проблем, чем решает.

Самая дорогая часть любого проекта — это время программиста, как на начальной стадии разработки, так и на этапе обслуживания. Мало того, что настройка производительности требует времени, которое можно было бы потратить на завершение проекта, так еще и часто создается код, который труднее понять, что в будущем отнимет больше времени, пока какой-нибудь другой разработчик попытается разобраться в ваших хитроумных оптимизациях.

Я готов признать, что есть проекты, предъявляющие особые требования к производительности, но есть вероятность, что ваш проект не входит в их число. Но не беспокойтесь, потому что в моих проектах таких требований тоже нет. Для обычных проектов дешевле купить больше серверов или хранилищ, чем настраивать дорогого разработчика.

Бенчмаркинг может быть образовательным, и вы можете многое узнать о проекте, поняв, как выполняется его код. Но время для образовательного сравнительного анализа находится в коротком окне между развертыванием и получением первого отчета о дефекте, которое в противном случае было бы потрачено на сортировку бумаги для принтера по цветам. До этого момента я советую сосредоточиться на написании кода, который легко понять и легко поддерживать.

```
package main

import (
    "testing"
    "math/rand"
    "time"
)

func BenchmarkSort(b *testing.B) {
    rand.Seed(time.Now().UnixNano())
    size := 250
```

```
    data := make([]int, size)
    for i := 0; i < b.N; i++ {
        for j := 0; j < size; j++ {
            data[j] = rand.Int()
        }
        sortAndTotal(data)
    }
}
```

Листинг 31-12 Содержимое файла benchmark_test.go в папке tests

Функция `BenchmarkSort` создает срез со случайными данными и передает его функции `sortAndTotal`, определенной в листинге 31-2. Чтобы выполнить тест, запустите команду, показанную в листинге 31-13, в папке `tests`.

```
go test -bench . -run notest
```

Листинг 31-13 Выполнение тестов

Точка после аргумента `-bench` приводит к выполнению всех тестов, обнаруженных инструментом `go test`. Точку можно заменить регулярным выражением для выбора конкретных эталонных показателей. По умолчанию также выполняются модульные тесты, но, поскольку я преднамеренно добавил ошибку в функцию `TestSum` в листинге 31-12, я использовал аргумент `-run`, чтобы указать значение, которое не будет соответствовать ни одному из имен тестовых функций в проекта, в результате чего будут выполняться только тесты.

Команда в листинге `31-13` находит и выполняет функцию `BenchmarkSort` и выдает вывод, аналогичный следующему, в зависимости от вашей системы:

```
goos: windows
goarch: amd64
pkg: tests
BenchmarkSort-12           23853                42642 ns/op
PASS
ok      tests   1.577s
```

За названием тестовой функции следует количество ЦП или ядер, которое в моей системе равно 12, но это не повлияет на результаты

теста, поскольку код не использует горутины:

```
...
BenchmarkSort-12            23853            42642 ns/op
...
```

Следующее поле сообщает значение N, которое было передано функции эталонного теста для получения этих результатов:

```
...
BenchmarkSort-12            23853            42642 ns/op
...
```

В моей системе инструменты тестирования запустили функцию BenchmarkSort со значением N, равным 23853. Это число будет меняться от теста к тесту и от системы к системе. Окончательное значение сообщает о продолжительности в наносекундах, необходимой для выполнения каждой итерации цикла тестирования:

```
...
BenchmarkSort-12            23853            42642 ns/op
...
```

Для этого тестового прогона тесту потребовалось 42 642 наносекунды.

Удаление установки из теста

Для каждой итерации цикла for функция BenchmarkSort должна генерировать случайные данные, и время, затраченное на создание этих данных, включается в результаты теста. Структура B определяет методы, описанные в таблице 31-5, которые используются для управления таймером, используемым для эталонного тестирования.

Таблица 31-5 В методы для контроля времени

Функция	Описание
StopTimer()	Этот метод останавливает таймер.
StartTimer()	Этот метод запускает таймер.
ResetTimer()	Этот метод сбрасывает таймер.

Метод ResetTimer полезен, когда эталонный тест требует некоторой первоначальной настройки, а другие методы полезны, когда есть накладные расходы, связанные с каждым тестируемым действием. В листинге 31-14 эти методы используются для исключения подготовки из результатов тестов.

```
package main

import (
    "testing"
    "math/rand"
    "time"
)

func BenchmarkSort(b *testing.B) {
    rand.Seed(time.Now().UnixNano())
    size := 250
    data := make([]int, size)
    b.ResetTimer()
    for i := 0; i < b.N; i++ {
        b.StopTimer()
        for j := 0; j < size; j++ {
            data[j] = rand.Int()
        }
        b.StartTimer()
        sortAndTotal(data)
    }
}
```

Листинг 31-14 Управление таймером в файле benchmark_test.go в папке tests

Таймер сбрасывается после установки случайного начального числа и инициализации среза. В цикле for метод StopTimer используется для остановки таймера до того, как срез будет заполнен случайными данными, а метод StartTimer используется для запуска таймера до вызова функции sortAndTotal. Запустите команду, показанную в листинге 31-14, в папке tests, и будет выполнен пересмотренный тест. В моей системе это дало следующие результаты:

```
goos: windows
goarch: amd64
pkg: tests
```

```
BenchmarkSort-12             35088                32095 ns/op
PASS
ok      tests   4.133s
```

Исключение работы, необходимой для подготовки к тесту, дало более точную оценку времени, необходимого для выполнения функции `sortAndTotal`.

Выполнение суббенчмаркингов

Функция бенчмаркинга может выполнять суббенчмакинги точно так же, как тестовая функция может запускать субтесты. Для быстрого ознакомления в таблице 31-6 описывается метод, используемый для запуска суббенчмарк-теста.

Таблица 31-6 В метод для запуска суббенчмарков

Функция	Описание
`Run(name, func)`	Вызов этого метода выполняет указанную функцию в качестве вспомогательного эталона. Метод блокируется во время выполнения эталонного теста.

Перечисление 31-15 обновляет функцию `BenchmarkSort`, так что выполняется ряд бенчмарков для различных размеров массивов.

```
package main

import (
    "testing"
    "math/rand"
    "time"
    "fmt"
)

func BenchmarkSort(b *testing.B) {
    rand.Seed(time.Now().UnixNano())
    sizes := []int { 10, 100, 250 }
    for _, size := range sizes {
        b.Run(fmt.Sprintf("Array Size %v", size), func(subB
*testing.B) {
            data := make([]int, size)
            subB.ResetTimer()
            for i := 0; i < subB.N; i++ {
```

```
                subB.StopTimer()
                for j := 0; j < size; j++ {
                    data[j] = rand.Int()
                }
                subB.StartTimer()
                sortAndTotal(data)
            }
        })
    }
}
```

Листинг 31-15 Выполнение суббенчмарков в файле benchmarks_test.go в папке tests

Выполнение этих бенчмарков может занять некоторое время. Вот результаты в моей системе, полученные с помощью команды, показанной в листинге 31-13:

```
goos: windows
goarch: amd64
pkg: tests
BenchmarkSort/Array_Size_10-
12              753120                1984 ns/op
BenchmarkSort/Array_Size_100-
12             110248              10953 ns/op
BenchmarkSort/Array_Size_250-
12              34369              31717 ns/op
PASS
ok      tests   61.453s
```

Журналирование ланных

Пакет `log` предоставляет простой API ведения журнала, который создает записи журнала и отправляет их в `io.Writer`, позволяя приложению генерировать данные журнала, не зная, где эти данные будут храниться. Наиболее полезные функции, определенные пакетом `log`, описаны в таблице 31-7.

Таблица 31-7 Полезные функции журнала

Функция	**Описание**

Функция	Описание
`Output()`	Эта функция возвращает `Writer`, которому будут передаваться сообщения журнала. По умолчанию сообщения журнала записываются в стандартный вывод.
`SetOutput(writer)`	Эта функция использует указанный `Writer` для ведения журнала.
`Flags()`	Эта функция возвращает флаги, используемые для форматирования сообщений журнала.
`SetFlags(flags)`	Эта функция использует указанные флаги для форматирования сообщений журнала.
`Prefix()`	Эта функция возвращает префикс, который применяется к сообщениям журнала. По умолчанию префикса нет.
`SetPrefix(prefix)`	Эта функция использует указанную строку в качестве префикса для сообщений журнала.
`Output(depth, message)`	Эта функция записывает указанное сообщение в `Writer`, возвращенный функцией `Output`, с указанной глубиной вызова, которая по умолчанию равна `2`. Глубина вызова используется для управления выбором файла кода и обычно не изменяется.
`Print(...vals)`	Эта функция создает сообщение журнала, вызывая `fmt.Sprint` и передавая результат функции `Output`.
`Printf(template, ...vals)`	Эта функция создает сообщение журнала, вызывая `fmt.Sprintf` и передавая результат функции `Output`.
`Fatal(...vals)`	Эта функция создает сообщение журнала, вызывая `fmt.Sprint`, передает результат в функцию `Output`, а затем завершает работу приложения.
`Fatalf(template, ...vals)`	Эта функция создает сообщение журнала, вызывая `fmt.Sprintf`, передает результат в функцию `Output`, а затем завершает работу приложения.
`Panic(...vals)`	Эта функция создает сообщение журнала, вызывая `fmt.Sprint`, а затем передает результат в функцию `Output`, а затем в функцию `panic`.
`Panicf(template, ...vals)`	Эта функция создает сообщение журнала, вызывая `fmt.Sprintf`, и передает результат в функцию `Output`, а затем в функцию `panic`.

Формат сообщений журнала управляется функцией `SetFlags`, для которой пакет `log` определяет константы, описанные в таблице 31-8.

Таблица 31-8 Константы пакета log

Функция	Описание
`Ldate`	Выбор этого флага включает дату в вывод журнала.
`Ltime`	При выборе этого флага время включается в вывод журнала.

Функция	Описание
Lmicroseconds	Выбор этого флага включает микросекунды во время.
Llongfile	Выбор этого флага включает имя файла кода, включая каталоги, и номер строки, в которой было зарегистрировано сообщение.
Lshortfile	Выбор этого флага включает имя файла кода, за исключением каталогов, и номер строки, в которой было зарегистрировано сообщение.
LUTC	При выборе этого флага для даты и времени используется UTC вместо местного часового пояса.
Lmsgprefix	При выборе этого флага префикс перемещается из его позиции по умолчанию, которая находится в начале сообщения журнала, непосредственно перед строкой, переданной функции Output.
LstdFlags	Эта константа представляет формат по умолчанию, который выбирает Ldate и Ltime.

В листинге 31-16 функции из таблицы 31-7 используются для выполнения простого логирования.

```
package main

import (
    "sort"
    //"fmt"
    "log"
)

func sortAndTotal(vals []int) (sorted []int, total int) {
    sorted = make([]int, len(vals))
    copy(sorted, vals)
    sort.Ints(sorted)
    for _, val := range sorted {
        total += val
        //total++
    }
    return
}

func main() {
    nums := []int { 100, 20, 1, 7, 84 }
    sorted, total := sortAndTotal(nums)
    log.Print("Sorted Data: ", sorted)
    log.Print("Total: ", total)
```

```
}

func init() {
    log.SetFlags(log.Lshortfile | log.Ltime)
}
```

Листинг 31-16 Журналирование сообщений в файле main.go в папке tests

Функция инициализации использует функцию `SetFlags` для выбора флагов `Lshortfile` и `Ltime`, которые будут включать имя файла и время в выходных данных журнала. В `main` функции сообщения журнала создаются с помощью функции `Print`. Скомпилируйте и запустите проект с помощью команды `go run .`, и вы увидите вывод, подобный следующему:

```
08:51:25 main.go:26: Sorted Data: [1 7 20 84 100]
08:51:25 main.go:27: Total: 212
```

Создание пользовательских регистраторов

Пакет `log` можно использовать для настройки различных параметров ведения журнала, чтобы разные части приложения могли записывать сообщения журнала в разные места назначения или использовать разные параметры форматирования. Функция, описанная в таблице 31-9, используется для создания пользовательского адресата регистрации.

Таблица 31-9 Функция пакета log для пользовательского ведения журнала

Функция	Описание
`New(writer, prefix, flags)`	Эта функция возвращает `Logger`, который будет записывать сообщения в указанный `Writer`, настроенный с указанным префиксом и флагами.

Результатом функции `New` является `Logger`, представляющий собой структуру, определяющую методы, соответствующие функциям, описанным в таблице 31-7. Функции в таблице 31-7 просто вызывают метод с тем же именем в регистраторе по умолчанию. В листинге 31-17 функция `New` используется для создания `Logger`.

```
package main

import (
```

```
    "sort"
    //"fmt"
    "log"
)

func sortAndTotal(vals []int) (sorted []int, total int) {
    var logger = log.New(log.Writer(), "sortAndTotal: ",
        log.Flags() | log.Lmsgprefix)
    logger.Printf("Invoked with %v values", len(vals))
    sorted = make([]int, len(vals))
    copy(sorted, vals)
    sort.Ints(sorted)
    logger.Printf("Sorted data: %v", sorted)
    for _, val := range sorted {
        total += val
        //total++
    }
    logger.Printf("Total: %v", total)
    return
}

func main() {
    nums := []int { 100, 20, 1, 7, 84 }
    sorted, total := sortAndTotal(nums)
    log.Print("Sorted Data: ", sorted)
    log.Print("Total: ", total)
}

func init() {
    log.SetFlags(log.Lshortfile | log.Ltime)
}
```

Листинг 31-17 Создание пользовательского регистратора в файле main.go в папке tests

Структура Logger создается с новым префиксом и добавлением флага Lmsgprefix с использованием Writer, полученного из функции Output, описанной в таблице 31-7. В результате сообщения журнала по-прежнему записываются в то же место назначения, но с дополнительным префиксом, обозначающим сообщения из функции sortAndTotal. Скомпилируйте и запустите проект, и вы увидите дополнительные сообщения журнала:

```
09:12:37 main.go:11: sortAndTotal: Invoked with 5 values
09:12:37 main.go:15: sortAndTotal: Sorted data: [1 7 20 84 100]
09:12:37 main.go:20: sortAndTotal: Total: 212
09:12:37 main.go:27: Sorted Data: [1 7 20 84 100]
09:12:37 main.go:28: Total: 212
```

Резюме

В этой главе я закончил описание наиболее полезных пакетов стандартных библиотек с модульным тестированием, эталонным тестированием и ведением журналов. Как я уже объяснял, я нахожу функции тестирования непривлекательными, и у меня есть серьезные сомнения по поводу бенчмаркинга, но оба набора функций хорошо интегрированы в инструменты Go, что упрощает их использование, если ваши взгляды на эти темы не совпадают с моими. Функции ведения журналов вызывают меньше споров, и я использую их в пользовательской платформе веб-приложений, которую я создаю в части 3.

Часть III
Применение Go

32. Создание веб-платформы

В этой главе я начинаю разработку пользовательской платформы веб-приложений, которую я продолжу в главах 33 и 34. В главах 35–38 я использую эту платформу для создания приложения `SportsStore`, которое я в той или иной форме включаю почти во все свои книги.

Цель этой части книги — показать, как Go применяется для решения проблем, возникающих в реальных проектах разработки. Для платформы веб-приложений это означает создание функций для ведения журналов, сеансов, HTML-шаблонов, авторизации и так далее. Для приложения SportsStore это означает использование базы данных продуктов, отслеживание выбора товаров пользователем, проверку введенных пользователем данных и выход из магазина.

Имейте в виду, что код в этих главах был написан специально для этой книги и протестирован только в той мере, в какой функции в последующих главах работают должным образом. Существуют хорошие сторонние пакеты, предоставляющие некоторые или все функции, созданные в этой части книги, и они являются хорошей отправной точкой для ваших проектов. Я рекомендую Gorilla Web Toolkit (`www.gorillatoolkit.org`), в котором есть несколько полезных пакетов (и я использую один из этих пакетов в главе 34).

Осторожно
Эти главы сложные и продвинутые, и важно точно следовать приведенным примерам. Если вы столкнулись с трудностями, то вам следует начать с проверки исправлений для этой книги в репозитории этой книги на GitHub (`https://github.com/apress/pro-go`), где я перечислю решения для любых возникающих проблем.

Создание проекта

Откройте командную строку, перейдите в удобное место и создайте новый каталог с именем `platform`. Перейдите в каталог `platform` и выполните команду, показанную в листинге 32-1.

Подсказка
Вы можете загрузить пример проекта для этой главы — и для всех остальных глав этой книги — с `https://github.com/apress/pro-go`. См. Главу 2 о том, как получить помощь, если у вас возникнут проблемы с запуском примеров.

```
go mod init platform
```

Листинг 32-1 Инициализация проекта

Добавьте файл с именем main.go в папку platform с содержимым, показанным в листинге 32-2.

```
package main

import (
    "fmt"
)

func writeMessage() {
    fmt.Println("Hello, Platform")
}

func main() {
    writeMessage()
}
```

Листинг 32-2 Содержимое файла main.go в папке platform

Запустите команду, показанную в листинге 32-3, в папке `platform`.

```
go run .
```

Листинг 32-3 Компиляция и выполнение проекта

Проект будет скомпилирован и выполнен и выдаст следующий результат:

```
Hello, Platform
```

Создание некоторых основных функций платформы

Для начала я собираюсь определить некоторые базовые службы, которые обеспечат основу для запуска веб-приложений.

Создание системы ведения журнала

Первая функция сервера, которую необходимо реализовать, — это ведение журнала. Пакет `log` в стандартной библиотеке Go предоставляет хороший набор базовых функций для создания журналов, но ему нужны дополнительные функции для фильтрации этих сообщений по деталям. Создайте папку `platform/logging` и добавьте в нее файл с именем `logging.go` с содержимым, показанным в листинге 32-4.

```
package logging

type LogLevel int

const (
    Trace LogLevel = iota
    Debug
    Information
    Warning
    Fatal
    None
)

type Logger interface {

    Trace(string)
    Tracef(string, ...interface{})

    Debug(string)
    Debugf(string, ...interface{})

    Info(string)
    Infof(string, ...interface{})

    Warn(string)
    Warnf(string, ...interface{})
```

```
    Panic(string)
    Panicf(string, ...interface{})
}
```

Листинг 32-4 Содержимое файла logging.go в папке logging

Этот файл определяет интерфейс `Logger`, который определяет методы регистрации сообщений с различными уровнями серьезности, которые задаются с использованием значения `LogLevel` в диапазоне от `Trace` до `Fatal`. Существует также уровень `None`, который указывает отсутствие вывода журнала. Для каждого уровня серьезности интерфейс `Logger` определяет один метод, который принимает простую строку, и один метод, который принимает строку шаблона и значения заполнителей.

Я определяю интерфейсы для всех функций, предоставляемых платформой, и использую эти интерфейсы для обеспечения реализации по умолчанию. Это позволит приложению заменить реализацию по умолчанию, если это необходимо, а также даст возможность предоставлять приложениям функции в виде служб, которые я опишу позже в этой главе.

Чтобы создать реализацию интерфейса `Logger` по умолчанию, добавьте файл с именем `logger_default.go` в папку `logging` с содержимым, показанным в листинге 32-5.

```
package logging

import (
    "log"
    "fmt"
)

type DefaultLogger struct {
    minLevel LogLevel
    loggers map[LogLevel]*log.Logger
    triggerPanic bool
}

func (l *DefaultLogger) MinLogLevel() LogLevel {
    return l.minLevel
}
```

```
func (l *DefaultLogger) write(level LogLevel, message string)
{
    if (l.minLevel <= level) {
        l.loggers[level].Output(2, message)
    }
}

func (l *DefaultLogger) Trace(msg string) {
    l.write(Trace, msg)
}

func   (l   *DefaultLogger)   Tracef(template   string,   vals
...interface{}) {
    l.write(Trace, fmt.Sprintf(template, vals...))
}

func (l *DefaultLogger) Debug(msg string) {
    l.write(Debug, msg)
}

func   (l   *DefaultLogger)   Debugf(template   string,   vals
...interface{}) {
    l.write(Debug, fmt.Sprintf(template, vals...))
}

func (l *DefaultLogger) Info(msg string) {
    l.write(Information, msg)
}

func   (l   *DefaultLogger)   Infof(template   string,   vals
...interface{}) {
    l.write(Information, fmt.Sprintf(template, vals...))
}

func (l *DefaultLogger) Warn(msg string) {
    l.write(Warning, msg)
}

func   (l   *DefaultLogger)   Warnf(template   string,   vals
...interface{}) {
    l.write(Warning, fmt.Sprintf(template, vals...))
}

func (l *DefaultLogger) Panic(msg string) {
```

```
    l.write(Fatal, msg)
    if (l.triggerPanic) {
        panic(msg)
    }
}

func (l *DefaultLogger) Panicf(template string, vals
...interface{}) {
    formattedMsg := fmt.Sprintf(template, vals...)
    l.write(Fatal, formattedMsg)
    if (l.triggerPanic) {
        panic(formattedMsg)
    }
}
```

Листинг 32-5 Содержимое файла logger_default.go в папке logging

Структура `DefaultLogger` реализует интерфейс `Logger`, используя функции, предоставляемые пакетом `log` в стандартной библиотеке, описанной в главе 31. Каждому уровню серьезности назначается `log.Logger`, что означает, что сообщения могут отправляться в разные места назначения или форматироваться по-разному. Добавьте файл с именем `default_create.go` в папку `logging` с кодом, показанным в листинге 32-6.

```
package logging

import (
    "log"
    "os"
)

func NewDefaultLogger(level LogLevel) Logger {
    flags := log.Lmsgprefix | log.Ltime
    return &DefaultLogger {
        minLevel: level,
        loggers: map[LogLevel]*log.Logger {
            Trace: log.New(os.Stdout, "TRACE ",  flags),
            Debug: log.New(os.Stdout, "DEBUG ",  flags),
            Information: log.New(os.Stdout, "INFO ",  flags),
            Warning: log.New(os.Stdout, "WARN ",  flags),
            Fatal: log.New(os.Stdout, "FATAL ",  flags),
        },
```

```
        triggerPanic: true,
    }
}
```

Листинг 32-6 Содержимое файла default_create.go в папке logging

Функция NewDefaultLogger создает DefaultLogger с минимальным уровнем важности и log.Loggers, которые записывают сообщения в стандартный вывод. В качестве простого теста в листинге 32-7 основная функция изменена таким образом, что она записывает свое сообщение, используя функцию ведения журнала.

```
package main

import (
    //"fmt"
    "platform/logging"
)

func writeMessage(logger logging.Logger) {
    logger.Info("Hello, Platform")
}

func main() {
    var logger logging.Logger = logging.NewDefaultLogger(logging.Information)
    writeMessage(logger)
}
```

Листинг 32-7 Использование функции ведения журнала в файле main.go в папке platform

Минимальный уровень серьезности Logger, созданного NewDefaultLogger, установлен на Information, что означает, что сообщения с более низким уровнем серьезности (Trace и Debug) будут отбрасываться. Скомпилируйте и запустите проект, и вы увидите следующий вывод, хотя и с другими временными метками:

```
18:28:46 INFO Hello, Platform
```

Создание системы конфигурации

Следующим шагом является добавление возможности настройки приложения, чтобы настройки не нужно было определять в файлах кода.

Создайте папку `platform/config` и добавьте в нее файл с именем `config.go` с содержимым, показанным в листинге 32-8.

```
package config

type Configuration interface {

    GetString(name string) (configValue string, found bool)
    GetInt(name string) (configValue int, found bool)
    GetBool(name string) (configValue bool, found bool)
    GetFloat(name string) (configValue float64, found bool)

    GetStringDefault(name, defVal string) (configValue string)
    GetIntDefault(name string, defVal int) (configValue int)
      GetBoolDefault(name string, defVal bool) (configValue
bool)
     GetFloatDefault(name string, defVal float64) (configValue
float64)

      GetSection(sectionName string) (section Configuration,
found bool)
}
```

Листинг 32-8 Содержимое файла config.go в папке config

Интерфейс `Configuration` определяет методы для получения параметров конфигурации с поддержкой получения значений строк, целых чисел, чисел с плавающей запятой и логического значения. Существует также набор методов, позволяющих указать значение по умолчанию. Данные конфигурации допускают вложенные разделы конфигурации, которые можно получить с помощью метода `GetSection`.

Определение файла конфигурации

Это помогает понять реализацию системы конфигурации, если вы видите тип файла конфигурации, который я собираюсь использовать. Добавьте файл с именем `config.json` в папку платформы с содержимым, показанным в листинге 32-9.

```
{
    "logging" : {
        "level": "debug"
    },
```

```
    "main" : {
        "message" : "Hello from the config file"
    }
}
```

Листинг 32-9 Содержимое файла config.json в папке platform

Этот файл конфигурации определяет два раздела конфигурации, названные `logging` и `main`. Секция `logging` содержит параметр конфигурации с одной строкой, именованный `level`. Секция `main` содержит один параметр конфигурации строки с именем `message`. Я добавлю настройки конфигурации по мере добавления функций в платформу и когда начну работу над приложением `SportsStore`, но этот файл показывает базовую структуру, которую использует файл конфигурации. При добавлении параметров конфигурации обратите особое внимание на кавычки и запятые, которые необходимы для JSON, но которые легко опустить.

Реализация интерфейса конфигурации

Чтобы создать реализацию интерфейса `Configuration`, добавьте файл с именем `config_default.go` в папку `config` с содержимым, показанным в листинге 32-10.

```
package config

import "strings"

type DefaultConfig struct {
    configData map[string]interface{}
}

func (c *DefaultConfig) get(name string) (result interface{},
found bool) {
    data := c.configData
    for _, key := range strings.Split(name, ":") {
        result, found = data[key]
         if newSection, ok := result.(map[string]interface{});
ok && found {
            data = newSection
        } else {
            return
        }
```

```
    }
    return
}

func  (c  *DefaultConfig)  GetSection(name  string)  (section
Configuration, found bool) {
    value, found := c.get(name)
    if (found) {
        if sectionData, ok := value.(map[string]interface{}) ;
ok {
            section = &DefaultConfig { configData: sectionData
}
        }
    }
    return
}

func (c *DefaultConfig) GetString(name string) (result string,
found bool) {
    value, found := c.get(name)
    if (found) { result = value.(string) }
    return
}

func (c *DefaultConfig) GetInt(name string) (result int, found
bool) {
    value, found := c.get(name)
    if (found) { result =  int(value.(float64)) }
    return
}

func  (c  *DefaultConfig)  GetBool(name  string)  (result  bool,
found bool) {
    value, found := c.get(name)
    if (found) { result = value.(bool) }
    return
}

func (c *DefaultConfig) GetFloat(name string) (result float64,
found bool) {
    value, found := c.get(name)
    if (found) { result = value.(float64) }
    return
}
```

Листинг 32-10 Содержимое файла config_default.go в папке config

Структура `DefaultConfig` реализует интерфейс `Configuration` с помощью карты. Вложенные разделы конфигурации также выражаются в виде карт. Отдельный параметр конфигурации можно запросить, отделив имя раздела от имени параметра, например `logging:level`, или можно запросить карту, содержащую все параметры, с помощью имени раздела, например `logging`. Чтобы определить методы, которые принимают значение по умолчанию, добавьте файл с именем `config_default_fallback.go` в папку `config` с содержимым, показанным в листинге 32-11.

```
package config

func (c *DefaultConfig) GetStringDefault(name, val string)
(result string) {
    result, ok := c.GetString(name)
    if !ok {
        result = val
    }
    return
}

func (c *DefaultConfig) GetIntDefault(name string, val int)
(result int) {
    result, ok := c.GetInt(name)
    if !ok {
        result = val
    }
    return
}

func (c *DefaultConfig) GetBoolDefault(name string, val bool)
(result bool) {
    result, ok := c.GetBool(name)
    if !ok {
        result = val
    }
    return
}
```

```
func  (c  *DefaultConfig)  GetFloatDefault(name  string,  val
float64) (result float64) {
    result, ok := c.GetFloat(name)
    if !ok {
        result = val
    }
    return
}
```

Листинг 32-11 Содержимое файла config_default_fallback.go в папке config

Чтобы определить функцию, которая будет загружать данные из файла конфигурации, добавьте файл с именем `config_json.go` в папку конфигурации с содержимым, показанным в листинге 32-12.

```
package config

import (
    "os"
    "strings"
    "encoding/json"
)

func Load(fileName string) (config Configuration,  err error)
{
    var data []byte
    data, err = os.ReadFile(fileName)
    if (err == nil) {
                                                    decoder      :=
json.NewDecoder(strings.NewReader(string(data)))
        m := map[string]interface{} {}
        err = decoder.Decode(&m)
        if (err == nil) {
            config = &DefaultConfig{ configData: m }
        }
    }
    return
}
```

Листинг 32-12 Содержимое файла config_json.go в папке config

Функция `Load` считывает содержимое файла, декодирует содержащийся в нем JSON в карту и использует карту для создания значения `DefaultConfig`.

Использование системы конфигурации

Чтобы получить уровень ведения журнала из системы конфигурации, внесите изменения, показанные в листинге 32-13, в файл `default_create.go` в папке журнала.

```
package logging

import (
    "log"
    "os"
    "strings"
    "platform/config"
)

func NewDefaultLogger(cfg config.Configuration) Logger {

    var level LogLevel = Debug
                    if     configLevelString,     found     :=
cfg.GetString("logging:level"); found {
        level = LogLevelFromString(configLevelString)
    }

    flags := log.Lmsgprefix | log.Ltime
    return &DefaultLogger {
        minLevel: level,
        loggers: map[LogLevel]*log.Logger {
            Trace: log.New(os.Stdout, "TRACE ",  flags),
            Debug: log.New(os.Stdout, "DEBUG ",  flags),
            Information: log.New(os.Stdout, "INFO ",  flags),
            Warning: log.New(os.Stdout, "WARN ",  flags),
            Fatal: log.New(os.Stdout, "FATAL ",  flags),
        },
        triggerPanic: true,
    }
}

func LogLevelFromString(val string) (level LogLevel) {
    switch strings.ToLower(val) {
        case "debug":
            level = Debug
        case "information":
            level = Information
        case "warning":
```

```
            level = Warning
        case "fatal":
            level = Fatal
        case "none":
            level = None
        default:
            level = Debug
    }
    return
}
```

Листинг 32-13 Использование системы конфигурации в файле default_create.go в папке logging

Нет хорошего способа представить значения `iota` в JSON, поэтому я использовал строку и определил функцию `LogLevelFromString` для преобразования параметра конфигурации в значение `LogLevel`. Перечисление 32-14 обновляет функцию `main` для загрузки и применения данных конфигурации, а также для использования системы конфигурации для чтения сообщения, которое она записывает.

```
package main

import (
            //"fmt"
            "platform/config"
            "platform/logging"
)

func writeMessage(logger logging.Logger, cfg config.Configuration) {
    section, ok := cfg.GetSection("main")
    if (ok) {
        message, ok := section.GetString("message")
        if (ok) {
            logger.Info(message)
        } else {
            logger.Panic("Cannot find configuration setting")
        }
    } else {
        logger.Panic("Config section not found")
    }
}
```

```
func main() {

    var cfg config.Configuration
    var err error
    cfg, err = config.Load("config.json")
    if (err != nil) {
        panic(err)
    }

    var logger logging.Logger = logging.NewDefaultLogger(cfg)
    writeMessage(logger, cfg)
}
```

Листинг 32-14 Чтение настроек конфигурации в файле main.go в папке platform

Конфигурация загружается из файла `config.json`, а реализация `Configuration` передается функции `NewDefaultLogger`, которая использует ее для чтения параметра уровня журнала.

Функция `writeMessage` демонстрирует использование раздела конфигурации, что может быть хорошим способом предоставить компоненту необходимые ему параметры, особенно если требуется несколько экземпляров с разными параметрами, каждый из которых может быть определен в своем собственном разделе.

Код в листинге 32-14 выдает следующий результат при компиляции и выполнении:

```
18:49:12 INFO Hello from the config file
```

Управление службами с внедрением зависимостей

Чтобы получить реализации интерфейсов `Logger` и `Configuration`, код в `main` функции должен знать, как создавать экземпляры структур, реализующих эти интерфейсы:

```
...
cfg, err = config.Load("config.json")
...
var logger logging.Logger = logging.NewDefaultLogger(cfg)
...
```

Это работоспособный подход, но он подрывает цель определения интерфейса, требует осторожности, чтобы экземпляры создавались согласованно, и усложняет процесс замены одной реализации интерфейса другой.

Я предпочитаю использовать внедрение зависимостей (DI), при котором код, зависящий от интерфейса, может получить реализацию без необходимости выбирать базовый тип или напрямую создавать экземпляр. Я собираюсь начать с *службы местоположения*, которая позже послужит основой для более продвинутых функций.

Во время запуска приложения интерфейсы, определенные приложением, будут добавлены в реестр вместе с фабричной функцией, которая создает экземпляры структуры реализации. Так, например, интерфейс platform.logger.Logger будет зарегистрирован в фабричной функции, которая вызывает функцию `NewDefaultLogger`. Когда интерфейс добавляется в реестр, он называется *службой* (сервисом).

Во время выполнения компоненты приложения, которым нужны функции, описанные службой, обращаются к реестру и запрашивают нужный интерфейс. Реестр вызывает фабричную функцию и возвращает созданную структуру, что позволяет компоненту приложения использовать функции интерфейса, не зная и не указывая, какая структура реализации будет использоваться или как она создается. Не волнуйтесь, если это не имеет смысла — это может быть трудной для понимания темой, и становится легче, когда вы видите ее в действии.

Определение жизненных циклов сервиса

Службы регистрируются с жизненными циклами, которые указывают, когда вызывается фабричная функция для создания новых значений структуры. Я собираюсь использовать три жизненных цикла службы, описанные в таблице 32-1.

Таблица 32-1 Жизненные циклы сервиса

Жизненный цикл	Описание
`Transient`	В этом жизненном цикле фабричная функция вызывается для каждого запроса на обслуживание.
`Singleton`	В этом жизненном цикле фабричная функция вызывается один раз, и каждый запрос получает один и тот же экземпляр структуры.

Жизненный цикл	Описание
Scoped	В этом жизненном цикле фабричная функция вызывается один раз для первого запроса в области, и каждый запрос в этой области получает один и тот же экземпляр структуры.

Создайте папку `platform/services` и добавьте в нее файл с именем `lifecycles.go` с содержимым, показанным в листинге 32-15.

```
package services

type lifecycle int

const (
    Transient lifecycle = iota
    Singleton
    Scoped
)
```

Листинг 32-15 Содержимое файла lifecycles.go в папке services

Я собираюсь реализовать жизненный цикл `Scoped`, используя пакет `context` в стандартной библиотеке, который я описал в главе 30. `Context` будет автоматически создаваться для каждого HTTP-запроса, полученного сервером, а это означает, что весь код обработки запросов, который обрабатывает этот request может совместно использовать один и тот же набор служб, так что, например, одна структура, предоставляющая информацию о сеансе, может использоваться во время обработки данного запроса.

Чтобы упростить работу с контекстами, добавьте файл с именем `context.go` в папку `services` с содержимым, показанным в листинге 32-16.

```
package services

import (
    "context"
    "reflect"
)

const ServiceKey = "services"
```

```
type serviceMap map[reflect.Type]reflect.Value

func NewServiceContext(c context.Context) context.Context {
    if (c.Value(ServiceKey) == nil) {
                    return  context.WithValue(c,  ServiceKey,
make(serviceMap))
    } else {
        return c
    }
}
```

Листинг 32-16 Содержимое файла context.go в папке services

Функция `NewServiceContext` извлекает контекст с помощью функции `WithValue`, добавляя карту, в которой хранятся службы, которые были разрешены. См. главу 30 для получения подробной информации о различных способах получения контекстов.

Определение внутренних сервисных функций

Я собираюсь обрабатывать регистрацию службы, проверяя фабричную функцию и используя ее результат для определения интерфейса, который она обрабатывает. Это пример типа фабричной функции, которая будет использоваться при регистрации нового сервиса:

```
...
func ConfigurationFactory() config.Configuration {
       //  TODO  create  struct  that  implements  Configuration
interface
}
...
```

Тип результата этой функции — `config.Configuration`. Использование отражения для проверки функции позволит мне получить тип результата и определить интерфейс, для которого это фабрика.

Некоторые фабричные функции будут зависеть от других сервисов. Вот еще один пример фабричной функции:

```
...
func Loggerfactory(cfg config.Configuration) logging.Logger {
    // TODO create struct that implements Logger interface
}
```

```
...
```

Эта фабричная функция разрешает запросы к интерфейсу `Logger`, но это зависит от реализации интерфейса `Configuration`. Это означает, что интерфейс `Configuration` должен быть разрешен для предоставления аргумента, необходимого для разрешения интерфейса `Logger`. Это пример *внедрения зависимостей*, когда зависимости фабричной функции — параметры — разрешаются, чтобы функция могла быть вызвана.

> **Примечание**
> Определение фабричных функций, зависящих от других служб, может изменить жизненные циклы вложенных служб. Например, если вы определяете одноэлементную службу, которая зависит от временной службы, вложенная служба будет разрешена только один раз при первом создании экземпляра одноэлементной службы. Это не проблема в большинстве проектов, но об этом следует помнить.

Добавьте файл с именем `core.go` в папку `services` с содержимым, показанным в листинге 32-17.

```
package services

import (
    "reflect"
    "context"
    "fmt"
)

type BindingMap struct {
    factoryFunc reflect.Value
    lifecycle
}

var services = make(map[reflect.Type]BindingMap)

func addService(life lifecycle, factoryFunc interface{}) (err
error) {
    factoryFuncType := reflect.TypeOf(factoryFunc)
```

```
        if    factoryFuncType.Kind()    ==    reflect.Func    &&
factoryFuncType.NumOut() == 1 {
        services[factoryFuncType.Out(0)] = BindingMap{
            factoryFunc: reflect.ValueOf(factoryFunc),
            lifecycle: life,
        }
    } else {
        err = fmt.Errorf("Type cannot be used as service: %v",
factoryFuncType)

    }
    return
}
var contextReference = (*context.Context)(nil)
var                    contextReferenceType                    =
reflect.TypeOf(contextReference).Elem()

func     resolveServiceFromValue(c     context.Context,     val
reflect.Value) (err error ){
    serviceType := val.Elem().Type()
    if serviceType == contextReferenceType {
        val.Elem().Set(reflect.ValueOf(c))
    } else if binding, found := services[serviceType]; found {
        if (binding.lifecycle == Scoped) {
            resolveScopedService(c, val, binding)
        } else {
                              val.Elem().Set(invokeFunction(c,
binding.factoryFunc)[0])
        }
    } else {
            err  =  fmt.Errorf("Cannot  find  service  %v",
serviceType)
    }
    return
}

func      resolveScopedService(c      context.Context,      val
reflect.Value,
        binding BindingMap) (err error) {
    sMap, ok := c.Value(ServiceKey).(serviceMap)
    if (ok) {
        serviceVal, ok := sMap[val.Type()]
        if (!ok) {
```

```
                              serviceVal   =   invokeFunction(c,
binding.factoryFunc)[0]
            sMap[val.Type()] = serviceVal
        }
        val.Elem().Set(serviceVal)
    } else {
         val.Elem().Set(invokeFunction(c, binding.factoryFunc)
[0])
    }
    return
}

func    resolveFunctionArguments(c    context.Context,        f
reflect.Value,
        otherArgs ...interface{}) []reflect.Value {
    params := make([]reflect.Value, f.Type().NumIn())
    i := 0
    if (otherArgs != nil) {
        for ; i < len(otherArgs); i++ {
            params[i] = reflect.ValueOf(otherArgs[i])
        }
    }
    for ; i < len(params); i++ {
        pType := f.Type().In(i)
        pVal := reflect.New(pType)
        err := resolveServiceFromValue(c, pVal)
        if err != nil {
            panic(err)
        }
        params[i] = pVal.Elem()
    }
    return params
}

func invokeFunction(c context.Context, f reflect.Value,
        otherArgs ...interface{}) []reflect.Value {
              return    f.Call(resolveFunctionArguments(c,    f,
otherArgs...))
}
```

Листинг 32-17 Содержимое файла core.go в папке services

Структура `BindingMap` представляет собой комбинацию фабричной функции, выраженной в виде `Reflect.Value`, и жизненного цикла.

Функция addService используется для регистрации службы, что она делает путем создания BindingMap и добавления к карте, назначенной переменной services.

Функция resolveServiceFromValue вызывается для разрешения службы, а ее аргументы — это Context и Value, являющиеся указателем на переменную, тип которой является интерфейсом, который нужно решить (это будет иметь больше смысла, когда вы увидите разрешение службы в действии). Чтобы разрешить службу, функция getServiceFromValue проверяет, есть ли BindingMap в карте служб, используя запрошенный тип в качестве ключа. Если есть BindingMap, то вызывается его фабричная функция, и значение присваивается через указатель.

Функция invokeFunction отвечает за вызов фабричной функции, используя функцию resolveFunctionArguments для проверки параметров фабричной функции и разрешения каждого из них. Эти функции принимают необязательные дополнительные аргументы, которые используются, когда функция должна быть вызвана с сочетанием служб и параметров с обычными значениями (в этом случае параметры, которым требуются обычные значения, должны быть определены в первую очередь).

Для служб с заданной областью требуется особое обращение. ResolveScopedService проверяет, содержит ли Context значение из предыдущего запроса на разрешение службы. Если нет, служба разрешается и добавляется в Context, чтобы ее можно было повторно использовать в той же области.

Определение функций регистрации службы

Ни одна из функций, определенных в листинге 32-17, не экспортируется. Чтобы создать функции, которые будут использоваться в остальной части приложения для регистрации служб, добавьте файл с именем registration.go в папку services с содержимым, показанным в листинге 32-18.

```
package services

import (
    "reflect"
    "sync"
)
```

```
func AddTransient(factoryFunc interface{}) (err error) {
    return addService(Transient, factoryFunc)
}

func AddScoped(factoryFunc interface{}) (err error) {
    return addService(Scoped, factoryFunc)
}

func AddSingleton(factoryFunc interface{}) (err error) {
    factoryFuncVal := reflect.ValueOf(factoryFunc)
        if factoryFuncVal.Kind() == reflect.Func &&
factoryFuncVal.Type().NumOut() == 1 {
        var results []reflect.Value
        once := sync.Once{}
        wrapper := reflect.MakeFunc(factoryFuncVal.Type(),
            func ([]reflect.Value) []reflect.Value {
                once.Do(func() {
                    results = invokeFunction(nil,
factoryFuncVal)
                })
                return results
            })
        err = addService(Singleton, wrapper.Interface())
    }
    return
}
```

Листинг 32-18 Содержимое файла registration.go в папке services

Функции `AddTransient` и `AddScoped` просто передают фабричную функцию функции `addService`. Для жизненного цикла синглтона требуется немного больше работы, и функция `AddSingleton` создает оболочку вокруг фабричной функции, которая гарантирует, что она выполняется только один раз, для первого запроса на разрешение службы. Это гарантирует, что создан только один экземпляр структуры реализации и что он не будет создан до тех пор, пока он не понадобится в первый раз.

Определение функций разрешения службы

Следующий набор функций включает в себя функции, позволяющие разрешать службы. Добавьте файл с именем `resolution.go` в папку `services` с содержимым, показанным в листинге 32-19.

```
package services

import (
    "reflect"
    "errors"
    "context"
)

func GetService(target interface{}) error {
    return GetServiceForContext(context.Background(), target)
}

func GetServiceForContext(c context.Context, target interface{}) (err error) {
    targetValue := reflect.ValueOf(target)
    if targetValue.Kind() == reflect.Ptr &&
            targetValue.Elem().CanSet() {
        err = resolveServiceFromValue(c, targetValue)
    } else {
        err = errors.New("Type cannot be used as target")
    }
    return
}
```

Листинг 32-19 Содержимое файла resolution.go в папке services

`GetServiceForContext` принимает контекст и указатель на значение, которое можно установить с помощью рефлексии. Для удобства функция `GetService` разрешает службу, используя фоновый контекст.

Регистрация и использование сервисов

Базовые функции службы готовы, что означает, что я могу зарегистрировать службы, а затем разрешить их. Добавьте файл с именем `services_default.go` в папку `services` с содержимым, показанным в листинге 32-20.

```
package services

import (
    "platform/logging"
    "platform/config"
)
```

```
func RegisterDefaultServices() {

    err := AddSingleton(func() (c config.Configuration) {
        c, loadErr :=  config.Load("config.json")
        if (loadErr != nil) {
            panic(loadErr)
        }
        return
    })

      err  =  AddSingleton(func(appconfig  config.Configuration)
logging.Logger {
        return logging.NewDefaultLogger(appconfig)
    })
    if (err != nil) {
        panic(err)
    }
}
```

Листинг 32-20 Содержимое файла functions.go в папке services

`RegisterDefaultServices` создает службы `Configuration` и `Logger`. Эти сервисы создаются с помощью функции `AddSingleton`, что означает, что один экземпляр структур, реализующих каждый интерфейс, будет общим для всего приложения. Листинг 32-21 обновляет функцию `main` для использования служб, а не для непосредственного создания экземпляров структур.

```
package main

import (
    //"fmt"
    "platform/config"
    "platform/logging"
    "platform/services"
)

func       writeMessage(logger       logging.Logger,       cfg
config.Configuration) {

    section, ok := cfg.GetSection("main")
    if (ok) {
        message, ok := section.GetString("message")
```

```
        if (ok) {
            logger.Info(message)
        } else {
            logger.Panic("Cannot find configuration setting")
        }
    } else {
        logger.Panic("Config section not found")
    }
}

func main() {

    services.RegisterDefaultServices()

    var cfg config.Configuration
    services.GetService(&cfg)

    var logger logging.Logger
    services.GetService(&logger)

    writeMessage(logger, cfg)
}
```

Листинг 32-21 Разрешение служб в файле main.go в папке platform

Разрешение службы выполняется путем передачи указателя на переменную, тип которой является интерфейсом. В листинге 32-21 функция `GetService` используется для получения реализаций интерфейсов `Repository` и `Logger` без необходимости знать, какой тип структуры будет использоваться, процесс, в котором она создана, или жизненные циклы службы.

Оба шага — создание переменной и передача указателя — необходимы для разрешения службы. Скомпилируйте и запустите проект, и вы увидите следующий вывод:

```
19:17:06 INFO Hello from the config file
```

Добавление поддержки для вызова функций

После того, как базовые функции службы будут готовы, можно будет легко создавать улучшения, которые упрощают и упрощают разрешение службы. Чтобы добавить поддержку прямого выполнения функций,

добавьте файл с именем `functions.go` в папку `services` с содержимым, показанным в листинге 32-22.

```
package services

import (
    "reflect"
    "errors"
    "context"
)

func Call(target interface{}, otherArgs ...interface{})
([]interface{}, error) {
        return CallForContext(context.Background(), target,
otherArgs...)
}

func CallForContext(c context.Context, target interface{},
otherArgs ...interface{}) (results []interface{}, err error) {
    targetValue := reflect.ValueOf(target)
    if (targetValue.Kind() == reflect.Func) {
            resultVals := invokeFunction(c, targetValue,
otherArgs...)
        results = make([]interface{}, len(resultVals))
        for i := 0; i < len(resultVals); i++ {
            results[i] = resultVals[i].Interface()
        }
    } else {
        err = errors.New("Only functions can be invoked")
    }
    return
}
```

Листинг 32-22 Содержимое файла functions.go в папке services

Функция `CallForContext` получает функцию и использует службы для создания значений, которые используются в качестве аргументов для вызова функции. Функция `Call` удобна для использования, когда `Context` недоступен. Реализация этой функции основана на коде, используемом для вызова фабричных функций в листинге 32-22. В листинге 32-23 показано, как прямой вызов функций может упростить использование сервисов.

```
package main

import (
    //"fmt"
    "platform/config"
    "platform/logging"
    "platform/services"
)

func writeMessage(logger logging.Logger, cfg config.Configuration) {

    section, ok := cfg.GetSection("main")
    if (ok) {
        message, ok := section.GetString("message")
        if (ok) {
            logger.Info(message)
        } else {
            logger.Panic("Cannot find configuration setting")
        }
    } else {
        logger.Panic("Config section not found")
    }
}

func main() {

    services.RegisterDefaultServices()

    // var cfg config.Configuration
    // services.GetService(&cfg)

    // var logger logging.Logger
    // services.GetService(&logger)

    services.Call(writeMessage)
}
```

Листинг 32-23 Вызов функции непосредственно в файле main.go в папке platform

Функция передается `Call`, который проверяет ее параметры и разрешает их с помощью сервисов. (Обратите внимание, что круглые скобки не следуют за именем функции, потому что это вызвало бы функцию, а не передало бы ее в `services.Call`.) Мне больше не нужно

запрашивать услуги напрямую, и я могу положиться на пакет services, который позаботится о деталях. Скомпилируйте и выполните код, и вы увидите следующий вывод:

```
19:19:08 INFO Hello from the config file
```

Добавление поддержки разрешения полей структуры

Последняя функция, которую я собираюсь добавить в пакет services, — это возможность разрешать зависимости от полей структуры. Добавьте файл с именем structs.go в папку services с содержимым, показанным в листинге 32-24.

```
package services

import (
    "reflect"
    "errors"
    "context"
)

func Populate(target interface{}) error {
    return PopulateForContext(context.Background(), target)
}

func PopulateForContext(c context.Context, target interface{})
(err error) {
    return PopulateForContextWithExtras(c, target,
        make(map[reflect.Type]reflect.Value))
}

func  PopulateForContextWithExtras(c  context.Context,  target
interface{},
        extras map[reflect.Type]reflect.Value) (err error) {
    targetValue := reflect.ValueOf(target)
    if targetValue.Kind() == reflect.Ptr &&
            targetValue.Elem().Kind() == reflect.Struct {
        targetValue = targetValue.Elem()
        for i := 0; i < targetValue.Type().NumField(); i++ {
            fieldVal := targetValue.Field(i)
            if fieldVal.CanSet() {
                if extra, ok := extras[fieldVal.Type()]; ok {
                    fieldVal.Set(extra)
```

```
                } else {
                    resolveServiceFromValue(c,  fieldVal.Addr()
)
                }
            }

        }
    } else {
        err = errors.New("Type cannot be used as target")
    }
    return
}
```

Листинг 32-24 Содержимое файла structs.go в папке services

Эти функции проверяют поля, определенные структурой, и пытаются разрешить их с помощью определенных служб. Любые поля, тип которых не является интерфейсом или для которых нет службы, пропускаются. Функция `PopulateForContextWithExtras` позволяет указывать дополнительные значения для полей структуры.

Listing 32-25 defines a struct whose fields declare dependencies on services.

```
package main

import (
    //"fmt"
    "platform/config"
    "platform/logging"
    "platform/services"
)

func      writeMessage(logger      logging.Logger,      cfg
config.Configuration) {

    section, ok := cfg.GetSection("main")
    if (ok) {
        message, ok := section.GetString("message")
        if (ok) {
            logger.Info(message)
        } else {
            logger.Panic("Cannot find configuration setting")
        }
```

```
    } else {
        logger.Panic("Config section not found")
    }
}

func main() {

    services.RegisterDefaultServices()

    services.Call(writeMessage)

    val := struct {
        message string
        logging.Logger
    }{
        message: "Hello from the struct",
    }
    services.Populate(&val)
    val.Logger.Debug(val.message)
}
```

Листинг 32-25 Внедрение структурных зависимостей в файл main.go в папке platform

Функция `main` определяет анонимную структуру и разрешает требуемые службы, передавая указатель на функцию `Populate`. В результате встроенные поля `Logger` заполняются с помощью службы. Функция `Populate` пропускает поле `message`, но значение определяется при инициализации структуры. Скомпилируйте и запустите проект, и вы увидите следующий вывод:

```
19:21:43 INFO Hello from the config file
19:21:43 DEBUG Hello from the struct
```

Резюме

В этой главе я начал разработку пользовательской платформы веб-приложений. Я создал функции ведения журнала и конфигурации, а также добавил поддержку сервисов и внедрения зависимостей. В следующей главе я продолжу разработку, создав конвейер обработки запросов и настраиваемую систему шаблонов.

33. Промежуточное ПО, шаблоны и обработчики

В этой главе я продолжаю разработку платформы веб-приложений, начатую в главе 32, добавляя поддержку обработки HTTP-запросов.

> **Подсказка**
> Вы можете загрузить пример проекта для этой главы — и для всех остальных глав этой книги — с `https://github.com/apress/pro-go`. См. Главу 2 о том, как получить помощь, если у вас возникнут проблемы с запуском примеров.

Создание конвейера запросов

Следующим шагом в создании платформы является создание веб-службы, которая будет обрабатывать HTTP-запросы от браузеров. Для подготовки я собираюсь создать простой конвейер, который будет содержать компоненты промежуточного программного обеспечения, которые могут проверять и изменять запросы.

При поступлении HTTP-запроса он будет передан каждому зарегистрированному компоненту ПО промежуточного слоя в конвейере, что даст каждому компоненту возможность обработать запрос и внести свой вклад в ответ. Компоненты также смогут завершать обработку запроса, предотвращая пересылку запроса оставшимся компонентам в конвейере.

Как только запрос достигает конца конвейера, он возвращается обратно по конвейеру, чтобы у компонентов была возможность внести дальнейшие изменения или выполнить дальнейшую работу, как показано на рисунке 33-1.

Рисунок 33-1 Конвейер обработки запросов

Определение интерфейса компонента промежуточного программного обеспечения

Создайте папку `platform/pipeline` и добавьте в нее файл с именем `component.go` с содержимым, показанным в листинге 33-1.

```
package pipeline
```

```
import (
    "net/http"
)

type ComponentContext struct {
    *http.Request
    http.ResponseWriter
    error
}

func (mwc *ComponentContext) Error(err error) {
    mwc.error = err
}

func (mwc *ComponentContext) GetError() error {
    return mwc.error
}

type MiddlewareComponent interface {

    Init()

            ProcessRequest(context *ComponentContext, next
func(*ComponentContext))
}
```

Листинг 33-1 Содержимое файла component.go в папке pipeline

Как следует из названия, интерфейс `MiddlewareComponent` описывает функциональные возможности, необходимые компоненту промежуточного программного обеспечения. Метод `Init` используется для выполнения любой одноразовой настройки, а другой метод с именем `ProcessRequest` отвечает за обработку HTTP-запросов. Параметры, определенные методом `ProcessRequest`, представляют собой указатель на структуру `ComponentContext` и функцию, которая передает запрос следующему компоненту в конвейере.

Все, что нужно компоненту для обработки запроса, предоставляется структурой `ComponentContext`, через которую можно получить доступ к `http.Request` и `http.ResponseWriter`. Структура `ComponentContext` также определяет неэкспортируемое поле `error`, которое используется для обозначения проблемы с обработкой запроса и устанавливается с помощью метода `Error`.

Создание конвейера запросов

Чтобы создать конвейер, который будет обрабатывать запросы, добавьте файл с именем `pipe.go` в папку `pipeline` с содержимым, показанным в листинге 33-2.

```
package pipeline

import (
    "net/http"
)

type RequestPipeline func(*ComponentContext)

var emptyPipeline RequestPipeline = func(*ComponentContext) { /* do
nothing */ }

func        CreatePipeline(components        ...MiddlewareComponent)
RequestPipeline {
    f := emptyPipeline
    for i := len(components) -1 ; i >= 0; i-- {
        currentComponent := components[i]
        nextFunc := f
        f = func(context *ComponentContext) {
            if (context.error == nil) {
                currentComponent.ProcessRequest(context, nextFunc)
            }
        }
        currentComponent.Init()
    }
    return f
}

func (pl RequestPipeline) ProcessRequest(req *http.Request,
        resp http.ResponseWriter) error {
    ctx := ComponentContext {
        Request: req,
        ResponseWriter: resp,
    }
    pl(&ctx)
    return ctx.error
}
```

Листинг 33-2 Содержимое файла pipe.go в папке pipeline

Функция `CreatePipeline` является наиболее важной частью этого листинга, поскольку она принимает ряд компонентов и соединяет их для создания функции, которая принимает указатель на структуру `ComponentContext`. Эта функция вызывает метод `ProcessRequest` первого компонента в конвейере со следующим аргументом, который вызывает метод `ProcessRequest` следующего компонента. Эта цепочка передает структуру `ComponentContext` всем компонентам по очереди, если только один из них не вызывает метод `Error`. Запросы обрабатываются с помощью метода `ProcessRequest`, который создает значение `ComponentContext` и использует его для запуска обработки запроса.

Создание базовых компонентов

Определение интерфейса компонента и конвейера простое, но оно обеспечивает гибкую основу, на которой могут быть написаны компоненты. Приложения могут определять и выбирать свои собственные компоненты, но есть некоторые основные функции, которые я собираюсь включить в платформу.

Создание сервисов компонента ПО промежуточного слоя

Создайте папку `platform/pipeline/basic` и добавьте в нее файл с именем `services.go` с содержимым, показанным в листинге 33-3.

```
package basic

import (
    "platform/pipeline"
    "platform/services"
)

type ServicesComponent struct {}

func (c *ServicesComponent) Init() {}

func (c *ServicesComponent) ProcessRequest(ctx
*pipeline.ComponentContext,
        next func(*pipeline.ComponentContext)) {
    reqContext := ctx.Request.Context()
    ctx.Request.WithContext(services.NewServiceContext(reqContext))
    next(ctx)
}
```

Листинг 33-3 Содержимое файла services.go в папке pipe/basic

Этот компонент промежуточного программного обеспечения изменяет `Context`, связанный с запросом, чтобы во время обработки запроса можно было использовать контекстно-зависимые службы. Метод `http.Request.Context` используется для получения стандартного `Context`, созданного с помощью запроса, который подготавливается для служб, а затем обновляется с помощью метода `WithContext`.

После подготовки контекста запрос передается по конвейеру путем вызова функции, полученной через параметр с именем `next`:

```
...
next(ctx)
...
```

Этот параметр дает компонентам промежуточного слоя контроль над обработкой запросов и позволяет изменять контекстные данные, которые

получают последующие компоненты. Это также позволяет компонентам сократить обработку запроса, не вызывая next функцию.

Создание компонента ПО промежуточного слоя ведения журналов

Затем добавьте файл с именем logging.go в папку basic с содержимым, показанным в листинге 33-4. Next, add a file named logging.go to the basic folder with the content shown in Listing 33-4.

```
package basic

import (
    "net/http"
    "platform/logging"
    "platform/pipeline"
    "platform/services"
)

type LoggingResponseWriter struct {
    statusCode int
    http.ResponseWriter
}

func (w *LoggingResponseWriter) WriteHeader(statusCode int) {
    w.statusCode = statusCode
    w.ResponseWriter.WriteHeader(statusCode)
}

func (w *LoggingResponseWriter) Write(b []byte) (int, error) {
    if (w.statusCode == 0) {
        w.statusCode = http.StatusOK
    }
    return w.ResponseWriter.Write(b)
}

type LoggingComponent struct {}

func (lc *LoggingComponent) Init() {}

func        (lc        *LoggingComponent)        ProcessRequest(ctx
*pipeline.ComponentContext,
        next func(*pipeline.ComponentContext))  {

    var logger logging.Logger
        err  :=  services.GetServiceForContext(ctx.Request.Context(),
&logger)
    if (err != nil) {
        ctx.Error(err)
```

```
        return
    }

    loggingWriter := LoggingResponseWriter{ 0, ctx.ResponseWriter}
    ctx.ResponseWriter = &loggingWriter

          logger.Infof("REQ   ---   %v   -   %v",   ctx.Request.Method,
ctx.Request.URL)
    next(ctx)
           logger.Infof("RSP   %v   %v",   loggingWriter.statusCode,
ctx.Request.URL )
}
```

Листинг 33-4 Содержимое файла logging.go в папке basic

Этот компонент регистрирует основные сведения о запросе и ответе с помощью службы `Logger`, созданной в главе 32. Интерфейс `ResponseWriter` не предоставляет доступ к коду состояния, отправленному в ответе, поэтому создается `LoggingResponseWriter` и передается следующему компоненту в конвейере.

Этот компонент выполняет действия до и после вызова функции `next`, регистрируя сообщение перед передачей запроса и регистрируя другое сообщение, в котором выводится код состояния после обработки запроса.

Этот компонент получает службу `Logger` при обработке запроса. Я мог бы получить `Logger` только один раз, но это работает только потому, что я знаю, что `Logger` был зарегистрирован как одноэлементная служба. Вместо этого я предпочитаю не делать предположений о жизненном цикле `Logger`, а это значит, что я не получу неожиданных результатов, если жизненный цикл изменится в будущем.

Создание компонента обработки ошибок

Конвейер запросов позволяет компонентам завершать обработку при возникновении ошибки. Чтобы определить компонент, который будет обрабатывать ошибку, добавьте файл с именем `errors.go` в папку `platform/pipeline/basic` с содержимым, показанным в листинге 33-5.

```
package basic

import (
    "fmt"
    "net/http"
    "platform/logging"
    "platform/pipeline"
    "platform/services"
)
```

```
type ErrorComponent struct {}

func     recoveryFunc    (ctx    *pipeline.ComponentContext,    logger
logging.Logger) {
    if arg := recover(); arg != nil {
        logger.Debugf("Error: %v", fmt.Sprint(arg))
        ctx.ResponseWriter.WriteHeader(http.StatusInternalServerError)
    }
}

func (c *ErrorComponent) Init() {}

func          (c          *ErrorComponent)          ProcessRequest(ctx
*pipeline.ComponentContext,
        next func(*pipeline.ComponentContext))  {

    var logger logging.Logger
    services.GetServiceForContext(ctx.Context(), &logger)
    defer recoveryFunc(ctx, logger)
    next(ctx)
    if (ctx.GetError() != nil) {
        logger.Debugf("Error: %v", ctx.GetError())
        ctx.ResponseWriter.WriteHeader(http.StatusInternalServerError)
    }
}
```

Листинг 33-5 Содержимое файла errors.go в папке basic

Этот компонент восстанавливается после любой паники, которая возникает, когда последующие компоненты обрабатывают запрос, а также обрабатывает любую ожидаемую ошибку. В обоих случаях ошибка регистрируется, а код состояния ответа указывает на ошибку.

Создание компонента статического файла

Почти все веб-приложения требуют поддержки обслуживания статических файлов, даже если это касается только таблиц стилей CSS. Стандартная библиотека содержит встроенную поддержку обслуживания файлов, что полезно, поскольку это задача, чреватая потенциальными проблемами. Но, к счастью, интегрировать функции стандартной библиотеки в конвейер запросов в примере проекта несложно. Добавьте файл с именем `files.go` в папку `basic` с содержимым, показанным в листинге 33-6.

```
package basic

import (
    "net/http"
    "platform/config"
```

```
    "platform/pipeline"
    "platform/services"
    "strings"
)

type StaticFileComponent struct {
    urlPrefix string
    stdLibHandler http.Handler
}

func (sfc *StaticFileComponent) Init() {
    var cfg config.Configuration
    services.GetService(&cfg)
           sfc.urlPrefix   =   cfg.GetStringDefault("files:urlprefix",
"/files/")
    path, ok := cfg.GetString("files:path")
    if (ok) {
        sfc.stdLibHandler = http.StripPrefix(sfc.urlPrefix,
            http.FileServer(http.Dir(path)))
    } else {
        panic ("Cannot load file configuration settings")
    }
}

func       (sfc       *StaticFileComponent)       ProcessRequest(ctx
*pipeline.ComponentContext,
    next func(*pipeline.ComponentContext)) {

    if  !strings.EqualFold(ctx.Request.URL.Path, sfc.urlPrefix) &&
            strings.HasPrefix(ctx.Request.URL.Path, sfc.urlPrefix) {
        sfc.stdLibHandler.ServeHTTP(ctx.ResponseWriter, ctx.Request)
    } else {
        next(ctx)
    }
}
```

Листинг 33-6 Содержимое файла files.go в папке basic

Этот обработчик использует метод `Init` для чтения параметров конфигурации, которые определяют префикс, используемый для файловых запросов, и каталог, из которого следует обслуживать файлы, а также использует обработчики, предоставляемые пакетом `net/http`, для обслуживания файлов.

Создание компонента ответа-заполнителя

Проект не содержит каких-либо компонентов промежуточного программного обеспечения, которые генерируют ответы, которые обычно определяются как часть приложения. Однако на данный момент мне нужен компонент-заполнитель, который будет генерировать простые ответы по мере разработки

других функций. Создайте папку `platform/placeholder` и добавьте в нее файл с именем `message_middleware.go` с содержимым, показанным в листинге 33-7.

```
package placeholder

import (
    "io"
    "errors"
    "platform/pipeline"
    "platform/config"
    "platform/services"
)

type SimpleMessageComponent struct {}

func (c *SimpleMessageComponent) Init() {}

func (c *SimpleMessageComponent) ProcessRequest(ctx
*pipeline.ComponentContext,
        next func(*pipeline.ComponentContext)) {

    var cfg config.Configuration
    services.GetService(&cfg)
    msg, ok := cfg.GetString("main:message")
    if (ok) {
        io.WriteString(ctx.ResponseWriter, msg)
    } else {
        ctx.Error(errors.New("Cannot find config setting"))
    }
    next(ctx)
}
```

Листинг 33-7 Содержимое файла message_middleware.go в папке placeholder

Этот компонент выдает простой текстовый ответ, которого достаточно, чтобы убедиться, что конвейер работает должным образом. Затем создайте папку `platform/placeholder/files` и добавьте в нее файл `hello.json` с содержимым, показанным в листинге 33-8.

```
{
    "message": "Hello from the JSON file"
}
```

Листинг 33-8 Содержимое файла hello.json в папке placeholder/files

Чтобы указать расположение, из которого будут считываться статические файлы, добавьте параметр, показанный в листинге 33-9, в файл `config.json` в папке `platform`.

```
{
    "logging" : {
        "level": "debug"
    },
    "main" : {
        "message" : "Hello from the config file"
    },
    "files": {
        "path": "placeholder/files"
    }
}
```

Листинг 33-9 Добавление параметра конфигурации в файл config.json в папке platform

Создание HTTP-сервера

Пришло время создать HTTP-сервер и использовать конвейер для обработки получаемых запросов. Создайте папку `platform/http` и добавьте в нее файл с именем `server.go` с содержимым, показанным в листинге 33-10.

```
package http

import (
    "fmt"
    "sync"
    "net/http"
    "platform/config"
    "platform/logging"
    "platform/pipeline"
)

type pipelineAdaptor struct {
    pipeline.RequestPipeline
}

func (p pipelineAdaptor) ServeHTTP(writer http.ResponseWriter,
    request *http.Request) {
        p.ProcessRequest(request, writer)
}

func  Serve(pl  pipeline.RequestPipeline,  cfg  config.Configuration,
logger logging.Logger ) *sync.WaitGroup {
    wg := sync.WaitGroup{}

    adaptor := pipelineAdaptor { RequestPipeline: pl }

    enableHttp := cfg.GetBoolDefault("http:enableHttp", true)
    if (enableHttp) {
        httpPort := cfg.GetIntDefault("http:port", 5000)
```

```
        logger.Debugf("Starting HTTP server on port %v", httpPort)
        wg.Add(1)
        go func() {
             err := http.ListenAndServe(fmt.Sprintf(":%v", httpPort),
adaptor)
            if (err != nil) {
                panic(err)
            }
        }()
    }
    enableHttps := cfg.GetBoolDefault("http:enableHttps", false)
    if (enableHttps) {
        httpsPort := cfg.GetIntDefault("http:httpsPort", 5500)
        certFile, cfok := cfg.GetString("http:httpsCert")
        keyFile, kfok := cfg.GetString("http:httpsKey")
        if cfok && kfok {
                  logger.Debugf("Starting HTTPS server on port %v",
httpsPort)
            wg.Add(1)
            go func() {
                    err := http.ListenAndServeTLS(fmt.Sprintf(":%v",
httpsPort),
                    certFile, keyFile, adaptor)
                if (err != nil) {
                    panic(err)
                }
            }()
        } else {
            panic("HTTPS certificate settings not found")
        }
    }
    return &wg
}
```

Листинг 33-10 Содержимое файла server.go в папке http

Функция `Serve` использует службу `Configuration` для считывания параметров HTTP и HTTPS и использует функции, предоставляемые стандартной библиотекой, для получения запросов и передачи их в конвейер для обработки. (Я включу поддержку HTTPS в главе 38, когда буду готовиться к развертыванию, но до тех пор я буду использовать настройки по умолчанию, которые прослушивают HTTP-запросы на порту 5000.)

Настройка приложения

Последним шагом является настройка конвейера, необходимого приложению, и использование его для настройки и запуска HTTP-сервера. Это задача, которая будет выполняться приложением после того, как я начну разработку в главе 35.

Однако сейчас добавьте файл с именем `startup.go` в папку `placeholder` с содержимым, показанным в листинге 33-11.

```
package placeholder

import (
    "platform/http"
    "platform/pipeline"
    "platform/pipeline/basic"
    "platform/services"
    "sync"
)

func createPipeline() pipeline.RequestPipeline {
    return pipeline.CreatePipeline(
        &basic.ServicesComponent{},
        &basic.LoggingComponent{},
        &basic.ErrorComponent{},
        &basic.StaticFileComponent{},
        &SimpleMessageComponent{},
    )
}

func Start() {
    results, err := services.Call(http.Serve, createPipeline())
    if (err == nil) {
        (results[0].(*sync.WaitGroup)).Wait()
    } else {
        panic(err)
    }
}
```

Листинг 33-11 Содержимое файла startup.go в папке placeholder

Функция `createPipeline` создает конвейер с ранее созданными компонентами промежуточного ПО. Функция `Start` вызывает `createPipeline` и использует результат для настройки и запуска HTTP-сервера. В листинге 33-12 функция `main` используется для завершения установки и запуска HTTP-сервера.

```
package main

import (
    "platform/services"
    "platform/placeholder"
)

func main() {
    services.RegisterDefaultServices()
```

```
    placeholder.Start()
}
```

Листинг 33-12 Завершение запуска приложения в файле main.go в папке platform

Скомпилируйте и запустите проект и используйте веб-браузер для запроса `http://localhost:5000`.

> **Работа с запросами разрешений брандмауэра Windows**
> Как объяснялось в предыдущих главах, Windows будет запрашивать разрешения брандмауэра каждый раз, когда проект компилируется с помощью команды `go run`. Вместо команды `go run` можно использовать простой сценарий Powershell, чтобы избежать этих запросов. Создайте файл `buildandrun.ps1` со следующим содержимым:
>
> ```
> $file = "./platform.exe"
>
> &go build -o $file
>
> if ($LASTEXITCODE -eq 0) {
>     &$file
> }
> ```
>
> Чтобы собрать и выполнить проект, используйте команду `./buildandrun.ps1` в папке `platform`.

HTTP-запрос будет получен сервером и передан по конвейеру, в результате чего будет получен ответ, показанный на рисунке 33-2. Запросите `http://localhost:5000/files/hello.json`, и вы увидите содержимое статического файла, также показанного на рисунке 33-2.

Вывод, подобный следующему, будет записан в стандартный вывод, показывающий, как сервер получает и обрабатывает запрос (вы также можете увидеть запросы для `/favicon.ico` в зависимости от вашего браузера):

```
20:10:12 DEBUG Starting HTTP server on port 5000
20:10:23 INFO REQ --- GET - /
20:10:23 INFO RSP 200 /
20:10:33 INFO REQ --- GET - /files/hello.json
20:10:33 INFO RSP 200 /files/hello.json
```

В настоящее время сервер одинаково отвечает на все запросы, кроме файлов, поэтому журнал показывает, что запросы к файлу `/favicon.ico` генерируют 200 ответов OK.

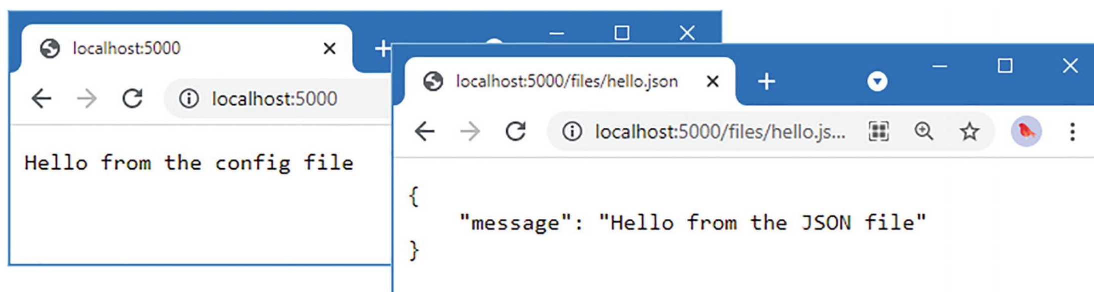

Рисунок 33-2 Получение ответа от HTTP-сервера

Оптимизация разрешения сервиса

В настоящее время компоненты промежуточного программного обеспечения должны напрямую разрешать службы, которые им требуются. Но поскольку система внедрения зависимостей может вызывать функции и заполнять структуры, небольшая дополнительная работа позволит компонентам объявлять службы, от которых они зависят, и получать их автоматически. Во-первых, необходим интерфейс, который позволит компонентам указывать, что им требуется внедрение зависимостей для обработки запросов, как показано в листинге 33-13.

```
package pipeline

import (
    "net/http"
)

type ComponentContext struct {
    *http.Request
    http.ResponseWriter
    error
}

func (mwc *ComponentContext) Error(err error) {
    mwc.error = err
}

func (mwc *ComponentContext) GetError() error {
    return mwc.error
}

type MiddlewareComponent interface {

    Init()
                    ProcessRequest(context    *ComponentContext,    next
func(*ComponentContext))
}
```

```
type ServicesMiddlwareComponent interface {
    Init()
    ImplementsProcessRequestWithServices()
}
```

Листинг 33-13 Определение интерфейса в файле component.go в папке pipeline

Реализуя метод с именем `ImplementsProcessRequestWithServices`, компоненты могут указать, что им требуются службы. Невозможно включить метод, которому требуются службы, в интерфейс, потому что каждому компоненту нужна своя сигнатура метода для требуемых служб. Вместо этого я собираюсь обнаружить `ServicesMiddlwareComponent`, а затем использовать отражение, чтобы определить, реализует ли компонент метод с именем `ProcessRequestWithServices`, первые два параметра которого совпадают с методом `ProcessRequest`, определенным интерфейсом `MiddlewareComponent`. В листинге 33-14 к функции, создающей конвейер, добавлена новая возможность, а также заполнение полей структуры компонента службами при подготовке конвейера.

```
package pipeline

import (
    "net/http"
    "platform/services"
    "reflect"
)

type RequestPipeline func(*ComponentContext)

var emptyPipeline RequestPipeline = func(*ComponentContext) { /* do
nothing */ }

func CreatePipeline(components ...interface{}) RequestPipeline {
    f := emptyPipeline
    for i := len(components) -1 ; i >= 0; i-- {
        currentComponent := components[i]
        services.Populate(currentComponent)
        nextFunc := f
                         if    servComp,    ok    :=    currentComponent.
(ServicesMiddlwareComponent ); ok {
                 f  =  createServiceDependentFunction(currentComponent,
nextFunc)
            servComp.Init()
                   }   else   if   stdComp,   ok   :=   currentComponent.
(MiddlewareComponent ); ok {
            f = func(context *ComponentContext) {
                if (context.error == nil) {
```

```
                    stdComp.ProcessRequest(context, nextFunc)
                }
            }
            stdComp.Init()
        } else {
            panic("Value is not a middleware component")
        }
    }
    return f
}

func createServiceDependentFunction(component interface{},
        nextFunc RequestPipeline) RequestPipeline {
                                                    method              :=
reflect.ValueOf(component).MethodByName("ProcessRequestWithServices")
    if (method.IsValid()) {
        return  func(context *ComponentContext) {
            if (context.error == nil) {
                                                              _,    err    :=
services.CallForContext(context.Request.Context(),
                    method.Interface(), context, nextFunc)
                if (err != nil) {
                    context.Error(err)
                }
            }
        }
    } else {
        panic("No ProcessRequestWithServices method defined")
    }
}

func (pl RequestPipeline) ProcessRequest(req *http.Request,
        resp http.ResponseWriter) error {
    ctx := ComponentContext {
        Request: req,
        ResponseWriter: resp,
    }
    pl(&ctx)
    return ctx.error
}
```

Листинг 33-14 Добавление поддержки для служб в файл pipe.go в папке pipeline

Эти изменения позволяют компоненту промежуточного программного обеспечения использовать преимущества внедрения зависимостей, так что зависимости от служб могут быть объявлены как параметры, как показано в листинге 33-15.

```
package basic

import (
    "net/http"
    "platform/logging"
    "platform/pipeline"
    //"platform/services"
)

type LoggingResponseWriter struct {
    statusCode int
    http.ResponseWriter
}

func (w *LoggingResponseWriter) WriteHeader(statusCode int) {
    w.statusCode = statusCode
    w.ResponseWriter.WriteHeader(statusCode)
}

func (w *LoggingResponseWriter) Write(b []byte) (int, error) {
    if (w.statusCode == 0) {
        w.statusCode = http.StatusOK
    }
    return w.ResponseWriter.Write(b)
}

type LoggingComponent struct {}

func (lc *LoggingComponent) ImplementsProcessRequestWithServices() {}

func (lc *LoggingComponent) Init() {}

func (lc *LoggingComponent) ProcessRequestWithServices(
    ctx *pipeline.ComponentContext,
    next func(*pipeline.ComponentContext),
    logger logging.Logger)  {

    // var logger logging.Logger
      //  err  :=  services.GetServiceForContext(ctx.Request.Context(),
&logger)
    // if (err != nil) {
    //     ctx.Error(err)
    //     return
    // }

    loggingWriter := LoggingResponseWriter{ 0, ctx.ResponseWriter}
    ctx.ResponseWriter = &loggingWriter
```

```
        logger.Infof("REQ   ---   %v   -   %v",   ctx.Request.Method,
ctx.Request.URL)
    next(ctx)
          logger.Infof("RSP   %v   %v",   loggingWriter.statusCode,
ctx.Request.URL )
}
```

Листинг 33-15 Использование внедрения зависимостей в файле logging.go в папке pipeline/basic

Определение метода `ImplementsProcessRequestWithServices` реализует интерфейс, который конвейер использует в качестве указания на наличие метода `ProcessRequestWithServices`, требующего внедрения зависимостей. Компоненты также могут полагаться на службы, разрешенные через их поля структуры, как показано в листинге 33-16.

```
package basic

import (
    "net/http"
    "platform/config"
    "platform/pipeline"
    //"platform/services"
    "strings"
)

type StaticFileComponent struct {
    urlPrefix string
    stdLibHandler http.Handler
    Config config.Configuration
}

func (sfc *StaticFileComponent) Init() {
    // var cfg config.Configuration
    // services.GetService(&cfg)
      sfc.urlPrefix = sfc.Config.GetStringDefault("files:urlprefix",
"/files/")
    path, ok := sfc.Config.GetString("files:path")
    if (ok) {
        sfc.stdLibHandler = http.StripPrefix(sfc.urlPrefix,
            http.FileServer(http.Dir(path)))
    } else {
        panic ("Cannot load file configuration settings")
    }
}

func       (sfc       *StaticFileComponent)       ProcessRequest(ctx
*pipeline.ComponentContext,
    next func(*pipeline.ComponentContext)) {
```

```
    if  !strings.EqualFold(ctx.Request.URL.Path, sfc.urlPrefix) &&
            strings.HasPrefix(ctx.Request.URL.Path, sfc.urlPrefix) {
        sfc.stdLibHandler.ServeHTTP(ctx.ResponseWriter, ctx.Request)
    } else {
        next(ctx)
    }
}
```

Листинг 33-16 Использование внедрения зависимостей в файле files.go в папке the pipeline/basic

Скомпилируйте и выполните проект и используйте браузер для запроса `http://localhost:5000` и `http://localhost:5000/files/hello.json`, что даст те же результаты, что и в предыдущем разделе. Вы можете увидеть результат 304 при запросе файла JSON, так как он не изменился с момента запроса в предыдущем разделе.

Создание HTML-ответов

Я описал функции обработки HTML-шаблонов в главе 23, но они работают не так, как я думаю об HTML-контенте. Я хочу иметь возможность определить шаблон HTML и указать общий макет, который будет использоваться в этом шаблоне. Это противоположно стандартному подходу пакета `html/template`, но поведение по умолчанию легко настроить для получения желаемого эффекта.

> **Примечание**
> Поскольку я изменяю порядок обработки шаблонов на обратный, шаблоны не могут использовать блочные функции для предоставления контента по умолчанию для шаблона, который переопределяется другим шаблоном.

Создание макета и шаблона

Процесс адаптации механизма шаблонов упрощается, когда вы знаете, какой будет структура шаблонов. Добавьте файл с именем `simple_message.html` в папку `platform/placeholder` с содержимым, показанным в листинге 33-17.

```
{{ layout "layout.html" }}

<h3>
    Hello from the template
</h3>
```

Листинг 33-17 Содержимое файла simple_message.html в папке placeholder

Этот шаблон задает требуемый макет с помощью выражения `layout`, но в остальном является стандартным шаблоном, использующим функции,

описанные в главе 23. Шаблон содержит элемент h3, содержимое которого включает действие, которое вставляет значение данных.

Чтобы определить макет, добавьте файл с именем layout.html в папку placeholder с содержимым, показанным в листинге 33-18.

```
<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>Pro Go</title>
</head>
<body>
    <h2>Hello from the layout</h2>
    {{ body }}
</body>
</html>
```

Листинг 33-18 Содержимое файла layout.html в папке placeholder

Макет содержит элементы, необходимые для определения HTML-документа, с добавлением действия, содержащего выражение body, которое будет вставлять содержимое выбранного шаблона в вывод.

Для рендеринга контента будет выбран и выполнен шаблон, который определит макет, который ему требуется. Макет также будет обработан и объединен с содержимым из шаблона для получения полного ответа в формате HTML. Это подход, который я предпочитаю, отчасти потому, что он позволяет избежать необходимости знать, какой макет требуется при выборе шаблона, а отчасти потому, что я привык к этому на других языках и платформах.

Реализация выполнения шаблона

Встроенный пакет шаблонов превосходен и упрощает поддержку модели, в которой шаблоны определяют свои макеты. Создайте папку platform/templates и добавьте в нее файл с именем template_executor.go с содержимым, показанным в листинге 33-19.

```
package templates

import "io"

type TemplateExecutor interface {

      ExecTemplate(writer io.Writer, name string, data interface{})
(err error)
}
```

Листинг 33-19 Содержимое файла template_executor.go в папке templates

Интерфейс TemplateProcessor определяет метод с именем ExecTemplate, который обрабатывает шаблон с использованием предоставленных значений данных и записывает содержимое в Writer. Чтобы создать реализацию интерфейса, добавьте файл с именем layout_executor.go в папку templates с содержимым, показанным в листинге 33-20.

```
package templates

import (
    "io"
    "strings"
    "html/template"
)

type LayoutTemplateProcessor struct {}

func (proc *LayoutTemplateProcessor) ExecTemplate(writer io.Writer,
        name string, data interface{}) (err error) {
    var sb strings.Builder
    layoutName := ""
    localTemplates := getTemplates()
    localTemplates.Funcs(map[string]interface{} {
        "body": insertBodyWrapper(&sb),
        "layout": setLayoutWrapper(&layoutName),
    })
    err = localTemplates.ExecuteTemplate(&sb, name, data)
    if (layoutName != "") {
        localTemplates.ExecuteTemplate(writer, layoutName, data)
    } else {
        io.WriteString(writer, sb.String())
    }
    return
}

var getTemplates func() (t *template.Template)

func insertBodyWrapper(body *strings.Builder) func() template.HTML {
    return func() template.HTML {
        return template.HTML(body.String())
    }
}

func setLayoutWrapper(val *string) func(string) string {
    return func(layout string) string {
        *val = layout
        return ""
    }
}
```

Листинг 33-20 Содержимое файла layout_executor.go в папке templates

Реализация метода `ExecTemplate` выполняет шаблон и сохраняет содержимое в файле `strings.Builder`. Для поддержки выражений макета и тела, описанных в предыдущем разделе, создаются пользовательские функции шаблона, например:

```
...
localTemplates.Funcs(map[string]interface{} {
    "body": insertBodyWrapper(&sb),
    "layout": setLayoutWrapper(&layoutName),
})
...
```

Когда встроенный механизм шаблонов встречает выражение `template`, он вызывает функцию, созданную `setLayoutWrapper`, которая устанавливает значение переменной, которая затем используется для выполнения указанного шаблона макета. Во время выполнения макета выражение `body` вызывает функцию, созданную функцией `insertBodyWrapper`, которая вставляет содержимое, сгенерированное исходным шаблоном, в выходные данные, полученные из макета. Чтобы предотвратить экранирование символов HTML встроенным механизмом шаблонов, результатом этой операции является значение `template.HTML`:

```
...
func insertBodyWrapper(body *strings.Builder) func() template.HTML {
    return func() template.HTML {
        return template.HTML(body.String())
    }
}
...
```

Как объяснялось в главе 23, система шаблонов Go автоматически кодирует содержимое, чтобы сделать его безопасным для включения в HTML-документы. Обычно это полезная функция, но в данном случае экранирование содержимого из шаблона при его вставке в макет предотвратит его интерпретацию как HTML.

Метод `ExecTemplate` получает загруженные шаблоны, вызывая функцию с именем `getTemplates`, для которой в листинге 33-20 определена переменная. Чтобы добавить поддержку загрузки шаблонов и создания значения функции, которое будет присвоено переменной `getTemplates`, добавьте файл с именем `template_loader.go` в папку `templates` с содержимым, показанным в листинге 33-21.

```
package templates
```

```
import (
    "html/template"
    "sync"
    "errors"
    "platform/config"
)

var once = sync.Once{}

func LoadTemplates(c config.Configuration) (err error) {
    path, ok := c.GetString("templates:path")
    if !ok {
        return errors.New("Cannot load template config")
    }
    reload := c.GetBoolDefault("templates:reload", false)
    once.Do(func() {
        doLoad := func() (t *template.Template) {
            t = template.New("htmlTemplates")
            t.Funcs(map[string]interface{} {
                "body": func() string { return "" },
                "layout": func() string { return "" },
            })
            t, err = t.ParseGlob(path)
            return
        }
        if (reload) {
            getTemplates = doLoad
        } else {
            var templates *template.Template
            templates = doLoad()
            getTemplates = func() *template.Template {
                t, _ := templates.Clone()
                return t
            }
        }
    })
    return
}
```

Листинг 33-21 Содержимое файла template_loader.go в папке templates

Функция `LoadTemplates` загружает шаблон из места, указанного в файле конфигурации. Существует также параметр конфигурации, который включает перезагрузку для каждого запроса, что не следует делать в развернутом проекте, но полезно во время разработки, поскольку это означает, что изменения в шаблонах можно увидеть без перезапуска приложения. Листинг 33-22 добавляет новые настройки в файл конфигурации.

```
{
    "logging" : {
        "level": "debug"
    },
    "main" : {
        "message" : "Hello from the config file"
    },
    "files": {
        "path": "placeholder/files"
    },
    "templates": {
        "path": "placeholder/*.html",
        "reload": true
    }
}
```

Листинг 33-22 Добавление настроек в файл config.json в папке platform

Значение, полученное через настройку `reload`, определяет функцию, назначенную переменной `getTemplates`. Если `reload` имеет значение `true`, то вызов `getTemplates` загрузит шаблоны с диска; если `false`, то будут клонированы ранее загруженные шаблоны.

Клонирование или повторная загрузка шаблонов необходимы для обеспечения правильной работы пользовательского `body` и функций `layout`. Функция `LoadTemplates` определяет функции-заполнители, чтобы можно было анализировать шаблоны при их загрузке.

Создание и использование службы шаблонов

Пользовательский механизм шаблонов будет доступен как услуга. Добавьте операторы, показанные в листинге 33-23, в файл `services_default.go` в папке `platform/services`.

```
package services

import (
    "platform/logging"
    "platform/config"
    "platform/templates"
)

func RegisterDefaultServices() {

    err := AddSingleton(func() (c config.Configuration) {
        c, loadErr :=  config.Load("config.json")
        if (loadErr != nil) {
            panic(loadErr)
        }
```

```
        return
    })

        err = AddSingleton(func(appconfig config.Configuration)
logging.Logger {
        return logging.NewDefaultLogger(appconfig)
    })
    if (err != nil) {
        panic(err)
    }

    err = AddSingleton(
        func(c config.Configuration) templates.TemplateExecutor {
            templates.LoadTemplates(c)
            return &templates.LayoutTemplateProcessor{}
        })
    if (err != nil) {
        panic(err)
    }
}
```

Листинг 33-23 Создание службы шаблонов в файле services_default.go в папке services

Чтобы убедиться, что механизм шаблонов работает, внесите изменения, показанные в листинге 33-24, в промежуточный программный компонент-заполнитель, созданный ранее в этой главе, чтобы он возвращал ответ в формате HTML, а не простую строку.

```
package placeholder

import (
    //"io"
    //"errors"
    "platform/pipeline"
    "platform/config"
    //"platform/services"
    "platform/templates"
)

type SimpleMessageComponent struct {
    Message string
    config.Configuration
}

func (lc *SimpleMessageComponent)
ImplementsProcessRequestWithServices() {}

func (c *SimpleMessageComponent) Init() {
```

```
    c.Message = c.Configuration.GetStringDefault("main:message",
        "Default Message")
}

func (c *SimpleMessageComponent) ProcessRequestWithServices(
    ctx *pipeline.ComponentContext,
    next func(*pipeline.ComponentContext),
    executor templates.TemplateExecutor)  {
    err := executor.ExecTemplate(ctx.ResponseWriter,
        "simple_message.html", c.Message)
    if (err != nil) {
        ctx.Error(err)
    } else {
        next(ctx)
    }
}
```

Листинг 33-24 Использование шаблона в файле message_middleware.go в папке placeholder

Компонент теперь реализует метод `ProcessRequestWithServices` и получает службы посредством внедрения зависимостей. Одна из запрашиваемых служб — это реализация интерфейса `TemplateExecutor`, который используется для отображения шаблона `simple_message.html`. Скомпилируйте и выполните проект и используйте браузер для запроса `http://localhost:5000`, и вы увидите ответ в формате HTML, показанный на рисунке 33-3.

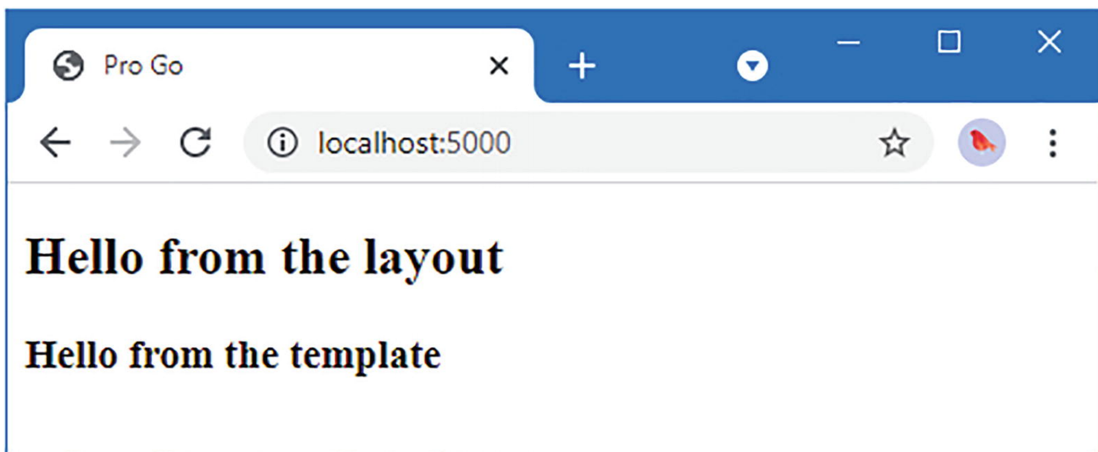

Рисунок 33-3 Создание HTML-ответа

Знакомство с обработчиками запросов

Следующим шагом является введение поддержки для определения логики, которая будет обрабатывать HTTP-запрос и давать соответствующий ответ, что позволит мне писать код, отвечающий на определенные URL-адреса, без необходимости повторять слишком много кода. Чтобы понять, как это будет

работать, лучше всего начать с примера обработчика запросов, который будет определен как тип с набором методов, обрабатывающих запросы. Добавьте файл с именем name_handler.go в папку-заполнитель с содержимым, показанным в листинге 33-25.

```
package placeholder

import (
    "fmt"
    "platform/logging"
)

var names = []string{"Alice", "Bob", "Charlie", "Dora"}

type NameHandler struct {
          logging.Logger
}

func (n NameHandler) GetName(i int) string {
    n.Logger.Debugf("GetName method invoked with argument: %v", i)
    if (i < len(names)) {
        return fmt.Sprintf("Name #%v: %v", i, names[i])
    } else {
        return fmt.Sprintf("Index out of bounds")
    }
}

func (n NameHandler) GetNames() string {
    n.Logger.Debug("GetNames method invoked")
    return fmt.Sprintf("Names: %v", names)
}

type NewName struct {
    Name string
    InsertAtStart bool
}

func (n NameHandler) PostName(new NewName) string {
    n.Logger.Debugf("PostName method invoked with argument %v", new)
    if (new.InsertAtStart) {
        names = append([] string { new.Name}, names... )
    } else {
        names = append(names, new.Name)
    }
    return fmt.Sprintf("Names: %v", names)
}
```

Листинг 33-25 Содержимое файла name_handler.go в папке placeholder

Структура NameHandler определяет три метода: GetName, GetNames и PostName. При запуске приложения будет проверен набор зарегистрированных обработчиков, и имена определяемых ими методов будут использоваться для создания маршрутов, соответствующих HTTP-запросам.

Первая часть имени каждого метода указывает метод HTTP, которому будет соответствовать маршрут, так что, например, метод GetName будет соответствовать запросам GET. Остальная часть имени метода будет использоваться в качестве первого сегмента пути URL, соответствующего маршруту, с добавлением дополнительных сегментов для параметров запросов GET.

В таблице 33-1 показаны сведения о запросах, которые будут обрабатываться методами, определенными в листинге 33-25.

Таблица 33-1 Запросы, сопоставленные примерами методов обработчика

Функция	HTTP метод	Пример URL
GetName	GET	/name/1
GetNames	GET	/names
PostName	POST	/names

Когда приходит запрос, соответствующий маршруту метода, значения параметров получаются из URL-адреса запроса, строки запроса и, если она присутствует, из формы запроса. Если тип параметра метода является структурой, то его поля будут заполнены теми же данными запроса.

Службы, необходимые для обработки запросов, объявляются как поля, определенные структурой обработчика. В листинге 33-25 структура NameHandler определяет поле, объявляющее зависимость от службы logging.Logger. Будет создан новый экземпляр структуры, его поля будут заполнены, а затем будет вызван метод, выбранный для обработки запроса.

Генерация URL-маршрутов

Первым шагом является добавление поддержки для создания маршрутов URL из методов обработчика запросов. Создайте папку platform/http/handling и добавьте в нее файл с именем route.go с содержимым, показанным в листинге 33-26.

```
package handling

import (
    "reflect"
    "regexp"
    "strings"
    "net/http"
)
```

```
type HandlerEntry struct {
    Prefix string
    Handler interface{}
}

type Route struct {
    httpMethod string
    prefix string
    handlerName string
    actionName string
    expression regexp.Regexp
    handlerMethod reflect.Method
}

var httpMethods = []string { http.MethodGet, http.MethodPost,
    http.MethodDelete, http.MethodPut }

func generateRoutes(entries ...HandlerEntry) []Route {
    routes := make([]Route, 0, 10)
    for _, entry := range entries {
        handlerType := reflect.TypeOf(entry.Handler)
        promotedMethods := getAnonymousFieldMethods(handlerType)

        for i := 0; i < handlerType.NumMethod(); i++ {
            method := handlerType.Method(i)
            methodName := strings.ToUpper(method.Name)
            for _, httpMethod := range httpMethods {
                if strings.Index(methodName, httpMethod) == 0 {
                               if  (matchesPromotedMethodName(method,
promotedMethods)) {
                        continue
                    }
                    route := Route{
                        httpMethod: httpMethod,
                        prefix: entry.Prefix,
                                                          handlerName:
strings.Split(handlerType.Name(), "Handler")[0],
                               actionName:  strings.Split(methodName,
httpMethod)[1],
                        handlerMethod: method,
                    }
                    generateRegularExpression(entry.Prefix, &route)
                    routes = append(routes, route)
                }
            }
        }
    }
    return routes
```

```
}

func matchesPromotedMethodName(method reflect.Method,
       methods []reflect.Method) bool {
    for _, m := range methods {
        if m.Name == method.Name {
            return true
        }
    }
    return false
}

func getAnonymousFieldMethods(target []reflect.Type) reflect.Method {
    methods := []reflect.Method {}
    for i := 0; i < target.NumField(); i++ {
        field := target.Field(i)
        if (field.Anonymous && field.IsExported()) {
            for j := 0; j < field.Type.NumMethod(); j++ {
                method := field.Type.Method(j)
                if (method.IsExported()) {
                    methods = append(methods, method)
                }
            }
        }
    }
    return methods
}

func generateRegularExpression(prefix string, route *Route) {
    if (prefix != "" && !strings.HasSuffix(prefix, "/")) {
        prefix += "/"
    }
    pattern := "(?i)" + "/" + prefix + route.actionName
    if (route.httpMethod == http.MethodGet) {
        for i := 1; i < route.handlerMethod.Type.NumIn(); i++ {
            if route.handlerMethod.Type.In(i).Kind() == reflect.Int {
                pattern += "/([0-9]*)"
            } else {
                pattern += "/([A-z0-9]*)"
            }
        }
    }
    pattern = "^" + pattern + "[/]?$"
    route.expression = *regexp.MustCompile(pattern)
}
```

Листинг 33-26 Содержимое файла route.go в папке http/handling

Маршруты будут настроены с необязательным префиксом, что позволит мне создавать разные URL-адреса для разных частей приложения, например, когда я ввожу управление доступом в главе 34. Структура HandlerEntry описывает обработчик и его префикс, а структура Route определяет обработанный результат для одного маршрута. Функция generateRoutes создает значения Route для методов, определенных обработчиком, полагаясь на функцию generateRegularExpression для создания и компиляции регулярных выражений, которые будут использоваться для сопоставления путей URL.

> **Примечание**
> Как отмечалось в главе 28, методы, продвинутые из анонимных встроенных полей, включаются при использовании отражения в структуре. Код в листинге 33-26 отфильтровывает эти продвинутые методы, чтобы предотвратить создание маршрутов, позволяющих использовать эти методы в HTTP-запросах.

Подготовка значений параметров для метода обработчика

Когда HTTP-запрос получен и сопоставлен с маршрутом, значения должны быть извлечены из запроса, чтобы их можно было использовать в качестве аргументов для метода обработчика. Все значения, которые могут быть получены из запроса, выражаются с использованием типа string Go, поскольку HTTP не поддерживает включение информации о типе в URL-адреса или данные формы. Я мог бы передать строковые значения из запроса в метод обработчика, но это просто означает, что каждый метод обработчика должен будет пройти процесс разбора строковых значений в требуемые типы. Вместо этого я собираюсь автоматически анализировать значения на основе типа параметра метода обработчика, что позволяет определить код один раз. Создайте папку http/handling/params и добавьте в нее файл с именем parser.go с содержимым, показанным в листинге 33-27.

```
package params

import (
    "reflect"
    "fmt"
    "strconv"
)

func parseValueToType(target reflect.Type, val string) (result
reflect.Value,
        err error) {
    switch target.Kind() {
        case reflect.String:
```

```
            result = reflect.ValueOf(val)
        case reflect.Int:
            iVal, convErr := strconv.Atoi(val)
            if convErr == nil {
                result = reflect.ValueOf(iVal)
            } else {
                return reflect.Value{}, convErr
            }
        case reflect.Float64:
            fVal, convErr := strconv.ParseFloat(val, 64)
            if (convErr == nil) {
                result = reflect.ValueOf(fVal)
            } else {
                return reflect.Value{}, convErr
            }
        case reflect.Bool:
            bVal, convErr := strconv.ParseBool(val)
            if (convErr == nil) {
                result = reflect.ValueOf(bVal)
            } else {
                return reflect.Value{}, convErr
            }
        default:
             err = fmt.Errorf("Cannot use type %v as handler method
parameter",
                target.Name())
        }
    return
}
```

Листинг 33-27 Содержимое файла parser.go в папке http/handling/params

Функция parseValueToType проверяет тип требуемого типа и использует функции, определенные пакетом strconv, для преобразования значения в ожидаемый тип. Я собираюсь поддерживать четыре основных типа: string, float64, int и bool. Я также буду поддерживать структуры, поля которых относятся к этим четырем типам. Функция parseValueToType возвращает error, если параметр определен с другим типом или значение, полученное в запросе, не может быть проанализировано.

Следующим шагом является использование функции parseValueToType для работы с методами обработчика, которые определяют параметры четырех поддерживаемых типов, таких как метод GetName, определенный в листинге 33-25:

```
...
func (n NameHandler) GetName(i int) string {
...
```

Значения для этого типа параметра будут получены из регулярного выражения, сгенерированного при регистрации обработчика. Добавьте файл с именем `simple_params.go` в папку `http/handling/params` с содержимым, показанным в листинге 33-28.

```
package params

import (
    "reflect"
    "errors"
)

func getParametersFromURLValues(funcType reflect.Type,
        urlVals []string) (params []reflect.Value, err error) {
    if (len(urlVals) == funcType.NumIn() -1) {
        params = make([]reflect.Value, funcType.NumIn() -1)
        for i := 0; i < len(urlVals); i++ {
            params[i], err = parseValueToType(funcType.In(i + 1),
urlVals[i])
            if (err != nil) {
                return
            }
        }
    } else {
        err = errors.New("Parameter number mismatch")
    }
    return
}
```

Листинг 33-28 Содержимое файла simple_params.go в папке http/handling/params

Функция `getParametersFromURLValues` проверяет параметры, определенные методом обработчика, и вызывает функцию `parseValueToType`, чтобы попытаться получить значение для каждого из них. Обратите внимание, что я пропускаю первый параметр, определенный методом. Как объяснялось в главе 28, при использовании отражения первым параметром является получатель, для которого вызывается метод.

Методы-обработчики, которым требуется доступ к значениям из строки запроса URL или данных формы, могут сделать это, определив параметр, тип которого является структурой с именами полей, которые соответствуют именам значений данных запроса, как этот метод, определенный в листинге 33-25:

```
...
type NewName struct {
    Name string
    InsertAtStart bool
}
```

```
func (n NameHandler) PostName(new NewName) string {
...
```

Этот параметр указывает, что методу обработчика требуются значения name и insertAtStart из запроса. Чтобы заполнить поля структуры из запроса, добавьте файл с именем struct_params.go в папку http/handling/params с содержимым, показанным в листинге 33-29.

```
package params

import (
    "reflect"
    "encoding/json"
    "io"
    "strings"
)

func populateStructFromForm(structVal reflect.Value,
        formVals map[string][]string) (err error) {
    for i := 0; i < structVal.Elem().Type().NumField(); i++ {
        field := structVal.Elem().Type().Field(i)
        for key, vals := range formVals {
            if strings.EqualFold(key, field.Name) && len(vals) > 0 {
                valField := structVal.Elem().Field(i)
                if (valField.CanSet()) {
                                                    valToSet,   convErr   :=
parseValueToType(valField.Type(), vals[0])
                    if (convErr == nil) {
                        valField.Set(valToSet)
                    } else {
                        err = convErr
                    }
                }
            }
        }
    }
    return
}

func populateStructFromJSON(structVal reflect.Value,
        reader io.ReadCloser) (err error) {
    return json.NewDecoder(reader).Decode(structVal.Interface())
}
```

Листинг 33-29 Содержимое файла struct_params.go в папке http/handling/params

Функция populateStructFromForm будет использоваться для любого метода обработчика, который требует структуру и устанавливает значения полей

структуры из карты. Функция populateStructFromJSON использует декодер JSON для чтения тела запроса и будет использоваться, когда запрос содержит полезные данные JSON. Чтобы применить эти функции, добавьте файл с именем processor.go в папку http/handling/params с содержимым, показанным в листинге 33-30.

```
package params

import (
    "net/http"
    "reflect"
)

func GetParametersFromRequest(request *http.Request, handlerMethod reflect.Method,
        urlVals []string) (params []reflect.Value, err error) {
    handlerMethodType := handlerMethod.Type
    params = make([]reflect.Value, handlerMethodType.NumIn() -1)
    if (handlerMethodType.NumIn() == 1) {
        return []reflect.Value {}, nil
    } else if handlerMethodType.NumIn() == 2 &&
            handlerMethodType.In(1).Kind() == reflect.Struct {
        structVal := reflect.New(handlerMethodType.In(1))
        err = request.ParseForm()
                if err == nil && getContentType(request) == "application/json" {
            err = populateStructFromJSON(structVal, request.Body)
        }
        if err == nil {
            err = populateStructFromForm(structVal, request.Form)
        }
        return []reflect.Value { structVal.Elem() }, err
    } else {
        return getParametersFromURLValues(handlerMethodType, urlVals)
    }
}

func getContentType(request *http.Request) (contentType string) {
    headerSlice := request.Header["Content-Type"]
    if headerSlice != nil && len(headerSlice) > 0 {
        contentType = headerSlice[0]
    }
    return
}
```

Листинг 33-30 Содержимое файла process.go в папке http/handling/params

GetParametersFromRequest экспортируется для использования в другом месте проекта. Он получает запрос, метод отраженного обработчика и срез, содержащий значения, соответствующие маршруту. Метод проверяется, чтобы увидеть, требуется ли параметр структуры, и параметры, необходимые методу, создаются с использованием ранее функций.

Сопоставление запросов с маршрутами

Последним шагом в этой главе является сопоставление входящих HTTP-запросов с маршрутами и выполнение метода-обработчика для генерации ответа. Добавьте файл с именем request_dispatch.go в папку http/handling с содержимым, показанным в листинге 33-31.

```
package handling

import (
    "platform/http/handling/params"
    "platform/pipeline"
    "platform/services"
    "net/http"
    "reflect"
    "strings"
    "io"
    "fmt"
)

func NewRouter(handlers ...HandlerEntry) *RouterComponent {
    return &RouterComponent{ generateRoutes(handlers...) }
}

type RouterComponent struct {
    routes []Route
}

func (router *RouterComponent) Init() {}

func      (router      *RouterComponent)      ProcessRequest(context
*pipeline.ComponentContext,
        next func(*pipeline.ComponentContext)) {
    for _, route := range router.routes {
                  if   (strings.EqualFold(context.Request.Method,
route.httpMethod)) {
                                                   matches     :=
route.expression.FindAllStringSubmatch(context.URL.Path, -1)
            if len(matches) > 0 {
                rawParamVals := []string {}
                if len(matches[0]) > 1 {
                    rawParamVals = matches[0][1:]
```

```
                }
                    err := router.invokeHandler(route, rawParamVals,
context)
                if (err == nil) {
                    next(context)
                } else {
                    context.Error(err)
                }
                return
            }
        }
    }
    context.ResponseWriter.WriteHeader(http.StatusNotFound)
}

func (router *RouterComponent) invokeHandler(route Route, rawParams
[]string,
        context *pipeline.ComponentContext) error {
                            paramVals,      err      :=
params.GetParametersFromRequest(context.Request,
        route.handlerMethod, rawParams)
    if (err == nil) {
        structVal := reflect.New(route.handlerMethod.Type.In(0))
                services.PopulateForContext(context.Context(),
structVal.Interface())
            paramVals = append([]reflect.Value { structVal.Elem() },
paramVals...)
        result := route.handlerMethod.Func.Call(paramVals)
                        io.WriteString(context.ResponseWriter,
fmt.Sprint(result[0].Interface()))
    }
    return err
}
```

Листинг 33-31 Содержимое файла request_dispatch.go в папке http/handling

Функция `NewRouter` используется для создания нового компонента промежуточного программного обеспечения, который обрабатывает запросы с использованием маршрутов, которые генерируются из ряда значений `HandlerEntry`. Структура `RouterComponent` реализует интерфейс `MiddlewareComponent`, а ее метод `ProcessRequest` сопоставляет маршруты с использованием метода HTTP и пути URL. Когда соответствующий маршрут найден, вызывается функция `invokeHandler`, которая подготавливает значения для параметров, определенных методом-обработчиком, который затем вызывается.

Этот компонент промежуточного программного обеспечения был написан с учетом того, что он применяется в конце конвейера, что означает, что ответ 404

— Not Found возвращается, если ни один из маршрутов не соответствует запросу.

Последнее, что следует отметить, это то, что ответ, созданный методом обработчика, просто записывается в виде строки, например:

```
...
io.WriteString(context.ResponseWriter,
fmt.Sprint(result[0].Interface()))
...
```

Это шаг назад по сравнению с шаблонами, представленными ранее в этой главе, но я рассмотрю это в главе 34. В листинге 33-32 изменена конфигурация заполнителя для использования нового компонента промежуточного программного обеспечения маршрутизации.

```
package placeholder

import (
    "platform/http"
    "platform/pipeline"
    "platform/pipeline/basic"
    "platform/services"
    "sync"
    "platform/http/handling"
)

func createPipeline() pipeline.RequestPipeline {
    return pipeline.CreatePipeline(
        &basic.ServicesComponent{},
        &basic.LoggingComponent{},
        &basic.ErrorComponent{},
        &basic.StaticFileComponent{},
        //&SimpleMessageComponent{},
        handling.NewRouter(
            handling.HandlerEntry{ "",  NameHandler{}},
        ),
    )
}

func Start() {
    results, err := services.Call(http.Serve, createPipeline())
    if (err == nil) {
        (results[0].(*sync.WaitGroup)).Wait()
    } else {
        panic(err)
    }
}
```

Листинг 33-32 Настройка приложения в файле startup.go в папке placeholder

Скомпилируйте и запустите проект и используйте браузер для запроса `http://localhost:5000/names`. Этот URL-адрес будет соответствовать маршруту для метода `GetNames`, определенному обработчиком запросов-заполнителей, и даст результат, показанный на рисунке 33-4.

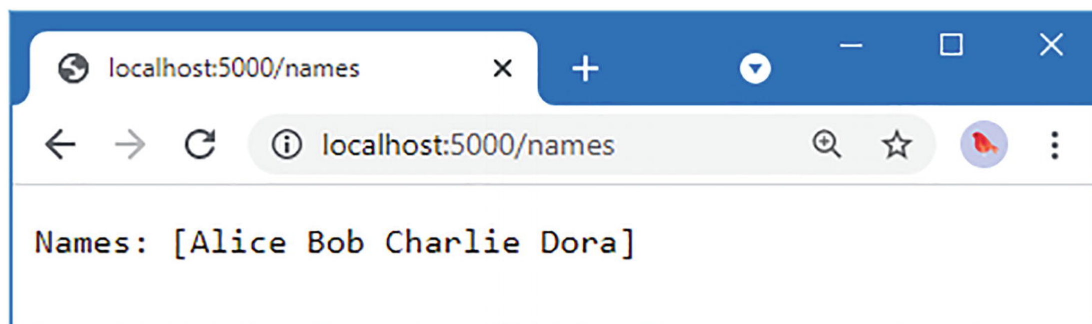

Рисунок 33-4 Использование обработчика запросов для генерации ответа

Чтобы проверить поддержку простых параметров метода обработчика, используйте браузер для запроса `http://localhost:5000/name/0` и `http://localhost:5000/name/100`. Обратите внимание, что именно name (в единственном числе), а не `names` (во множественном числе) в этих URL-адресах создают ответы, показанные на рисунке 33-5.

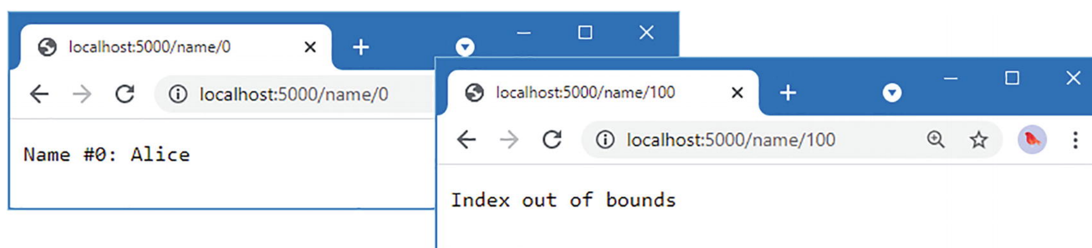

Рисунок 33-5 Нацеливание на метод обработчика запросов с помощью простого параметра

Чтобы проверить отправку запроса POST, выполните команду, показанную в листинге 33-33, из командной строки.

```
curl --header "Content-Type: application/json" --request POST --data
'{"name" : "Edith","insertatstart" : false}'
http://localhost:5000/name
```

Листинг 33-33 Отправка запроса POST с данными JSON

Если вы используете Windows, вместо этого выполните команду, показанную в листинге 33-34, в командной строке PowerShell.

```
Invoke-WebRequest http://localhost:5000/name -Method Post -Body `
```

```
(@{name="Edith";insertatstart=$false} | ConvertTo-Json) `
-ContentType "application/json"
```

Листинг 33-34 Отправка запроса POST с данными JSON в Windows

Эти команды отправляют на сервер один и тот же запрос, результат которого можно увидеть, запросив `http://localhost:5000/names`, как показано на рисунке 33-6.

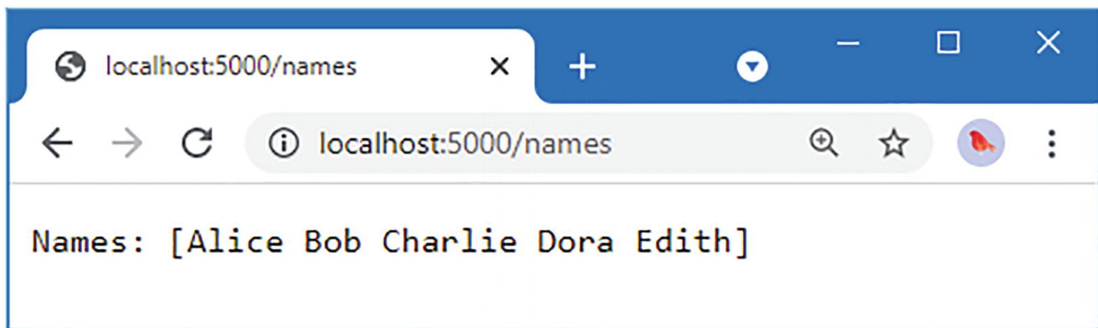

Рисунок 33-6 Эффект отправки POST-запроса

Резюме

В этой главе я продолжил разработку платформы веб-приложений, создав конвейер, который использует компоненты промежуточного программного обеспечения для обработки запросов. Я добавил поддержку шаблонов, которые могут указывать свои макеты, и представил обработчики запросов, которые я буду развивать в следующей главе.

34. Действия, сеансы и авторизация

В этой главе я завершаю разработку пользовательской платформы веб-приложений, начатую в главе 32 и продолженную в главе 33.

> **Подсказка**
> Вы можете загрузить пример проекта для этой главы — и для всех остальных глав этой книги — с `https://github.com/apress/pro-go`. См. Главу 2 о том, как получить помощь, если у вас возникнут проблемы с запуском примеров.

Представляем результаты действий

На данный момент платформа обрабатывает ответы, сгенерированные обработчиками запросов, записывая их в виде строк. Я не хочу заставлять каждый метод обработчика иметь дело со спецификой генерации ответа, потому что большинство ответов будут похожими — по большей части рендеринг шаблона — и я не хочу каждый раз дублировать один и тот же код.

Вместо этого я собираюсь добавить поддержку результатов действий, которые представляют собой инструкции о том, какой тип ответа требуется, а также любую дополнительную информацию, необходимую для его получения. Когда метод-обработчик хочет отобразить шаблон в качестве своего ответа, он вернет результат действия, который выбирает шаблон, и действие будет выполнено без необходимости понимания методом-обработчиком того, как это происходит. Создайте папку `platform/http/actionresults` и добавьте в нее файл с именем `actionresult.go` с содержимым, показанным в листинге 34-1.

```
package actionresults

import (
    "context"
    "net/http"
)

type ActionContext struct {
    context.Context
    http.ResponseWriter
}

type ActionResult interface {

    Execute(*ActionContext) error
}
```

Листинг 34-1 Содержимое файла actionresult.go в папке http/actionresults

Интерфейс ActionResult определяет метод Execute, который будет использоваться для генерации ответа с использованием средств, предоставляемых структурой ActionContext, а именно Context (для получения услуг) и ResponseWriter (для генерации ответа).

Листинг 34-2 обновляет код, который вызывает методы обработчика, чтобы он выполнял результаты действия, когда они используются.

```
package handling

import (
    "platform/http/handling/params"
    "platform/pipeline"
    "platform/services"
    "net/http"
    "reflect"
    "strings"
    "io"
    "fmt"
    "platform/http/actionresults"
)

// ...functions and types omitted for brevity...

func (router *RouterComponent) invokeHandler(route Route, rawParams
[]string,
      context *pipeline.ComponentContext) error {
    paramVals, err := params.GetParametersFromRequest(context.Request,
        route.handlerMethod, rawParams)
    if (err == nil) {
        structVal := reflect.New(route.handlerMethod.Type.In(0))
                            services.PopulateForContext(context.Context(),
structVal.Interface())
              paramVals = append([]reflect.Value { structVal.Elem() },
paramVals...)
        result := route.handlerMethod.Func.Call(paramVals)
        if len(result) > 0 {
                             if action, ok := result[0].Interface().
(actionresults.ActionResult); ok {
                        err = services.PopulateForContext(context.Context(),
action)
                if (err == nil) {
                    err = action.Execute(&actionresults.ActionContext{
                        context.Context(), context.ResponseWriter })
                }
            } else {
                io.WriteString(context.ResponseWriter,
                    fmt.Sprint(result[0].Interface()))
            }
        }
    }
    return err
```

```
}
```

Листинг 34-2 выполнение действий в файле request_dispatch.go в папке http/handling

Структура, реализующая интерфейс `ActionResult`, передается функции `services.PopulateForContext`, чтобы ее поля заполнялись службами, а затем вызывается метод `Execute` для получения результата.

Определение общих результатов действий

Чаще всего ответ создается с использованием шаблона, поэтому добавьте файл с именем `templateresult.go` в папку `platform/http/actionresults` с содержимым, показанным в листинге 34-3.

```
package actionresults

import (
    "platform/templates"
)

func NewTemplateAction(name string, data interface{}) ActionResult {
    return &TemplateActionResult{ templateName:  name, data: data }
}

type TemplateActionResult struct {
    templateName string
    data interface{}
    templates.TemplateExecutor
}

func (action *TemplateActionResult) Execute(ctx *ActionContext) error {
    return action.TemplateExecutor.ExecTemplate(ctx.ResponseWriter,
        action.templateName, action.data)
}
```

Листинг 34-3 Содержимое файла templateresult.go в папке http/actionresults

Структура `TemplateActionResult` — это действие, которое отображает шаблон при его выполнении. В его полях указывается имя шаблона, данные, которые будут переданы исполнителю шаблона, и служба исполнителя шаблона. `NewTemplateAction` создает новый экземпляр структуры `TemplateActionResult`.

Другим распространенным результатом является перенаправление, которое часто выполняется после обработки запроса POST или PUT. Чтобы создать результат такого типа, добавьте файл с именем `redirectresult.go` в папку `platform/http/actionresults` с содержимым, показанным в листинге 34-4.

```
package actionresults

import "net/http"

func NewRedirectAction(url string) ActionResult {
    return &RedirectActionResult{ url: url}
```

```
}

type RedirectActionResult struct {
    url string
}

func (action *RedirectActionResult) Execute(ctx *ActionContext) error {
    ctx.ResponseWriter.Header().Set("Location", action.url)
    ctx.ResponseWriter.WriteHeader(http.StatusSeeOther)
    return nil
}
```

Листинг 34-4 Содержимое файла redirectresult.go в папке http/actionresults

Это действие приводит к результату с ответом `303 See Other`. Это перенаправление, которое указывает новый URL-адрес и гарантирует, что браузер не будет повторно использовать метод HTTP или URL-адрес из исходного запроса.

Следующий результат действия, который необходимо определить в этом разделе, позволит методу-обработчику возвращать результат JSON, что будет полезно при создании веб-службы в главе 38. Создайте файл с именем `jsonresult.go` в папке `platform/http/actionresults` с содержимое показано в листинге 34-5.

```
package actionresults

import "encoding/json"

func NewJsonAction(data interface{}) ActionResult {
    return &JsonActionResult{ data: data}
}

type JsonActionResult struct {
    data interface{}
}

func (action *JsonActionResult) Execute(ctx *ActionContext) error {
    ctx.ResponseWriter.Header().Set("Content-Type", "application/json")
    encoder := json.NewEncoder(ctx.ResponseWriter)
    return encoder.Encode(action.data)
}
```

Листинг 34-5 Содержимое файла jsonresult.go в папке http/actionresults

Этот результат действия устанавливает заголовок `Content-Type`, чтобы указать, что ответ содержит JSON и использует кодировщик из пакета `enconding/json` для сериализации данных и отправки их клиенту.

Последнее встроенное действие позволит обработчику запроса указать, что произошла ошибка и что нормальный ответ не может быть создан. Добавьте файл с именем `errorresult.go` в папку `platform/http/actionresults` с содержимым, показанным в листинге 34-6.

```
package actionresults
```

```
func NewErrorAction(err error) ActionResult {
        return &ErrorActionResult{err}
}

type ErrorActionResult struct {
        error
}

func (action *ErrorActionResult) Execute(*ActionContext) error {
        return action.error
}
```

Листинг 34-6 Содержимое файла errorresult.go в папке http/actionresults

Этот результат действия не генерирует ответ, а просто передает ошибку из метода обработчика запроса на остальную часть платформы.

Обновление заполнителей для использования результатов действий

Чтобы убедиться, что результаты действия работают должным образом, в листинге 34-7 изменяются результаты методов обработчика-заполнителя.

```
package placeholder

import (
    "fmt"
    "platform/logging"
    "platform/http/actionresults"
)

var names = []string{"Alice", "Bob", "Charlie", "Dora"}

type NameHandler struct {
        logging.Logger
}

func (n NameHandler) GetName(i int) actionresults.ActionResult {
    n.Logger.Debugf("GetName method invoked with argument: %v", i)
    var response string
    if (i < len(names)) {
        response = fmt.Sprintf("Name #%v: %v", i, names[i])
    } else {
        response =  fmt.Sprintf("Index out of bounds")
    }
        return   actionresults.NewTemplateAction("simple_message.html",
response)
}

func (n NameHandler) GetNames() actionresults.ActionResult {
    n.Logger.Debug("GetNames method invoked")
    return actionresults.NewTemplateAction("simple_message.html", names)
}
```

```
type NewName struct {
    Name string
    InsertAtStart bool
}

func (n NameHandler) PostName(new NewName) actionresults.ActionResult {
    n.Logger.Debugf("PostName method invoked with argument %v", new)
    if (new.InsertAtStart) {
        names = append([] string { new.Name}, names... )
    } else {
        names = append(names, new.Name)
    }
    return actionresults.NewRedirectAction("/names")
}

func (n NameHandler) GetJsonData() actionresults.ActionResult {
    return actionresults.NewJsonAction(names)
}
```

Листинг 34-7 Использование результатов действий в файле name_handler.go в папке placeholder

Эти изменения означают, что методы `GetName` и `GetNames` возвращают результаты действия шаблона, метод `PostName` возвращает перенаправление, предназначенное для метода `GetNames`, а новый метод `GetJsonData` возвращает данные JSON. Последнее изменение заключается в добавлении выражения к шаблону-заполнителю, как показано в листинге 34-8.

```
{{ layout "layout.html" }}

<h3>
    {{ . }}
</h3>
```

Листинг 34-8 Обновление шаблона в файле simple_message.html в папке placeholder

Скомпилируйте и запустите проект и используйте браузер для запроса `http://localhost:5000/names`. Теперь ответ представляет собой HTML-документ, созданный путем выполнения шаблона, как показано на рисунке 34-1. Запросите `http://localhost:5000/jsondata`, и ответом будут данные JSON, как показано на рисунке 34-1.

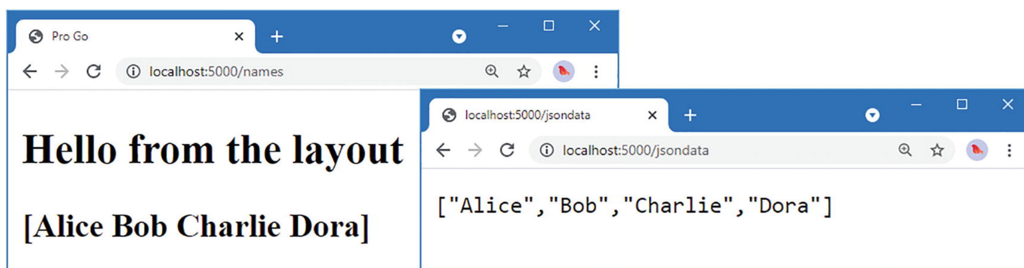

Рисунок 34-1 Использование результатов действий для получения ответов

Вызов обработчиков запросов из шаблонов

В последующих главах я собираюсь включить содержимое шаблона из одного обработчика в выходные данные другого обработчика, чтобы можно было отображать сведения о корзине, например, как часть шаблона, представляющего список продуктов. Это неудобная функция для реализации, но она избавит обработчиков от необходимости предоставлять своим шаблонам данные, которые напрямую не связаны с их назначением. В листинге 34-9 изменен интерфейс, используемый для службы шаблонов.

```
package templates

import "io"

type TemplateExecutor interface {

      ExecTemplate(writer io.Writer, name string, data interface{}) (err
error)

    ExecTemplateWithFunc(writer io.Writer, name string,
        data interface{}, handlerFunc InvokeHandlerFunc) (err error)
}

type InvokeHandlerFunc func(handlerName string, methodName string,
    args ...interface{}) interface{}
```

Листинг 34-9 Изменение интерфейса шаблона в файле template_executor.go в папке templates

Метод ExecTemplate был изменен таким образом, что он определяет метод ExecTemplateWithFunc, который принимает аргумент InvokeHandlerFunc, который будет использоваться для вызова метода обработчика в шаблоне. Для поддержки новой функции в листинге 34-10 определена новая функция-заполнитель, которая позволит анализировать шаблоны, если они содержат ключевое слово, запускающее обработчик.

```
package templates

import (
    "html/template"
    "sync"
    "errors"
    "platform/config"
)

var once = sync.Once{}

func LoadTemplates(c config.Configuration) (err error) {
    path, ok := c.GetString("templates:path")
    if !ok {
        return errors.New("Cannot load template config")
    }
```

```
    reload := c.GetBoolDefault("templates:reload", false)
    once.Do(func() {
        doLoad := func() (t *template.Template) {
            t = template.New("htmlTemplates")
            t.Funcs(map[string]interface{} {
                "body": func() string { return "" },
                "layout": func() string { return "" },
                "handler": func() interface{} { return "" },
            })
            t, err = t.ParseGlob(path)
            return
        }
        if (reload) {
            getTemplates = doLoad
        } else {
            var templates *template.Template
            templates = doLoad()
            getTemplates = func() *template.Template {
                t, _ := templates.Clone()
                return t
            }
        }
    })
    return
}
```

Листинг 34-10 Добавление функции-заполнителя в файл template_loader.go в папке templates

Как видно из листинга, я собираюсь использовать ключевое слово `handler` для вызова метода обработчика из шаблона. Листинг 34-11 обновляет исполнитель шаблона для поддержки ключевого слова `handler`.

```
package templates

import (
    "io"
    "strings"
    "html/template"
)

type LayoutTemplateProcessor struct {}

var emptyFunc = func(handlerName, methodName string,
    args ...interface{}) interface{} { return "" }

func (proc *LayoutTemplateProcessor) ExecTemplate(writer io.Writer,
        name string, data interface{}) (err error) {
    return proc.ExecTemplateWithFunc(writer, name, data, emptyFunc)
}

func    (proc    *LayoutTemplateProcessor)    ExecTemplateWithFunc(writer
io.Writer,
```

```
        name string, data interface{},
        handlerFunc InvokeHandlerFunc) (err error) {

    var sb strings.Builder
    layoutName := ""
    localTemplates := getTemplates()
    localTemplates.Funcs(map[string]interface{} {
        "body": insertBodyWrapper(&sb),
        "layout": setLayoutWrapper(&layoutName),
        "handler": handlerFunc,
    })
    err = localTemplates.ExecuteTemplate(&sb, name, data)
    if (layoutName != "") {
        localTemplates.ExecuteTemplate(writer, layoutName, data)
    } else {
        io.WriteString(writer, sb.String())
    }
    return
}

var getTemplates func() (t *template.Template)

func insertBodyWrapper(body *strings.Builder) func() template.HTML {
    return func() template.HTML {
        return template.HTML(body.String())
    }
}

func setLayoutWrapper(val *string) func(string) string {
    return func(layout string) string {
        *val = layout
        return ""
    }
}
```

Листинг 34-11 Обновление выполнения шаблона в файле layout_executor.go в папке templates

Листинг 34-12 обновляет результат действия шаблона, чтобы он вызывал метод `ExecTemplate` с новым аргументом.

```
package actionresults

import (
    "platform/templates"
)

func NewTemplateAction(name string, data interface{}) ActionResult {
    return &TemplateActionResult{ templateName:  name, data: data }
}

type TemplateActionResult struct {
    templateName string
```

```
    data interface{}
    templates.TemplateExecutor
    templates.InvokeHandlerFunc
}

func (action *TemplateActionResult) Execute(ctx *ActionContext) error {
                                                                          return
action.TemplateExecutor.ExecTemplateWithFunc(ctx.ResponseWriter,
        action.templateName, action.data, action.InvokeHandlerFunc)
}
```

Листинг 34-12 Добавление аргумента в файл templateresult.go в папке http/actionresults

Метод `Execute` использует функцию служб для получения значения `InvokeHandlerFunc`, которое затем передается исполнителю шаблона.

Обновление обработки запросов

Чтобы реализовать эту функцию, мне нужно создать службу для типа `InvokeHandlerFunc`. Добавьте файл с именем `handler_func.go` в папку `platform/http` с содержимым, показанным в листинге `34-13`.

```
package handling

import (
    "context"
    "fmt"
    "html/template"
    "net/http"
    "platform/http/actionresults"
    "platform/services"
    "platform/templates"
    "reflect"
    "strings"
)

func createInvokehandlerFunc(ctx context.Context,
        routes []Route) templates.InvokeHandlerFunc {
       return func(handlerName, methodName string, args ...interface{})
interface{} {
        var err error
        for _, route := range routes {
            if strings.EqualFold(handlerName, route.handlerName) &&
                                             strings.EqualFold(methodName,
route.handlerMethod.Name) {
                paramVals := make([]reflect.Value, len(args))
                for i := 0; i < len(args); i++ {
                    paramVals[i] = reflect.ValueOf(args[i])
                }
                structVal := reflect.New(route.handlerMethod.Type.In(0))
                services.PopulateForContext(ctx, structVal.Interface())
                paramVals = append([]reflect.Value { structVal.Elem() },
```

```
                    paramVals...)
                result := route.handlerMethod.Func.Call(paramVals)
                if action, ok := result[0].Interface().
                        (*actionresults.TemplateActionResult); ok {
                    invoker := createInvokehandlerFunc(ctx, routes)
                    err = services.PopulateForContextWithExtras(ctx,
                    action,
                    map[reflect.Type]reflect.Value {
                        reflect.TypeOf(invoker): reflect.ValueOf(invoker),
                    })
                                writer  :=  &stringResponseWriter{  Builder:
&strings.Builder{} }
                    if err == nil {
                        err = action.Execute(&actionresults.ActionContext{
                            Context: ctx,
                            ResponseWriter: writer,
                        })
                        if err == nil {
                                                    return  (template.HTML)
(writer.Builder.String())
                        }
                    }
                } else {
                    return fmt.Sprint(result[0])
                }
            }
        }
        if err == nil {
                err  =  fmt.Errorf("No  route  found  for  %v  %v",  handlerName,
methodName)
        }
        panic(err)
    }
}

type stringResponseWriter struct {
    *strings.Builder
}
func (sw *stringResponseWriter) Write(data []byte) (int, error) {
    return sw.Builder.Write(data)
}
func (sw *stringResponseWriter) WriteHeader(statusCode int) {}
func    (sw    *stringResponseWriter)    Header()    http.Header    {    return
http.Header{}}
```

Листинг 34-13 Содержимое файла handler_func.go в папке http/handling

`createInvokehandlerFunc` создает функцию, которая использует набор маршрутов для поиска и выполнения метода обработчика. Вывод обработчика — это строка, которую можно включить в шаблон.

Листинг 34-14 обновляет код, который выполняет результаты действия, чтобы предоставить функцию, которую можно использовать для вызова обработчика.

```
...
func  (router  *RouterComponent)  invokeHandler(route  Route,  rawParams
[]string,
      context *pipeline.ComponentContext) error {
    paramVals, err := params.GetParametersFromRequest(context.Request,
        route.handlerMethod, rawParams)
    if (err == nil) {
        structVal := reflect.New(route.handlerMethod.Type.In(0))
                            services.PopulateForContext(context.Context(),
structVal.Interface())
              paramVals  =  append([]reflect.Value  {  structVal.Elem()  },
paramVals...)
        result := route.handlerMethod.Func.Call(paramVals)
        if len(result) > 0 {
                             if  action,  ok  :=  result[0].Interface().
(actionresults.ActionResult); ok {
                    invoker := createInvokehandlerFunc(context.Context(),
router.routes)
                                                                 err    =
services.PopulateForContextWithExtras(context.Context(),
                    action,
                    map[reflect.Type]reflect.Value {
                        reflect.TypeOf(invoker): reflect.ValueOf(invoker),
                    })
                if (err == nil) {
                    err = action.Execute(&actionresults.ActionContext{
                        context.Context(), context.ResponseWriter  })
                }
            } else {
                io.WriteString(context.ResponseWriter,
                    fmt.Sprint(result[0].Interface()))
            }
        }
    }
    return err
}
...
```

Листинг 34-14 Обновление выполнения результатов в файле request_dispatch.go в папке http/handling

Я мог бы создать службу для функции, которая вызывает обработчики, но я хочу убедиться, что действие получает функцию, которая вызывает обработчики, используя URL-маршрутизатор, обрабатывающий запрос. Как вы увидите позже в этой главе, я собираюсь использовать несколько URL-маршрутов для обработки различных типов запросов, и я не хочу, чтобы обработчики, управляемые одним маршрутизатором, вызывали методы обработчиков, управляемых другим маршрутизатором.

Настройка приложения

Некоторые изменения необходимы, чтобы убедиться, что шаблон может вызывать метод обработчика. Сначала создайте новый обработчик запросов, добавив файл с

именем day_handler.go в папку placeholder с содержимым, показанным в листинге 34-15.

```
package placeholder

import (
    "platform/logging"
    "time"
    "fmt"
)

type DayHandler struct {
    logging.Logger
}

func (dh DayHandler) GetDay() string {
    return  fmt.Sprintf("Day: %v", time.Now().Day())
}
```

Листинг 34-15 Содержимое файла day_handler.go в папке placeholder

Затем зарегистрируйте новый обработчик запросов, как показано в листинге 34-16.

```
...
func createPipeline() pipeline.RequestPipeline {
    return pipeline.CreatePipeline(
        &basic.ServicesComponent{},
        &basic.LoggingComponent{},
        &basic.ErrorComponent{},
        &basic.StaticFileComponent{},
        //&SimpleMessageComponent{},
        handling.NewRouter(
            handling.HandlerEntry{ "",  NameHandler{}},
            handling.HandlerEntry{ "",  DayHandler{}},
        ),
    )
}
...
```

Листинг 34-16 Регистрация нового обработчика в файле startup.go в папке placeholder

Наконец, добавьте выражение, которое вызывает метод GetDay, определенный в листинге 34-15, как показано в листинге 34-17.

```
{{ layout "layout.html" }}

<h3>
    {{ . }}
</h3>

{{ handler "day" "getday"}}
```

Листинг 34-17 Добавление выражения в файл simple_message.html в папку placeholder

Скомпилируйте и запустите приложение и запросите http://localhost:5000/names; вы увидите, что результат, полученный при отображении шаблона simple_message.html, содержит результат метода GetDay, как показано на рисунке 34-2, хотя и с дополнительными выходными данными, отражающими день запуска примера.

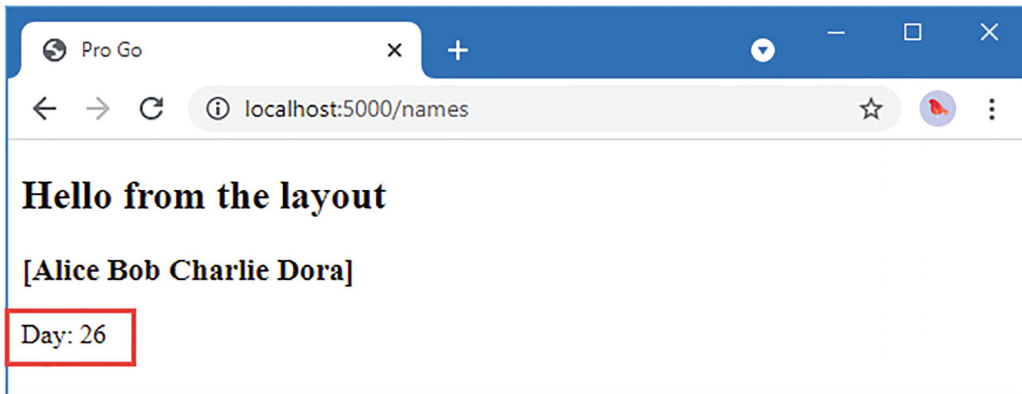

Рисунок 34-2 Вызов обработчика из шаблона

Создание URL-адресов из маршрутов

When I wanted to redirect the browser to a new URL in Listing 34-7, I had to specify the URL like this:

```
...
return actionresults.NewRedirectAction("/names")
...
```

Это не идеально, потому что это означает, что изменение конфигурации маршрутизации может сломать такой жестко закодированный URL-адрес. Более надежный подход — добавить поддержку для указания метода обработчика и создания URL-адреса на основе связанной с ним конфигурации маршрутизации. Добавьте файл с именем url_generation.go в папку http/handling с содержимым, показанным в листинге 34-18.

```
package handling

import (
    "fmt"
    "net/http"
    "strings"
    "errors"
    "reflect"
)

type URLGenerator interface {

    GenerateUrl(method interface{}, data ...interface{}) (string, error)
```

```
    GenerateURLByName(handlerName, methodName string,
        data ...interface{}) (string, error)

    AddRoutes(routes []Route)
}

type routeUrlGenerator struct {
    routes []Route
}

func (gen *routeUrlGenerator) AddRoutes(routes []Route) {
    if gen.routes == nil {
        gen.routes = routes
    } else {
        gen.routes = append(gen.routes, routes...)
    }
}

func (gen *routeUrlGenerator) GenerateUrl(method interface{},
        data ...interface{}) (string, error) {
    methodVal := reflect.ValueOf(method)
    if methodVal.Kind() == reflect.Func &&
            methodVal.Type().In(0).Kind() == reflect.Struct {
        for _, route := range gen.routes {
            if route.handlerMethod.Func.Pointer() == methodVal.Pointer() {
                return generateUrl(route, data...)
            }
        }
    }
    return "", errors.New("No matching route")
}

func  (gen  *routeUrlGenerator)  GenerateURLByName(handlerName,  methodName
string,
        data ...interface{}) (string, error) {
    for _, route := range gen.routes {
        if strings.EqualFold(route.handlerName, handlerName) &&
                    strings.EqualFold(route.httpMethod + route.actionName,
methodName) {
            return generateUrl(route, data...)
        }
    }
    return "", errors.New("No matching route")
}

func generateUrl(route Route, data ...interface{}) (url string, err error)
{
    url = "/" + route.prefix
    if (!strings.HasPrefix(url, "/")) {
        url = "/" + url
    }
    if (!strings.HasSuffix(url, "/")) {
```

```
        url += "/"
    }
    url+= strings.ToLower(route.actionName)
        if  len(data)  >  0  &&  !strings.EqualFold(route.httpMethod,
http.MethodGet) {
        err = errors.New("Only GET handler can have data values")
    } else if strings.EqualFold(route.httpMethod, http.MethodGet) &&
            len(data) != route.handlerMethod.Type.NumIn() -1 {
            err  =  errors.New("Number  of  data  values  doesn't  match  method
params")
    } else {
        for _, val := range data {
            url = fmt.Sprintf("%v/%v", url, val)
        }
    }
    return
}
```

Листинг 34-18 Содержимое файла url_generation.go в папке http/handling

Интерфейс `URLGenerator` определяет методы с именами `GenerateURL` и `GenerateURLByName`. Метод `GenerateURL` получает функцию-обработчик и использует ее для поиска маршрута, а метод `GenerateURLByName` находит функцию-обработчик, используя строковые значения. Структура `routeUrlGenerator` реализует методы `URLGenerator`, используя маршруты для создания URL-адресов.

Создание службы генератора URL

Я хочу создать службу для интерфейса `URLGenerator`, но я хочу, чтобы она была доступна только тогда, когда конвейер запросов настроен на использование функций маршрутизации, определенных в главе 33. В листинге 34-19 служба настраивается при создании экземпляра компонента промежуточного программного обеспечения маршрутизации.

```
...
func NewRouter(handlers ...HandlerEntry) *RouterComponent {
    routes := generateRoutes(handlers...)

    var urlGen URLGenerator
    services.GetService(&urlGen)
    if urlGen == nil {
        services.AddSingleton(func () URLGenerator {
            return &routeUrlGenerator { routes: routes }
        })
    } else {
        urlGen.AddRoutes(routes)
    }
    return &RouterComponent{ routes: routes }
}
...
```

Листинг 34-19 Создание службы в файле request_dispatch.go в папке http/handling

Новый сервис означает, что я могу генерировать URL программно, как показано в листинге 34-20.

```
package placeholder

import (
    "fmt"
    "platform/logging"
    "platform/http/actionresults"
    "platform/http/handling"
)

var names = []string{"Alice", "Bob", "Charlie", "Dora"}

type NameHandler struct {
              logging.Logger
    handling.URLGenerator
}

func (n NameHandler) GetName(i int) actionresults.ActionResult {
    n.Logger.Debugf("GetName method invoked with argument: %v", i)
    var response string
    if (i < len(names)) {
        response = fmt.Sprintf("Name #%v: %v", i, names[i])
    } else {
        response =  fmt.Sprintf("Index out of bounds")
    }
            return   actionresults.NewTemplateAction("simple_message.html",
response)
}

func (n NameHandler) GetNames() actionresults.ActionResult {
    n.Logger.Debug("GetNames method invoked")
    return actionresults.NewTemplateAction("simple_message.html", names)
}

type NewName struct {
    Name string
    InsertAtStart bool
}

func (n NameHandler) PostName(new NewName) actionresults.ActionResult {
    n.Logger.Debugf("PostName method invoked with argument %v", new)
    if (new.InsertAtStart) {
        names = append([] string { new.Name}, names... )
    } else {
        names = append(names, new.Name)
    }
    return n.redirectOrError(NameHandler.GetNames)
}

func (n NameHandler) GetRedirect() actionresults.ActionResult {
```

```
    return n.redirectOrError(NameHandler.GetNames)
}

func (n NameHandler) GetJsonData() actionresults.ActionResult {
    return actionresults.NewJsonAction(names)
}

func (n NameHandler) redirectOrError(handler interface{},
        data ...interface{}) actionresults.ActionResult {
    url, err := n.GenerateUrl(handler)
    if (err == nil) {
        return actionresults.NewRedirectAction(url)
    } else {
        return actionresults.NewErrorAction(err)
    }
}
```

Листинг 34-20 Создание URL-адреса в файле name_handler.go в папке placeholder

Новая служба позволяет динамически генерировать URL-адреса, отражающие определенные маршруты. Тестировать запрос POST неудобно, поэтому в листинге 34-20 добавлен новый метод обработчика с именем `GetRedirect`, который получает запрос GET и выполняет перенаправление на URL-адрес, созданный путем указания метода `GetNames`:

```
...
return n.redirectOrError(NameHandler.GetNames)
...
```

Обратите внимание, что круглые скобки не используются при выборе метода обработчика, поскольку именно метод, а не результат его вызова, требуется для создания URL-адреса.

Скомпилируйте и запустите проект и используйте браузер для запроса `http://localhost:5000/redirect`. Браузер будет автоматически перенаправлен на URL-адрес, предназначенный для метода `GetNames`, как показано на рисунке 34-3.

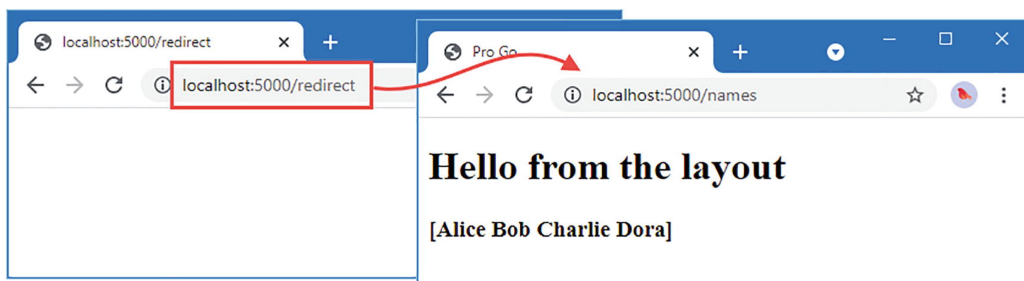

Рисунок 34-3 Создание URL-адреса перенаправления

Определение альтернативных маршрутов

Поддержка создания URL-адресов упрощает процесс определения маршрутов, соответствующих URL-адресу методу обработчика, в дополнение к тем маршрутам, которые создаются непосредственно обработчиком. Например, существует пробел в URL-адресах, поддерживаемых маршрутами-заполнителями, что означает, что запросы для URL-адреса по умолчанию, `http://localhost:5000/`, приводят к результату `404 — Not Found`. В этом разделе я собираюсь добавить поддержку для определения дополнительных маршрутов, которые не являются производными непосредственно от структур обработчиков и их методов, что позволит устранить пробелы, подобные этому.

Добавьте файл с именем `alias_route.go` в папку `platform/http/handling` с содержимым, показанным в листинге 34-21.

```
package handling

import (
    "platform/http/actionresults"
    "platform/services"
    "net/http"
    "reflect"
    "regexp"
    "fmt"
)

func (rc *RouterComponent) AddMethodAlias(srcUrl string,
        method interface{}, data ...interface{}) *RouterComponent {
    var urlgen URLGenerator
    services.GetService(&urlgen)
    url, err := urlgen.GenerateUrl(method, data...)
    if (err == nil) {
        return rc.AddUrlAlias(srcUrl, url)
    } else {
        panic(err)
    }
}

func (rc *RouterComponent) AddUrlAlias(srcUrl string,
        targetUrl string) *RouterComponent {
    aliasFunc := func(interface{}) actionresults.ActionResult {
        return actionresults.NewRedirectAction(targetUrl)
    }
    alias := Route {
        httpMethod: http.MethodGet,
        handlerName: "Alias",
        actionName: "Redirect",
        expression: *regexp.MustCompile(fmt.Sprintf("^%v[/]?$", srcUrl)),
        handlerMethod: reflect.Method{
            Type: reflect.TypeOf(aliasFunc),
            Func: reflect.ValueOf(aliasFunc),
        },
    }
```

```
    rc.routes = append([]Route { alias},  rc.routes... )
    return rc
}
```

Листинг 34-21 Содержимое файла alias_route.go в папке http/handling

Этот файл определяет дополнительные методы для структуры `RouterComponent`. Метод `AddUrlAlias` создает `Route`, но делает это путем создания `Reflect.Method`, который вызывает функцию, которая создает результат действия перенаправления. Легко забыть, что типы, определенные пакетом `Reflect`, являются обычными структурами и интерфейсами Go, а `Method` — это просто структура, и я могу установить поля `Type` и `Func` так, чтобы моя функция-псевдоним выглядела как обычный метод для код, выполняющий маршруты.

Метод `AddMethodAlias` позволяет создать маршрут с использованием URL-адреса и метода обработчика. Служба `URLGenerator` используется для создания URL-адреса для метода обработчика, который передается методу `AddUrlAlias`.

В листинге 34-22 к набору маршрутов-заполнителей добавлен псевдоним, чтобы запросы на URL-адрес по умолчанию перенаправлялись и обрабатывались методом обработчика `GetNames`.

```
package placeholder

import (
    "platform/http"
    "platform/pipeline"
    "platform/pipeline/basic"
    "platform/services"
    "sync"
    "platform/http/handling"
)

func createPipeline() pipeline.RequestPipeline {
    return pipeline.CreatePipeline(
        &basic.ServicesComponent{},
        &basic.LoggingComponent{},
        &basic.ErrorComponent{},
        &basic.StaticFileComponent{},
        //&SimpleMessageComponent{},
        handling.NewRouter(
            handling.HandlerEntry{ "",  NameHandler{}},
            handling.HandlerEntry{ "",  DayHandler{}},
        ).AddMethodAlias("/", NameHandler.GetNames),
    )
}

func Start() {
    results, err := services.Call(http.Serve, createPipeline())
    if (err == nil) {
        (results[0].(*sync.WaitGroup)).Wait()
    } else {
        panic(err)
```

```
    }
}
```

Листинг 34-22 Определение альтернативного маршрута в файле startup.go в папке placeholder

Скомпилируйте и запустите проект и используйте браузер для запроса `http://localhost:5000`. Вместо ответа 404 браузер перенаправляется, как показано на рисунке 34-4.

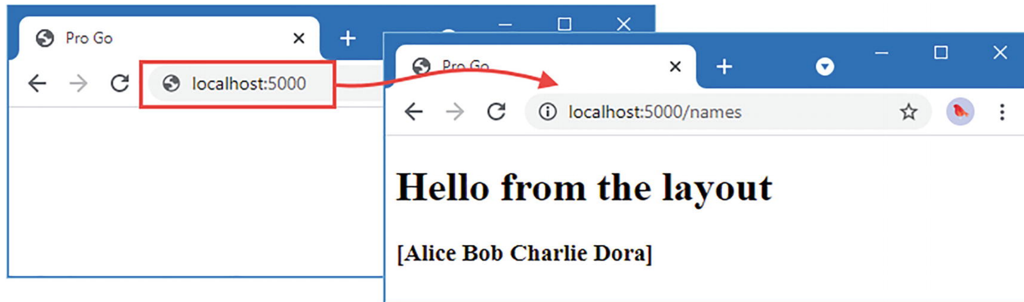

Рисунок 34-4 Эффект псевдонима маршрута

Проверка данных запроса

Как только приложение начинает принимать данные от пользователей, возникает необходимость в валидации. Пользователи будут вводить что угодно в поле формы, иногда потому, что инструкции неясны, а также потому, что они работают над процессом, чтобы как можно быстрее добраться до конца. Определяя проверку как услугу, я могу свести к минимуму объем кода, который приходится реализовывать отдельным обработчикам.

Поскольку служба не может знать, какие требования проверки требуются обработчикам, мне нужно каким-то образом описать их как часть типов данных, которые обрабатывают обработчики. Самый простой подход — использовать теги структуры, с помощью которых можно выразить некоторые основные требования проверки.

Создайте папку `platform/validation` и добавьте в нее файл с именем `validator.go` с содержимым, показанным в листинге 34-23.

```
package validation

type Validator interface {
    Validate(data interface{}) (ok bool, errs []ValidationError)
}

type ValidationError struct {
    FieldName string
    Error error
}

type ValidatorFunc func(fieldName string, value interface{},
    arg string) (bool, error)
```

```
func DefaultValidators() map[string]ValidatorFunc {
    return map[string]ValidatorFunc {
        "required": required,
        "min": min,
    }
}
```

Листинг 34-23 Содержимое файла validator.go в папке validation

Интерфейс `Validator` будет использоваться для обеспечения проверки как услуги, при этом отдельные проверки проверки будут выполняться функциями `ValidatorFunc`. Я собираюсь определить два валидатора, `required` и `min`, которые будут гарантировать, что значение предоставлено для строкового значения, и обеспечить минимальное значение для значений `int` и `float64` и минимальную длину для строковых значений. Дополнительные валидаторы могут быть определены по мере необходимости, но этих двух будет достаточно для этого проекта. Чтобы определить функции валидатора, добавьте значение файла с именем `validator_functions.go` в папку `platform/validation` с содержимым, показанным в листинге 34-24.

```
package validation

import (
    "errors"
    "fmt"
    "strconv"
)

func required(fieldName string, value interface{},
        arg string) (valid bool, err error) {
    if str, ok := value.(string); ok {
        valid = str != ""
        err = fmt.Errorf("A value is required")
    } else {
        err = errors.New("The required validator is for strings")
    }
    return
}

func min(fieldName string, value interface{}, arg string) (valid bool, err
error) {
    minVal, err := strconv.Atoi(arg)
    if err != nil {
        panic("Invalid arguments for validator: " + arg)
    }
    err = fmt.Errorf("The minimum value is %v", minVal)
    if iVal, iValOk := value.(int); iValOk {
        valid = iVal >= minVal
    } else if fVal, fValOk := value.(float64); fValOk {
        valid = fVal >= float64(minVal)
    } else if strVal, strValOk := value.(string); strValOk {
```

```
        err = fmt.Errorf("The minimum length is %v characters", minVal)
        valid = len(strVal) >= minVal
    } else {
         err = errors.New("The min validator is for int, float64, and str
values")
    }
    return
}
```

Листинг 34-24 Содержимое файла validator_functions.go в папке validation

Для выполнения проверки каждая функция получает имя проверяемого поля структуры, значение, полученное из запроса, и необязательные аргументы, которые настраивают процесс проверки. Чтобы создать реализацию и функции, которые будут настраивать службу, добавьте файл с именем `tag_validator.go` в папку `platform/validation` с содержимым, показанным в листинге 34-25.

```
package validation

import (
    "reflect"
    "strings"
)

func NewDefaultValidator(validators map[string]ValidatorFunc) Validator {
    return &TagValidator{ DefaultValidators() }
}

type TagValidator struct {
    validators map[string]ValidatorFunc
}

func (tv *TagValidator) Validate(data interface{}) (ok bool,
         errs []ValidationError) {
    errs = []ValidationError{}
    dataVal := reflect.ValueOf(data)
    if (dataVal.Kind() == reflect.Ptr) {
        dataVal = dataVal.Elem()
    }
    if (dataVal.Kind() != reflect.Struct) {
        panic("Only structs can be validated")
    }
    for i := 0; i < dataVal.NumField(); i++ {
        fieldType := dataVal.Type().Field(i)
        validationTag, found := fieldType.Tag.Lookup("validation")
        if found {
            for _, v := range strings.Split(validationTag, ",") {
                var name, arg string = "", ""
                if strings.Contains(v, ":") {
                    nameAndArgs := strings.SplitN(v, ":", 2)
                    name = nameAndArgs[0]
                    arg = nameAndArgs[1]
```

```
            } else {
                name = v
            }
            if validator, ok := tv.validators[name]; ok {
                valid, err := validator(fieldType.Name,
                    dataVal.Field(i).Interface(), arg )
                if (!valid) {
                    errs = append(errs, ValidationError{
                        FieldName: fieldType.Name,
                        Error: err,
                    })
                }
            } else {
                panic("Unknown validator: " + name)
            }
        }
    }
}
ok = len(errs) == 0
return
}
```

Листинг 34-25 Содержимое файла tag_validator.go в папке validation

Структура `TagValidator` реализует интерфейс `Validator`, ища тег структуры с именем `validation` и анализируя его, чтобы увидеть, какая проверка требуется для каждого поля структуры. Используется каждый указанный валидатор, а ошибки собираются и возвращаются как результат метода `Validate`. Функция `NewDefaultValidation` создает экземпляр структуры и используется для создания службы проверки, как показано в листинге 34-26.

```
package services

import (
    "platform/logging"
    "platform/config"
    "platform/templates"
    "platform/validation"
)

func RegisterDefaultServices() {

   // ...statements omitted for brevity...

    err = AddSingleton(
        func() validation.Validator {
                                                                        return
validation.NewDefaultValidator(validation.DefaultValidators())
        })
    if (err != nil) {
        panic(err)
    }
```

```
}
```

Листинг 34-26 Регистрация службы проверки в файле services_default.go в папке services

Я зарегистрировал новую службу как синглтон, используя валидаторы, возвращаемые функцией `DefaultValidators`.

Выполнение проверки данных

Требуется некоторая подготовка, чтобы убедиться, что проверка данных работает. Во-первых, в листинге 34-27 создается новый метод обработчика и применяется тег структуры проверки к обработчику запроса-заполнителя.

```
package placeholder

import (
    "fmt"
    "platform/logging"
    "platform/http/actionresults"
    "platform/http/handling"
    "platform/validation"
)

var names = []string{"Alice", "Bob", "Charlie", "Dora"}

type NameHandler struct {
            logging.Logger
    handling.URLGenerator
    validation.Validator
}

func (n NameHandler) GetName(i int) actionresults.ActionResult {
    n.Logger.Debugf("GetName method invoked with argument: %v", i)
    var response string
    if (i < len(names)) {
        response = fmt.Sprintf("Name #%v: %v", i, names[i])
    } else {
        response =  fmt.Sprintf("Index out of bounds")
    }
          return   actionresults.NewTemplateAction("simple_message.html",
response)
}

func (n NameHandler) GetNames() actionresults.ActionResult {
    n.Logger.Debug("GetNames method invoked")
    return actionresults.NewTemplateAction("simple_message.html", names)
}

type NewName struct {
    Name string `validation:"required,min:3"`
    InsertAtStart bool
}
```

```
func (n NameHandler) GetForm() actionresults.ActionResult {
    postUrl, _ := n.URLGenerator.GenerateUrl(NameHandler.PostName)
    return actionresults.NewTemplateAction("name_form.html", postUrl)
}

func (n NameHandler) PostName(new NewName) actionresults.ActionResult {
    n.Logger.Debugf("PostName method invoked with argument %v", new)
    if ok, errs := n.Validator.Validate(&new); !ok {
          return actionresults.NewTemplateAction("validation_errors.html",
errs)
    }
    if (new.InsertAtStart) {
        names = append([] string { new.Name}, names... )
    } else {
        names = append(names, new.Name)
    }
    return n.redirectOrError(NameHandler.GetNames)
}

func (n NameHandler) GetRedirect() actionresults.ActionResult {
    return n.redirectOrError(NameHandler.GetNames)
}

func (n NameHandler) GetJsonData() actionresults.ActionResult {
    return actionresults.NewJsonAction(names)
}

func (n NameHandler) redirectOrError(handler interface{},
        data ...interface{}) actionresults.ActionResult {
    url, err := n.GenerateUrl(handler)
    if (err == nil) {
        return actionresults.NewRedirectAction(url)
    } else {
        return actionresults.NewErrorAction(err)
    }
}
```

Листинг 34-27 Подготовка к проверке в файле name_handler.go в папке placeholder

Тег проверки был добавлен в поле `Name`, применяя `required` и `min` валидаторы, что означает, что требуется значение с минимальным количеством трех символов. Чтобы упростить проверку проверки, я добавил метод-обработчик с именем `GetForm`, который отображает шаблон с именем `name_form.html`. Когда данные получены методом `PostName`, они проверяются с помощью службы, а шаблон `validation_errors.html` используется для формирования ответа при наличии ошибок проверки.

Добавьте файл с именем `name_form.html` в папку-заполнитель с содержимым, показанным в листинге 34-28.

```
{{ layout "layout.html" }}
```

```
<form method="POST" action="{{ . }}">
    <div style="padding: 5px;">
        <label>Name:</label>
        <input name="name" />
    </div>
    <div style="padding: 5px;">
        <label>Insert At Front:</label>
        <input name="insertatstart" type="checkbox" value="true" />
    </div>
    <div style="padding: 5px;">
        <button type="submit">Submit</button>
    </div>
</form>
```

Листинг 34-28 Содержимое файла name_form.html в папке placeholder

Этот шаблон создает простую HTML-форму, которая отправляет данные на URL-адрес, полученный от метода-обработчика. Добавьте файл с именем `validation_errors.html` в папку `placeholder` с содержимым, показанным в листинге 34-29.

```
{{ layout "layout.html" }}

<h3>Validation Errors</h3>

<ul>
    {{ range . }}
        <li>{{.FieldName}}: {{ .Error }}</li>
    {{ end }}
</ul>
```

Листинг 34-29 Содержимое файла validation_errors.html в папке placeholder

Срез ошибок проверки, полученных от метода обработчика, отображается в списке. Скомпилируйте и запустите проект и используйте браузер для запроса `http://localhost:5000/form`. Нажмите кнопку **Submit**, не вводя значение в поле **Name**, и вы увидите ошибки как от `required`, так и от `min` валидаторов, как показано на рисунке 34-5.

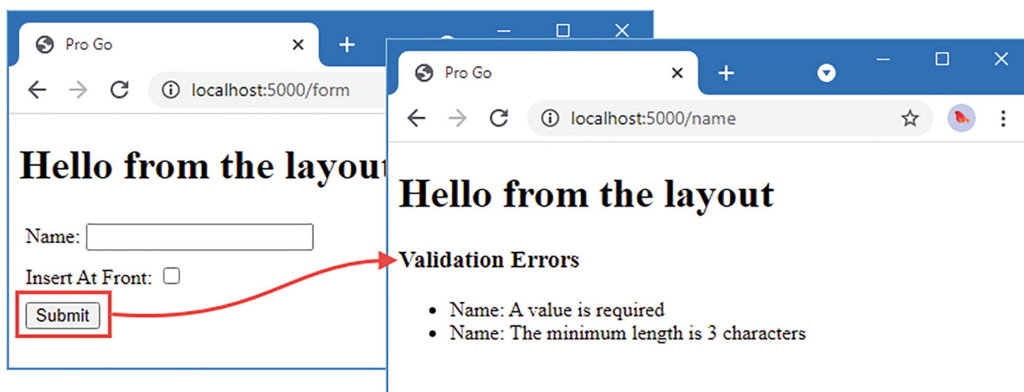

Рисунок 34-5 Отображение ошибок проверки

Если вы введете имя, содержащее менее трех символов, вы увидите предупреждение только от валидатора `min`. Если вы введете имя, состоящее из трех и более символов, оно будет добавлено в список имен, как показано на рисунке 34-6.

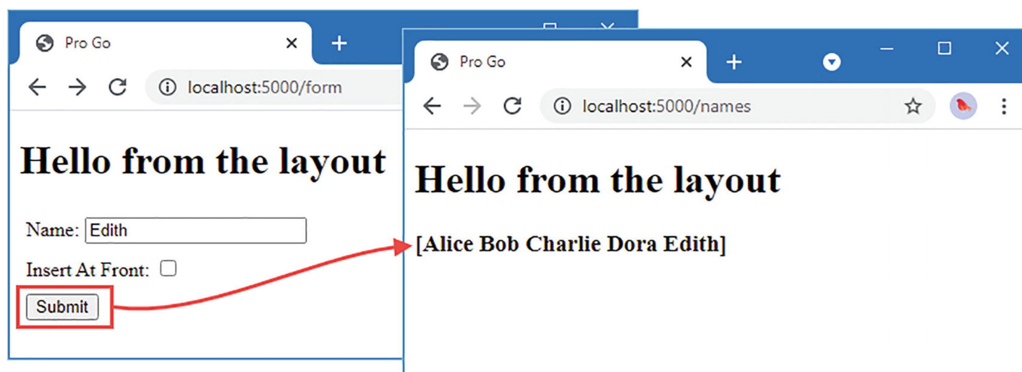

Рисунок 34-6 Прохождение проверки данных

Добавление сеансов

Сеансы используют файлы cookie для идентификации связанных HTTP-запросов, что позволяет отразить результаты одного действия пользователя в последующих действиях. Как бы я ни рекомендовал писать собственную платформу для изучения Go и стандартной библиотеки, это не распространяется на функции, связанные с безопасностью, где важен хорошо спроектированный и тщательно протестированный код. Файлы cookie и сеансы могут показаться не связанными с безопасностью, но они составляют основу, с помощью которой многие приложения идентифицируют пользователей после проверки их учетных данных. Небрежно написанная функция сеанса может позволить пользователям получить доступ для обхода контроля доступа или доступа к данным других пользователей.

В главе 32 я рекомендовал веб-инструментарий Gorilla как хорошее место для начала в качестве альтернативы написанию собственного фреймворка. Один из пакетов, предоставляемых набором инструментов Gorilla, называется `sessions` и обеспечивает поддержку безопасного создания сеансов и управления ими. Именно этот пакет я собираюсь использовать для добавления поддержки сеансов в этой главе. Запустите команду, показанную в листинге 34-30, в папке `platform`, чтобы загрузить и установить пакет `sessions`.

```
go get github.com/gorilla/sessions
```

Листинг 34-30 Установка пакета

Отсрочка записи данных ответа

Использование файлов cookie для сеансов представляет собой проблему конвейерного подхода, который я использовал для обработки запросов. Сеансы получаются перед выполнением метода обработчика, изменяются во время выполнения, а затем файл cookie сеанса обновляется после завершения метода обработчика. Это проблема, потому что обработчик запишет данные в `ResponseWriter`, после чего невозможно

обновить куки в заголовке. Добавьте файл кода с именем deferredwriter.go в папку конвейера с содержимым, показанным в листинге 34-31. (Это средство записи похоже на то, которое я создал для вызова обработчиков в шаблонах. Я предпочитаю определять отдельные типы при перехвате данных запроса и ответа, потому что способ использования перехваченных данных может меняться со временем.)

```
package pipeline

import (
    "net/http"
    "strings"
)

type DeferredResponseWriter struct {
    http.ResponseWriter
    strings.Builder
    statusCode int
}

func (dw *DeferredResponseWriter) Write(data []byte) (int, error) {
    return dw.Builder.Write(data)
}

func (dw *DeferredResponseWriter) FlushData()  {
    if (dw.statusCode == 0) {
        dw.statusCode = http.StatusOK
    }
    dw.ResponseWriter.WriteHeader(dw.statusCode)
    dw.ResponseWriter.Write([]byte(dw.Builder.String()))
}

func (dw *DeferredResponseWriter) WriteHeader(statusCode int) {
    dw.statusCode = statusCode
}
```

Листинг 34-31 Содержимое файла deferredwriter.go в папке pipeline

DeferredResponseWriter — это оболочка вокруг ResponseWriter, которая не записывает ответ до тех пор, пока не будет вызван метод FlushData, до которого данные хранятся в памяти. В листинге 34-32 DeferredResponseWriter используется при создании контекста, передаваемого компонентам промежуточного слоя.

```
...
func (pl RequestPipeline) ProcessRequest(req *http.Request,
        resp http.ResponseWriter) error {
    deferredWriter := &DeferredResponseWriter{ ResponseWriter:  resp }
    ctx := ComponentContext {
        Request: req,
        ResponseWriter: deferredWriter,
    }
    pl(&ctx)
```

```
    if (ctx.error == nil) {
        deferredWriter.FlushData()
    }
    return ctx.error
}
...
```

Листинг 34-32 Использование модифицированного модуля записи в файле pipe.go в папке pipeline

Это изменение позволяет устанавливать заголовки ответов, когда запрос возвращается по конвейеру.

Создание интерфейса сеанса, службы и промежуточного программного обеспечения

Я собираюсь предоставить доступ к сеансам как к сервису и использовать интерфейс, чтобы другие части платформы не зависели напрямую от пакета инструментов Gorilla, что позволяет легко использовать другой пакет сеансов, если это необходимо.

Создайте папку `platform/sessions` и добавьте файл с именем `session.go` с содержимым, показанным в листинге 34-33.

```
package sessions

import (
    "context"
    "platform/services"
    gorilla "github.com/gorilla/sessions"
)

const SESSION__CONTEXT_KEY string = "pro_go_session"

func RegisterSessionService() {
    err := services.AddScoped(func(c context.Context) Session {
        val := c.Value(SESSION__CONTEXT_KEY)
        if s, ok := val.(*gorilla.Session); ok {
            return &SessionAdaptor{ gSession: s}
        } else {
            panic("Cannot get session from context ")
        }
    })
    if (err != nil) {
        panic(err)
    }
}

type Session interface {
    GetValue(key string) interface{}
    GetValueDefault(key string, defVal interface{}) interface{}
    SetValue(key string, val interface{})
}

type SessionAdaptor struct {
```

```
    gSession *gorilla.Session
}

func (adaptor *SessionAdaptor) GetValue(key string) interface{} {
    return adaptor.gSession.Values[key]
}

func (adaptor *SessionAdaptor) GetValueDefault(key string,
        defVal interface{}) interface{} {
    if val, ok := adaptor.gSession.Values[key]; ok {
        return val
    }
    return defVal
}

func (adaptor *SessionAdaptor) SetValue(key string, val interface{}) {
    if val == nil {
        adaptor.gSession.Values[key] = nil
    } else {
        switch typedVal := val.(type) {
            case int, float64, bool, string:
                adaptor.gSession.Values[key] = typedVal
            default:
                    panic("Sessions only support int, float64, bool, and
string values")
        }
    }
}
```

Листинг 34-33 Содержимое файла session.go в папке sessions

Чтобы избежать конфликта имен, я импортировал пакет инструментов Gorilla, используя имя `gorilla`. Интерфейс `Session` определяет методы для получения и установки значений сеанса, и этот интерфейс реализован и сопоставлен с функциями Gorilla структурой `SessionAdaptor`. Функция `RegisterSessionService` регистрирует одноэлементную службу, которая получает сеанс из пакета Gorilla из текущего `Context` и заключает его в `SessionAdaptor`.

Любые данные, связанные с сеансом, будут сохранены в файле `cookie`. Чтобы избежать проблем со структурами и срезами, метод `SetValue` будет принимать только значения `int`, `float64`, `bool` и `string`, а также поддержку `nil` для удаления значения из сеанса.

Компонент промежуточного программного обеспечения будет отвечать за создание сеанса при передаче запроса по конвейеру и за сохранение сеанса при обратном пути. Добавьте файл с именем `session_middleware.go` в папку `platform/sessions` с содержимым, показанным в листинге `34-34`.

Примечание
Я использую самый простой вариант хранения сеансов, что означает, что данные сеанса сохраняются в cookie-файле ответа, отправляемом в браузеры. Это

ограничивает диапазон типов данных, которые можно безопасно хранить в сеансе, и подходит только для сеансов, в которых хранятся небольшие объемы данных. Доступны дополнительные хранилища сеансов, которые хранят данные в базе данных, что может решить эти проблемы. См. `https://github.com/gorilla/sessions` для получения списка доступных пакетов хранилища.

```
package sessions

import (
    "context"
    "time"
    "platform/config"
    "platform/pipeline"
    gorilla "github.com/gorilla/sessions"
)

type SessionComponent struct {
    store *gorilla.CookieStore
    config.Configuration
}

func (sc *SessionComponent) Init() {
    cookiekey, found := sc.Configuration.GetString("sessions:key")
    if !found {
        panic("Session key not found in configuration")
    }
    if sc.GetBoolDefault("sessions:cyclekey", true) {
        cookiekey += time.Now().String()
    }
    sc.store = gorilla.NewCookieStore([]byte(cookiekey))
}

func (sc *SessionComponent) ProcessRequest(ctx *pipeline.ComponentContext,
        next func(*pipeline.ComponentContext)) {
    session, _ := sc.store.Get(ctx.Request, SESSION__CONTEXT_KEY)
      c  := context.WithValue(ctx.Request.Context(), SESSION__CONTEXT_KEY,
session)
    ctx.Request = ctx.Request.WithContext(c)
    next(ctx)
    session.Save(ctx.Request, ctx.ResponseWriter)
}
```

Листинг 34-34 Содержимое файла session_middleware.go в папке sessions

Метод `Init` создает хранилище файлов cookie, что является одним из способов, которыми пакет Gorilla поддерживает сохранение сеансов. Метод `ProcessRequest` получает сессию из хранилища перед передачей запроса по конвейеру со `next` функцией параметра. Сеанс сохраняется в хранилище, когда запрос возвращается по конвейеру.

Если параметр конфигурации session:cyclekey имеет значение true, то имя, используемое для файлов cookie сеанса, будет включать время инициализации компонента промежуточного программного обеспечения. Это полезно во время разработки, поскольку это означает, что сеансы сбрасываются при каждом запуске приложения.

Создание обработчика, использующего сеансы

Чтобы обеспечить простую проверку работы функции сеанса, добавьте файл с именем counter_handler.go в папку placeholder с содержимым, показанным в листинге 34-35.

```
package placeholder

import (
    "fmt"
    "platform/sessions"
)

type CounterHandler struct {
    sessions.Session
}

func (c CounterHandler) GetCounter() string {
    counter := c.Session.GetValueDefault("counter", 0).(int)
    c.Session.SetValue("counter", counter + 1)
    return fmt.Sprintf("Counter: %v", counter)
}
```

Листинг 34-35 Содержимое файла counter_handler.go в папке placeholder

Обработчик объявляет свою зависимость от Session, определяя поле структуры, которое будет заполнено при создании экземпляра структуры для обработки запроса. Метод GetCounter получает значение с именем counter из сеанса, увеличивает его и обновляет сеанс перед использованием значения в качестве ответа.

Настройка приложения

Чтобы настроить службу сеанса и конвейер запросов, внесите изменения, показанные в листинге 34-36, в файл startup.go в папке placeholder.

```
package placeholder

import (
    "platform/http"
    "platform/pipeline"
    "platform/pipeline/basic"
    "platform/services"
    "sync"
    "platform/http/handling"
    "platform/sessions"
)
```

```
func createPipeline() pipeline.RequestPipeline {
    return pipeline.CreatePipeline(
        &basic.ServicesComponent{},
        &basic.LoggingComponent{},
        &basic.ErrorComponent{},
        &basic.StaticFileComponent{},
        &sessions.SessionComponent{},
        //&SimpleMessageComponent{},
        handling.NewRouter(
            handling.HandlerEntry{ "",  NameHandler{}},
            handling.HandlerEntry{ "",  DayHandler{}},
            handling.HandlerEntry{ "",  CounterHandler{}},
        ).AddMethodAlias("/", NameHandler.GetNames),
    )
}

func Start() {
    sessions.RegisterSessionService()
    results, err := services.Call(http.Serve, createPipeline())
    if (err == nil) {
        (results[0].(*sync.WaitGroup)).Wait()
    } else {
        panic(err)
    }
}
```

Листинг 34-36 Настройка сеансов в файле startup.go в папке placeholder

Наконец, добавьте параметр конфигурации, показанный в листинге 34-37, в файл `config.json`. Пакет сеанса Gorilla использует ключ для защиты данных сеанса. В идеале это должно храниться за пределами папки проекта, чтобы случайно не попасть в общедоступный репозиторий исходного кода, но для простоты я включил его в файл конфигурации.

```
{
    "logging" : {
        "level": "debug"
    },
    "main" : {
        "message" : "Hello from the config file"
    },
    "files": {
        "path": "placeholder/files"
    },
    "templates": {
        "path": "placeholder/*.html",
        "reload": true
    },
    "sessions": {
        "key": "MY_SESSION_KEY",
        "cyclekey": true
    }
```

```
}
```

Листинг 34-37 Определение ключа сеанса в файле config.json в папке platform

Скомпилируйте и запустите проект и используйте браузер для запроса `http://localhost:5000/counter`. Каждый раз, когда вы перезагружаете браузер, значение, хранящееся в сеансе, будет увеличиваться, как показано на рисунке 34-7.

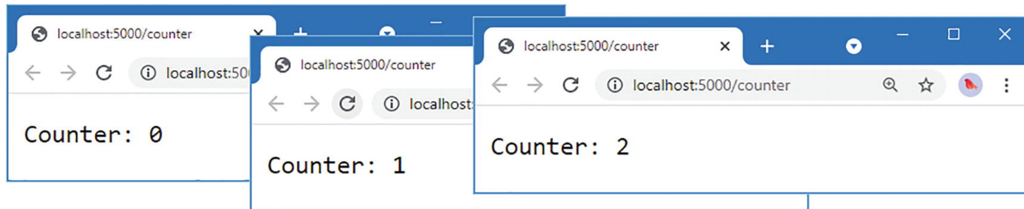

Рисунок 34-7 Использование сессий

Добавление авторизации пользователя

Последняя функция, необходимая для платформы, — поддержка авторизации с возможностью ограничения доступа к URL-адресам для определенных пользователей. В этом разделе я определяю интерфейсы, описывающие пользователей, и добавляю поддержку использования этих интерфейсов для управления доступом.

Важно не путать авторизацию с аутентификацией и управлением пользователями. Авторизация — это процесс принудительного управления доступом, который является темой этого раздела.

Аутентификация — это процесс получения и проверки учетных данных пользователя, чтобы их можно было идентифицировать для авторизации. Управление пользователями — это процесс управления данными пользователя, включая пароли и другие учетные данные.

В этой книге я создаю только заполнитель для аутентификации и вообще не занимаюсь управлением пользователями. В реальных проектах аутентификацию и управление пользователями должен обеспечивать проверенный сервис, которых доступно множество. Эти сервисы предоставляют API-интерфейсы HTTP, которые легко использовать с помощью стандартной библиотеки Go, функции которой для выполнения HTTP-запросов были описаны в главе 25.

Определение основных типов авторизации

Создайте папку `platform/authorization/identity` и добавьте файл с именем `user.go` с содержимым, показанным в листинге 34-38.

```
package identity

type User interface {

    GetID() int

    GetDisplayName() string
```

```
    InRole(name string) bool

    IsAuthenticated() bool
}
```

Листинг 34-38 Содержимое файла user.go в папке authorization/identity

`User` интерфейс будет представлять аутентифицированного пользователя, чтобы можно было оценить запросы к ограниченным ресурсам. Чтобы создать реализацию `User` интерфейса по умолчанию, которая будет полезна для приложений с простыми требованиями к авторизации, добавьте файл с именем `basic_user.go` в папку `authorization/identity` с содержимым, показанным в листинге 34-39.

```
package identity

import "strings"

var UnauthenticatedUser User = &basicUser{}

func NewBasicUser(id int, name string, roles ...string) User {
    return &basicUser {
        Id: id,
        Name: name,
        Roles: roles,
        Authenticated: true,
    }
}

type basicUser struct {
    Id int
    Name string
    Roles []string
    Authenticated bool
}

func (user *basicUser) GetID() int {
    return user.Id
}

func (user *basicUser) GetDisplayName() string {
    return user.Name
}

func (user *basicUser) InRole(role string) bool {
    for _, r := range user.Roles {
        if strings.EqualFold(r, role) {
            return true
        }
    }
    return false
}
```

```
func (user *basicUser) IsAuthenticated() bool {
    return user.Authenticated
}
```

Листинг 34-39 Содержимое файла basic_user.go в папке authorization/identity

Функция `NewBasicUser` создает простую реализацию `User` интерфейса, а переменная `UnauthenticatedUser` будет использоваться для представления пользователя, не вошедшего в приложение.

Добавьте файл с именем `signin_mgr.go` в папку `platform/authorization/identity` с содержимым, показанным в листинге 34-40.

```
package identity

type SignInManager interface {

    SignIn(user User) error
    SignOut(user User) error
}
```

Листинг 34-40 Содержимое файла signin_mgr.go в папке authorization/identity

Интерфейс `SignInManager` будет использоваться для определения службы, которую приложение будет использовать для входа пользователя в приложение и выхода из него. Подробная информация о том, как пользователь аутентифицируется, остается на усмотрение приложения.

Добавьте файл с именем `user_store.go` в папку `platform/authorization/identity` с содержимым, показанным в листинге 34-41.

```
package identity

type UserStore interface {

    GetUserByID(id int) (user User, found bool)

    GetUserByName(name string) (user User, found bool)
}
```

Листинг 34-41 Содержимое файла user_store.go в папке authorization/identity

Хранилище пользователей обеспечивает доступ к пользователям, известным приложению, которых можно найти по идентификатору или имени.

Далее мне нужен интерфейс, который будет использоваться для описания требования контроля доступа. Добавьте файл с именем `auth_condition.go` в папку `platform/authorization/identity` с содержимым, показанным в листинге 34-42.

```
package identity

type AuthorizationCondition interface {

    Validate(user User) bool
```

```
}
```

Листинг 34-42 Содержимое файла auth_condition.go в папке authorization/identity

Интерфейс `AuthorizationCondition` будет использоваться для оценки того, имеет ли вошедший пользователь доступ к защищенному URL-адресу, и будет использоваться как часть процесса обработки запроса.

Реализация интерфейсов платформы

Следующим шагом будет реализация интерфейсов, которые платформа будет предоставлять для авторизации. Добавьте файл с именем `sessionsignin.go` в папку `platform/authorization` с содержимым, показанным в листинге 34-43.

```
package authorization

import (
    "platform/authorization/identity"
    "platform/services"
    "platform/sessions"
    "context"
)

const USER_SESSION_KEY string = "USER"

func RegisterDefaultSignInService() {
                err    :=    services.AddScoped(func(c    context.Context)
identity.SignInManager {
        return &SessionSignInMgr{ Context : c}
    })
    if (err != nil) {
        panic(err)
    }
}

type SessionSignInMgr struct {
    context.Context
}

func (mgr *SessionSignInMgr) SignIn(user identity.User) (err error) {
    session, err := mgr.getSession()
    if err == nil {
        session.SetValue(USER_SESSION_KEY, user.GetID())
    }
    return
}

func (mgr *SessionSignInMgr) SignOut(user identity.User) (err error) {
    session, err := mgr.getSession()
    if err == nil {
        session.SetValue(USER_SESSION_KEY, nil)
    }
```

```
    return
}

func (mgr *SessionSignInMgr) getSession() (s sessions.Session, err error)
{
    err = services.GetServiceForContext(mgr.Context, &s)
    return
}
```

Листинг 34-43 Содержимое файла sessionsignin.go в папке authorization

Структура `SessionSignInMgr` реализует интерфейс `SignInManager`, сохраняя идентификатор вошедшего пользователя в сеансе и удаляя его, когда пользователь выходит из системы. Использование сеансов гарантирует, что пользователь останется в системе до тех пор, пока он не выйдет из системы или пока не истечет срок действия сеанса. Функция `RegisterDefaultSignInService` создает службу с заданной областью для интерфейса `SignInManager`, которая разрешается с помощью структуры `SessionSignInMgr`.

Чтобы предоставить службу, которая представляет вошедшего в систему пользователя, добавьте файл с именем `user_service.go` в папку `platform/authorization` с содержимым, показанным в листинге 34-44.

```
package authorization

import (
    "platform/services"
    "platform/sessions"
    "platform/authorization/identity"
)

func RegisterDefaultUserService() {
    err := services.AddScoped(func(session sessions.Session,
            store identity.UserStore) identity.User {
        userID, found := session.GetValue(USER_SESSION_KEY).(int)
        if found {
            user, userFound := store.GetUserByID(userID)
            if (userFound) {
                return user
            }
        }
        return identity.UnauthenticatedUser
    })
    if (err != nil) {
        panic(err)
    }
}
```

Листинг 34-44 Содержимое файла user_service.go в папке authorization

Функция `RegisterDefaultUserService` создает службу с заданной областью для `User` интерфейса, которая считывает значение, хранящееся в текущем сеансе, и

использует его для запроса службы UserStore.

Чтобы создать простое условие доступа, которое проверяет, находится ли пользователь в роли, добавьте файл с именем role_condition.go в папку platform/authorization с содержимым, показанным в листинге 34-45.

```
package authorization

import ("platform/authorization/identity")

func NewRoleCondition(roles ...string) identity.AuthorizationCondition {
    return &roleCondition{ allowedRoles: roles}
}

type roleCondition struct {
    allowedRoles []string
}

func (c *roleCondition) Validate(user identity.User) bool {
    for _, allowedRole := range c.allowedRoles {
        if user.InRole(allowedRole) {
            return true
        }
    }
    return false
}
```

Листинг 34-45 Содержимое файла role_condition.go в папке authorization

Функция NewRoleCondition принимает набор ролей, которые используются для создания условия, возвращающего значение true, если пользователь был назначен какой-либо из них.

Реализация контроля доступа

Следующим шагом является добавление поддержки для определения ограничения доступа и применения его к запросам. Добавьте файл с именем auth_middleware.go в папку platform/authorization с содержимым, показанным в листинге 34-46.

```
package authorization

import (
    "net/http"
    "platform/authorization/identity"
    "platform/config"
    "platform/http/handling"
    "platform/pipeline"
    "strings"
    "regexp"
)

func             NewAuthComponent(prefix          string,           condition
identity.AuthorizationCondition,
```

```
        requestHandlers ...interface{}) *AuthMiddlewareComponent {

    entries := []handling.HandlerEntry {}
    for _, handler := range requestHandlers {
        entries = append(entries, handling.HandlerEntry{prefix, handler})
    }

    router := handling.NewRouter(entries...)

    return &AuthMiddlewareComponent{
        prefix: "/" + prefix ,
        condition:  condition,
        RequestPipeline: pipeline.CreatePipeline(router),
        fallbacks: map[*regexp.Regexp]string {},
    }
}

type AuthMiddlewareComponent struct {
    prefix string
    condition identity.AuthorizationCondition
    pipeline.RequestPipeline
    config.Configuration
    authFailURL string
    fallbacks map[*regexp.Regexp]string
}

func (c *AuthMiddlewareComponent) Init() {
                                                c.authFailURL,      _      =
c.Configuration.GetString("authorization:failUrl")
}

func (*AuthMiddlewareComponent) ImplementsProcessRequestWithServices() {}

func (c *AuthMiddlewareComponent) ProcessRequestWithServices(
        context *pipeline.ComponentContext,
        next func(*pipeline.ComponentContext),
        user identity.User) {

    if strings.HasPrefix(context.Request.URL.Path, c.prefix) {
        for expr, target := range c.fallbacks {
            if expr.MatchString(context.Request.URL.Path) {
                http.Redirect(context.ResponseWriter, context.Request,
                    target, http.StatusSeeOther)
                    return
            }
        }
        if c.condition.Validate(user) {
                                c.RequestPipeline.ProcessRequest(context.Request,
context.ResponseWriter)
        } else {
            if c.authFailURL != "" {
                http.Redirect(context.ResponseWriter, context.Request,
```

```
                    c.authFailURL, http.StatusSeeOther)
            } else if user.IsAuthenticated() {
                context.ResponseWriter.WriteHeader(http.StatusForbidden)
            } else {
                context.ResponseWriter.WriteHeader(http.StatusUnauthorized)
            }
        }
    } else {
        next(context)
    }
}

func (c *AuthMiddlewareComponent) AddFallback(target string,
        patterns ...string) *AuthMiddlewareComponent {
    for _, p := range patterns {
        c.fallbacks[regexp.MustCompile(p)] = target
    }
    return c
}
```

Листинг 34-46 Содержимое файла auth_middleware.go в папке authorization

Структура `AuthMiddlewareComponent` — это промежуточный компонент, который создает ветвь в конвейере запросов с маршрутизатором URL-адресов, обработчики которого получают запрос только при выполнении условия авторизации.

Реализация функций заполнителя приложения

Следуя шаблону, установленному для более ранних функций, я собираюсь создать базовые реализации функций авторизации, которые будет предоставлять приложение, использующее платформу. Добавьте файл с именем `placeholder_store.go` на `platform/placeholder` с содержимым, показанным в листинге 34-47.

```
package placeholder

import (
    "platform/services"
    "platform/authorization/identity"
    "strings"
)

func RegisterPlaceholderUserStore() {
    err := services.AddSingleton(func () identity.UserStore {
        return &PlaceholderUserStore{}
    })
    if (err != nil) {
        panic(err)
    }
}

var users = map[int]identity.User {
    1: identity.NewBasicUser(1, "Alice", "Administrator"),
```

```
    2: identity.NewBasicUser(2, "Bob"),
}

type PlaceholderUserStore struct {}

func (store *PlaceholderUserStore) GetUserByID(id int) (identity.User,
bool) {
    user, found := users[id]
    return user, found
}

func (store *PlaceholderUserStore) GetUserByName(name string)
(identity.User, bool) {
    for _, user := range users {
        if strings.EqualFold(user.GetDisplayName(), name) {
            return user, true
        }
    }
    return nil, false
}
```

Листинг 34-47 Содержимое файла placeholder_store.go в папке placeholder

Структура `PlaceholderUserStore` реализует интерфейс `UserStore` со статически определенными данными для двух пользователей, `Alice` и `Bob`, и используется функцией `RegisterPlaceholderUserStore` для создания одноэлементной службы.

Создание обработчика аутентификации

Чтобы разрешить простую аутентификацию, добавьте файл с именем `authentication_handler.go` в папку-заполнитель с содержимым, показанным в листинге 34-48.

```
package placeholder

import (
    "platform/http/actionresults"
    "platform/authorization/identity"
    "fmt"
)

type AuthenticationHandler struct {
    identity.User
    identity.SignInManager
    identity.UserStore
}

func (h AuthenticationHandler) GetSignIn() actionresults.ActionResult {
    return actionresults.NewTemplateAction("signin.html",
    fmt.Sprintf("Signed in as: %v", h.User.GetDisplayName()))
}
```

```
type Credentials struct {
    Username string
    Password string
}

func    (h    AuthenticationHandler)    PostSignIn(creds    Credentials)
actionresults.ActionResult {
    if creds.Password == "mysecret" {
        user, ok := h.UserStore.GetUserByName(creds.Username)
        if (ok) {
            h.SignInManager.SignIn(user)
            return actionresults.NewTemplateAction("signin.html",
                fmt.Sprintf("Signed in as: %v", user.GetDisplayName()))
        }
    }
    return actionresults.NewTemplateAction("signin.html", "Access Denied")
}

func (h AuthenticationHandler) PostSignOut() actionresults.ActionResult {
    h.SignInManager.SignOut(h.User)
    return actionresults.NewTemplateAction("signin.html", "Signed out")
}
```

Листинг 34-48 Содержимое файла authentication_handler.go в папке placeholder

Этот обработчик запросов имеет встроенный пароль — `mysecret` — для всех пользователей. Метод `GetSignIn` отображает шаблон для сбора имени пользователя и пароля. Метод `PostSignIn` проверяет пароль и удостоверяется, что в магазине есть пользователь с указанным именем, прежде чем выполнять вход пользователя в приложение. Метод `PostSignOut` подписывает пользователя из приложения. Чтобы создать шаблон, используемый обработчиком, добавьте файл с именем `signin.html` в папку `placeholder` с содержимым, показанным в листинге 34-49.

```
{{ layout "layout.html" }}

{{ if ne . "" }}
    <h3 style="padding: 10px;">{{. }}</h3>
{{ end }}

<form method="POST" action="/signin">
    <div style="padding: 5px;">
        <label>Username:</label>
        <input name="username" />
    </div>
    <div style="padding: 5px;">
        <label>Password:</label>
        <input name="password" />
    </div>
    <div style="padding: 5px;">
        <button type="submit">Sign In</button>
        <button type="submit" formaction="/signout">Sign Out</button>
    </div>
```

```
</form>
```

Листинг 34-49 Содержимое файла signin.html в папке placeholder

Шаблон отображает базовую HTML-форму с сообщением, предоставленным методом обработчика, который ее отображает.

Настройка приложения

Остается только настроить приложение для создания защищенного обработчика и настроить функции авторизации, как показано в листинге 34-50.

```
package placeholder

import (
    "platform/http"
    "platform/pipeline"
    "platform/pipeline/basic"
    "platform/services"
    "sync"
    "platform/http/handling"
    "platform/sessions"
    "platform/authorization"
)

func createPipeline() pipeline.RequestPipeline {
    return pipeline.CreatePipeline(
        &basic.ServicesComponent{},
        &basic.LoggingComponent{},
        &basic.ErrorComponent{},
        &basic.StaticFileComponent{},
        &sessions.SessionComponent{},
        //&SimpleMessageComponent{},
        authorization.NewAuthComponent(
            "protected",
            authorization.NewRoleCondition("Administrator"),
            CounterHandler{},
        ),
        handling.NewRouter(
            handling.HandlerEntry{ "",  NameHandler{}},
            handling.HandlerEntry{ "",  DayHandler{}},
            //handling.HandlerEntry{ "",  CounterHandler{}},
            handling.HandlerEntry{ "", AuthenticationHandler{}},
        ).AddMethodAlias("/", NameHandler.GetNames),
    )
}

func Start() {
    sessions.RegisterSessionService()
    authorization.RegisterDefaultSignInService()
    authorization.RegisterDefaultUserService()
    RegisterPlaceholderUserStore()
    results, err := services.Call(http.Serve, createPipeline())
```

```
    if (err == nil) {
        (results[0].(*sync.WaitGroup)).Wait()
    } else {
        panic(err)
    }
}
```

Листинг 34-50 Настройка приложения в файле startup.go в папке placeholder

Изменения создают ветвь конвейера с префиксом `/protected`, которая доступна только пользователям, которым назначена роль `Administrator`. `CounterHandler`, определенный ранее в этой главе, является единственным обработчиком ветки. `AuthenticationHandler` добавляется в основную ветвь конвейера.

Скомпилируйте и запустите приложение и используйте браузер для запроса `http://localhost:5000/protected/counter`. Это защищенный метод обработчика, и, поскольку зарегистрированного пользователя нет, будет показан результат, показанный на рисунке 34-8.

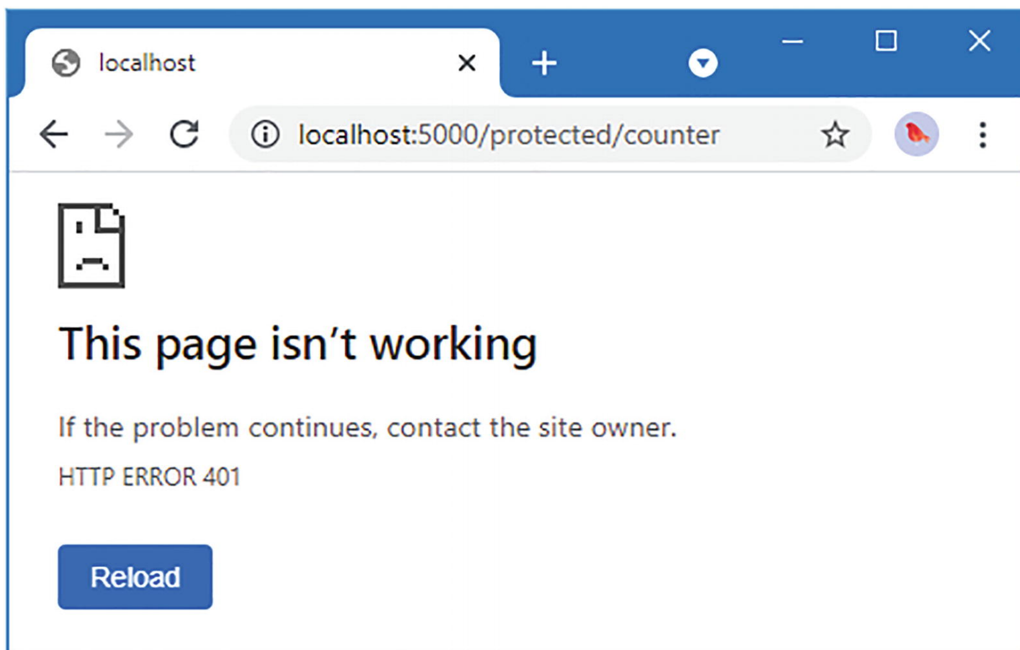

Рисунок 34-8 Неаутентифицированный запрос

Ответ 401 отправляется, когда пользователь, не прошедший проверку подлинности, запрашивает защищенный ресурс и известен как ответ на вызов, который часто используется для предоставления пользователю возможности войти в систему.

Затем запросите

`http://localhost:5000/signin`

, введите bob в поле **Username**, введите mysecret в поле **Password** и нажмите **Sign In**, как показано на рисунке 34-9. Запросите

http://localhost:5000/protected/counter, и вы получите ответ 403, который отправляется, когда пользователь, уже представивший свои учетные данные, запрашивает доступ к защищенному ресурсу.

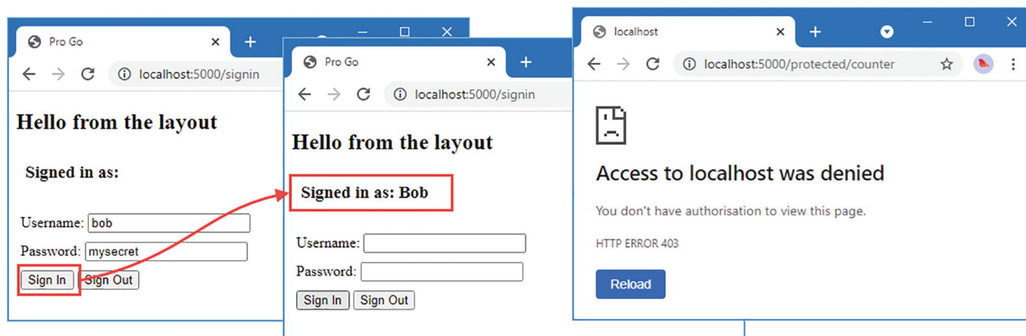

Рисунок 34-9 Неавторизованный запрос

Наконец, запросите http://localhost:5000/signin, введите alice в поле **Username** и mysecret в поле **Password** и нажмите **Sign In**, как показано на рисунке 34-10. Запросите http://localhost:5000/protected/counter, и вы получите ответ от обработчика, также показанного на рисунке 34-10, поскольку Alice находится в роли Adminstrator.

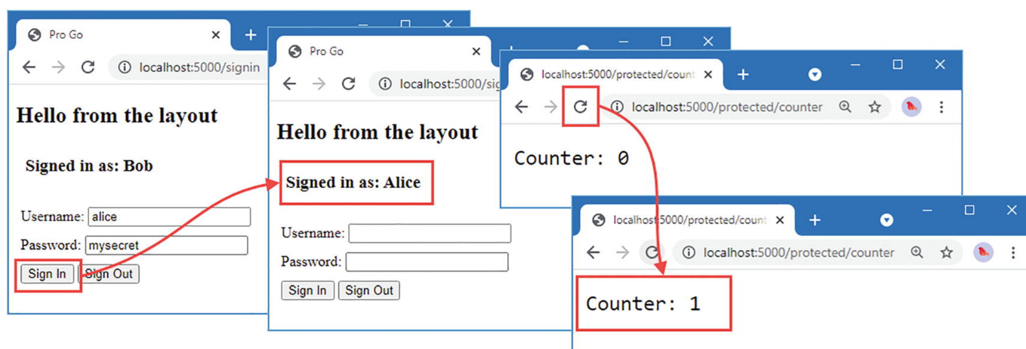

Рисунок 34-10 Авторизованный запрос

Резюме

В этой главе я завершил разработку собственной среды веб-приложений, добавив поддержку результатов действий, проверки данных, сеансов и авторизации. В следующей главе я начинаю процесс использования платформы для создания интернет-магазина.

35. SportsStore: настоящее приложение

В этой главе я начинаю разработку приложения SportsStore, которое представляет собой интернет-магазин спортивных товаров. Это пример, который я включаю во многие свои книги, что позволяет мне продемонстрировать, как один и тот же набор функций реализуется в разных языках и средах.

Создание проекта SportsStore

Я собираюсь создать приложение, использующее проект платформы, созданный в главах 32, но определенное в собственном проекте. Откройте командную строку и используйте ее для создания папки с именем `sportsstore` в той же папке, что и папка `platform`. Перейдите в папку `sportsstore` и выполните команду, показанную в листинге 35-1.

> **Подсказка**
> Вы можете загрузить пример проекта для этой главы — и для всех остальных глав этой книги — с `https://github.com/apress/pro-go`. См. Главу 2 о том, как получить помощь, если у вас возникнут проблемы с запуском примеров.

```
go mod init sportsstore
```

Листинг 35-1 Инициализация проекта

Эта команда создает файл `go.mod`. Чтобы объявить зависимость от проекта платформы, выполните команды, показанные в листинге 35-2, в папке `sportsstore`.

```
go mod edit -require="platform@v1.0.0"
go mod edit -replace="platform@v1.0.0"="../platform"
go get -d "platform@v1.0.0"
```

Листинг 35-2 Создание зависимости

Откройте файл `go.mod`, и вы увидите действие этих команд, как показано в листинге 35-3.

```
module sportsstore

go 1.17

require platform v1.0.0

require (
	github.com/gorilla/securecookie v1.1.1 // indirect
	github.com/gorilla/sessions v1.2.1 // indirect
)

replace platform v1.0.0 => ../platform
```

Листинг 35-3 Действие команд go в файле go.mod в папке sportsstore

Директива require объявляет зависимость от модуля platform. В реальных проектах это можно указать как URL-адрес вашего репозитория контроля версий, например URL-адрес GitHub. Этот проект не будет передан системе контроля версий, поэтому я просто использовал название platform.

Директива replace указывает локальный путь, по которому можно найти модуль platform. Когда инструменты Go устраняют зависимость от пакета в модуле platform, они делают это с использованием папки platform, которая находится на том же уровне, что и папка sportsstore.

Проект platform имеет зависимости от сторонних пакетов, которые необходимо разрешить, прежде чем их можно будет использовать. Это было сделано командой go get, создавшей директиву require, которая объявляет косвенные зависимости от пакетов, используемых для реализации сеансов в главе 34.

Настройка приложения

Добавьте файл с именем config.json в папку sportsstore и используйте его для определения параметров конфигурации, показанных в листинге 35-4.

```
{
    "logging" : {
        "level": "debug"
    },
    "files": {
        "path": "files"
    },
    "templates": {
        "path": "templates/*.html",
        "reload": true
    },
    "sessions": {
        "key": "MY_SESSION_KEY",
        "cyclekey": true
    }
}
```

Листинг 35-4 Содержимое файла config.json в папке sportsstore

Затем добавьте файл с именем main.go в папку sportsstore с содержимым, показанным в листинге 35-5.

```
package main

import (
          "platform/services"
    "platform/logging"
)

func writeMessage(logger logging.Logger) {
    logger.Info("SportsStore")
}
```

```
func main() {
    services.RegisterDefaultServices()
    services.Call(writeMessage)
}
```

Листинг 35-5 Содержимое файла main.go в папке sportsstore

Скомпилируйте и выполните проект с помощью команды, показанной в листинге 35-6, в папке `sportsstore`.

```
go run .
```

Листинг 35-6 Компиляция и выполнение проекта

Метод `main` устанавливает службы `platform` по умолчанию и вызывает `writeMessage`, выводя следующий результат:

```
07:55:03 INFO SportsStore
```

Запуск модели данных

Почти у всех проектов есть какая-то модель данных, и именно с нее я обычно начинаю разработку. Мне нравится начинать с нескольких простых типов данных, а затем начинать работать над тем, чтобы сделать их доступными для остальной части проекта. По мере добавления функций в приложение я возвращаюсь к модели данных и расширяю ее возможности.

Создайте папку `sportsstore/models` и добавьте в нее файл с именем `product.go` с содержимым, показанным в листинге 35-7.

```
package models

type Product struct {
    ID int
    Name string
    Description string
    Price float64
    *Category
}
```

Листинг 35-7 Содержимое файла product.go в папке models

Я предпочитаю определять один тип в каждом файле вместе со всеми связанными функциями конструктора или методами, связанными с этим типом. Чтобы создать тип данных для встроенного поля `Category`, добавьте файл с именем `category.go` в папку моделей с содержимым, показанным в листинге 35-8.

```
package models

type Category struct {
    ID int
    CategoryName string
}
```

Листинг 35-8 Содержимое файла category.go в папке models

При определении типов для встроенных полей я стараюсь выбирать имена полей, которые будут полезны при повышении уровня поля. В данном случае имя поля CategoryName было выбрано таким образом, чтобы оно не конфликтовало с полями, определенными окружающим типом Product, даже если это имя не то, которое я выбрал бы для автономного типа.

Определение интерфейса репозитория

Мне нравится использовать репозиторий как способ отделить источник данных в приложении от кода, который их потребляет. Добавьте файл с именем repository.go в папку sportsstore/models с содержимым, показанным в листинге 35-9.

```
package models

type Repository interface {

    GetProduct(id int) Product

    GetProducts() []Product

    GetCategories() []Category

    Seed()
}
```

Листинг 35-9 Содержимое файла repository.go в папке models

Я создам сервис для интерфейса Repository, который позволит мне легко менять источник данных, используемых в приложении.

Обратите внимание, что методы GetProduct, GetProducts и GetCategories, определенные в листинге 35-9, не возвращают указатели. Я предпочитаю использовать значения, чтобы код, использующий данные, не вносил изменения с помощью указателей, влияющих на данные, управляемые репозиторием. Этот подход означает, что значения данных будут дублироваться, но гарантирует отсутствие странных эффектов, вызванных случайными изменениями через общую ссылку. Иными словами, я не хочу, чтобы репозиторий предоставлял доступ к данным без обмена ссылками с кодом, который использует эти данные.

Реализация (временного) репозитория

Я буду хранить данные SportsStore в реляционной базе данных, но я предпочитаю начать с простой реализации репозитория в памяти, которую я использую до тех пор, пока не будут реализованы некоторые основные функции приложения.

По мере разработки проекта неизбежны изменения в подходе, и если я начну с базы данных для репозитория, то мне не захочется вносить изменения в написанные мной SQL-запросы. Это означает, что в конечном итоге я адаптирую код приложения, чтобы обойти ограничения SQL, что, как я знаю, не имеет смысла, но я также знаю, что я все равно это сделаю. Вы можете быть более дисциплинированным, но я получаю наилучшие результаты, работая с простым репозиторием в памяти, а затем пишу SQL только тогда, когда я понимаю, какой будет окончательная форма данных.

Создайте папку sportsstore/models/repo и добавьте в нее файл с именем memory_repo.go с содержимым, показанным в листинге 35-10.

```
package repo

import (
    "platform/services"
    "sportsstore/models"
)

func RegisterMemoryRepoService() {
    services.AddSingleton(func() models.Repository {
        repo := &MemoryRepo{}
        repo.Seed()
        return repo
    })
}

type MemoryRepo struct {
    products []models.Product
    categories []models.Category
}

func (repo *MemoryRepo) GetProduct(id int) (product models.Product) {
    for _, p := range repo.products {
        if (p.ID == id) {
            product = p
            return
        }
    }
    return
}

func (repo *MemoryRepo) GetProducts() (results []models.Product) {
    return repo.products
}

func (repo *MemoryRepo) GetCategories() (results []models.Category) {
    return repo.categories
}
```

Листинг 35-10 Содержимое memory_repo.go в папке models/repo

Структура MemoryRepo определяет большую часть функций, необходимых для реализации интерфейса репозитория, сохраняя значения в срезе. Чтобы реализовать метод Seed, добавьте файл с именем memory_repo_seed.go в папку repo с содержимым, показанным в листинге 35-11.

```
package repo

import (
    "fmt"
    "math/rand"
    "sportsstore/models"
```

```
)

func (repo *MemoryRepo) Seed() {
    repo.categories = make([]models.Category, 3)
    for i := 0; i < 3; i++ {
        catName := fmt.Sprintf("Category_%v", i + 1)
        repo.categories[i]= models.Category{ID: i + 1, CategoryName: catName}
    }

    for i := 0; i < 20; i++ {
        name := fmt.Sprintf("Product_%v", i + 1)
        price := rand.Float64() * float64(rand.Intn(500))
        cat := &repo.categories[rand.Intn(len(repo.categories))]
        repo.products = append(repo.products, models.Product{
            ID: i + 1,
            Name: name, Price: price,
            Description: fmt.Sprintf("%v (%v)", name, cat.CategoryName),
            Category: cat,
        })
    }
}
```

Листинг 35-11 Содержимое файла memory_repo_seed.go в папке models/repo

Я определил этот метод отдельно, чтобы не указывать код заполнения при добавлении функций в репозиторий.

Отображение списка продуктов

Первым шагом в отображении контента является отображение списка продуктов для продажи. Создайте папку `sportsstore/store` и добавьте в нее файл с именем `product_handler.go` с содержимым, показанным в листинге 35-12.

```
package store

import (
    "sportsstore/models"
    "platform/http/actionresults"
)

type ProductHandler struct {
    Repository models.Repository
}

type ProductTemplateContext struct {
    Products []models.Product
}

func (handler ProductHandler) GetProducts() actionresults.ActionResult {
    return actionresults.NewTemplateAction("product_list.html",
        ProductTemplateContext {
            Products: handler.Repository.GetProducts(),
        })
```

```
}
```

Листинг 35-12 Содержимое файла product_handler.go в папке store

Метод GetProducts отображает шаблон с именем product_list.html, передавая значение ProductTemplateContext, которое я буду использовать для предоставления дополнительной информации в шаблон позже.

> **Подсказка**
> Маршруты не генерируются для методов, которые продвигаются из анонимных встроенных полей структуры, чтобы случайно не создавать маршруты и не раскрывать внутреннюю работу обработчиков запросов для HTTP-запросов. Одним из следствий этого решения является то, что оно также исключает методы, определенные структурой, которая имеет то же имя, что и продвинутый метод. Именно по этой причине я присвоил имя полю Products, определенному структурой ProductHandler. Если бы я этого не сделал, то метод GetProducts не использовался бы для генерации маршрута, потому что он совпадает с именем метода, определенного интерфейсом models.Repository.

Создание шаблона и макета

Чтобы определить шаблон, создайте папку sportsstore/templates и добавьте в нее файл с именем product_list.html с содержимым, показанным в листинге 35-13.

```
{{ layout "store_layout.html" }}

{{ range .Products }}
    <div>
         {{.ID}}, {{ .Name }}, {{ printf "$%.2f" .Price }}, {{ .CategoryName
}}
    </div>
{{ end }}
```

Листинг 35-13 Содержимое файла product_list.html в папке templates

Макет использует выражение range для поля Product структуры, предоставленной обработчиком, для создания элемента div для каждого Product в Repository.

Чтобы создать макет, указанный в листинге 35-13, добавьте файл с именем store_layout.html в папку sportsstore/templates с содержимым, показанным в листинге 35-14..

```
<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>SportsStore</title>
</head>
<body>
    {{ body }}
</body>
</html>
```

Листинг 35-14 Содержимое файла store_layout.html в папке templates

Настройка приложения

Чтобы зарегистрировать службы и создать конвейер, требуемый приложением SportsStore, замените содержимое файла main.go тем, что показано в листинге 35-15.

```
package main

import (
    "sync"
    "platform/http"
    "platform/http/handling"
    "platform/services"
    "platform/pipeline"
    "platform/pipeline/basic"
    "sportsstore/store"
    "sportsstore/models/repo"
)

func registerServices() {
    services.RegisterDefaultServices()
    repo.RegisterMemoryRepoService()
}

func createPipeline() pipeline.RequestPipeline {
    return pipeline.CreatePipeline(
        &basic.ServicesComponent{},
        &basic.LoggingComponent{},
        &basic.ErrorComponent{},
        &basic.StaticFileComponent{},
        handling.NewRouter(
            handling.HandlerEntry{ "",  store.ProductHandler{}},
        ).AddMethodAlias("/", store.ProductHandler.GetProducts),
    )
}

func main() {
    registerServices()
    results, err := services.Call(http.Serve, createPipeline())
    if (err == nil) {
        (results[0].(*sync.WaitGroup)).Wait()
    } else {
        panic(err)
    }
}
```

Листинг 35-15 Замена содержимого файла main.go в папке sportsstore

Службы по умолчанию регистрируются вместе с хранилищем памяти. Конвейер содержит основные компоненты, созданные в главе 34, с маршрутизатором, настроенным с помощью ProductHandler.

Скомпилируйте и запустите проект и используйте браузер для запроса http://localhost:5000, который даст ответ, показанный на рисунке 35-1.

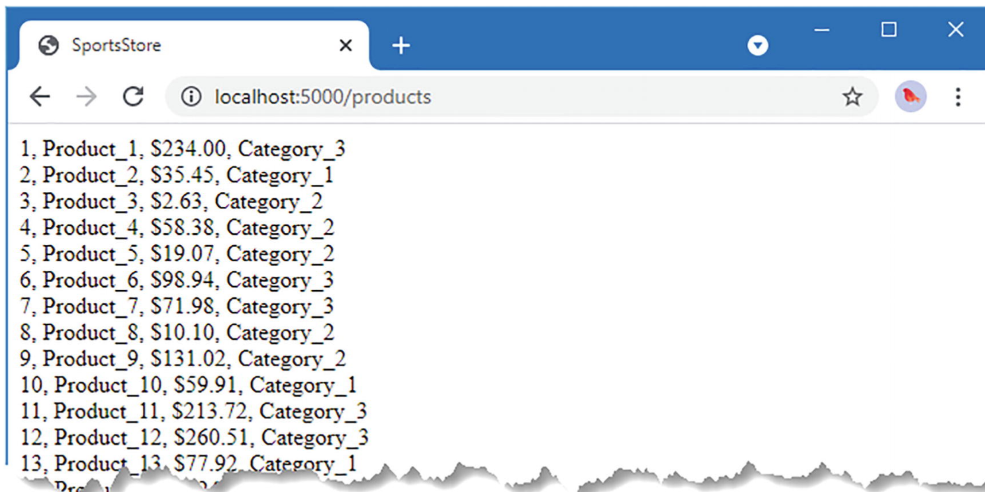

Рисунок 35-1 Отображение списка продуктов

Работа с запросами разрешений брандмауэра Windows

Как объяснялось в предыдущих главах, Windows будет запрашивать разрешения брандмауэра каждый раз, когда проект компилируется с помощью команды `go run`, чего можно избежать, используя простой сценарий PowerShell. Напомню, вот содержимое скрипта, который я сохраняю как `buildandrun.ps1`:

```
$file = "./sportsstore.exe"

&go build -o $file

if ($LASTEXITCODE -eq 0) {
    &$file
}
```

Чтобы собрать и выполнить проект, используйте команду `./buildandrun.ps1` в папке `sportsstore`.

Добавление пагинации

Вывод на рисунке 35-1 показывает, что все продукты в репозитории отображаются в одном списке. Следующим шагом является добавление поддержки разбиения на страницы, чтобы пользователю было представлено небольшое количество продуктов и он мог перемещаться между страницами. Мне нравится вносить изменения в репозиторий, а затем работать до тех пор, пока не будет достигнут шаблон, отображающий данные. В листинге 35-16 к интерфейсу `Repository` добавлен метод, который позволяет запрашивать страницу значений `Product`.

```
package models

type Repository interface {

    GetProduct(id int) Product
```

```
    GetProducts() []Product

    GetProductPage(page, pageSize int) (products []Product, totalAvailable
int)

    GetCategories() []Category

    Seed()
}
```

Листинг 35-16 Добавление метода в файл репозитория.go в папке models

Метод `GetProductPage` возвращает срез `Product` и общее количество элементов в репозитории. Перечисление 35-17 реализует новый метод в репозитории памяти.

```
package repo

import (
    "platform/services"
    "sportsstore/models"
    "math"
)

func RegisterMemoryRepoService() {
    services.AddSingleton(func() models.Repository {
        repo := &MemoryRepo{}
        repo.Seed()
        return repo
    })
}

type MemoryRepo struct {
    products []models.Product
    categories []models.Category
}

func (repo *MemoryRepo) GetProduct(id int) (product models.Product) {
    for _, p := range repo.products {
        if (p.ID == id) {
            product = p
            return
        }
    }
    return
}

func (repo *MemoryRepo) GetProducts() (results []models.Product) {
    return repo.products
}

func (repo *MemoryRepo) GetCategories() (results []models.Category) {
    return repo.categories
}
```

```
func (repo *MemoryRepo) GetProductPage(page, pageSize int) ([]models.Product,
int) {
    return getPage(repo.products, page, pageSize), len(repo.products)
}

func getPage(src []models.Product, page, pageSize int) []models.Product {
    start := (page -1) * pageSize
    if page > 0 && len(src) > start {
        end := (int)(math.Min((float64)(len(src)), (float64)(start +
pageSize)))
        return src[start : end]
    }
    return []models.Product{}
}
```

Листинг 35-17 Реализация метода в файле memory_repo.go в папке models/repo

Листинг 35-18 обновляет обработчик запроса, чтобы он выбирал страницу данных и передал ее шаблону вместе с дополнительными полями структуры, необходимыми для поддержки разбиения на страницы.

```
package store

import (
    "sportsstore/models"
    "platform/http/actionresults"
    "platform/http/handling"
    "math"
)

const pageSize = 4

type ProductHandler struct {
    Repository models.Repository
    URLGenerator handling.URLGenerator
}

type ProductTemplateContext struct {
    Products []models.Product
    Page int
    PageCount int
    PageNumbers []int
    PageUrlFunc func(int) string
}

func (handler ProductHandler) GetProducts(page int)
actionresults.ActionResult {
    prods, total := handler.Repository.GetProductPage(page, pageSize)
    pageCount := int(math.Ceil(float64(total) / float64(pageSize)))
    return actionresults.NewTemplateAction("product_list.html",
        ProductTemplateContext {
            Products: prods,
            Page: page,
            PageCount: pageCount,
```

```
            PageNumbers: handler.generatePageNumbers(pageCount),
            PageUrlFunc: handler.createPageUrlFunction(),
        })
}

func (handler ProductHandler) createPageUrlFunction() func(int) string {
    return func(page int) string {
                                                        url,      _      :=
handler.URLGenerator.GenerateUrl(ProductHandler.GetProducts, page)
        return url
    }
}

func  (handler  ProductHandler)  generatePageNumbers(pageCount  int)  (pages
[]int) {
    pages = make([]int, pageCount)
    for i := 0; i < pageCount; i++ {
        pages[i] = i + 1
    }
    return
}
```

Листинг 35-18 Обновление метода обработчика в файле product_handler.go в папке store

В листинге 35-18 появилось много новых операторов, потому что обработчик должен предоставить гораздо больше информации шаблону для поддержки нумерации страниц. Метод `GetProducts` был изменен, чтобы принимать параметр, который используется для получения страницы данных. Дополнительные поля, определенные для структуры, передаваемой в шаблон, включают в себя выбранную страницу, функцию для создания URL-адресов для перехода на страницу и срез, содержащий последовательность чисел (что необходимо, поскольку шаблоны могут использовать диапазоны, но не циклы `for` для создания контента). Листинг 35-19 обновляет шаблон для использования новой информации.

```
{{ layout "store_layout.html" }}
{{ $context := . }}

{{ range .Products }}
    <div>
         {{.ID}}, {{ .Name }}, {{ printf "$%.2f" .Price }}, {{ .CategoryName
}}
    </div>
{{ end }}

{{ range .PageNumbers}}
    {{ if eq $context.Page .}}
        {{ . }}
    {{ else }}
        <a href="{{ call $context.PageUrlFunc . }}">{{ . }}</a>
    {{ end }}
{{ end }}
```

Листинг 35-19 Поддержка нумерации страниц в файле product_list.html в папке templates

Я определил переменную `$context`, чтобы всегда иметь легкий доступ к значению структуры, переданному в шаблон методом обработчика. Новое выражение `range` перечисляет список номеров страниц и отображает ссылку навигации для всех из них, кроме текущей выбранной страницы. URL-адрес для ссылки создается путем вызова функции, назначенной полю `PageUrlFunc` контекстной структуры.

Затем необходимо изменить псевдонимы, установленные для системы маршрутизации, чтобы URL-адрес по умолчанию и URL-адрес `/products` инициировали перенаправление на первую страницу продуктов, как показано в листинге 35-20.

```
...
func createPipeline() pipeline.RequestPipeline {
    return pipeline.CreatePipeline(
        &basic.ServicesComponent{},
        &basic.LoggingComponent{},
        &basic.ErrorComponent{},
        &basic.StaticFileComponent{},
        handling.NewRouter(
            handling.HandlerEntry{ "",  store.ProductHandler{}},
        ).AddMethodAlias("/", store.ProductHandler.GetProducts, 1).
            AddMethodAlias("/products", store.ProductHandler.GetProducts, 1),
    )
}
...
```

Листинг 35-20 Обновление псевдонимов в файле main.go в папке sportsstore

Скомпилируйте и запустите проект и используйте браузер для запроса `http://localhost:5000`. Вам будут представлены продукты, отображаемые на четырех страницах, с навигационными ссылками, которые запрашивают другие страницы, как показано на рисеуrt 35-2.

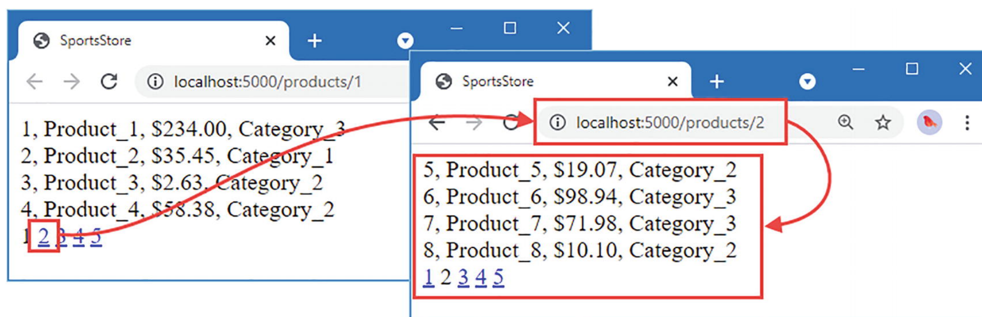

Рисунок 35-2 Добавление поддержки пагинации

Стилизация содержимого шаблона

Прежде чем добавлять какие-либо дополнительные функции в приложение, я собираюсь рассмотреть внешний вид продуктов в списке. Я собираюсь использовать Bootstrap, популярный CSS-фреймворк, который мне нравится использовать. Bootstrap применяет стили, используя атрибуты `class` HTML-элементов, и подробно описан на `https://getbootstrap.com`.

Установка CSS-файла Bootstrap

В Go нет хорошего способа установки пакетов за пределами экосистемы Go. Чтобы добавить файл CSS в проект, создайте папку `sportsstore/files` и с помощью командной строки запустите команду, показанную в листинге 35-21, в папке `sportsstore`.

```
curl
https://cdnjs.cloudflare.com/ajax/libs/bootstrap/5.1.1/css/bootstrap.min.css
--output files/bootstrap.min.css
```

Листинг 35-21 Загрузка таблицы стилей CSS

Если вы используете Windows, используйте команду PowerShell, показанную в листинге 35-22.

```
Invoke-WebRequest                          -Uri                          `
"https://cdnjs.cloudflare.com/ajax/libs/bootstrap/5.1.1/css/bootstrap.min.css"
`
-OutFile "files/bootstrap.min.css"
```

Листинг 35-22 Загрузка таблицы стилей CSS в Windows

Обновление макета

Добавьте элемент, показанный в листинге 35-23, в файл `store_layout.html` в папке `templates`.

```
<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>SportsStore</title>
    <link href="/files/bootstrap.min.css" rel="stylesheet" />
</head>
<body>
    <div class="bg-dark text-white p-2">
        <span class="navbar-brand ml-2">SPORTS STORE</span>
    </div>
    <div class="row m-1 p-1">
        <div id="sidebar" class="col-3">
            {{ template "left_column" . }}
        </div>
        <div class="col-9">
            {{ template "right_column" . }}
        </div>
    </div>
</body>
</html>
```

Листинг 35-23 Добавление Bootstrap в файл store_layout.html в папке templates

Новые элементы добавляют элемент `link` для CSS-файла Bootstrap и используют функции Bootstrap для создания заголовка и двухколоночного макета. Содержимое столбцов получается из шаблонов с именами `left_column` и `right_column`.

Стилизация содержимого шаблона

Роль шаблона `product_list.html` должна измениться, чтобы соответствовать ожиданиям макета и определить шаблоны для левого и правого столбцов в макете, как показано в листинге 35-24.

```
{{ layout "store_layout.html" }}

{{ define "left_column" }}
    Put something useful here
{{end}}

{{ define "right_column" }}
    {{ $context := . }}
    {{ range $context.Products }}
        <div class="card card-outline-primary m-1 p-1">
            <div class="bg-faded p-1">
                <h4>
                    {{ .Name }}
                                <span class="badge rounded-pill bg-primary"
style="float:right">
                        <small>{{ printf "$%.2f" .Price }}</small>
                    </span>
                </h4>
            </div>
            <div class="card-text p-1">{{ .Description }}</div>
        </div>
    {{ end }}
    {{ template "page_buttons.html" $context }}
{{end}}
```

Листинг 35-24 Создание содержимого столбца в файле product_list.html в папке templates

Новая структура определяет заполнитель для левого столбца и создает список стилизованных продуктов в правом столбце.

Я определил отдельный шаблон для кнопок пагинации. Добавьте файл с именем `page_buttons.html` в папку `templates` с содержимым, показанным в листинге 35-25.

```
{{ $context := . }}
<div class="btn-group pull-right m-1">
    {{ range .PageNumbers}}
        {{ if eq $context.Page .}}
        <a class="btn btn-primary">{{ . }}</a>
        {{ else }}
            <a href="{{ call $context.PageUrlFunc . }}"
                class="btn btn-outline-primary">{{ . }}</a>
        {{ end }}
    {{ end }}
</div>
```

Листинг 35-25 Содержимое файла page_buttons.html в папке templates

Скомпилируйте и запустите проект и запросите `http://localhost:5000`. Вы увидите стилизованное содержимое, показанное на рисунке 35-3.

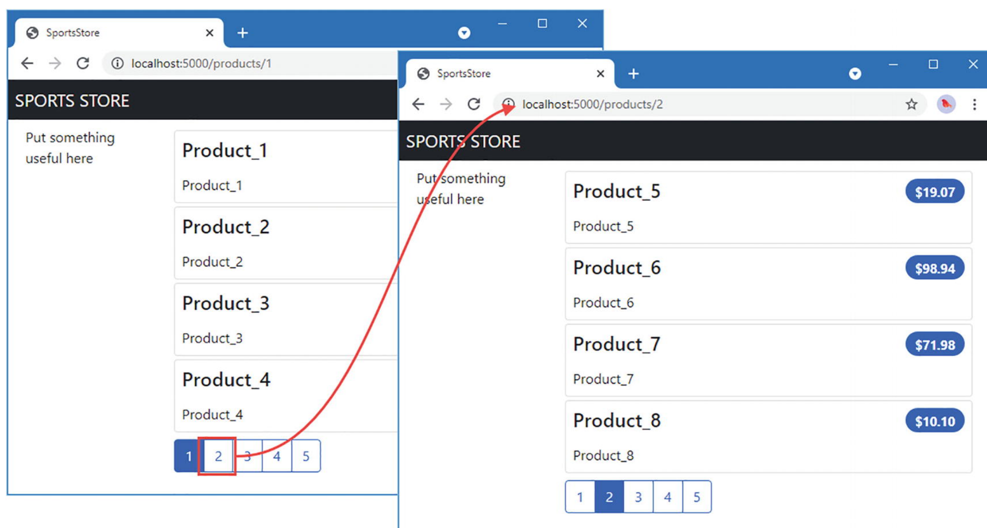

Рисунок 35-3 Стилизация содержимого

Добавление поддержки фильтрации категорий

Следующим шагом является замена содержимого заполнителя в левом столбце кнопками, позволяющими пользователю выбирать категорию, по которой следует фильтровать продукты, отображаемые в списке. Для начала добавьте метод, показанный в листинге 35-26, в интерфейс `Repository`.

```
package models

type Repository interface {

    GetProduct(id int) Product

    GetProducts() []Product

     GetProductPage(page, pageSize int) (products []Product, totalAvailable
int)

      GetProductPageCategory(categoryId int, page, pageSize int) (products
[]Product,
        totalAvailable int)

    GetCategories() []Category

    Seed()
}
```

Листинг 35-26 Добавление метода в файл репозитория.go в папке models

Новый метод позволяет указать категорию при запросе страницы. Листинг 35-27 реализует новый метод в репозитории памяти.

```
package repo
```

```
import (
    "platform/services"
    "sportsstore/models"
    "math"
)

func RegisterMemoryRepoService() {
    services.AddSingleton(func() models.Repository {
        repo := &MemoryRepo{}
        repo.Seed()
        return repo
    })
}

type MemoryRepo struct {
    products []models.Product
    categories []models.Category
}

func (repo *MemoryRepo) GetProduct(id int) (product models.Product) {
    for _, p := range repo.products {
        if (p.ID == id) {
            product = p
            return
        }
    }
    return
}

func (repo *MemoryRepo) GetProducts() (results []models.Product) {
    return repo.products
}

func (repo *MemoryRepo) GetCategories() (results []models.Category) {
    return repo.categories
}

func (repo *MemoryRepo) GetProductPage(page, pageSize int) ([]models.Product,
int) {
    return getPage(repo.products, page, pageSize), len(repo.products)
}

func (repo *MemoryRepo) GetProductPageCategory(category int, page,
        pageSize int) (products []models.Product, totalAvailable int) {
    if category == 0 {
        return repo.GetProductPage(page, pageSize)
    } else {
        filteredProducts := make([]models.Product, 0, len(repo.products))
        for _, p := range repo.products {
            if p.Category.ID == category {
                filteredProducts = append(filteredProducts, p)
            }
        }
```

```
                        return   getPage(filteredProducts,   page,   pageSize),
len(filteredProducts)
    }
}

func getPage(src []models.Product, page, pageSize int) []models.Product {
    start := (page -1) * pageSize
    if page > 0 && len(src) > start {
          end  :=  (int)(math.Min((float64)(len(src)),  (float64)(start  +
pageSize)))
        return src[start : end]
    }
    return []models.Product{}
}
```

Листинг 35-27 Реализация метода в файле memory_repository.go в папке models

Новый метод перечисляет данные о продукте, фильтруя выбранную категорию, а затем выбирает указанную страницу данных.

Обновление обработчика запросов

Следующим шагом является изменение метода обработчика запроса, чтобы он получал параметр категории и использовал его для получения отфильтрованных данных, которые затем передаются в шаблон вместе с дополнительными данными контекста, необходимыми для создания кнопок навигации, которые позволяют выбрать другую категорию. выбрано, как показано в листинге 35-28..

```
package store

import (
    "sportsstore/models"
    "platform/http/actionresults"
    "platform/http/handling"
    "math"
)

const pageSize = 4

type ProductHandler struct {
    Repository models.Repository
    URLGenerator handling.URLGenerator
}

type ProductTemplateContext struct {
    Products []models.Product
    Page int
    PageCount int
    PageNumbers []int
    PageUrlFunc func(int) string
    SelectedCategory int
}

func (handler ProductHandler) GetProducts(category,
```

```
        page int) actionresults.ActionResult {
    prods, total := handler.Repository.GetProductPageCategory(category,
        page, pageSize)
    pageCount := int(math.Ceil(float64(total) / float64(pageSize)))
    return actionresults.NewTemplateAction("product_list.html",
        ProductTemplateContext {
            Products: prods,
            Page: page,
            PageCount: pageCount,
            PageNumbers: handler.generatePageNumbers(pageCount),
            PageUrlFunc: handler.createPageUrlFunction(category),
            SelectedCategory: category,
        })
}

func (handler ProductHandler) createPageUrlFunction(category int) func(int)
string {
    return func(page int) string {
                                                        url,      _      :=
handler.URLGenerator.GenerateUrl(ProductHandler.GetProducts,
            category, page)
        return url
    }
}

func (handler ProductHandler) generatePageNumbers(pageCount int) (pages
[]int) {
    pages = make([]int, pageCount)
    for i := 0; i < pageCount; i++ {
        pages[i] = i + 1
    }
    return
}
```

Листинг 35-28 Добавление поддержки фильтрации категорий в файле product_handler.go в папке store

Мне также пришлось обновить существующую функцию, которая генерирует URL-адреса для выбора страницы, и ввести функцию, которая генерирует URL-адреса для выбора новой категории.

Создание обработчика категории

Причина, по которой я добавил поддержку вызова обработчиков из шаблонов, заключалась в том, что я мог отображать автономный контент, такой как кнопки категорий. Добавьте файл с именем `category_handler.go` в папку `sportsstore/store` с содержимым, показанным в листинге 35-29.

```
package store

import (
    "sportsstore/models"
    "platform/http/actionresults"
    "platform/http/handling"
)
```

```
type CategoryHandler struct {
    Repository models.Repository
    URLGenerator handling.URLGenerator
}

type categoryTemplateContext struct {
    Categories []models.Category
    SelectedCategory int
    CategoryUrlFunc func(int) string
}

func (handler CategoryHandler) GetButtons(selected int)
actionresults.ActionResult {
    return actionresults.NewTemplateAction("category_buttons.html",
        categoryTemplateContext {
            Categories: handler.Repository.GetCategories(),
            SelectedCategory: selected,
            CategoryUrlFunc: handler.createCategoryFilterFunction(),
        })
}

func (handler CategoryHandler) createCategoryFilterFunction() func(int)
string {
    return func(category int) string {
        url, _ :=
handler.URLGenerator.GenerateUrl(ProductHandler.GetProducts,
            category, 1)
        return url
    }
}
```

Листинг 35-29 Содержимое файла category_handler.go в папке store

Набор категорий, для которых требуются кнопки, обработчик получает через репозиторий, полученный как сервис. Выбранная категория получается через параметр метода-обработчика.

Чтобы создать шаблон, отображаемый методом обработчика `GetButtons`, добавьте файл с именем `category_buttons.html` в папку templates с содержимым, показанным в листинге 35-30.

```
{{ $context := . }}

<div class="d-grid gap-2">
        <a
    {{ if eq $context.SelectedCategory 0}}
        class="btn btn-primary"
    {{ else }}
        class="btn btn-outline-primary"
    {{ end }}
        href="{{ call $context.CategoryUrlFunc 0 }}">All</a>
    {{ range $context.Categories }}
            <a
        {{ if eq $context.SelectedCategory .ID}}
            class="btn btn-primary"
```

```
        {{ else }}
            class="btn btn-outline-primary"
        {{ end }}
              href="{{ call $context.CategoryUrlFunc .ID }}">{{ .CategoryName
}}</a>
    {{ end }}
</div>
```

Листинг 35-30 Содержимое файла category_buttons.html в папке templates

Обычно я предпочитаю помещать полные элементы в предложения блоков `if/else/end`, но, как показывает этот шаблон, вы можете использовать условие, чтобы выбрать только ту часть элемента, которая отличается, в данном случае это атрибут `class`. Хотя дублирования меньше, я нахожу это более трудным для чтения, но оно служит для демонстрации того, что вы можете использовать систему шаблонов так, как это соответствует вашим личным предпочтениям.

Отображение навигации по категориям в шаблоне списка товаров

В листинге 35-31 показаны изменения, необходимые для шаблона, в котором перечислены продукты, чтобы включить функции фильтра категорий.

```
{{ layout "store_layout.html" }}

{{ define "left_column" }}
    {{ $context := . }}
    {{ handler "category" "getbuttons" $context.SelectedCategory}}
{{end}}

{{ define "right_column" }}
    {{ $context := . }}
    {{ range $context.Products }}
        <div class="card card-outline-primary m-1 p-1">
            <div class="bg-faded p-1">
                <h4>
                    {{ .Name }}
                                <span class="badge rounded-pill bg-primary"
style="float:right">
                        <small>{{ printf "$%.2f" .Price }}</small>
                    </span>
                </h4>
            </div>
            <div class="card-text p-1">{{ .Description }}</div>
        </div>
    {{ end }}
    {{ template "page_buttons.html" $context }}
{{end}}
```

Листинг 35-31 Отображение категорий в файле product_list.html в папке templates

Изменения заменяют сообщение-заполнитель ответом от метода `GetButtons`, определенного в листинге 35-30.

Регистрация обработчика и обновление псевдонимов

Последнее изменение заключается в обновлении псевдонимов, которые сопоставляют URL-адреса с методом обработчика, как показано в листинге 35-32.

```
...
func createPipeline() pipeline.RequestPipeline {
    return pipeline.CreatePipeline(
        &basic.ServicesComponent{},
        &basic.LoggingComponent{},
        &basic.ErrorComponent{},
        &basic.StaticFileComponent{},
        handling.NewRouter(
            handling.HandlerEntry{ "",  store.ProductHandler{}},
            handling.HandlerEntry{ "",  store.CategoryHandler{}},
        ).AddMethodAlias("/", store.ProductHandler.GetProducts, 0, 1).
            AddMethodAlias("/products[/]?[A-z0-9]*?",
                store.ProductHandler.GetProducts, 0, 1),
    )
}
...
```

Листинг 35-32 Обновление псевдонимов маршрутов в файле main.go в папке

Скомпилируйте и выполните проект и запросите `http://localhost:5000`, и вы увидите кнопки категорий и сможете выбирать продукты из одной категории, как показано на рисунке 35-4.

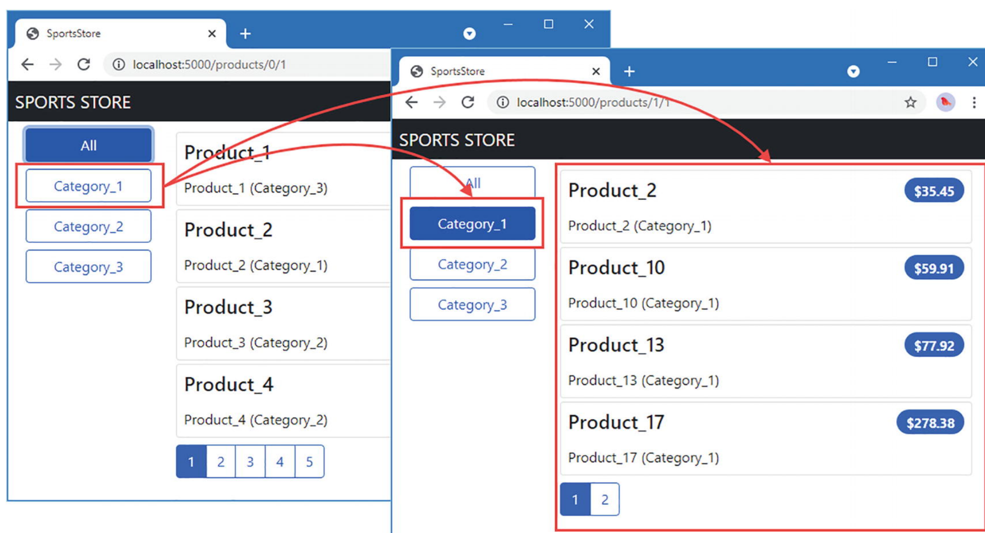

Рисунок 35-4 Фильтрация по категории

Резюме

В этой главе я начал разработку приложения SportsStore, используя платформу, созданную в главах 32–34. Я начал с базовой модели данных и репозитория и создал обработчик, который отображает продукты с поддержкой разбивки на страницы и фильтрации по категориям. В следующей главе я продолжу разработку приложения SportsStore.

36. SportsStore: корзина и база данных

В этой главе я продолжаю разработку приложения SportsStore, добавляя поддержку корзины покупок и добавляя базу данных вместо временного репозитория, созданного в главе 35.

> **Подсказка**
> Вы можете загрузить пример проекта для этой главы — и для всех остальных глав этой книги — с `https://github.com/apress/pro-go`. См. Главу 2 о том, как получить помощь, если у вас возникнут проблемы с запуском примеров.

Создание корзины покупок

Приложение SportsStore работает хорошо, но я не могу продавать какие-либо продукты, пока не реализую корзину для покупок, которая позволит пользователям собирать свой выбор перед оплатой.

Определение модели корзины и репозитория

Чтобы определить тип данных тележки, создайте папку `sportsstore/store/cart` и добавьте в нее файл с именем `cart.go` с содержимым, показанным в листинге 36-1.

```
package cart

import "sportsstore/models"

type CartLine struct {
    models.Product
    Quantity int
}

func (cl *CartLine) GetLineTotal() float64 {
    return cl.Price * float64(cl.Quantity)
}

type Cart interface {
    AddProduct(models.Product)
    GetLines() []*CartLine
    RemoveLineForProduct(id int)
    GetItemCount() int
    GetTotal() float64
```

```
    Reset()
}

type BasicCart struct {
    lines []*CartLine
}

func (cart *BasicCart) AddProduct(p models.Product) {
    for _, line := range cart.lines {
        if (line.Product.ID == p.ID) {
            line.Quantity++
            return
        }
    }
    cart.lines = append(cart.lines, &CartLine{
        Product: p, Quantity: 1,
    })
}

func (cart *BasicCart) GetLines() []*CartLine {
    return cart.lines
}

func (cart *BasicCart) RemoveLineForProduct(id int) {
    for index, line := range cart.lines {
        if (line.Product.ID == id) {
             cart.lines = append(cart.lines[0: index], cart.lines[index
+ 1:]...)
        }
    }
}

func (cart *BasicCart) GetItemCount() (total int) {
    for _, l := range cart.lines {
        total += l.Quantity
    }
    return
}

func (cart *BasicCart) GetTotal() (total float64) {
    for _, line := range cart.lines {
        total += float64(line.Quantity) * line.Product.Price
    }
    return
}

func (cart *BasicCart) Reset() {
    cart.lines = []*CartLine{}
}
```

Листинг 36-1 Содержимое файла cart.go в папке store/cart

Интерфейс `Cart` будет предоставляться как служба, и я определил структуру `BasicCart`, которая реализует методы `Cart` с использованием среза. Чтобы определить службу, добавьте файл с именем `cart_service.go` в папку `sportsstore/store/cart` с содержимым, показанным в листинге 36-2.

```
package cart

import (
    "platform/services"
    "platform/sessions"
    "sportsstore/models"
    "encoding/json"
    "strings"
)

const CART_KEY string = "cart"

func RegisterCartService() {
    services.AddScoped(func(session sessions.Session) Cart {
        lines := []*CartLine {}
        sessionVal := session.GetValue(CART_KEY)
        if strVal, ok := sessionVal.(string); ok {
            json.NewDecoder(strings.NewReader(strVal)).Decode(&lines)
        }
        return &sessionCart{
            BasicCart: &BasicCart{ lines: lines},
            Session: session,
        }
    })
}

type sessionCart struct {
    *BasicCart
    sessions.Session
}

func (sc *sessionCart) AddProduct(p models.Product) {
    sc.BasicCart.AddProduct(p)
    sc.SaveToSession()
}

func (sc *sessionCart) RemoveLineForProduct(id int) {
    sc.BasicCart.RemoveLineForProduct(id)
    sc.SaveToSession()
}

func (sc *sessionCart) SaveToSession() {
```

```
    builder := strings.Builder{}
    json.NewEncoder(&builder).Encode(sc.lines)
    sc.Session.SetValue(CART_KEY, builder.String())
}

func (sc *sessionCart) Reset() {
    sc.lines = []*CartLine{}
    sc.SaveToSession()
}
```

Листинг 36-2 Содержимое файла cart_service.go в папке store/cart

Структура `sessionCart` реагирует на изменения, добавляя JSON-представление своих значений `CartLine` в сеанс. Функция `RegisterCartService` создает службу `Cart` с ограниченной областью действия, которая создает `sessionCart` и заполняет ее строки из данных сеанса JSON.

Создание обработчика запроса корзины

Добавьте файл с именем `cart_handler.go` в папку `sportsstore/store` с содержимым, показанным в листинге 36-3.

```
package store

import (
    "platform/http/actionresults"
    "platform/http/handling"
    "sportsstore/models"
    "sportsstore/store/cart"
)

type CartHandler struct {
    models.Repository
    cart.Cart
    handling.URLGenerator
}

type CartTemplateContext struct {
    cart.Cart
    ProductListUrl string
    CartUrl string
    CheckoutUrl string
    RemoveUrl string
}

func (handler CartHandler) GetCart() actionresults.ActionResult {
                return    actionresults.NewTemplateAction("cart.html",
CartTemplateContext {
        Cart: handler.Cart,
```

```
                                                    ProductListUrl:
handler.mustGenerateUrl(ProductHandler.GetProducts, 0, 1),
                                                         RemoveUrl:
handler.mustGenerateUrl(CartHandler.PostRemoveFromCart),
    })
}

type CartProductReference struct {
    ID int
}

func  (handler  CartHandler)  PostAddToCart(ref  CartProductReference)
actionresults.ActionResult {
    p := handler.Repository.GetProduct(ref.ID)
    handler.Cart.AddProduct(p)
    return actionresults.NewRedirectAction(
        handler.mustGenerateUrl(CartHandler.GetCart))
}

func (handler CartHandler) PostRemoveFromCart(ref CartProductReference)
actionresults.ActionResult {
    handler.Cart.RemoveLineForProduct(ref.ID)
    return actionresults.NewRedirectAction(
        handler.mustGenerateUrl(CartHandler.GetCart))
}

func  (handler  CartHandler)  mustGenerateUrl(method  interface{},  data
...interface{}) string {
    url, err := handler.URLGenerator.GenerateUrl(method, data...)
    if (err != nil) {
        panic(err)
    }
    return url
}
```

Листинг 36-3 Содержимое файла cart_handler.go в папке store

Метод `GetCart` отображает шаблон, отображающий содержимое корзины пользователя. Будет вызван метод `PostAddToCart` для добавления товара в корзину, после чего браузер будет перенаправлен на метод `GetCart`. Чтобы создать шаблон, используемый методом `GetCart`, добавьте файл с именем `cart.html` в папку шаблонов с содержимым, показанным в листинге 36-4.

```
{{ layout "simple_layout.html" }}
{{ $context := . }}

<div class="p-1">
    <h2>Your cart</h2>
    <table class="table table-bordered table-striped">
        <thead>
```

```
            <tr>
                <th>Quantity</th><th>Item</th>
                <th class="text-end">Price</th>
                <th class="text-end">Subtotal</th>
                <th />
            </tr>
        </thead>
        <tbody>
            {{ range $context.Cart.GetLines }}
                <tr>
                    <td class="text-start">{{ .Quantity }}</td>
                    <td class="text-start">{{ .Name }}</td>
                       <td class="text-end">{{ printf "$%.2f" .Price }}
</td>
                    <td class="text-end">
                        {{ printf "$%.2f" .GetLineTotal }}
                    </td>
                    <td>
                                                  <form  method="POST"  action="
{{  $context.RemoveUrl }}">
                                       <input type="hidden" name="id" value="{{
.ID }}" />
                                          <button class="btn btn-sm btn-danger"
type="submit">
                                Remove
                            </button>
                        </form>
                    </td>
                </tr>

            {{ end }}
        </tbody>
        <tfoot>
            <tr>
                <td colspan="3" class="text-end">Total:</td>
                <td class="text-end">
                    {{ printf "$%.2f" $context.Cart.GetTotal }}
                </td>
            </tr>
        </tfoot>
    </table>
    <div class="text-center">
         <a class="btn btn-secondary" href="{{ $context.ProductListUrl
}}">
            Continue shopping
        </a>
    </div>
</div>
```

Листинг 36-4 Содержимое файла cart.html в папке templates

Этот шаблон создает таблицу HTML со строками для каждого из продуктов, выбранных пользователем. Также есть кнопка, которая возвращает пользователя к списку продуктов, чтобы можно было сделать дальнейший выбор. Чтобы создать макет, используемый для этого шаблона, добавьте файл с именем `simple_layout.html` в папку `templates` с содержимым, показанным в листинге 36-5.

```
<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>SportsStore</title>
    <link href="/files/bootstrap.min.css" rel="stylesheet" />
</head>
<body>
    <div class="bg-dark text-white p-2">
        <div class="container-fluid">
            <div class="row">
                <div class="col navbar-brand">SPORTS STORE</div>
            </div>
        </div>
    </div>
    {{ body }}
</body>
</html>
```

Листинг 36-5 Содержимое файла simple_layout.html в папке templates

Этот макет отображает заголовок SportsStore, но не применяет макет столбца, который используется для списка продуктов..

Добавление товаров в корзину

Каждый продукт будет отображаться с кнопкой **Add To Cart**, которая отправит запрос методу `PostAddToCart`, созданному в листинге 36-3. Сначала добавьте элементы, показанные в листинге 36-6, которые определяют кнопку и форму, которую она отправляет.

```
{{ layout "store_layout.html" }}

{{ define "left_column" }}
    {{ $context := . }}
    {{ handler "category" "getbuttons" $context.SelectedCategory}}
{{end}}

{{ define "right_column" }}
    {{ $context := . }}
    {{ range $context.Products }}
        <div class="card card-outline-primary m-1 p-1">
```

```
            <div class="bg-faded p-1">
                <h4>
                    {{ .Name }}
                           <span class="badge rounded-pill bg-primary"
style="float:right">
                        <small>{{ printf "$%.2f" .Price }}</small>
                    </span>
                </h4>
            </div>
            <div class="card-text p-1">
                  <form method="POST" action="{{ $context.AddToCartUrl
}}">
                    {{ .Description }}
                    <input type="hidden" name="id" value="{{.ID}}" />
                     <button type="submit"class="btn btn-success btn-sm
pull-right"
                        style="float:right">
                            Add To Cart
                    </button>
                </form>
            </div>
        </div>
    {{ end }}
    {{ template "page_buttons.html" $context }}
{{end}}
```

Листинг 36-6 Добавление формы в файл product_list.html в папке templates

Чтобы предоставить шаблону URL-адрес, который используется в форме, внесите изменения, показанные в листинге 36-7, в его обработчик.

```
package store

import (
    "sportsstore/models"
    "platform/http/actionresults"
    "platform/http/handling"
    "math"
)

const pageSize = 4

type ProductHandler struct {
    Repository models.Repository
    URLGenerator handling.URLGenerator
}

type ProductTemplateContext struct {
    Products []models.Product
    Page int
```

```
    PageCount int
    PageNumbers []int
    PageUrlFunc func(int) string
    SelectedCategory int
    AddToCartUrl string
}

func (handler ProductHandler) GetProducts(category,
        page int) actionresults.ActionResult {
    prods, total := handler.Repository.GetProductPageCategory(category,
        page, pageSize)
    pageCount := int(math.Ceil(float64(total) / float64(pageSize)))
    return actionresults.NewTemplateAction("product_list.html",
        ProductTemplateContext {
            Products: prods,
            Page: page,
            PageCount: pageCount,
            PageNumbers: handler.generatePageNumbers(pageCount),
            PageUrlFunc: handler.createPageUrlFunction(category),
            SelectedCategory: category,
            AddToCartUrl: mustGenerateUrl(handler.URLGenerator,
                CartHandler.PostAddToCart),
        })
}

func (handler ProductHandler) createPageUrlFunction(category int)
func(int) string {
    return func(page int) string {
        url, _ :=
handler.URLGenerator.GenerateUrl(ProductHandler.GetProducts,
            category, page)
        return url
    }
}

func (handler ProductHandler) generatePageNumbers(pageCount int) (pages
[]int) {
    pages = make([]int, pageCount)
    for i := 0; i < pageCount; i++ {
        pages[i] = i + 1
    }
    return
}

func mustGenerateUrl(generator handling.URLGenerator, target
interface{}) string {
    url, err := generator.GenerateUrl(target)
    if (err != nil) {
        panic(err)
    }
```

```
    return url;
}
```

Листинг 36-7 Добавление данных контекста в файл product_handler.go в папке store

Изменения добавляют новое свойство в структуру контекста, используемую для передачи данных в шаблон, что позволяет обработчику предоставлять URL-адрес, который можно использовать в форме HTML.

Настройка приложения

Последним шагом для запуска базовой функции корзины является настройка служб, промежуточного программного обеспечения и обработчика, необходимых для сеансов и корзины, как показано в листинге 36-8.

```
package main

import (
    "sync"
    "platform/http"
    "platform/http/handling"
    "platform/services"
    "platform/pipeline"
    "platform/pipeline/basic"
    "sportsstore/store"
    "sportsstore/models/repo"
    "platform/sessions"
    "sportsstore/store/cart"
)

func registerServices() {
    services.RegisterDefaultServices()
    repo.RegisterMemoryRepoService()
    sessions.RegisterSessionService()
    cart.RegisterCartService()
}

func createPipeline() pipeline.RequestPipeline {
    return pipeline.CreatePipeline(
        &basic.ServicesComponent{},
        &basic.LoggingComponent{},
        &basic.ErrorComponent{},
        &basic.StaticFileComponent{},
        &sessions.SessionComponent{},
        handling.NewRouter(
            handling.HandlerEntry{ "",  store.ProductHandler{}},
            handling.HandlerEntry{ "",  store.CategoryHandler{}},
            handling.HandlerEntry{ "", store.CartHandler{}},
             ).AddMethodAlias("/", store.ProductHandler.GetProducts, 0,
1).
```

```
            AddMethodAlias("/products[/]?[A-z0-9]*?",
                store.ProductHandler.GetProducts, 0, 1),
    )
}

func main() {
    registerServices()
    results, err := services.Call(http.Serve, createPipeline())
    if (err == nil) {
        (results[0].(*sync.WaitGroup)).Wait()
    } else {
        panic(err)
    }
}
```

Листинг 36-8 Настройка приложения для корзины в файле main.go в папке sportsstore

Скомпилируйте и запустите проект и используйте браузер для запроса `http://localhost:5000`. Продукты показаны с помощью кнопки **Add To Cart**, при нажатии которой продукт добавляется в корзину и перенаправляет браузер для отображения содержимого корзины, как показано на рисунке 36-1.

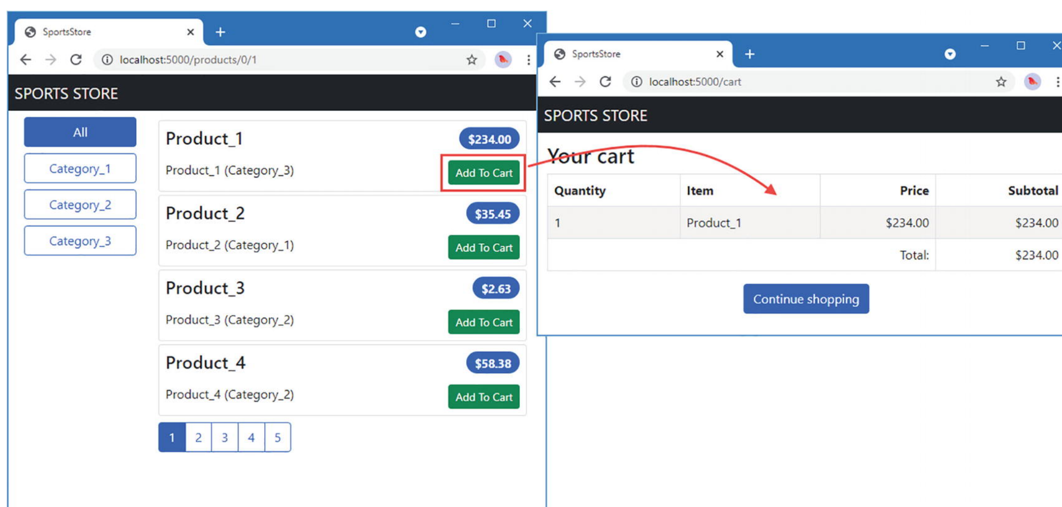

Рисунок 36-1 Создание корзины магазина

Добавление виджета «Сводка корзины»

Пользователи ожидают увидеть сводную информацию о выбранных ими продуктах при просмотре списка доступных продуктов. Добавьте метод, показанный в листинге 36-9, в обработчик запроса `CartHandler`.

```
package store

import (
    "platform/http/actionresults"
    "platform/http/handling"
```

```
    "sportsstore/models"
    "sportsstore/store/cart"
)

type CartHandler struct {
    models.Repository
    cart.Cart
    handling.URLGenerator
}

type CartTemplateContext struct {
    cart.Cart
    ProductListUrl string
    CartUrl string
}

func (handler CartHandler) GetCart() actionresults.ActionResult {
                 return    actionresults.NewTemplateAction("cart.html",
CartTemplateContext {
        Cart: handler.Cart,
                                                          ProductListUrl:
handler.mustGenerateUrl(ProductHandler.GetProducts, 0, 1),
    })
}

func (handler CartHandler) GetWidget() actionresults.ActionResult {
           return   actionresults.NewTemplateAction("cart_widget.html",
CartTemplateContext {
        Cart: handler.Cart,
        CartUrl: handler.mustGenerateUrl(CartHandler.GetCart),
    })
}

// ...statements omitted for brevity...
```

Листинг 36-9 Добавление метода в файл cart_handler.go в папке store

Чтобы определить шаблон, используемый новым методом, добавьте файл с именем `cart_widget.html` в папку шаблонов с содержимым, показанным в листинге 36-10.

```
{{ $context := . }}
{{ $count := $context.Cart.GetItemCount }}
    <small class="navbar-text">
        {{ if gt $count 0 }}
            <b>Your cart:</b>
            {{ $count }} item(s)
            {{ printf "$%.2f" $context.Cart.GetTotal }}
        {{ else }}
            <span class="px-2 text-secondary">(empty cart)</span>
```

```
        {{ end }}
    </small>
<a href={{ $context.CartUrl }}
        class="btn btn-sm btn-secondary navbar-btn">
    <i class="fa fa-shopping-cart"></i>
</a>
```

Листинг 36-10 Содержимое файла cart_widget.html в папке templates

Вызов обработчика и добавление таблицы стилей значков CSS

В листинге 36-10 вызывается метод `GetWidget` для вставки виджета корзины в макет. Для шаблона виджета корзины требуется значок корзины покупок, который предоставляется отличным пакетом Font Awesome. В главе 35 я скопировал CSS-файл Bootstrap, чтобы его можно было обслуживать, используя функции статических файлов, предоставляемые веб-платформой, но для пакета Font Awesome требуется несколько файлов, поэтому в листинге 36-11 добавлен элемент ссылки с URL-адресом. для сети распространения контента. (Это означает, что вы должны быть в сети, чтобы увидеть значки. См. `https://fontawesome.com` для получения подробной информации о том, как загрузить файлы, которые можно установить в папку `sportsstore/files`.)

```
<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>SportsStore</title>
    <link href="/files/bootstrap.min.css" rel="stylesheet" />
    <link rel="stylesheet"
href="https://cdnjs.cloudflare.com/ajax/libs/font-
awesome/5.15.4/css/all.min.css"  />
</head>
<body>
    <div class="bg-dark text-white p-2">
        <div class="container-fluid">
            <div class="row">
                <div class="col navbar-brand">SPORTS STORE</div>
                <div class="col-6 navbar-text text-end">
                    {{ handler "cart" "getwidget" }}
                </div>
            </div>
        </div>
    </div>
    <div class="row m-1 p-1">
        <div id="sidebar" class="col-3">
            {{ template "left_column" . }}
        </div>
        <div class="col-9">
            {{ template "right_column" . }}
        </div>
```

```
    </div>
</body>
</html>
```

Листинг 36-11 Добавление ссылки на таблицу стилей в файл store_layout.html в папке templates

Скомпилируйте и запустите проект, и вы увидите виджет, отображаемый в заголовке страницы. Виджет укажет, что корзина пуста. Нажмите одну из кнопок **Add To Cart**, а затем нажмите кнопку **Continue Shopping**, чтобы увидеть результат выбора продукта, показанный на рисунке 36-2.

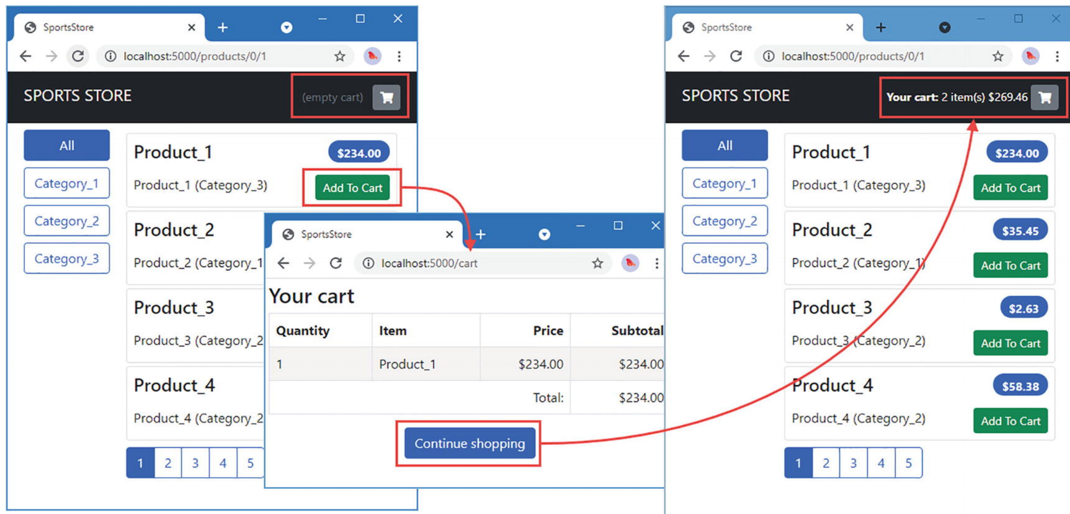

Рисунок 36-2 Отображение виджета корзины

Использование репозитория базы данных

Большинство основных функций реализовано, и пришло время отказаться от временного репозитория, который я создал в главе 35, и заменить его тем, который использует постоянную базу данных. Я собираюсь использовать SQLite. Используйте командную строку для запуска команды, показанной в листинге 36-12, в папке `sportsstore`, чтобы загрузить и установить драйвер SQLite, который также включает среду выполнения SQLite.

```
go get modernc.org/sqlite
```

Листинг 36-12 Установка драйвера SQLite и пакета базы данных

Создание типов репозиториев

Добавьте файл с именем `sql_repo.go` в папку `models/repo` с содержимым, показанным в листинге 36-13, в котором определяются основные типы репозитория SQL.

```
package repo
```

```
import (
    "database/sql"
    "platform/config"
    "platform/logging"
    "context"
)

type SqlRepository struct {
    config.Configuration
    logging.Logger
    Commands SqlCommands
    *sql.DB
    context.Context
}

type SqlCommands struct {
    Init,
    Seed,
    GetProduct,
    GetProducts,
    GetCategories,
    GetPage,
    GetPageCount,
    GetCategoryPage,
    GetCategoryPageCount *sql.Stmt
}
```

Листинг 36-13 Содержимое файла sql_repo.go в папке models/repo

Структура SqlRepository будет использоваться для реализации интерфейса Repository и будет предоставляться остальной части приложения в качестве службы. Эта структура определяет поле *sql.DB, обеспечивающее доступ к базе данных, и поле Commands, представляющее собой набор полей *sql.Stmt, которые будут заполнены подготовленными операторами, необходимыми для реализации функций интерфейса Repository.

Открытие базы данных и загрузка команд SQL

В главе 26 я определил команды SQL как строки Go. В реальных проектах я предпочитаю определять команды SQL в текстовых файлах с расширением .sql, что означает, что мой редактор может выполнять проверку синтаксиса. Это означает, что мне нужно открыть базу данных, а затем найти и обработать файлы SQL, соответствующие полям, определенным структурой SqlCommands, определенной в листинге 36-13. Добавьте файл с именем sql_loader.go в папку models/repo с содержимым, показанным в листинге 36-14.

```
package repo

import (
```

```
    "os"
    "database/sql"
    "reflect"
    "platform/config"
    "platform/logging"
    _ "modernc.org/sqlite"
)

func  openDB(config  config.Configuration,  logger  logging.Logger)  (db
*sql.DB,
        commands *SqlCommands, needInit bool) {
    driver := config.GetStringDefault("sql:driver_name", "sqlite")
    connectionStr, found := config.GetString("sql:connection_str")
    if !found {
        logger.Panic("Cannot read SQL connection string from config")
        return
    }
    if _, err := os.Stat(connectionStr); os.IsNotExist(err) {
        needInit = true
    }
    var err error
    if db, err = sql.Open(driver, connectionStr); err == nil {
        commands = loadCommands(db, config, logger)
    } else {
        logger.Panic(err.Error())
    }
    return
}

func loadCommands(db *sql.DB, config config.Configuration,
        logger logging.Logger) (commands *SqlCommands)  {
    commands = &SqlCommands {}
    commandVal := reflect.ValueOf(commands).Elem()
    commandType := reflect.TypeOf(commands).Elem()
    for i := 0; i < commandType.NumField(); i++ {
        commandName := commandType.Field(i).Name
        logger.Debugf("Loading SQL command: %v", commandName)
        stmt := prepareCommand(db, commandName, config, logger)
        commandVal.Field(i).Set(reflect.ValueOf(stmt))
    }
    return commands
}

func     prepareCommand(db     *sql.DB,     command     string,     config
config.Configuration,
        logger logging.Logger) *sql.Stmt {
    filename, found := config.GetString("sql:commands:" + command)
    if !found {
             logger.Panicf("Config  does  not  contain  location  for  SQL
command: %v",
```

```
            command)
    }
    data, err := os.ReadFile(filename)
    if err != nil {
        logger.Panicf("Cannot read SQL command file: %v", filename)
    }
    statement, err := db.Prepare(string(data))
    if (err != nil) {
        logger.Panicf(err.Error())
    }
    return statement
}
```

Листинг 36-14 Содержимое файла sql_loader.go в папке models/repo

Функция openDB считывает имя драйвера базы данных и строку подключения из системы конфигурации и открывает базу данных перед вызовом функции loadCommands. Функция loadCommands использует рефлексию для получения списка полей, определенных структурой SqlCommands, и вызывает команду prepareCommand для каждого из них. Функция prepareCommand получает имя файла, содержащего SQL для команды из системы конфигурации, считывает содержимое файла и создает подготовленный оператор, который присваивается полю SqlCommands.

Определение начального числа и операторов инициализации

Для каждой функции, требуемой интерфейсом Repository, мне нужно определить файл SQL, содержащий запрос, и определить метод Go, который будет его выполнять. Я собираюсь начать с команд Seed и Init. Команда Seed требуется для интерфейса репозитория, но функция Init специфична для структуры SqlRepository и будет использоваться для создания схемы базы данных. Добавьте файл с именем sql_initseed.go в папку models/repo с содержимым, показанным в листинге 36-15.

Обратите внимание, что все запросы, используемые репозиторием, используют методы, принимающие аргумент context.Context (ExecContext, QueryContext и т. д.). Платформа, созданная в главах 32–34, передает значения Context компонентам промежуточного программного обеспечения и обработчикам запросов, поэтому я использовал их при выполнении запросов к базе данных.

```
package repo

func (repo *SqlRepository) Init() {
     if _, err := repo.Commands.Init.ExecContext(repo.Context); err !=
nil {
        repo.Logger.Panic("Cannot exec init command")
    }
}
```

```
func (repo *SqlRepository) Seed() {
     if _, err := repo.Commands.Seed.ExecContext(repo.Context); err !=
nil {
        repo.Logger.Panic("Cannot exec seed command")
    }
}
```

Листинг 36-15 Содержимое файла sql_initseed.go в папке models/repo

Чтобы создать SQL-команды, используемые этими методами, создайте папку `sportsstore/sql` и добавьте в нее файл с именем `init_db.sql` с содержимым, показанным в листинге 36-16.

```
DROP TABLE IF EXISTS Products;
DROP TABLE IF EXISTS Categories;

CREATE TABLE IF NOT EXISTS Categories (
    Id INTEGER NOT NULL PRIMARY KEY,        Name TEXT
);

CREATE TABLE IF NOT EXISTS Products (
    Id INTEGER NOT NULL PRIMARY KEY,
    Name TEXT, Description TEXT,
    Category INTEGER, Price decimal(8, 2),
    CONSTRAINT CatRef FOREIGN KEY(Category) REFERENCES Categories (Id)
);
```

Листинг 36-16 Содержимое файла init_db.sql в папке sql

Этот файл содержит операторы, которые удаляют и воссоздают таблицы `Categories` и `Products`. Добавьте файл `seed_db.sql` в папку `sportsstore/sql` с содержимым, показанным в листинге 36-17.

```
INSERT INTO Categories(Id, Name) VALUES
        (1, "Watersports"), (2, "Soccer"), (3, "Chess");

INSERT INTO Products(Id, Name, Description, Category, Price) VALUES
        (1, "Kayak", "A boat for one person", 1, 275),
        (2, "Lifejacket", "Protective and fashionable", 1, 48.95),
        (3, "Soccer Ball", "FIFA-approved size and weight", 2, 19.50),
          (4, "Corner Flags", "Give your playing field a professional
touch", 2, 34.95),
        (5, "Stadium", "Flat-packed 35,000-seat stadium", 2, 79500),
        (6, "Thinking Cap", "Improve brain efficiency by 75%", 3, 16),
              (7, "Unsteady Chair", "Secretly give your opponent a
disadvantage", 3, 29.95),
        (8, "Human Chess Board", "A fun game for the family", 3, 75),
        (9, "Bling-Bling King", "Gold-plated, diamond-studded King", 3,
1200);
```

Листинг 36-17 Содержимое файла seed_db.sql в папке sql

Файл содержит операторы INSERT, которые создают три категории и девять продуктов, используя значения, знакомые всем, кто читал другие мои книги.

Определение основных запросов

Чтобы завершить репозиторий, мне нужно проработать методы, требуемые интерфейсом Repository, определить реализацию этого метода на Go и SQL-запрос, который он будет использовать. Добавьте файл с именем sql_basic_methods.go в папку models/repo с содержимым, показанным в листинге 36-18.

```
package repo

import "sportsstore/models"

func (repo *SqlRepository) GetProduct(id int) (p models.Product) {
    row := repo.Commands.GetProduct.QueryRowContext(repo.Context, id)
    if row.Err() == nil {
        var err error
        if p, err = scanProduct(row); err != nil {
            repo.Logger.Panicf("Cannot scan data: %v", err.Error())
        }
    } else {
         repo.Logger.Panicf("Cannot exec GetProduct command: %v",
row.Err().Error())
    }
    return
}

func (repo *SqlRepository) GetProducts() (results []models.Product) {
    rows, err := repo.Commands.GetProducts.QueryContext(repo.Context)
    if err == nil {
        if results, err = scanProducts(rows); err != nil {
            repo.Logger.Panicf("Cannot scan data: %v", err.Error())
            return
        }
    } else {
        repo.Logger.Panicf("Cannot exec GetProducts command: %v", err)
    }
    return
}

func (repo *SqlRepository) GetCategories() []models.Category {
    results := make([]models.Category, 0, 10)
    rows, err := repo.Commands.GetCategories.QueryContext(repo.Context)
    if err == nil {
        for rows.Next() {
            c := models.Category{}
            if err := rows.Scan(&c.ID, &c.CategoryName); err != nil {
```

```
                repo.Logger.Panicf("Cannot scan data: %v", err.Error())
            }
            results = append(results, c)
        }
    } else {
           repo.Logger.Panicf("Cannot exec GetCategories command: %v",
err)
    }
    return results
}
```

Листинг 36-18 Содержимое файла sql_basic_methods.go в папке models/repo

В листинге 36-18 реализованы методы `GetProduct`, `GetProducts` и `GetCategories`. Чтобы определить функции, которые сканируют значения `Product` из результатов SQL, добавьте файл с именем `sql_scan.go` в папку `models/repo` с содержимым, показанным в листинге 36-19.

```
package repo

import (
    "database/sql"
    "sportsstore/models"
)

func scanProducts(rows *sql.Rows) (products []models.Product, err
error) {
    products = make([]models.Product, 0, 10)
    for rows.Next() {
        p := models.Product{ Category: &models.Category{}}
        err = rows.Scan(&p.ID, &p.Name, &p.Description, &p.Price,
            &p.Category.ID, &p.Category.CategoryName)
        if (err == nil) {
            products = append(products, p)
        } else {
            return
        }
    }
    return
}

func scanProduct(row *sql.Row) (p models.Product, err error) {
    p = models.Product{ Category: &models.Category{}}
         err = row.Scan(&p.ID, &p.Name, &p.Description, &p.Price,
&p.Category.ID,
        &p.Category.CategoryName)
    return p, err
}
```

Листинг 36-19 Содержимое файла sql_scan.go в папке models/repo

Функция scanProducts сканирует значения при наличии нескольких строк, а функция scanProduct делает то же самое для результатов с одной строкой.

Определение файлов SQL для базовых запросов

Теперь идет процесс определения файлов SQL для каждого запроса. Добавьте файл с именем get_product.sql в папку sportsstore/sql с содержимым, показанным в листинге 36-20.

```
SELECT      Products.Id,      Products.Name,      Products.Description,
Products.Price,
    Categories.Id, Categories.Name
FROM Products, Categories
WHERE Products.Category = Categories.Id
AND Products.Id = ?
```

Листинг 36-20 Содержимое файла get_product.sql в папке sql

Этот запрос создает одну строку, содержащую сведения о продукте с указанным Id. Добавьте файл с именем get_products.sql в папку sportsstore/sql с содержимым, показанным в листинге 36-21.

```
SELECT      Products.Id,      Products.Name,      Products.Description,
Products.Price,
    Categories.Id, Categories.Name
FROM Products, Categories
WHERE Products.Category = Categories.Id
ORDER BY Products.Id
```

Листинг 36-21 Содержимое файла get_products.sql в папке sql

Этот запрос создает строки для всех продуктов в базе данных. Затем добавьте файл с именем get_categories.sql в папку sportsstore/sql с содержимым, показанным в листинге 36-22.

```
SELECT Categories.Id, Categories.Name
FROM Categories ORDER BY Categories.Id
```

Листинг 36-22 Содержимое файла get_categories.sql в папке sql

Этот запрос выбирает все строки в папке Categories.

Определение постраничных запросов

Методы для постраничных данных более сложны, поскольку они должны выполнять один запрос для страницы данных и один запрос для получения общего количества доступных результатов. Добавьте файл с именем sql_page_methods.go в папку sportsstore/models/repo с содержимым, показанным в листинге 36-23.

```
package repo
```

```
import "sportsstore/models"

func (repo *SqlRepository) GetProductPage(page,
        pageSize int) (products []models.Product, totalAvailable int) {
    rows, err := repo.Commands.GetPage.QueryContext(repo.Context,
        pageSize, (pageSize * page) - pageSize)
    if err == nil {
        if products, err = scanProducts(rows); err != nil {
            repo.Logger.Panicf("Cannot scan data: %v", err.Error())
            return
        }
    } else {
          repo.Logger.Panicf("Cannot exec GetProductPage command: %v",
err)
        return
    }
    row := repo.Commands.GetPageCount.QueryRowContext(repo.Context)
    if row.Err() == nil {
        if err := row.Scan(&totalAvailable); err != nil {
            repo.Logger.Panicf("Cannot scan data: %v", err.Error())
        }
    } else {
           repo.Logger.Panicf("Cannot exec GetPageCount command: %v",
row.Err().Error())
    }
    return
}

func (repo *SqlRepository) GetProductPageCategory(categoryId int, page,
        pageSize int) (products []models.Product, totalAvailable int) {
    if (categoryId == 0) {
        return repo.GetProductPage(page, pageSize)
    }
                                         rows,          err           :=
repo.Commands.GetCategoryPage.QueryContext(repo.Context, categoryId,
        pageSize, (pageSize * page) - pageSize)
    if err == nil {
        if products, err = scanProducts(rows); err != nil {
            repo.Logger.Panicf("Cannot scan data: %v", err.Error())
            return
        }
    } else {
          repo.Logger.Panicf("Cannot exec GetProductPage command: %v",
err)
        return
    }
                                                     row              :=
repo.Commands.GetCategoryPageCount.QueryRowContext(repo.Context,
        categoryId)
    if row.Err() == nil {
```

```
        if err := row.Scan(&totalAvailable); err != nil {
            repo.Logger.Panicf("Cannot scan data: %v", err.Error())
        }
    } else {
          repo.Logger.Panicf("Cannot exec GetCategoryPageCount command:
%v",
            row.Err().Error())
    }
    return
}
```

Листинг 36-23 Содержимое файла sql_page_methods.go в папке models/repo

Чтобы определить основной SQL-запрос, используемый методом `GetProductPage`, добавьте файл с именем `get_product_page.sql` в папку `sportsstore/sql` с содержимым, показанным в листинге 36-24.

```
SELECT      Products.Id,      Products.Name,      Products.Description,
Products.Price,
    Categories.Id, Categories.Name
FROM Products, Categories
WHERE Products.Category = Categories.Id
ORDER BY Products.Id
LIMIT ? OFFSET ?
```

Листинг 36-24 Содержимое файла get_product_page.sql в папке sql

Чтобы определить запрос, используемый для получения общего количества продуктов в базе данных, добавьте файл с именем `get_page_count.sql` в папку `sportsstore/sql` с содержимым, показанным в листинге 36-25.

```
SELECT COUNT (Products.Id)
FROM Products, Categories
WHERE Products.Category = Categories.Id;
```

Листинг 36-25 Содержимое файла get_page_count.sql в папке sql

Чтобы определить основной запрос, используемый методом `GetProductPageCategory`, добавьте файл с именем `get_category_product_page.sql` в папку `sportsstore/sql` с содержимым, показанным в листинге 36-26.

```
SELECT      Products.Id,      Products.Name,      Products.Description,
Products.Price,
    Categories.Id, Categories.Name
FROM Products, Categories
WHERE Products.Category = Categories.Id AND       Products.Category =
?
ORDER BY Products.Id
LIMIT ? OFFSET ?
```

Листинг 36-26 Содержимое файла get_category_product_page.sql в папке sql

Чтобы определить запрос, определяющий количество товаров в определенной категории, добавьте файл с именем `get_category_product_page_count.sql` в папку `sportsstore/sql` с содержимым, показанным в листинге 36-27.

```
SELECT COUNT (Products.Id)
FROM Products, Categories
WHERE Products.Category = Categories.Id AND Products.Category = ?
```

Листинг 36-27 Содержимое файла get_category_product_page_count.sql в папке sql

Определение службы репозитория SQL

Чтобы определить функцию, которая будет регистрировать службу репозитория, добавьте файл с именем `sql_service.go` в папку `sportssstore/models/repo` с содержимым, показанным в листинге 36-28.

```
package repo

import (
    "sync"
    "context"
    "database/sql"
    "platform/services"
    "platform/config"
    "platform/logging"
    "sportsstore/models"
)

func RegisterSqlRepositoryService() {
    var db *sql.DB
    var commands *SqlCommands
    var needInit bool
    loadOnce := sync.Once {}
    resetOnce := sync.Once {}
             services.AddScoped(func   (ctx   context.Context,   config
config.Configuration,
            logger logging.Logger) models.Repository {
        loadOnce.Do(func () {
            db, commands, needInit = openDB(config, logger)
        })
        repo := &SqlRepository{
            Configuration: config,
            Logger: logger,
            Commands: *commands,
            DB: db,
            Context: ctx,
        }
        resetOnce.Do(func() {
```

```
            if needInit || config.GetBoolDefault("sql:always_reset",
true) {
                repo.Init()
                repo.Seed()
            }
        })
        return repo
    })
}
```

Листинг 36-28 Содержимое файла sql_service.go в папке models/repo

База данных открывается при первом разрешении зависимости от интерфейса `Repository`, поэтому команды подготавливаются только один раз. Параметр конфигурации указывает, следует ли сбрасывать базу данных каждый раз при запуске приложения, что полезно во время разработки, и это делается путем выполнения метода `Init`, за которым следует метод `Seed`.

Настройка приложения для использования репозитория SQL

В листинге 36-29 определены параметры конфигурации, которые указывают расположение файлов SQL. Код, который загружает эти файлы, будет паниковать, если эти файлы не могут быть загружены, поэтому важно убедиться, что указанные пути совпадают с теми, которые использовались для создания файлов.

```
{
    "logging" : {
        "level": "debug"
    },
    "files": {
        "path": "files"
    },
    "templates": {
        "path": "templates/*.html",
        "reload": true
    },
    "sessions": {
        "key": "MY_SESSION_KEY",
        "cyclekey": true
    },
    "sql": {
        "connection_str": "store.db",
        "always_reset": true,
        "commands": {
            "Init":                 "sql/init_db.sql",
            "Seed":                 "sql/seed_db.sql",
            "GetProduct":           "sql/get_product.sql",
            "GetProducts":          "sql/get_products.sql",
            "GetCategories":        "sql/get_categories.sql",
            "GetPage":              "sql/get_product_page.sql",
```

```
            "GetPageCount":        "sql/get_page_count.sql",
            "GetCategoryPage":     "sql/get_category_product_page.sql",
                                         "GetCategoryPageCount":
"sql/get_category_product_page_count.sql"
        }
    }
}
```

Листинг 36-29 Определение параметров конфигурации в файле config.json в папке sportsstore

Последним изменением является регистрация репозитория SQL, чтобы он использовался для разрешения зависимостей в интерфейсе репозитория, и закомментирование оператора, регистрирующего временный репозиторий, как показано в листинге 36-30.

```
...
func registerServices() {
    services.RegisterDefaultServices()
    //repo.RegisterMemoryRepoService()
    repo.RegisterSqlRepositoryService()
    sessions.RegisterSessionService()
    cart.RegisterCartService()
}
...
```

Листинг 36-30 Изменение службы репозитория в файле main.go в папке sportsstore

Скомпилируйте и выполните проект и используйте браузер для запроса `http://localhost:5000`, и вы увидите данные, которые считываются из базы данных, как показано на рисунке 36-3.

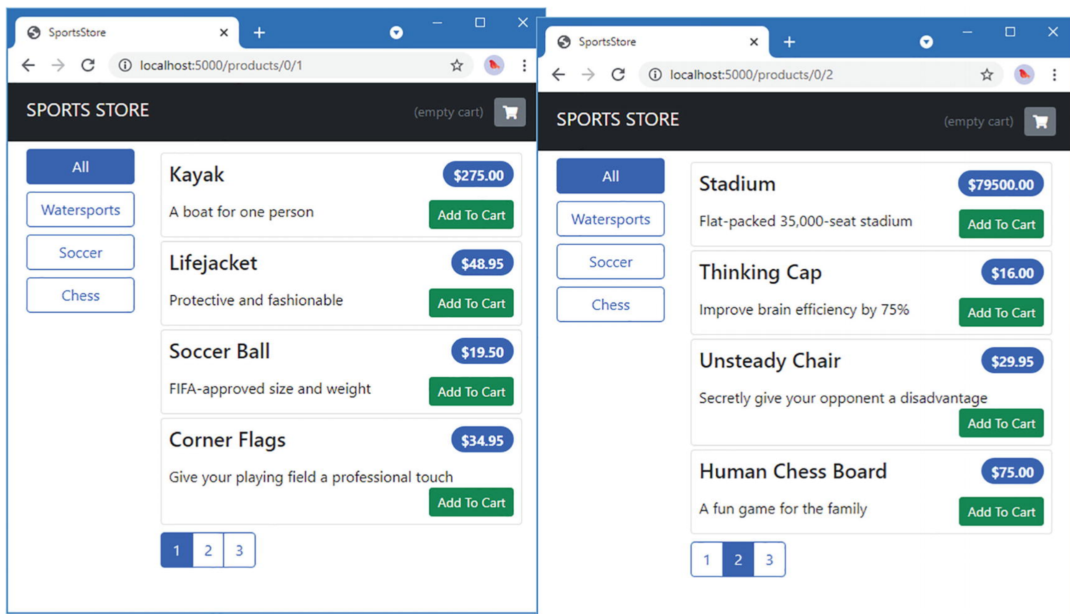

Рисунок 36-3 Использование данных из базы данных

Резюме

В этой главе я продолжил разработку приложения SportsStore, добавив поддержку корзины покупок и заменив временный репозиторий тем, который использует базу данных SQL. В следующей главе я продолжу разработку приложения SportsStore.

37. SportsStore: оформление заказа и администрирование

В этой главе я продолжаю разработку приложения SportsStore, добавляя процесс оформления заказа и приступая к работе над функциями администрирования.

Подсказка
Вы можете загрузить пример проекта для этой главы — и для всех остальных глав этой книги — с `https://github.com/apress/pro-go`. См. Главу 2 о том, как получить помощь, если у вас возникнут проблемы с запуском примеров.

Создание процесса оформления заказа

Чтобы завершить работу с магазином, мне нужно позволить пользователю проверить и выполнить заказ. В этом разделе я расширим модель данных, чтобы описать сведения о доставке и создам обработчики для сбора этих сведений и использования их для хранения заказа в базе данных. Конечно, большинство сайтов электронной коммерции не остановились бы на этом, и я не предоставлял поддержки для обработки кредитных карт или других форм оплаты. Но я хочу, чтобы все было сосредоточено на Go, поэтому будет достаточно простой записи в базе данных.

Определение модели

Чтобы определить тип, который будет представлять сведения о доставке пользователя и выбранные продукты, добавьте файл с именем `order.go` в папку `models` с содержимым, показанным в листинге 37-1.

```
package models

type Order struct {
    ID int
    ShippingDetails
    Products []ProductSelection
    Shipped bool
}
```

```
type ShippingDetails struct {
    Name string `validation:"required"`
    StreetAddr string `validation:"required"`
    City string `validation:"required"`
    State string `validation:"required"`
    Zip string `validation:"required"`
    Country string `validation:"required"`
}

type ProductSelection struct{
    Quantity int
    Product
}
```

Листинг 37-1 Содержимое файла order.go в папке models

Тип `Order` определяет поле `ShippingDetails`, которое будет использоваться для представления сведений о доставке клиента и которое было определено с помощью тегов структуры для функции проверки платформы. Существует также поле `Products`, которое будет использоваться для хранения продуктов и количества, заказанного клиентом.

Расширение репозитория

Следующим шагом является расширение репозитория, чтобы его можно было использовать для хранения и извлечения заказов. Добавьте методы, показанные в листинге 37-2, в файл репозиторий.go в папке `sportsstore/models`.

```
package models

type Repository interface {

    GetProduct(id int) Product

    GetProducts() []Product

        GetProductPage(page, pageSize int) (products []Product,
totalAvailable int)

       GetProductPageCategory(categoryId int, page, pageSize int)
(products []Product,
        totalAvailable int)

    GetCategories() []Category

    GetOrder(id int) Order
    GetOrders() []Order
    SaveOrder(*Order)
```

```
    Seed()
}
```

Листинг 37-2 Добавление методов интерфейса в файл репозитория.go в папке models

В листинге 37-3 показаны изменения, необходимые для файла SQL, которые создают новые таблицы для хранения данных заказа.

```
DROP TABLE IF EXISTS OrderLines;
DROP TABLE IF EXISTS Orders;
DROP TABLE IF EXISTS Products;
DROP TABLE IF EXISTS Categories;

CREATE TABLE IF NOT EXISTS Categories (
    Id INTEGER NOT NULL PRIMARY KEY,       Name TEXT
);

CREATE TABLE IF NOT EXISTS Products (
    Id INTEGER NOT NULL PRIMARY KEY,
    Name TEXT, Description TEXT,
    Category INTEGER, Price decimal(8, 2),
     CONSTRAINT CatRef FOREIGN KEY(Category) REFERENCES Categories
(Id)
);

CREATE TABLE IF NOT EXISTS OrderLines (
    Id INTEGER NOT NULL PRIMARY KEY,
    OrderId INT, ProductId INT, Quantity INT,
     CONSTRAINT OrderRef FOREIGN KEY(ProductId) REFERENCES Products
(Id)
    CONSTRAINT OrderRef FOREIGN KEY(OrderId) REFERENCES Orders (Id)
);

CREATE TABLE IF NOT EXISTS Orders (
    Id INTEGER NOT NULL PRIMARY KEY,
    Name TEXT NOT NULL,
    StreetAddr TEXT NOT NULL,
    City TEXT NOT NULL,
    Zip TEXT NOT NULL,
    Country TEXT NOT NULL,
    Shipped BOOLEAN
);
```

Листинг 37-3 Добавление таблиц в файл init_db.sql в папку sql

Чтобы определить некоторые начальные данные, добавьте операторы, показанные в листинге 37-4, в файл `seed_db.sql` в папке `sportsstore/sql`.

```
INSERT INTO Categories(Id, Name) VALUES
        (1, "Watersports"), (2, "Soccer"), (3, "Chess");

INSERT INTO Products(Id, Name, Description, Category, Price) VALUES
        (1, "Kayak", "A boat for one person", 1, 275),
        (2, "Lifejacket", "Protective and fashionable", 1, 48.95),
        (3, "Soccer Ball", "FIFA-approved size and weight", 2, 19.50),
        (4, "Corner Flags", "Give your playing field a professional touch", 2, 34.95),
        (5, "Stadium", "Flat-packed 35,000-seat stadium", 2, 79500),
        (6, "Thinking Cap", "Improve brain efficiency by 75%", 3, 16),
        (7, "Unsteady Chair", "Secretly give your opponent a disadvantage", 3, 29.95),
        (8, "Human Chess Board", "A fun game for the family", 3, 75),
        (9, "Bling-Bling King", "Gold-plated, diamond-studded King", 3, 1200);

INSERT INTO Orders(Id, Name, StreetAddr, City, Zip, Country, Shipped) VALUES
        (1, "Alice", "123 Main St", "New Town", "12345", "USA", false),
        (2, "Bob", "The Grange", "Upton", "UP12 6YT", "UK", false);

INSERT INTO OrderLines(Id, OrderId, ProductId, Quantity) VALUES
        (1, 1, 1, 1), (2, 1, 2, 2), (3, 1, 8, 1), (4, 2, 5, 2);
```

Листинг 37-4 Добавление начальных данных в файл seed_db.sql в папке sql

Отключение временного репозитория

Временный репозиторий, созданный в главе 35, больше не определяет все методы, указанные в интерфейсе Repository. В реальном проекте я обычно переключаюсь обратно на репозиторий памяти при добавлении новой функции, например заказов, а затем снова переключаюсь на SQL, как только понимаю, что требуется. Но для этого проекта я просто закомментирую код, создающий сервис в памяти, как показано в листинге 37-5, чтобы он не вызывал ошибки компилятора.

```
package repo

import (
//    "platform/services"
    "sportsstore/models"
    "math"
)
```

```
// func RegisterMemoryRepoService() {
//     services.AddSingleton(func() models.Repository {
//         repo := &MemoryRepo{}
//         repo.Seed()
//         return repo
//     })
// }

type MemoryRepo struct {
    products []models.Product
    categories []models.Category
}

// ...other statements omitted for brevity...
```

Листинг 37-5 Комментирующий код в файле memory_repo.go в папке models/repo

Определение методов и команд репозитория

Следующим шагом является определение и реализация новых методов `Repository` и файлов SQL, на которые они будут опираться. В листинге 37-6 к структуре, используемой для загрузки файлов SQL для базы данных, добавлены новые команды.

```
package repo

import (
    "database/sql"
    "platform/config"
    "platform/logging"
    "context"
)

type SqlRepository struct {
    config.Configuration
    logging.Logger
    Commands SqlCommands
    *sql.DB
    context.Context
}

type SqlCommands struct {
    Init,
    Seed,
    GetProduct,
    GetProducts,
    GetCategories,
    GetPage,
```

```
    GetPageCount,
    GetCategoryPage,
    GetCategoryPageCount,
    GetOrder,
    GetOrderLines,
    GetOrders,
    GetOrdersLines,
    SaveOrder,
    SaveOrderLine *sql.Stmt
}
```

Листинг 37-6 Добавление команд в файл sql_repo.go в папке models/repo

Определение файлов SQL

Добавьте файл с именем `get_order.sql` в папку `sportsstore/sql` с содержимым, показанным в листинге 37-7.

```
SELECT Orders.Id, Orders.Name, Orders.StreetAddr, Orders.City,
Orders.Zip,
    Orders.Country, Orders.Shipped
FROM Orders
WHERE Orders.Id = ?
```

Листинг 37-7 Содержимое файла get_order.sql в папке sql

Этот запрос извлекает детали заказа. Чтобы определить запрос, который будет получать сведения о заказанных продуктах, добавьте файл с именем `get_order_lines.sql` в папку `sportsstore/sql` с содержимым, показанным в листинге 37-8.

```
SELECT OrderLines.Quantity, Products.Id, Products.Name,
Products.Description,
    Products.Price, Categories.Id, Categories.Name
FROM Orders, OrderLines, Products, Categories
WHERE Orders.Id = OrderLines.OrderId
    AND OrderLines.ProductId = Products.Id
    AND Products.Category = Categories.Id
    AND Orders.Id = ?
ORDER BY Products.Id
```

Листинг 37-8 Содержимое файла get_order_lines.sql в папке sql

Чтобы определить запрос, который будет получать все заказы в базе данных, добавьте файл с именем `get_orders.sql` в папку `sportsstore/sql` с содержимым, показанным в листинге 37-9.

```
SELECT Orders.Id, Orders.Name, Orders.StreetAddr, Orders.City,
Orders.Zip, Orders.Country, Orders.Shipped
```

```
FROM Orders
ORDER BY Orders.Shipped, Orders.Id
```

Листинг 37-9 Содержимое папки get_orders.sql в папке sql

Чтобы определить запрос, который будет получать все сведения о продуктах, связанных со всеми заказами, добавьте файл с именем `get_orders_lines.sql` в папку `sportsstore/sql` с содержимым, показанным в листинге 37-10.

```
SELECT Orders.Id, OrderLines.Quantity, Products.Id, Products.Name,
            Products.Description,   Products.Price,   Categories.Id,
Categories.Name
FROM Orders, OrderLines, Products, Categories
WHERE Orders.Id = OrderLines.OrderId
    AND OrderLines.ProductId = Products.Id
    AND Products.Category = Categories.Id
ORDER BY Orders.Id
```

Листинг 37-10 Содержимое файла get_orders_lines.sql в папке sql

Чтобы определить оператор, в котором будет храниться заказ, добавьте файл с именем `save_order.sql` в папку `sportsstore/sql` с содержимым, показанным в листинге 37-11.

```
INSERT INTO Orders(Name, StreetAddr, City, Zip, Country, Shipped)
VALUES (?, ?, ?, ?, ?, ?)
```

Листинг 37-11 Содержимое файла save_order.sql в папке sql

Чтобы определить оператор, в котором будут храниться сведения о выборе продукта, связанного с заказом, добавьте файл с именем `save_order_line.sql` в папку `sportsstore/sql` с содержимым, показанным в листинге 37-12.

```
INSERT INTO OrderLines(OrderId, ProductId, Quantity)
VALUES (?, ?, ?)
```

Листинг 37-12 Содержимое файла save_order_line.sql в папке sql

В листинге 37-13 добавлены параметры конфигурации для новых файлов SQL.

```
...
"sql": {
    "connection_str": "store.db",
    "always_reset": true,
    "commands": {
        "Init":                 "sql/init_db.sql",
        "Seed":                 "sql/seed_db.sql",
```

```
        "GetProduct":            "sql/get_product.sql",
        "GetProducts":           "sql/get_products.sql",
        "GetCategories":         "sql/get_categories.sql",
        "GetPage":               "sql/get_product_page.sql",
        "GetPageCount":          "sql/get_page_count.sql",
        "GetCategoryPage":       "sql/get_category_product_page.sql",
                                              "GetCategoryPageCount":
"sql/get_category_product_page_count.sql",
        "GetOrder": "sql/get_order.sql",
        "GetOrderLines": "sql/get_order_lines.sql",
        "GetOrders": "sql/get_orders.sql",
        "GetOrdersLines": "sql/get_orders_lines.sql",
        "SaveOrder": "sql/save_order.sql",
        "SaveOrderLine": "sql/save_order_line.sql"
    }
}
...
```

Листинг 37-13 Добавление настроек конфигурации в файл config.json в папке sportsstore

Реализация методов репозитория

Добавьте файл с именем `sql_orders_one.go` в папку `sportsstore/models/repo` с содержимым, показанным в листинге 37-14.

```
package repo

import "sportsstore/models"

func (repo *SqlRepository) GetOrder(id int) (order models.Order) {
    order = models.Order { Products: []models.ProductSelection {}}
    row := repo.Commands.GetOrder.QueryRowContext(repo.Context, id)
    if row.Err() == nil {
         err := row.Scan(&order.ID, &order.Name, &order.StreetAddr,
&order.City,
            &order.Zip, &order.Country, &order.Shipped)
        if (err != nil) {
                  repo.Logger.Panicf("Cannot scan order data: %v",
err.Error())
            return
        }
                                            lineRows,    err     :=
repo.Commands.GetOrderLines.QueryContext(repo.Context, id)
        if (err == nil) {
            for lineRows.Next() {
                ps := models.ProductSelection {
                                 Product: models.Product{ Category:
&models.Category{}},
                }
```

```
                    err = lineRows.Scan(&ps.Quantity, &ps.Product.ID,
&ps.Product.Name,
                     &ps.Product.Description,&ps.Product.Price,
                                                 &ps.Product.Category.ID,
&ps.Product.Category.CategoryName)
                if err == nil {
                    order.Products = append(order.Products, ps)
                } else {
                    repo.Logger.Panicf("Cannot scan order line data:
%v",
                        err.Error())
                }
            }
        } else {
            repo.Logger.Panicf("Cannot exec GetOrderLines command:
%v", err.Error())
        }
    } else {
            repo.Logger.Panicf("Cannot exec GetOrder command: %v",
row.Err().Error())
    }
    return
}
```

Листинг 37-14 Содержимое файла sql_orders_one.go в папке models/repo

Этот метод запрашивает в базе данных заказ, а затем снова запрашивает сведения о выборе продуктов, связанных с этим заказом. Затем добавьте файл с именем `sql_orders_all.go` в папку `sportsstore/models/repo` с содержимым, показанным в листинге 37-15.

```
package repo

import "sportsstore/models"

func (repo *SqlRepository) GetOrders() []models.Order {
    orderMap := make(map[int]*models.Order, 10)
                                    orderRows,        err          :=
repo.Commands.GetOrders.QueryContext(repo.Context)
    if err != nil {
            repo.Logger.Panicf("Cannot exec GetOrders command: %v",
err.Error())
    }
    for orderRows.Next() {
         order := models.Order { Products: []models.ProductSelection
{}}
                    err   :=   orderRows.Scan(&order.ID,   &order.Name,
&order.StreetAddr, &order.City,
```

```
            &order.Zip, &order.Country, &order.Shipped)
        if (err != nil) {
                repo.Logger.Panicf("Cannot scan order data: %v",
err.Error())
            return []models.Order {}
        }
        orderMap[order.ID] = &order
    }

                                    lineRows,       err         :=
repo.Commands.GetOrdersLines.QueryContext(repo.Context)
    if (err != nil) {
        repo.Logger.Panicf("Cannot exec GetOrdersLines command: %v",
err.Error())
    }
    for lineRows.Next() {
        var order_id int
        ps := models.ProductSelection {
            Product: models.Product{ Category: &models.Category{} },
        }
        err = lineRows.Scan(&order_id, &ps.Quantity, &ps.Product.ID,
                        &ps.Product.Name,  &ps.Product.Description,
&ps.Product.Price,
                                            &ps.Product.Category.ID,
&ps.Product.Category.CategoryName)
        if err == nil {
                                     orderMap[order_id].Products   =
append(orderMap[order_id].Products, ps)
        } else {
              repo.Logger.Panicf("Cannot scan order line data: %v",
err.Error())
        }
    }
    orders := make([]models.Order, 0, len(orderMap))
    for _, o := range orderMap {
        orders = append(orders, *o)
    }
    return orders
}
```

Листинг 37-15 Содержимое файла sql_orders_all.go в папке models/repo

Этот метод запрашивает базу данных для всех заказов и связанных с ними продуктов. Чтобы реализовать последний метод, добавьте файл с именем `sql_orders_save.go` в папку `sportsstore/models/repo` с содержимым, показанным в листинге 37-16.

```
package repo
```

```
import "sportsstore/models"

func (repo *SqlRepository) SaveOrder(order *models.Order) {
    tx, err := repo.DB.Begin()
    if err != nil {
                repo.Logger.Panicf("Cannot  create  transaction:  %v",
err.Error())
        return
    }
    result, err :=  tx.StmtContext(repo.Context,
         repo.Commands.SaveOrder).Exec(order.Name, order.StreetAddr,
order.City,
            order.Zip, order.Country, order.Shipped)
    if err != nil {
            repo.Logger.Panicf("Cannot  exec  SaveOrder  command: %v",
err.Error())
        tx.Rollback()
        return
    }
    id, err := result.LastInsertId()
    if err != nil {
                 repo.Logger.Panicf("Cannot  get  inserted  ID:  %v",
err.Error())
        tx.Rollback()
        return
    }
                   statement     :=     tx.StmtContext(repo.Context,
repo.Commands.SaveOrderLine)
    for _, sel := range order.Products {
        _, err := statement.Exec(id, sel.Product.ID, sel.Quantity)
        if err != nil {
              repo.Logger.Panicf("Cannot exec SaveOrderLine command:
%v", err.Error())
            tx.Rollback()
            return
        }
    }
    err = tx.Commit()
    if err != nil {
          repo.Logger.Panicf("Transaction cannot be committed: %v",
err.Error())
        err = tx.Rollback()
        if err != nil {
              repo.Logger.Panicf("Transaction cannot be rolled back:
%v", err.Error())
        }
    }
    order.ID = int(id)
```

```
}
```

Листинг 37-16 Содержимое файла sql_orders_save.go в папке models/repo

Этот метод использует транзакцию, чтобы обеспечить добавление нового заказа и связанных с ним продуктов в базу данных. Если транзакция не удалась, то изменения откатываются.

Создание обработчика запросов и шаблонов

Следующим шагом является определение обработчика запросов, который позволит пользователю предоставить информацию о доставке и оформить заказ. Как отмечалось в начале этой главы, сохранение заказа завершит процесс оформления заказа, хотя в реальных интернет-магазинах пользователю будет предложено произвести оплату. Добавьте файл с именем `order_handler.go` в папку `sportsstore/store` с содержимым, показанным в листинге 37-17.

```
package store

import (
    "encoding/json"
    "platform/http/actionresults"
    "platform/http/handling"
    "platform/sessions"
    "platform/validation"
    "sportsstore/models"
    "sportsstore/store/cart"
    "strings"
)

type OrderHandler struct {
    cart.Cart
    sessions.Session
    Repository models.Repository
    URLGenerator handling.URLGenerator
    validation.Validator
}

type OrderTemplateContext struct {
    models.ShippingDetails
    ValidationErrors [][]string
    CancelUrl string
}

func (handler OrderHandler) GetCheckout() actionresults.ActionResult
{
    context := OrderTemplateContext {}
```

```
     jsonData := handler.Session.GetValueDefault("checkout_details",
"")
    if jsonData != nil {
                          json.NewDecoder(strings.NewReader(jsonData.
(string))).Decode(&context)
    }
        context.CancelUrl  =  mustGenerateUrl(handler.URLGenerator,
CartHandler.GetCart)
    return actionresults.NewTemplateAction("checkout.html", context)
}

func        (handler        OrderHandler)        PostCheckout(details
models.ShippingDetails) actionresults.ActionResult {
    valid, errors := handler.Validator.Validate(details)
    if (!valid) {
        ctx := OrderTemplateContext {
            ShippingDetails: details,
            ValidationErrors: [][]string {},
        }
        for _, err := range errors {
            ctx.ValidationErrors = append(ctx.ValidationErrors,
                []string { err.FieldName, err.Error.Error()})
        }

        builder := strings.Builder{}
        json.NewEncoder(&builder).Encode(ctx)
                         handler.Session.SetValue("checkout_details",
builder.String())
        redirectUrl := mustGenerateUrl(handler.URLGenerator,
            OrderHandler.GetCheckout)
        return actionresults.NewRedirectAction(redirectUrl)
    } else {
        handler.Session.SetValue("checkout_details", "")
    }
    order := models.Order {
        ShippingDetails: details,
        Products: []models.ProductSelection {},
    }
    for _, cl := range handler.Cart.GetLines() {
                          order.Products   =   append(order.Products,
models.ProductSelection {
            Quantity: cl.Quantity,
            Product: cl.Product,
        })
    }
    handler.Repository.SaveOrder(&order)
    handler.Cart.Reset()
```

```
                                     targetUrl,         _          :=
handler.URLGenerator.GenerateUrl(OrderHandler.GetSummary,
        order.ID)
    return actionresults.NewRedirectAction(targetUrl)
}

func      (handler      OrderHandler)      GetSummary(id      int)
actionresults.ActionResult {
                                     targetUrl,         _          :=
handler.URLGenerator.GenerateUrl(ProductHandler.GetProducts,
        0, 1)
     return actionresults.NewTemplateAction("checkout_summary.html",
struct {
        ID int
        TargetUrl string
    }{ ID: id, TargetUrl: targetUrl})
}
```

Листинг 37-17 Содержимое файла order_handler.go в папке магазина

Этот обработчик определяет три метода. Метод `GetCheckout` отобразит HTML-форму, позволяющую пользователю ввести данные о доставке, и отобразит все ошибки проверки, возникшие в результате предыдущих попыток оформления заказа.

Метод `PostCheckout` является целью формы, отображаемой методом `GetCheckout`. Этот метод проверяет данные, предоставленные пользователем, и при наличии ошибок перенаправляет браузер обратно к методу `GetCheckout`. Я использую сеанс для передачи данных из метода `PostCheckout` в метод `GetCheckout`, кодируя и декодируя данные как JSON, чтобы их можно было сохранить в файле cookie сеанса.

Если ошибок проверки нет, метод `PostCheckout` создает `Order`, используя сведения о доставке, предоставленные пользователем, и сведения о продукте, полученные из `Cart`, которую обработчик получает в качестве услуги. `Order` хранится с использованием репозитория, а браузер перенаправляется на метод `GetSummary`, который отображает шаблон, отображающий сводку.

Чтобы создать шаблон сведений о доставке, добавьте файл с именем `checkout.html` в папку `sportsstore/templates` с содержимым, показанным в листинге 37-18.

```
{{ layout "simple_layout.html" }}
{{ $context := .}}
{{ $details := .ShippingDetails }}

<div class="p-2">
    <h2>Check out now</h2>
    Please enter your details, and we'll ship your goods right away!
```

```
</div>

{{ if gt (len $context.ValidationErrors) 0}}
    <ul class="text-danger mt-3">
        {{ range $context.ValidationErrors }}
            <li>
                {{ index . 0 }}: {{ index . 1 }}
            </li>
        {{ end }}
    </ul>
{{ end }}

<form method="POST" class="p-2">
    <h3>Ship to</h3>
    <div class="form-group">
        <label class="form-label">Name:</label>
                <input  name="name"  class="form-control"  value="{{
$details.Name }}" />
    </div>
    <div class="form-group">
        <label>Street Address:</label>
        <input name="streetaddr" class="form-control"
            value="{{ $details.StreetAddr }}" />
    </div>
    <div class="form-group">
        <label>City:</label>
                <input  name="city"  class="form-control"  value="{{
$details.City }}" />
    </div>
    <div class="form-group">
        <label>State:</label>
               <input  name="state"  class="form-control"  value="{{
$details.State }}" />
    </div>
    <div class="form-group">
        <label>Zip:</label>
                <input  name="zip"  class="form-control"  value="{{
$details.Zip }}" />
    </div>
    <div class="form-group">
        <label>Country:</label>
              <input  name="country"  class="form-control"  value="{{
$details.Country }}" />
    </div>
    <div class="text-center py-1">
        <a class="btn btn-secondary m-1" href="{{ $context.CancelUrl
}}">Cancel</a>
```

```
                          <button class="btn btn-primary m-1" type="submit">Submit</button>
    </div>
</form>
```

Листинг 37-18 Содержимое файла checkout.html в папке templates

Чтобы создать шаблон, отображаемый в конце процесса оформления заказа, добавьте файл с именем `checkout_summary.html` в папку `sportsstore/templates` с содержимым, показанным в листинге 37-19.

```
{{ layout "simple_layout.html" }}
{{ $context := . }}

<div class="text-center m-3">
    <h2>Thanks!</h2>
    <p>Thanks for placing order #{{ $context.ID }} </p>
    <p>We'll ship your goods as soon as possible.</p>
    <a class="btn btn-primary" href="{{ $context.TargetUrl }}">
        Return to Store
    </a>
</div>
```

Листинг 37-19 Содержимое файла checkout_summary.html в папке templates

Этот шаблон включает ссылку, которая вернет пользователя к списку продуктов. Метод `PostCheckout` сбрасывает корзину пользователя, позволяя пользователю снова начать процесс покупки.

Интеграция процесса оформления заказа

Чтобы пользователь мог начать процесс оформления заказа из сводки корзины, внесите изменения, показанные в листинге 37-20.

```
...
func (handler CartHandler) GetCart() actionresults.ActionResult {
               return actionresults.NewTemplateAction("cart.html", CartTemplateContext {
        Cart: handler.Cart,
                                                     ProductListUrl: handler.mustGenerateUrl(ProductHandler.GetProducts, 0, 1),
                                                          RemoveUrl: handler.mustGenerateUrl(CartHandler.PostRemoveFromCart),
                                                        CheckoutUrl: handler.mustGenerateUrl(OrderHandler.GetCheckout),
    })
}
...
```

Листинг 37-20 Добавление свойства контекста в файл cart_handler.go в папке store

Это изменение задает значение свойства `context`, чтобы дать шаблону URL-адрес для нацеливания на обработчик проверки. В листинге 37-21 добавлена ссылка, использующая URL.

```
...
<div class="text-center">
    <a class="btn btn-primary" href="{{ $context.ProductListUrl }}">
        Continue shopping
    </a>
     <a class="btn btn-danger" href="{{ $context.CheckoutUrl }}">Checkout</a>
</div>
...
```

Листинг 37-21 Добавление элемента в файл cart.html в папку templates

Регистрация обработчика запросов

В листинге 37-22 обработчик запросов регистрируется, чтобы он мог получать запросы.

```
...
func createPipeline() pipeline.RequestPipeline {
    return pipeline.CreatePipeline(
        &basic.ServicesComponent{},
        &basic.LoggingComponent{},
        &basic.ErrorComponent{},
        &basic.StaticFileComponent{},
        &sessions.SessionComponent{},
        handling.NewRouter(
            handling.HandlerEntry{ "",  store.ProductHandler{}},
            handling.HandlerEntry{ "",  store.CategoryHandler{}},
            handling.HandlerEntry{ "", store.CartHandler{}},
            handling.HandlerEntry{ "", store.OrderHandler{}},
          ).AddMethodAlias("/", store.ProductHandler.GetProducts, 0, 1).
            AddMethodAlias("/products[/]?[A-z0-9]*?",
                store.ProductHandler.GetProducts, 0, 1),
    )
}
...
```

Листинг 37-22 Регистрация нового обработчика в файле main.go в папке sportsstore

Скомпилируйте и запустите проект и используйте браузер для запроса `http://localhost:5000`. Добавьте товары в корзину и нажмите кнопку `Checkout`, после чего появится форма, показанная на рисунке 37-1.

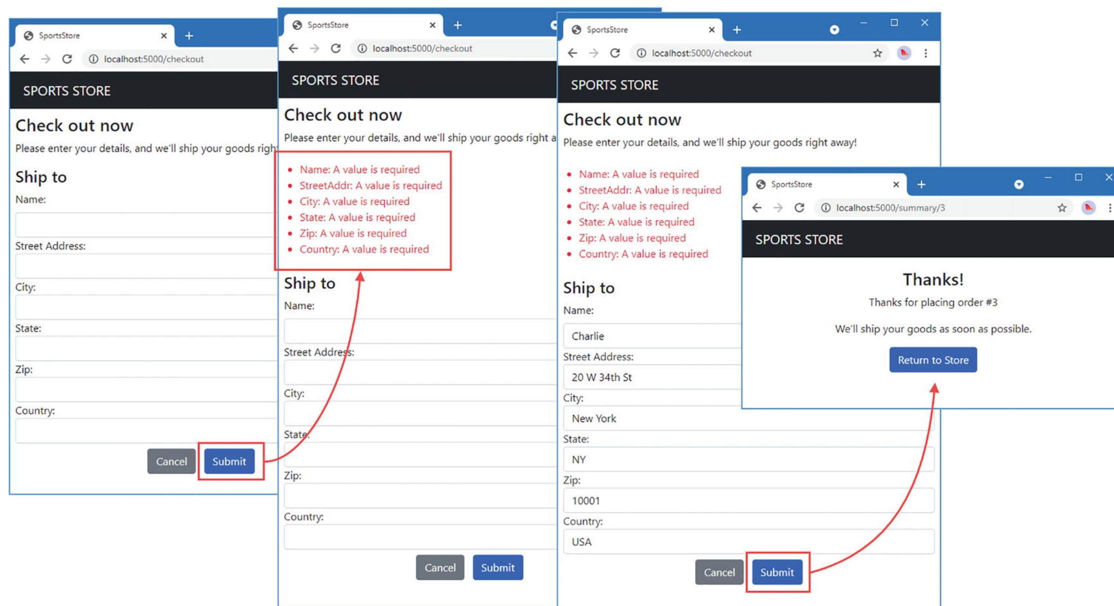

Рисунок 37-1 Процесс оформления заказа

Создание функций администрирования

В приложении SportsStore есть базовый процесс перечисления продуктов и оформления заказа, и теперь пришло время создать функции администрирования. Я собираюсь начать с некоторых базовых шаблонов и обработчиков, которые создают замещающий контент.

Создайте папку `sportsstore/admin` и добавьте в нее файл `main_handler.go` с содержимым, показанным в листинге 37-23.

```
package admin

import (
    "platform/http/actionresults"
    "platform/http/handling"
)

var sectionNames = []string { "Products", "Categories", "Orders",
"Database"}

type AdminHandler struct {
    handling.URLGenerator
}

type AdminTemplateContext struct {
    Sections []string
    ActiveSection string
    SectionUrlFunc func(string) string
```

```
}

func    (handler    AdminHandler)    GetSection(section    string)
actionresults.ActionResult {
              return   actionresults.NewTemplateAction("admin.html",
AdminTemplateContext {
        Sections: sectionNames,
        ActiveSection: section,
        SectionUrlFunc: func(sec string) string {
                                                 sectionUrl,    _    :=
handler.GenerateUrl(AdminHandler.GetSection, sec)
            return sectionUrl
        },
    })
}
```

Листинг 37-23 Содержимое файла main_handler.go в папке admin

Целью этого обработчика является отображение шаблона общих функций администрирования с кнопками для перемещения между различными разделами функций. Добавьте файл с именем `admin.html` в папку `sportsstore/templates` с содержимым, показанным в листинге 37-24.

```
{{ $context := . }}
<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>SportsStore</title>
    <link href="/files/bootstrap.min.css" rel="stylesheet" />
</head>
<body>
    <div class="bg-info text-white p-2">
        <div class="container-fluid">
            <div class="row">
                         <div  class="col  navbar-brand">SPORTS  STORE
Administration</div>
            </div>
        </div>
    </div>
    <div class="row m-1 p-1">
        <div id="sidebar" class="col-3">
            <div class="d-grid gap-2">
                {{ range $context.Sections }}
                    <a href="{{ call $context.SectionUrlFunc . }}"
                        {{ if eq . $context.ActiveSection }}
                            class="btn btn-info">
                        {{ else }}
```

```
                            class="btn btn-outline-info">
                        {{ end }}
                        {{ . }}
                    </a>
                {{ end }}
            </div>
        </div>
        <div class="col-9">
            {{ if eq $context.ActiveSection ""}}
                <h6 class="p-2">
                          Welcome to the SportsStore Administration
Features
                </h6>
            {{ else }}
                {{ handler $context.ActiveSection "getdata" }}
            {{ end }}
        </div>
    </div>
</body>
</html>
```

Листинг 37-24 Содержимое файла admin.html в папке templates

Этот шаблон использует другую цветовую схему для обозначения функций администрирования и отображает макет из двух столбцов с кнопками разделов с одной стороны и выбранной функцией администрирования с другой. Выбранная функция отображается с помощью функции `handler`.

Добавьте файл с именем `products_handler.go` в папку `sportsstore/admin` с содержимым, показанным в листинге 37-25.

```
package admin

type ProductsHandler struct {}

func (handler ProductsHandler) GetData() string {
    return "This is the products handler"
}
```

Листинг 37-25 Содержимое файла products_handler.go в папке admin

Добавьте файл с именем `category_handler.go` в папку `sportsstore/admin` с содержимым, показанным в листинге 37-26.

```
package admin

type CategoriesHandler struct {}

func (handler CategoriesHandler) GetData() string {
```

```
    return "This is the categories handler"
}
```

Листинг 37-26 Содержимое файла category_handler.go в папке admin

Добавьте файл с именем `orders_handler.go` в папку `sportsstore/admin` с содержимым, показанным в листинге 37-27.

```
package admin

type OrdersHandler struct {}

func (handler OrdersHandler) GetData() string {
    return "This is the orders handler"
}
```

Листинг 37-27 Содержимое файла orders_handler.go в папке admin

Чтобы завершить набор обработчиков, добавьте файл с именем `database_handler.go` в папку `sportsstore/admin` с содержимым, показанным в листинге 37-28.

```
package admin

type DatabaseHandler struct {}

func (handler DatabaseHandler) GetData() string {
    return "This is the database handler"
}
```

Листинг 37-28 Содержимое файла database_handler.go в папке admin

Я добавлю элементы управления доступом для функций администрирования в главе 38, а сейчас я собираюсь зарегистрировать новые обработчики, чтобы к ним мог получить доступ любой, как показано в листинге 37-29.

```
package main

import (
    "sync"
    "platform/http"
    "platform/http/handling"
    "platform/services"
    "platform/pipeline"
    "platform/pipeline/basic"
    "sportsstore/store"
    "sportsstore/models/repo"
```

```
    "platform/sessions"
    "sportsstore/store/cart"
    "sportsstore/admin"
)

func registerServices() {
    services.RegisterDefaultServices()
    //repo.RegisterMemoryRepoService()
    repo.RegisterSqlRepositoryService()
    sessions.RegisterSessionService()
    cart.RegisterCartService()
}

func createPipeline() pipeline.RequestPipeline {
    return pipeline.CreatePipeline(
        &basic.ServicesComponent{},
        &basic.LoggingComponent{},
        &basic.ErrorComponent{},
        &basic.StaticFileComponent{},
        &sessions.SessionComponent{},
        handling.NewRouter(
            handling.HandlerEntry{ "",  store.ProductHandler{}},
            handling.HandlerEntry{ "",  store.CategoryHandler{}},
            handling.HandlerEntry{ "", store.CartHandler{}},
            handling.HandlerEntry{ "", store.OrderHandler{}},
            handling.HandlerEntry{ "admin", admin.AdminHandler{}},
                                   handling.HandlerEntry{   "admin",
admin.ProductsHandler{}},
                                   handling.HandlerEntry{   "admin",
admin.CategoriesHandler{}},
            handling.HandlerEntry{ "admin", admin.OrdersHandler{}},
                                   handling.HandlerEntry{   "admin",
admin.DatabaseHandler{}},
             ).AddMethodAlias("/", store.ProductHandler.GetProducts,
0, 1).
AddMethodAlias("/products[/]?[A-z0-9]*?",
store.ProductHandler.GetProducts, 0, 1).
                                        AddMethodAlias("/admin[/]?",
admin.AdminHandler.GetSection, ""),
    )
}

func main() {
    registerServices()
    results, err := services.Call(http.Serve, createPipeline())
    if (err == nil) {
        (results[0].(*sync.WaitGroup)).Wait()
    } else {
```

```
        panic(err)
    }
}
```

Листинг 37-29 Регистрация обработчиков администрирования в файле main.go в папке sportsstore

Скомпилируйте и запустите проект и используйте браузер для запроса `http://localhost:5000/admin`, что даст ответ, показанный на рисунке 37-2. Нажатие кнопок навигации в левом столбце вызывает различные обработчики в правом столбце.

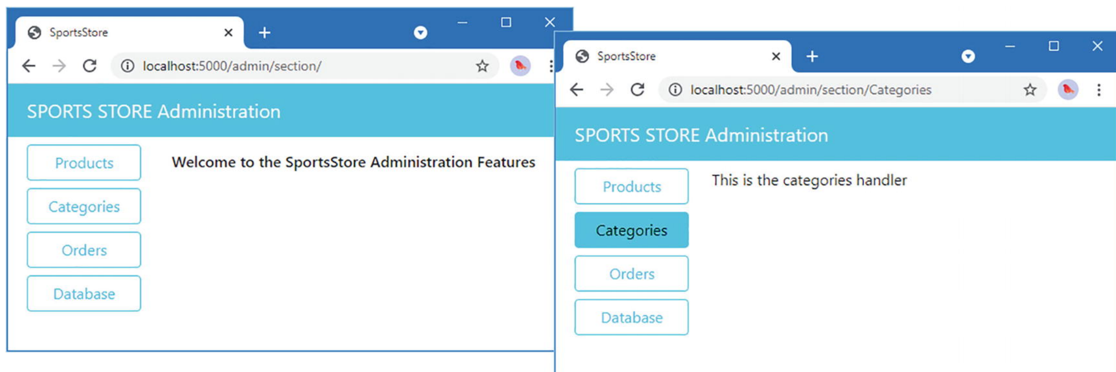

Рисунок 37-2 Начало работы над функциями администрирования

Создание функции администрирования продукта

Функция администрирования продуктов позволит добавлять новые продукты в магазин и изменять существующие продукты. Для простоты я не позволю удалять продукты из базы данных, которая была создана с использованием отношений внешнего ключа между таблицами.

Расширение репозитория

Первый шаг — расширить `Repository`, чтобы я мог вносить изменения в базу данных. В листинге 37-30 к интерфейсу `Repository` добавлен новый метод.

```
package models

type Repository interface {

    GetProduct(id int) Product
    GetProducts() []Product
    SaveProduct(*Product)

        GetProductPage(page,  pageSize  int)  (products  []Product,
totalAvailable int)
```

```
        GetProductPageCategory(categoryId  int,  page,  pageSize  int)
(products []Product,
        totalAvailable int)

    GetCategories() []Category

    GetOrder(id int) Order
    GetOrders() []Order
    SaveOrder(*Order)

    Seed()
}
```

Листинг 37-30 Определение метода в файле repository.go в папке models

Чтобы определить SQL, который будет использоваться для хранения новых продуктов, добавьте файл с именем `save_product.sql` в папку `sportsstore/sql` с содержимым, показанным в листинге 37-31.

```
INSERT INTO Products(Name, Description, Category, Price)
VALUES (?, ?, ?, ?)
```

Листинг 37-31 Содержимое файла save_product.sql в папке sql

Чтобы определить SQL, который будет использоваться для изменения существующих продуктов, добавьте файл с именем `update_product.sql` в папку `sportsstore/sql` с содержимым, показанным в листинге 37-32.

```
UPDATE Products
SET Name = ?, Description = ?, Category = ?, Price =?
WHERE Id == ?
```

Листинг 37-32 Содержимое файла update_product.sql в папке sql

В листинге 37-33 добавлены новые команды, обеспечивающие доступ к файлам SQL для изменения данных о продукте.

```
package repo

import (
    "database/sql"
    "platform/config"
    "platform/logging"
    "context"
)

type SqlRepository struct {
    config.Configuration
    logging.Logger
```

```
    Commands SqlCommands
    *sql.DB
    context.Context
}

type SqlCommands struct {
    Init,
    Seed,
    GetProduct,
    GetProducts,
    GetCategories,
    GetPage,
    GetPageCount,
    GetCategoryPage,
    GetCategoryPageCount,
    GetOrder,
    GetOrderLines,
    GetOrders,
    GetOrdersLines,
    SaveOrder,
    SaveOrderLine,
    SaveProduct,
    UpdateProduct *sql.Stmt
}
```

Листинг 37-33 Добавление команд в файл sql_repo.go в папке models/repo

В листинге 37-34 добавлены параметры конфигурации, указывающие расположение файлов SQL для новых команд.

```
...
"sql": {
    "connection_str": "store.db",
    "always_reset": true,
    "commands": {
        "Init":                 "sql/init_db.sql",
        "Seed":                 "sql/seed_db.sql",
        "GetProduct":           "sql/get_product.sql",
        "GetProducts":          "sql/get_products.sql",
        "GetCategories":        "sql/get_categories.sql",
        "GetPage":              "sql/get_product_page.sql",
        "GetPageCount":         "sql/get_page_count.sql",
         "GetCategoryPage":      "sql/get_category_product_page.sql",
                                                     "GetCategoryPageCount":
"sql/get_category_product_page_count.sql",
        "GetOrder":             "sql/get_order.sql",
        "GetOrderLines":        "sql/get_order_lines.sql",
        "GetOrders":            "sql/get_orders.sql",
```

```
        "GetOrdersLines":       "sql/get_orders_lines.sql",
        "SaveOrder":            "sql/save_order.sql",
        "SaveOrderLine":        "sql/save_order_line.sql",
        "SaveProduct":          "sql/save_product.sql",
        "UpdateProduct":        "sql/update_product.sql"
    }
}
...
```

Листинг 37-34 Добавление настроек конфигурации в файл config.json в папке sportsstore

Чтобы использовать команды SQL для реализации метода репозитория, добавьте файл с именем `sql_products_save.go` в папку `sportsstore/models/repo` с содержимым, показанным в листинге 37-35.

```
package repo

import "sportsstore/models"

func (repo *SqlRepository) SaveProduct(p *models.Product) {

    if (p.ID == 0) {
                                             result,     err     :=
repo.Commands.SaveProduct.ExecContext(repo.Context, p.Name,
            p.Description, p.Category.ID, p.Price)
        if err == nil {
            id, err := result.LastInsertId()
            if err == nil {
                p.ID = int(id)
                return
            } else {
                   repo.Logger.Panicf("Cannot get inserted ID: %v",
err.Error())
            }
        } else {
               repo.Logger.Panicf("Cannot exec SaveProduct command:
%v", err.Error())
        }
    } else {
                                             result,     err     :=
repo.Commands.UpdateProduct.ExecContext(repo.Context, p.Name,
            p.Description, p.Category.ID, p.Price, p.ID)
        if err == nil {
            affected, err := result.RowsAffected()
            if err == nil && affected != 1 {
                  repo.Logger.Panicf("Got unexpected rows affected:
%v", affected)
            } else if err != nil {
```

```
                  repo.Logger.Panicf("Cannot get rows affected: %v",
err)
            }
        } else {
                repo.Logger.Panicf("Cannot exec Update command: %v",
err.Error())
        }
    }
}
```

Листинг 37-35 Содержимое файла sql_products_save.go в папке models/repo

Если `ID` свойство `Product`, полученное этим методом, равно нулю, то данные добавляются в базу данных; в противном случае выполняется обновление.

Реализация обработчика запросов продуктов

Следующим шагом является удаление ответа-заполнителя из обработчика запроса и добавление реальной функциональности, которая позволит администратору просматривать и редактировать данные о `Product`. Замените содержимое файла `products_handler.go` в папке `sportsstore/admin` содержимым, показанным в листинге 37-36. (Убедитесь, что вы редактируете файл в папке `admin`, а не файл в папке `store` с таким же именем.)

```
package admin

import (
    "sportsstore/models"
    "platform/http/actionresults"
    "platform/http/handling"
    "platform/sessions"
)

type ProductsHandler struct {
    models.Repository
    handling.URLGenerator
    sessions.Session
}

type ProductTemplateContext struct {
    Products []models.Product
    EditId int
    EditUrl string
    SaveUrl string
}

const PRODUCT_EDIT_KEY string = "product_edit"
```

```
func (handler ProductsHandler) GetData() actionresults.ActionResult {
    return actionresults.NewTemplateAction("admin_products.html",
            ProductTemplateContext {
        Products: handler.GetProducts(),
        EditId: handler.Session.GetValueDefault(PRODUCT_EDIT_KEY, 0).
(int),
        EditUrl: mustGenerateUrl(handler.URLGenerator,
             ProductsHandler.PostProductEdit),
        SaveUrl: mustGenerateUrl(handler.URLGenerator,
             ProductsHandler.PostProductSave),
    })
}

type EditReference struct {
    ID int
}

func  (handler  ProductsHandler)  PostProductEdit(ref  EditReference)
actionresults.ActionResult {
    handler.Session.SetValue(PRODUCT_EDIT_KEY, ref.ID)
                                                                   return
actionresults.NewRedirectAction(mustGenerateUrl(handler.URLGenerator,
         AdminHandler.GetSection, "Products"))
}

type ProductSaveReference struct {
    Id int
    Name, Description string
    Category int
    Price float64
}

func (handler ProductsHandler) PostProductSave(
        p ProductSaveReference) actionresults.ActionResult {

    handler.Repository.SaveProduct(&models.Product{
        ID: p.Id, Name: p.Name, Description: p.Description,
        Category: &models.Category{ ID: p.Category },
        Price: p.Price,
    })
    handler.Session.SetValue(PRODUCT_EDIT_KEY, 0)
                                                                   return
actionresults.NewRedirectAction(mustGenerateUrl(handler.URLGenerator,
        AdminHandler.GetSection, "Products"))
}

func mustGenerateUrl(gen handling.URLGenerator, target interface{},
        data ...interface{}) string {
```

```
    url, err := gen.GenerateUrl(target, data...)
    if (err != nil) {
        panic(err)
    }
    return url
}
```

Листинг 37-36 Добавление функций в файл products_handler.go в папке admin

Метод GetData отображает шаблон с именем admin_products.html с данными контекста, которые содержат значения Product в базе данных, значение int, используемое для обозначения ID продукта, который пользователь хочет изменить, и URL-адреса, используемые для навигации. Чтобы создать шаблон, добавьте файл с именем admin_products.html в папку sportsstore/templates с содержимым, показанным в листинге 37-37.

```
{{ $context := . }}
<table class="table table-sm table-striped table-bordered">
    <thead>
        <tr>
            <th>ID</th><th>Name</th><th>Description</th>
                <th>Category</th><th class="text-end">Price</th><th>
</th>
        </tr>
    </thead>
    <tbody>
        {{ range $context.Products }}
            {{ if ne $context.EditId .ID}}
                <tr>
                    <td>{{ .ID }}</td>
                    <td>{{ .Name }}</td>
                    <td>
                        <span class="d-inline-block text-truncate"
                            style="max-width: 200px;">
                               {{ .Description }}
                        </span>
                    </td>
                    <td>{{ .CategoryName }}</td>
                    <td class="text-end">{{ printf "$%.2f" .Price }}
</td>
                    <td class="text-center">
                                         <form method="POST" action="{{
$context.EditUrl }}">
                              <input type="hidden" name="id" value="{{
.ID }}" />
                                <button class="btn btn-sm btn-warning"
type="submit">
                                  Edit
```

```
                                </button>
                            </form>
                        </td>
                    </tr>
                {{ else }}
                    <tr>
                         <form method="POST" action="{{ $context.SaveUrl
}}" >
                            <input type="hidden" name="id" value="{{ .ID
}}" />
                            <td>
                                    <input class="form-control" disabled
value="{{.ID}}"
                                    size="3"/>
                            </td>
                             <td><input name="name" class="form-control"
size=12
                                value="{{ .Name }}" /></td>
                              <td><input name="description" class="form-
control"
                                   size=15 value="{{ .Description }}" />
</td>
                                <td>{{ handler "categories" "getselect"
.Category.ID }}</td>
                             <td><input name="price" class="form-control
text-end"
                                size=7 value="{{ .Price }}"/></td>
                            <td>
                                   <button class="btn btn-sm btn-danger"
type="submit">
                                    Save
                                </button>
                            </td>
                        </form>
                    </tr>
                {{ end }}
            {{ end }}
        </tbody>
        {{ if eq $context.EditId 0}}
            <tfoot>
                      <tr><td  colspan="6"  class="text-center">Add  New
Product</td></tr>
                <tr>
                    <form method="POST" action="{{ $context.SaveUrl }}"
>
                       <td>-</td>
                            <td><input name="name" class="form-control"
size=12 /></td>
```

```
                         <td><input name="description" class="form-control"
                       size=15 /></td>
                      <td>{{ handler "categories" "getselect" 0 }}
</td>
                       <td><input name="price" class="form-control"
size=7 /></td>
                 <td>
                            <button class="btn btn-sm btn-danger"
type="submit">
                            Save
                       </button>
                 </td>
             </form>
         </tr>
     </tfoot>
    {{ end }}
</table>
```

Листинг 37-37 Содержимое файла admin_products.html в папке templates

Этот шаблон создает HTML-шаблон, содержащий все продукты, а также встроенный редактор для изменения существующих продуктов и еще один для создания новых продуктов. Для обеих задач требуется элемент `select`, который позволяет пользователю выбрать категорию, которая создается путем вызова метода, определенного в `CategoriesHandler`. Листинг 37-38 добавляет этот метод в обработчик запросов.

```
package admin

import (
    "platform/http/actionresults"
    "sportsstore/models"
)

type CategoriesHandler struct {
    models.Repository
}

func (handler CategoriesHandler) GetData() string {
    return "This is the categories handler"
}

func    (handler    CategoriesHandler)    GetSelect(current    int)
actionresults.ActionResult {
      return actionresults.NewTemplateAction("select_category.html",
struct {
        Current int
```

```
        Categories []models.Category
    }{ Current: current, Categories: handler.GetCategories()})
}
```

Листинг 37-38 Добавление поддержки для элемента Select в файле category_handler.go в папке admin

Чтобы определить шаблон, используемый методом GetSelect, добавьте файл с именем select_category.html в папку sportsstore/templates с содержимым, показанным в листинге 37-39.

```
{{ $context := . }}

<select class="form-select" name="category" value="{{
$context.Current }}">
    <option value="0">Select a category</option>
    {{ range $context.Categories }}
        <option value="{{.ID}}" {{ if eq $context.Current .ID
}}selected{{end}}>
            {{.CategoryName}}
        </option>
    {{ end }}
</select>
```

Листинг 37-39 Содержимое файла select_category.html в папке templates

Скомпилируйте и запустите проект, используйте браузер для запроса http://localhost:5000/admin и нажмите кнопку **Products**. Вы увидите список продуктов, который был прочитан из базы данных. Нажмите одну из кнопок **Edit**, чтобы выбрать продукт для редактирования, введите новые значения в поля формы и нажмите кнопку **Submit**, чтобы сохранить изменения в базе данных, также показанные на рисунке 37-3.

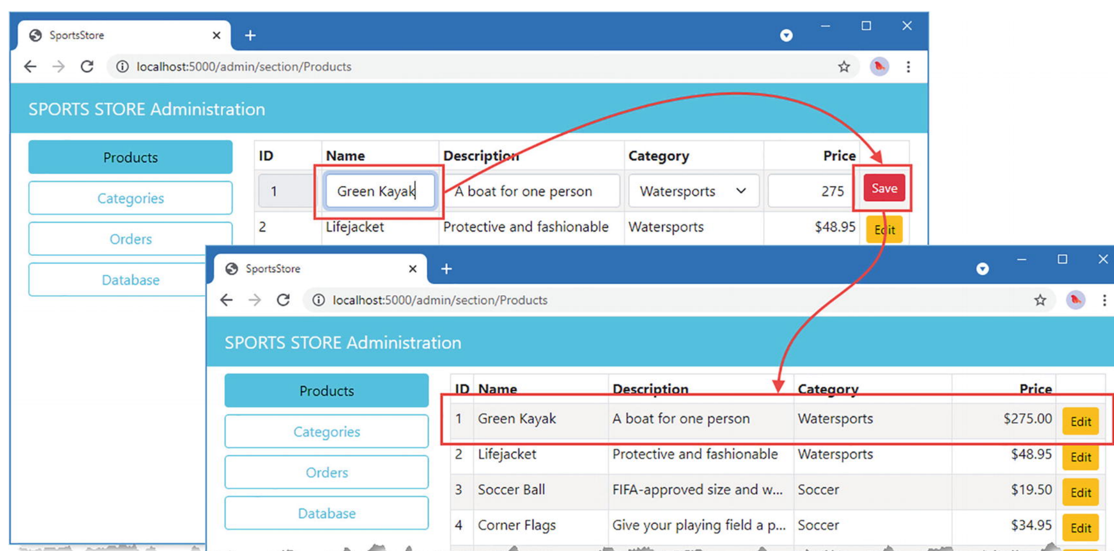

Рисунок 37-3 Editing a product

> **Примечание**
> Приложение SportsStore настроено на сброс базы данных при каждом запуске, а это означает, что любые изменения, которые вы вносите в базу данных, будут отброшены. Я отключил эту функцию при подготовке приложения к развертыванию в главе 38.

Если для редактирования не выбран ни один продукт, можно использовать форму в нижней части таблицы для создания новых продуктов в базе данных, как показано на рисунке 37-4.

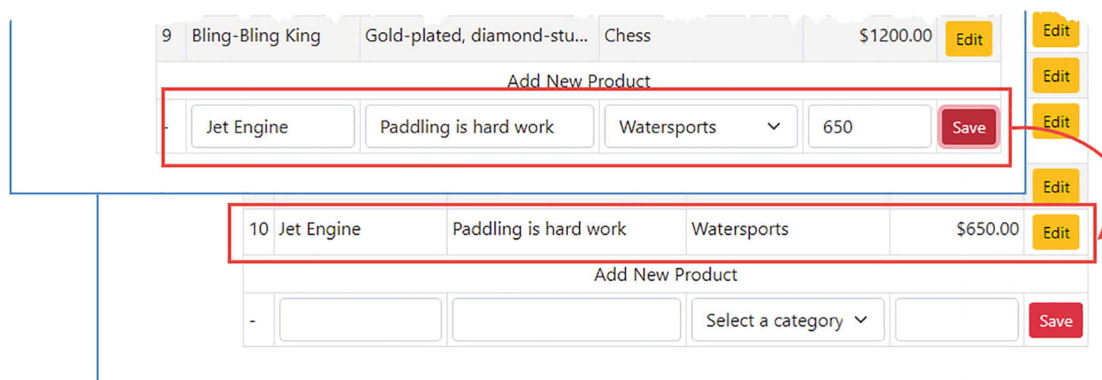

Рисунок 37-4 Добавление продукта

Создание функции администрирования категорий

Я собираюсь применить базовый шаблон, установленный в предыдущем разделе, для реализации других функций администрирования.

Расширение репозитория

В листинге 37-40 к интерфейсу `Repository` добавлен метод, который будет хранить `Category`.

```
package models

type Repository interface {

    GetProduct(id int) Product
    GetProducts() []Product
    SaveProduct(*Product)

        GetProductPage(page, pageSize int) (products []Product,
totalAvailable int)

       GetProductPageCategory(categoryId int, page, pageSize int)
(products []Product,
        totalAvailable int)

    GetCategories() []Category
    SaveCategory(*Category)

    GetOrder(id int) Order
    GetOrders() []Order
    SaveOrder(*Order)

    Seed()
}
```

Листинг 37-40 Добавление метода в файл репозитория.go в папке models

Чтобы определить SQL, который будет использоваться для хранения новых категорий в базе данных, добавьте файл с именем `save_category.sql` в папку `sportsstore/sql` с содержимым, показанным в листинге 37-41.

```
INSERT INTO Categories(Name) VALUES (?)
```

Листинг 37-41 Содержимое файла save_category.sql в папке sql

Чтобы определить SQL, который будет использоваться для изменения существующих категорий, добавьте файл с именем `update_category.sql` в папку `sportsstore/sql` с содержимым, показанным в листинге 37-42.

```
UPDATE Categories SET Name = ? WHERE Id == ?
```

Листинг 37-42 Содержимое файла update_category.sql в папке sql

В листинге 37-43 добавлены новые команды, обеспечивающие доступ к файлам SQL.

```
...
type SqlCommands struct {
    Init,
    Seed,
    GetProduct,
    GetProducts,
    GetCategories,
    GetPage,
    GetPageCount,
    GetCategoryPage,
    GetCategoryPageCount,
    GetOrder,
    GetOrderLines,
    GetOrders,
    GetOrdersLines,
    SaveOrder,
    SaveOrderLine,
    SaveProduct,
    UpdateProduct,
    SaveCategory,
    UpdateCategory *sql.Stmt
}
...
```

Листинг 37-43 Добавление команд в файл sql_repo.go в папке models/repo

В листинге 37-44 добавлены параметры конфигурации, указывающие расположение файлов SQL для новых команд.

```
...
"sql": {
    "connection_str": "store.db",
    "always_reset": true,
    "commands": {
        "Init":                 "sql/init_db.sql",
        "Seed":                 "sql/seed_db.sql",
        "GetProduct":           "sql/get_product.sql",
        "GetProducts":          "sql/get_products.sql",
        "GetCategories":        "sql/get_categories.sql",
        "GetPage":              "sql/get_product_page.sql",
        "GetPageCount":         "sql/get_page_count.sql",
        "GetCategoryPage":      "sql/get_category_product_page.sql",
                                             "GetCategoryPageCount":
"sql/get_category_product_page_count.sql",
        "GetOrder":             "sql/get_order.sql",
```

```
        "GetOrderLines":        "sql/get_order_lines.sql",
        "GetOrders":            "sql/get_orders.sql",
        "GetOrdersLines":       "sql/get_orders_lines.sql",
        "SaveOrder":            "sql/save_order.sql",
        "SaveOrderLine":        "sql/save_order_line.sql",
        "SaveProduct":          "sql/save_product.sql",
        "UpdateProduct":        "sql/update_product.sql",
        "SaveCategory":         "sql/save_category.sql",
        "UpdateCategory":       "sql/update_category.sql"
    }
}
...
```

Листинг 37-44 Добавление настроек конфигурации в файл config.json в папке sportsstore

Чтобы реализовать новый метод интерфейса, добавьте файл с именем `sql_category_save.go` в папку `sportsstore/models/repo` с содержимым, показанным в листинге 37-45.

```
package repo

import "sportsstore/models"

func (repo *SqlRepository) SaveCategory(c *models.Category) {
    if (c.ID == 0) {
                                                result,     err     :=
repo.Commands.SaveCategory.ExecContext(repo.Context,
            c.CategoryName)
        if err == nil {
            id, err := result.LastInsertId()
            if err == nil {
                c.ID = int(id)
                return
            } else {
                    repo.Logger.Panicf("Cannot get inserted ID: %v",
err.Error())
            }
        } else {
                repo.Logger.Panicf("Cannot exec SaveCategory command:
%v", err.Error())
        }
    } else {
                                                result,     err     :=
repo.Commands.UpdateCategory.ExecContext(repo.Context,
            c.CategoryName, c.ID)
        if err == nil {
            affected, err := result.RowsAffected()
            if err == nil && affected != 1 {
```

```
                    repo.Logger.Panicf("Got unexpected rows affected:
%v", affected)
            } else if err != nil {
                repo.Logger.Panicf("Cannot get rows affected: %v",
err)
            }
        } else {
            repo.Logger.Panicf("Cannot exec UpdateCategory command:
%v", err.Error())
        }
    }
}
```

Листинг 37-45 Содержимое файла sql_category_save.go в папке models/repo

Если свойство `ID` полученной этим методом `Category` равно нулю, то данные добавляются в базу данных; в противном случае выполняется обновление.

Реализация обработчика запроса категории

Замените содержимое файла `category_handler.go` в папке `sportsstore/admin` кодом, показанным в листинге 37-46.

```
package admin

import (
    "sportsstore/models"
    "platform/http/actionresults"
    "platform/http/handling"
    "platform/sessions"
)

type CategoriesHandler struct {
    models.Repository
    handling.URLGenerator
    sessions.Session
}

type CategoryTemplateContext struct {
    Categories []models.Category
    EditId int
    EditUrl string
    SaveUrl string
}

const CATEGORY_EDIT_KEY string = "category_edit"
```

```
func (handler CategoriesHandler) GetData() actionresults.ActionResult
{
    return actionresults.NewTemplateAction("admin_categories.html",
        CategoryTemplateContext {
            Categories: handler.Repository.GetCategories(),
                                                                    EditId:
handler.Session.GetValueDefault(CATEGORY_EDIT_KEY, 0).(int),
            EditUrl: mustGenerateUrl(handler.URLGenerator,
                 CategoriesHandler.PostCategoryEdit),
            SaveUrl: mustGenerateUrl(handler.URLGenerator,
                 CategoriesHandler.PostCategorySave),
        })
}

func (handler CategoriesHandler) PostCategoryEdit(ref EditReference)
actionresults.ActionResult {
    handler.Session.SetValue(CATEGORY_EDIT_KEY, ref.ID)
                                                                     return
actionresults.NewRedirectAction(mustGenerateUrl(handler.URLGenerator,
         AdminHandler.GetSection, "Categories"))
}

func (handler CategoriesHandler) PostCategorySave(
        c models.Category) actionresults.ActionResult {
    handler.Repository.SaveCategory(&c)
    handler.Session.SetValue(CATEGORY_EDIT_KEY, 0)
                                                                     return
actionresults.NewRedirectAction(mustGenerateUrl(handler.URLGenerator,
        AdminHandler.GetSection, "Categories"))
}

func    (handler    CategoriesHandler)    GetSelect(current    int)
actionresults.ActionResult {
      return  actionresults.NewTemplateAction("select_category.html",
struct {
        Current int
        Categories []models.Category
    }{ Current: current, Categories: handler.GetCategories()})
}
```

Листинг 37-46 Замена содержимого файла category_handler.go в папке admin

Чтобы определить шаблон, используемый этим обработчиком, добавьте файл с именем `admin_categories.html` в папку `sportsstore/templates` с содержимым, показанным в листинге 37-47.

```
{{ $context := . }}
<table class="table table-sm table-striped table-bordered">
```

```
    <thead><tr><th>ID</th><th>Name</th><th></th></tr></thead>
    <tbody>
        {{ range $context.Categories }}
            {{ if ne $context.EditId .ID}}
                <tr>
                    <td>{{ .ID }}</td>
                    <td>{{ .CategoryName }}</td>
                    <td class="text-center">
                                    <form method="POST" action="{{
$context.EditUrl }}">
                            <input type="hidden" name="id" value="{{
.ID }}" />
                              <button class="btn btn-sm btn-warning"
type="submit">
                                Edit
                            </button>
                        </form>
                    </td>
                </tr>
            {{ else }}
                <tr>
                     <form method="POST" action="{{ $context.SaveUrl
}}" >
                        <input type="hidden" name="id" value="{{ .ID
}}" />
                        <td>
                            <input class="form-control" disabled
                                value="{{.ID}}" size="3"/>
                        </td>
                         <td><input name="categoryname" class="form-
control" size=12
                            value="{{ .CategoryName }}" /></td>
                    <td class="text-center">
                               <button class="btn btn-sm btn-danger"
type="submit">
                                Save
                            </button>
                        </td>
                    </form>
                </tr>
            {{end }}
        {{ end }}
    </tbody>
    {{ if eq $context.EditId 0}}
        <tfoot>
                    <tr><td colspan="6" class="text-center">Add New
Category</td></tr>
            <tr>
```

```
                 <form method="POST" action="{{ $context.SaveUrl }}"
>
                    <td>-</td>
                         <td><input name="categoryname" class="form-
control"
                        size=12 /></td>
                    <td class="text-center">
                               <button class="btn btn-sm btn-danger"
type="submit">
                            Save
                        </button>
                    </td>
                </form>
            </tr>
        </tfoot>
    {{ end }}
</table>
```

Листинг 37-47 Содержимое файла admin_categories.html в папке templates

Скомпилируйте и запустите проект, используйте браузер для запроса `http://localhost:5000/admin` и нажмите кнопку **Categories**. Вы увидите список категорий, который был прочитан из базы данных, и сможете редактировать и создавать категории, как показано на рисунке 37-5.

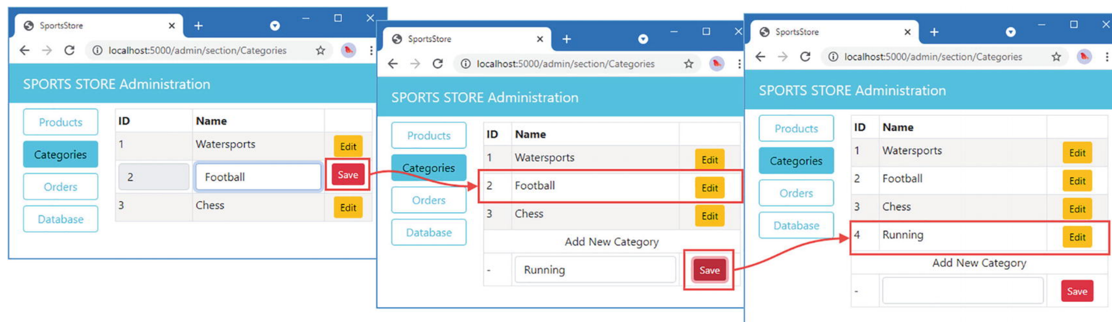

Рисунок 37-5 Управление категориями

Резюме

В этой главе я продолжил разработку приложения SportsStore, добавив процесс оплаты и начав работу над функциями администрирования. В следующей главе я дополню эти функции, добавлю поддержку контроля доступа и подготовлю приложение к развертыванию.

38. SportsStore: завершение и развертывание

В этой главе я завершаю разработку приложения SportsStore и готовлю его к развертыванию.

> **Подсказка**
> Вы можете загрузить пример проекта для этой главы — и для всех остальных глав этой книги — с https://github.com/apress/pro-go. См. Главу 2 о том, как получить помощь, если у вас возникнут проблемы с запуском примеров.

Завершение функций администрирования

Два из четырех разделов администрирования, определенных в главе 37, еще не реализованы. В этом разделе я определяю обе эти функции одновременно, отражая тот факт, что они проще, чем функции продукта и категории.

Расширение репозитория

Для реализации функций администрирования требуются два новых метода репозитория, как показано в листинге 38-1.

```
package models

type Repository interface {

    GetProduct(id int) Product
    GetProducts() []Product
    SaveProduct(*Product)

        GetProductPage(page, pageSize int) (products []Product,
totalAvailable int)

       GetProductPageCategory(categoryId int, page, pageSize int)
(products []Product,
        totalAvailable int)

    GetCategories() []Category
    SaveCategory(*Category)

    GetOrder(id int) []Order
```

```
    GetOrders() Order
    SaveOrder(*Order)
    SetOrderShipped(*Order)

    Seed()
    Init()
}
```

Листинг 38-1 Добавление интерфейса в файл репозитория.go в папке models

Метод SetOrderShipped будет использоваться для обновления существующего Order, чтобы указать, когда он был отправлен. Метод Init соответствует имени метода, уже определенному реализацией интерфейса SQL, и будет использоваться, чтобы позволить администратору подготовить базу данных к первому использованию после ее развертывания.

Чтобы определить SQL, который будет использоваться для обновления существующих заказов, добавьте файл с именем update_order.sql в папку sportsstore/sql с содержимым, показанным в листинге 38-2.

```
UPDATE Orders SET Shipped = ? WHERE Id == ?
```

Листинг 38-2 Содержимое файла update_order.sql в папке sql

В листинге 38-3 добавлена новая команда, чтобы к SQL, определенному в листинге 38-2, можно было обращаться так же, как и к другим операторам SQL.

```
...
type SqlCommands struct {
    Init,
    Seed,
    GetProduct,
    GetProducts,
    GetCategories,
    GetPage,
    GetPageCount,
    GetCategoryPage,
    GetCategoryPageCount,
    GetOrder,
    GetOrderLines,
    GetOrders,
    GetOrdersLines,
    SaveOrder,
    SaveOrderLine,
    UpdateOrder,
    SaveProduct,
    UpdateProduct,
    SaveCategory,
    UpdateCategory *sql.Stmt
```

```
}
...
```

Листинг 38-3 Добавление новой команды в файл sql_repo.go в папке models/repo

В листинге 38-4 добавлен параметр конфигурации, указывающий расположение SQL, необходимого для новой команды.

```
...
"sql": {
    "connection_str": "store.db",
    "always_reset": true,
    "commands": {
        "Init": "sql/init_db.sql",
        "Seed": "sql/seed_db.sql",
        "GetProduct": "sql/get_product.sql",
        "GetProducts": "sql/get_products.sql",
        "GetCategories": "sql/get_categories.sql",
        "GetPage": "sql/get_product_page.sql",
        "GetPageCount": "sql/get_page_count.sql",
        "GetCategoryPage": "sql/get_category_product_page.sql",
                                             "GetCategoryPageCount":
"sql/get_category_product_page_count.sql",
        "GetOrder": "sql/get_order.sql",
        "GetOrderLines": "sql/get_order_lines.sql",
        "GetOrders": "sql/get_orders.sql",
        "GetOrdersLines": "sql/get_orders_lines.sql",
        "SaveOrder": "sql/save_order.sql",
        "SaveOrderLine": "sql/save_order_line.sql",
        "SaveProduct":          "sql/save_product.sql",
        "UpdateProduct":        "sql/update_product.sql",
        "SaveCategory":         "sql/save_category.sql",
        "UpdateCategory":       "sql/update_category.sql",
        "UpdateOrder":          "sql/update_order.sql"
    }
}
...
```

Листинг 38-4 Добавление параметра конфигурации в файл config.json в папке sportsstore

Чтобы реализовать метод репозитория, добавьте файл с именем `sql_order_update.go` в папку `sportsstore/models/repo` с содержимым, показанным в листинге 38-5.

```
package repo

import "sportsstore/models"
```

```
func (repo *SqlRepository) SetOrderShipped(o *models.Order) {
                                        result,         err         :=
repo.Commands.UpdateOrder.ExecContext(repo.Context,
        o.Shipped, o.ID)
    if err == nil {
        rows, err :=result.RowsAffected()
        if err != nil {
                   repo.Logger.Panicf("Cannot  get  updated  ID:  %v",
err.Error())
        } else if rows != 1 {
              repo.Logger.Panicf("Got unexpected rows affected: %v",
rows)
        }
    } else {
         repo.Logger.Panicf("Cannot  exec  UpdateOrder  command:  %v",
err.Error())
    }
}
```

Листинг 38-5 Содержимое файла sql_order_update.go в папке models/repo

Реализация обработчиков запросов

Чтобы добавить поддержку управления заказами, замените содержимое файла `orders_handler.go` в папке `sportsstore/admin` содержимым, показанным в листинге 38-6.

```
package admin

import (
    "platform/http/actionresults"
    "platform/http/handling"
    "sportsstore/models"
)

type OrdersHandler struct {
    models.Repository
    handling.URLGenerator
}

func (handler OrdersHandler) GetData() actionresults.ActionResult {
         return  actionresults.NewTemplateAction("admin_orders.html",
struct {
        Orders []models.Order
         CallbackUrl string
    }{
        Orders: handler.Repository.GetOrders(),
        CallbackUrl: mustGenerateUrl(handler.URLGenerator,
```

```
            OrdersHandler.PostOrderToggle),
    })
}

func  (handler  OrdersHandler)  PostOrderToggle(ref  EditReference)
actionresults.ActionResult {
    order := handler.Repository.GetOrder(ref.ID)
    order.Shipped = !order.Shipped
    handler.Repository.SetOrderShipped(&order)
                                                                 return
actionresults.NewRedirectAction(mustGenerateUrl(handler.URLGenerator,
        AdminHandler.GetSection, "Orders"))
}
```

Листинг 38-6 Новое содержимое файла orders_handler.go в папке admin

Единственное изменение, которое будет разрешено для заказов, — это изменить значение поля **Shipped**, указывающее, что заказ был отправлен. Замените содержимое файла `database_handler.go` содержимым, показанным в листинге 38-7.

```
package admin

import (
    "platform/http/actionresults"
    "platform/http/handling"
    "sportsstore/models"
)

type DatabaseHandler struct {
    models.Repository
    handling.URLGenerator
}

func (handler DatabaseHandler) GetData() actionresults.ActionResult {
       return  actionresults.NewTemplateAction("admin_database.html",
struct {
        InitUrl, SeedUrl string
    }{
        InitUrl: mustGenerateUrl(handler.URLGenerator,
            DatabaseHandler.PostDatabaseInit),
        SeedUrl: mustGenerateUrl(handler.URLGenerator,
           DatabaseHandler.PostDatabaseSeed),
    })
}

func        (handler        DatabaseHandler)        PostDatabaseInit()
actionresults.ActionResult {
```

```
    handler.Repository.Init()
                                                                    return
actionresults.NewRedirectAction(mustGenerateUrl(handler.URLGenerator,
        AdminHandler.GetSection, "Database"))
}

func       (handler       DatabaseHandler)       PostDatabaseSeed()
actionresults.ActionResult {
    handler.Repository.Seed()
                                                                    return
actionresults.NewRedirectAction(mustGenerateUrl(handler.URLGenerator,
        AdminHandler.GetSection, "Database"))
}
```

Листинг 38-7 Новое содержимое файла database_handler.go в папке admin

Существуют методы-обработчики для каждой из операций, которые можно выполнять с базой данных, что позволит администратору быстро запустить приложение после того, как оно будет подготовлено для развертывания далее в этой главе.

Создание шаблонов

Чтобы создать шаблон, используемый для управления заказами, добавьте файл с именем `admin_orders.html` в папку `sportsstore/templates` с содержимым, показанным в листинге 38-8.

```
{{ $context := .}}

<table class="table table-sm table-striped table-bordered">
    <tr><th>ID</th><th>Name</th><th>Address</th><th/></tr>
    <tbody>
        {{ range $context.Orders }}
            <tr>
                <td>{{ .ID }}</td>
                <td>{{ .Name }}</td>
                <td>{{ .StreetAddr }}, {{ .City }}, {{ .State }},
                     {{ .Country }}, {{ .Zip }}</td>
                <td>
                                       <form  method="POST"  action="
{{$context.CallbackUrl}}">
                              <input  type="hidden"  name="id"  value="
{{.ID}}" />
                        {{ if .Shipped }}
                              <button class="btn-btn-sm btn-warning"
type="submit">
                                Ship Order
                            </button>
```

```
                            {{ else }}
                                   <button class="btn-btn-sm btn-danger"
type="submit">
                                    Mark Unshipped
                                </button>
                            {{ end }}
                        </form>
                    </td>
                </tr>
                  <tr><th colspan="2"/><th>Quantity</th><th>Product</th>
</tr>
                {{ range .Products }}
                    <tr>
                        <td colspan="2"/>
                        <td>{{ .Quantity }}</td>
                        <td>{{ .Product.Name }}</td>
                    </tr>
                {{ end }}
            {{ end }}
    </tbody>
</table>
```

Листинг 38-8 Содержимое файла admin_orders.html в папке templates

Шаблон отображает заказы в виде таблицы вместе с подробной информацией о продуктах, которые каждый из них содержит. Чтобы создать шаблон, используемый для управления базой данных, добавьте файл с именем `admin_database.html` в папку `sportsstore/templates` с содержимым, показанным в листинге 38-9.

```
{{ $context := . }}

<form method="POST">
    <button class="btn btn-danger m-3 p-2" type="submit"
            formaction="{{ $context.InitUrl}}">
        Initialize Database
    </button>
    <button class="btn btn-warning m-3 p-2" type="submit"
            formaction="{{ $context.SeedUrl}}">
        Seed Database
    </button>
</form>
```

Листинг 38-9 Содержимое файла admin_database.html в папке templates

Скомпилируйте и выполните проект, используйте браузер для запроса `http://localhost:5000/admin` и нажмите кнопку **Orders**, чтобы просмотреть заказы в базе данных и изменить статус их доставки, как показано на рисунке

38-1. Нажмите кнопку **Database**, и вы сможете сбросить и заполнить базу данных, что также показано на рисунке 38-1.

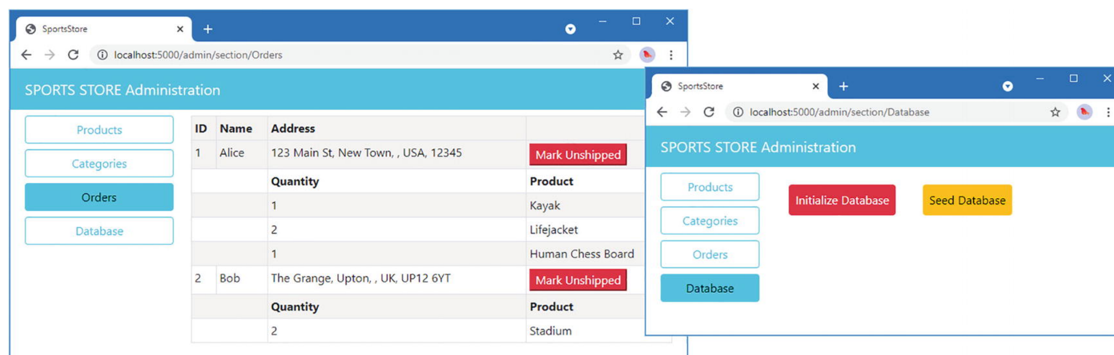

Рисунок 38-1 Завершение функций администрирования

Ограничение доступа к функциям администрирования

Предоставление открытого доступа к функциям администрирования упрощает разработку, но никогда не должно быть разрешено в рабочей среде. Теперь, когда функции администрирования завершены, пришло время убедиться, что они доступны только авторизованным пользователям.

Создание пользовательского хранилища и обработчика запросов

Как я объяснял ранее, я не реализую настоящую систему аутентификации, которую сложно сделать надежно и которая выходит за рамки этой книги. Вместо этого я собираюсь следовать подходу, аналогичному тому, который я использовал в проекте `platform`, и полагаться на аппаратные учетные данные для аутентификации пользователя. Создайте папку `sportsstore/admin/auth` и добавьте в нее файл с именем `user_store.go` с содержимым, показанным в листинге 38-10.

```
package auth

import (
    "platform/services"
    "platform/authorization/identity"
    "strings"
)

func RegisterUserStoreService() {
    err := services.AddSingleton(func () identity.UserStore {
        return &userStore{}
    })
```

```
    if (err != nil) {
        panic(err)
    }
}

var users = map[int]identity.User {
    1: identity.NewBasicUser(1, "Alice", "Administrator"),
}

type userStore struct {}

func (store *userStore) GetUserByID(id int) (identity.User, bool) {
    user, found := users[id]
    return user, found
}

func (store *userStore) GetUserByName(name string) (identity.User, bool) {
    for _, user := range users {
        if strings.EqualFold(user.GetDisplayName(), name) {
            return user, true
        }
    }
    return nil, false
}
```

Листинг 38-10 Содержимое файла user_store.go в папке admin/auth

Чтобы создать обработчик запросов аутентификации, добавьте файл с именем `auth_handler.go` в папку `sportsstore/admin` с содержимым, показанным в листинге 38-11.

```
package admin

import (
    "platform/authorization/identity"
    "platform/http/actionresults"
    "platform/http/handling"
    "platform/sessions"
)

type AuthenticationHandler struct {
    identity.User
    identity.SignInManager
    identity.UserStore
    sessions.Session
    handling.URLGenerator
}
```

```
const SIGNIN_MSG_KEY string = "signin_message"

func (handler AuthenticationHandler) GetSignIn()
actionresults.ActionResult {
     message := handler.Session.GetValueDefault(SIGNIN_MSG_KEY, "").
(string)
    return actionresults.NewTemplateAction("signin.html", message)
}

type Credentials struct {
    Username string
    Password string
}

func (handler AuthenticationHandler) PostSignIn(creds Credentials)
actionresults.ActionResult {
    if creds.Password == "mysecret" {
        user, ok := handler.UserStore.GetUserByName(creds.Username)
        if (ok) {
            handler.Session.SetValue(SIGNIN_MSG_KEY, "")
            handler.SignInManager.SignIn(user)
            return actionresults.NewRedirectAction("/admin/section/")
        }
    }
    handler.Session.SetValue(SIGNIN_MSG_KEY, "Access Denied")
                                                                return
actionresults.NewRedirectAction(mustGenerateUrl(handler.URLGenerator,
        AuthenticationHandler.GetSignIn))
}

func (handler AuthenticationHandler) PostSignOut(creds Credentials)
actionresults.ActionResult {
        handler.SignInManager.SignOut(handler.User)
    return actionresults.NewRedirectAction("/")
}
```

Листинг 38-11 Содержимое файла auth_handler.go в папке admin

Метод `GetSignIn` отображает шаблон, который запрашивает у пользователя учетные данные и отображает сообщение, которое хранится в сеансе. Метод `PostSignIn` получает учетные данные из формы и либо подписывает пользователя в приложении, либо добавляет сообщение в сеанс и перенаправляет браузер, чтобы пользователь мог повторить попытку.

Чтобы создать шаблон, позволяющий пользователям входить в приложение, добавьте файл с именем `signin.html` в папку `sportsstore/templates` с содержимым, показанным в листинге 38-12.

```
{{ layout "simple_layout.html" }}

{{ if ne . "" }}
    <h3 class="text-danger p-2">{{ . }}</h3>
{{ end }}

<form method="POST" class="m-2">
    <div class="form-group">
        <label>Username:</label>
        <input class="form-control"  name="username" />
    </div>
    <div class="form-group">
        <label>Password:</label>
         <input class="form-control" name="password" type="password"
/>
    </div>
    <div class="my-2">
              <button  class="btn  btn-secondary"  type="submit">Sign
In</button>
    </div>
</form>
```

Листинг 38-12 Содержимое файла signin.html в папке templates

Этот шаблон запрашивает у пользователя имя учетной записи и пароль, которые отправляются обратно в обработчик запросов.

Чтобы позволить пользователю выйти из приложения, добавьте файл с именем `signout_handler.go` в папку `sportsstore/admin` с содержимым, показанным в листинге 38-13.

```
package admin

import (
          "platform/authorization/identity"
          "platform/http/actionresults"
    "platform/http/handling"
)

type SignOutHandler struct {
    identity.User
    handling.URLGenerator
}

func        (handler        SignOutHandler)        GetUserWidget()
actionresults.ActionResult {
          return actionresults.NewTemplateAction("user_widget.html",
struct {
            identity.User
```

```
            SignoutUrl string}{
                handler.User,
                mustGenerateUrl(handler.URLGenerator,
                    AuthenticationHandler.PostSignOut),
            })
    }
```

Листинг 38-13 Содержимое файла signout_handler.go в папке admin

Чтобы создать шаблон, который позволит пользователю выйти из системы, добавьте файл с именем `user_widget.html` в папку `sportsstore/templates` с содержимым, показанным в листинге 38-14.

```
{{ $context := . }}

{{ if $context.User.IsAuthenticated }}
    <form method="POST" action="{{$context.SignoutUrl}}">
         <button class="btn btn-sm btn-outline-secondary text-white"
type="submit">
            Sign Out
        </button>
    </form>
{{ end }}
```

Листинг 38-14 Содержимое файла user_widget.html в папке templates

В листинге 38-15 пользовательский виджет добавляется к макету, используемому для функций администрирования.

```
...
<div class="bg-info text-white p-2">
    <div class="container-fluid">
        <div class="row">
                        <div class="col navbar-brand">SPORTS STORE
Administration</div>
            <div class="col-6 navbar-text text-end">
                {{ handler "signout" "getuserwidget" }}
            </div>
        </div>
    </div>
</div>
...
```

Листинг 38-15 Добавление виджета в файл admin.html в папке templates

Настройка приложения

В листинге 38-16 добавлен параметр конфигурации, указывающий URL-адрес, который будет использоваться при запросе ограниченного URL-адреса, что

обеспечивает более полезную альтернативу возврату кода состояния.

```
{
    "logging" : {
        "level": "debug"
    },
    "files": {
        "path": "files"
    },
    "templates": {
        "path": "templates/*.html",
        "reload": true
    },
    "sessions": {
        "key": "MY_SESSION_KEY",
        "cyclekey": true
    },
    "sql": {

        // ...setting omitted for brevity...

    },
    "authorization": {
        "failUrl": "/signin"
    }
}
```

Листинг 38-16 Добавление параметра конфигурации в файл config.json в папке sportsstore

Указанный URL-адрес предложит пользователю ввести свои учетные данные. В листинге 38-17 конвейер запросов реконфигурируется таким образом, чтобы функции администрирования были защищены.

```
package main

import (
    "sync"
    "platform/http"
    "platform/http/handling"
    "platform/services"
    "platform/pipeline"
    "platform/pipeline/basic"
    "sportsstore/store"
    "sportsstore/models/repo"
    "platform/sessions"
    "sportsstore/store/cart"
    "sportsstore/admin"
    "platform/authorization"
```

```
    "sportsstore/admin/auth"
)

func registerServices() {
    services.RegisterDefaultServices()
    //repo.RegisterMemoryRepoService()
    repo.RegisterSqlRepositoryService()
    sessions.RegisterSessionService()
    cart.RegisterCartService()
    authorization.RegisterDefaultSignInService()
    authorization.RegisterDefaultUserService()
    auth.RegisterUserStoreService()
}

func createPipeline() pipeline.RequestPipeline {
    return pipeline.CreatePipeline(
        &basic.ServicesComponent{},
        &basic.LoggingComponent{},
        &basic.ErrorComponent{},
        &basic.StaticFileComponent{},
        &sessions.SessionComponent{},

        authorization.NewAuthComponent(
            "admin",
            authorization.NewRoleCondition("Administrator"),
            admin.AdminHandler{},
            admin.ProductsHandler{},
            admin.CategoriesHandler{},
            admin.OrdersHandler{},
            admin.DatabaseHandler{},
            admin.SignOutHandler{},
        ).AddFallback("/admin/section/", "^/admin[/]?$"),

        handling.NewRouter(
            handling.HandlerEntry{ "",  store.ProductHandler{}},
            handling.HandlerEntry{ "",  store.CategoryHandler{}},
            handling.HandlerEntry{ "", store.CartHandler{}},
            handling.HandlerEntry{ "", store.OrderHandler{}},
                                   //   handling.HandlerEntry{   "admin",
admin.AdminHandler{}},
                                   //   handling.HandlerEntry{   "admin",
admin.ProductsHandler{}},
                                   //   handling.HandlerEntry{   "admin",
admin.CategoriesHandler{}},
                                   //   handling.HandlerEntry{   "admin",
admin.OrdersHandler{}},
                                   //   handling.HandlerEntry{   "admin",
admin.DatabaseHandler{}},
```

```
                                        handling.HandlerEntry{   "",
admin.AuthenticationHandler{}},
          ).AddMethodAlias("/", store.ProductHandler.GetProducts, 0,
1).
            AddMethodAlias("/products[/]?[A-z0-9]*?",
                store.ProductHandler.GetProducts, 0, 1),    )
}

func main() {
    registerServices()
    results, err := services.Call(http.Serve, createPipeline())
    if (err == nil) {
        (results[0].(*sync.WaitGroup)).Wait()
    } else {
        panic(err)
    }
}
```

Листинг 38-17 Настройка приложения в файле main.go в папке sportsstore

Скомпилируйте и запустите приложение и используйте браузер для запроса `http://localhost:5000/admin`. При появлении запроса войдите в систему как пользователь `alice` с паролем `mysecret`, и вам будет предоставлен доступ к функциям администрирования, как показано на рисунке 38-2.

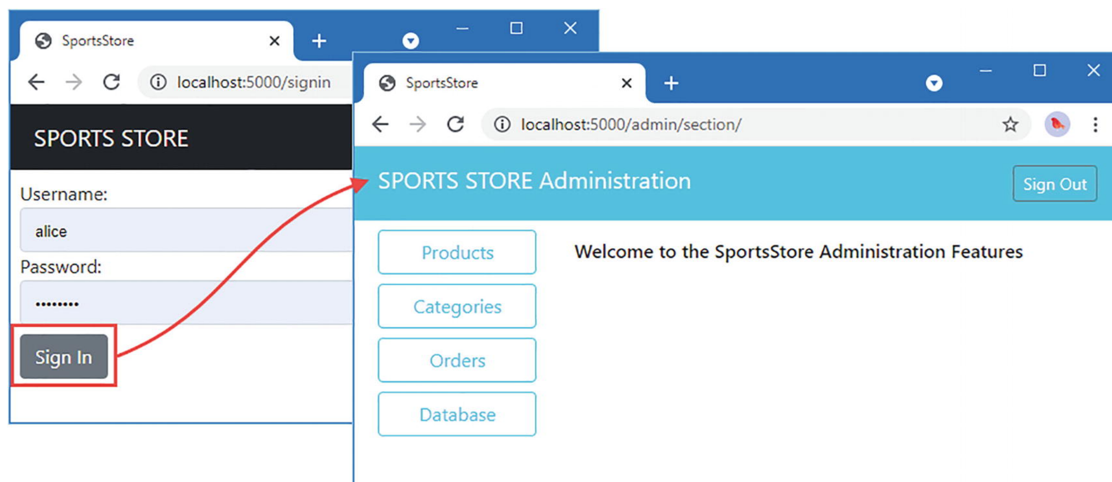

Рисунок 38-2 Вход в приложение

Создание веб-службы

Последняя функция, которую я собираюсь добавить, — это простой веб-сервис, просто чтобы показать, как это можно сделать. Я не собираюсь использовать авторизацию для защиты веб-службы, что может быть сложным процессом, зависящим от типа клиентов, которым, как ожидается, потребуется доступ. Это

означает, что любой пользователь сможет изменять базу данных. Если вы развертываете реальную веб-службу, вы можете использовать файлы cookie почти так же, как я сделал в этом примере. Если ваши клиенты не поддерживают файлы cookie, можно использовать веб-токены JSON (JWT), как описано на странице https://jwt.io.

Чтобы создать веб-службу, добавьте файл с именем `rest_handler.go` в папку `sportsstore/store` с содержимым, показанным в листинге 38-18.

```
package store

import (
    "sportsstore/models"
    "platform/http/actionresults"
    "net/http"
)

type StatusCodeResult struct {
    code int
}

func (action *StatusCodeResult) Execute(ctx *actionresults.ActionContext) error {
    ctx.ResponseWriter.WriteHeader(action.code)
    return nil
}

type RestHandler struct {
    Repository models.Repository
}

func (h RestHandler) GetProduct(id int) actionresults.ActionResult {
    return actionresults.NewJsonAction(h.Repository.GetProduct(id))
}

func (h RestHandler) GetProducts() actionresults.ActionResult {
    return actionresults.NewJsonAction(h.Repository.GetProducts())
}

type ProductReference struct {
    models.Product
    CategoryID int
}

func (h RestHandler) PostProduct(p ProductReference) actionresults.ActionResult {
    if p.ID == 0 {
        return actionresults.NewJsonAction(h.processData(p))
```

```
    } else {
        return &StatusCodeResult{ http.StatusBadRequest }
    }
}

func     (h     RestHandler)     PutProduct(p     ProductReference)
actionresults.ActionResult {
    if p.ID > 0 {
        return actionresults.NewJsonAction(h.processData(p))
    } else {
        return &StatusCodeResult{ http.StatusBadRequest }
    }
}

func (h RestHandler) processData(p ProductReference) models.Product
{
    product := p.Product
    product.Category = &models.Category {
        ID: p.CategoryID,
    }
    h.Repository.SaveProduct(&product)
    return h.Repository.GetProduct(product.ID)
}
```

Листинг 38-18 Содержимое файла rest_handler.go в папке store

Структура `StatusCodeResult` — это результат действия, который отправляет код состояния HTTP, что полезно для веб-служб. Обработчик запросов определяет методы, которые позволяют извлекать один продукт и все продукты с помощью запросов GET, создавать новые продукты с помощью запросов POST и изменять существующие продукты с помощью запросов PUT. В листинге 38-19 регистрируется новый обработчик с префиксом `/api`.

```
...
handling.NewRouter(
     handling.HandlerEntry{ "",  store.ProductHandler{}},
     handling.HandlerEntry{ "",  store.CategoryHandler{}},
     handling.HandlerEntry{ "", store.CartHandler{}},
     handling.HandlerEntry{ "", store.OrderHandler{}},
     handling.HandlerEntry{ "", admin.AuthenticationHandler{}},
     handling.HandlerEntry{ "api", store.RestHandler{}},
).AddMethodAlias("/", store.ProductHandler.GetProducts, 0, 1).
    AddMethodAlias("/products[/]?[A-z0-9]*?",
    store.ProductHandler.GetProducts, 0, 1),
...
```

Листинг 38-19 Регистрация обработчика в файле main.go в папке sportsstore

Скомпилируйте и запустите проект. Откройте новую командную строку и выполните команду, показанную в листинге 38-20, чтобы добавить новый продукт в базу данных.

```
curl --header "Content-Type: application/json" --request POST --data
'{"name" : "Jet Engine","description": "Paddling is hard work",
"price":650, "categoryid":1}' http://localhost:5000/api/product
```

Листинг 38-20 Добавление нового продукта

Если вы используете Windows, откройте новое окно PowerShell и выполните команду, показанную в листинге 38-21.

```
Invoke-RestMethod http://localhost:5000/api/product -Method POST -
Body (@{ Name="Jet Engine"; Description="Paddling is hard work";
Price=650; CategoryId=1 } | ConvertTo-Json) -ContentType
"application/json"
```

Листинг 38-21 Добавление нового продукта в Windows

Чтобы увидеть эффект изменения, выполните команду, показанную в листинге 38-22.

```
curl http://localhost:5000/api/product/10
```

Листинг 38-22 Запрос данных

Если вы используете Windows, выполните команду, показанную в листинге 38-23, в окне PowerShell.

```
Invoke-RestMethod http://localhost:5000/api/product/10
```

Листинг 38-23 Запрос данных в Windows

Вы также можете использовать браузер, чтобы увидеть эффект изменения. Запрос `http://localhost:5000/admin`. Войдите в систему как пользователь `alice` с паролем `mysecret` и нажмите кнопку `Products`. Последняя строка таблицы будет содержать продукт, созданный с помощью веб-сервиса, как показано на рисунке 38-3.

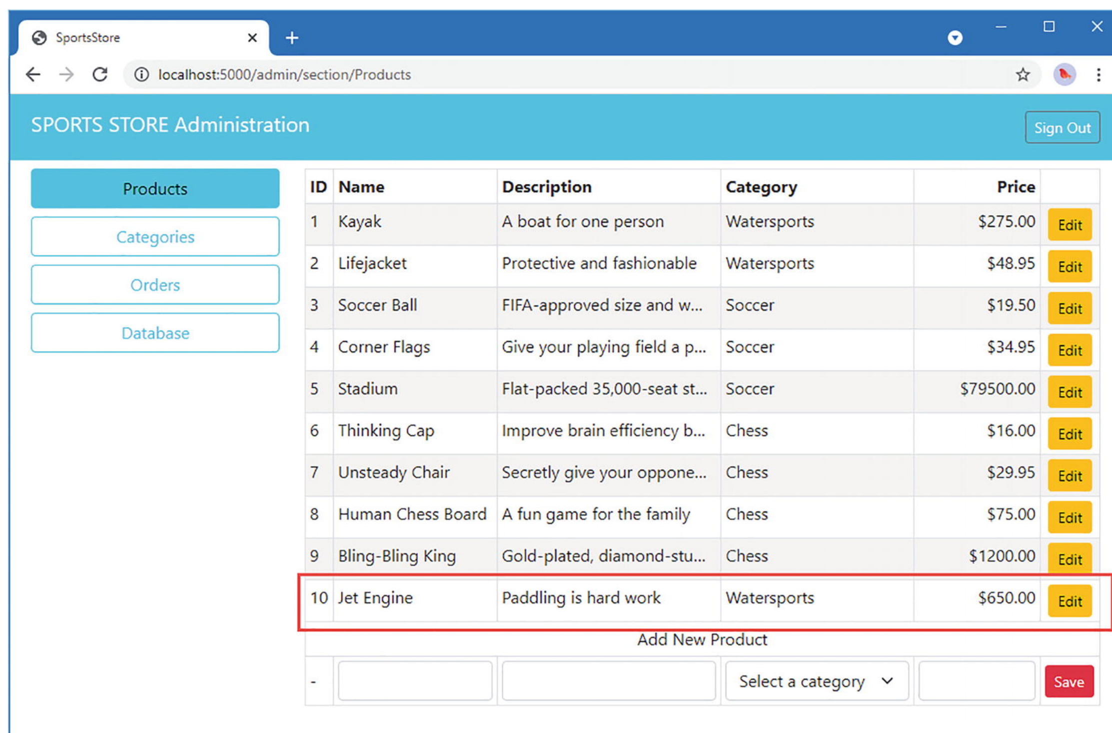

Рисунок 38-3 Проверка эффекта изменения базы данных

Подготовка к развертыванию

В этом разделе я подготовлю приложение SportsStore и создам контейнер, который можно развернуть в рабочей среде. Это не единственный способ развертывания приложения Go, но я выбрал контейнеры Docker, потому что они широко используются и подходят для веб-приложений. Это не полное руководство по развертыванию, но оно даст вам представление о процессе подготовки приложения.

Установка сертификатов

Первый шаг — добавить сертификаты, которые будут использоваться для HTTPS. Как объяснялось в главе 24, вы можете создать самозаверяющий сертификат, если у вас нет реального доступного сертификата, или вы можете использовать файлы сертификатов из репозитория GitHub для этой книги (которые содержат самоподписанный сертификат, который я создал).

Настройка приложения

Наиболее важным изменением является изменение конфигурации приложения для отключения функций, которые удобны при разработке, но которые не следует использовать при развертывании, а также для включения HTTPS, как показано в листинге 38-24.

```
{
    "logging" : {
        "level": "information"
    },
    "files": {
        "path": "files"
    },
    "templates": {
        "path": "templates/*.html",
        "reload": false
    },
    "sessions": {
        "key": "MY_SESSION_KEY",
        "cyclekey": false
    },
    "sql": {
        "connection_str": "store.db",
        "always_reset": false,
        "commands": {
            "Init": "sql/init_db.sql",
            "Seed": "sql/seed_db.sql",
            "GetProduct": "sql/get_product.sql",
            "GetProducts": "sql/get_products.sql",
            "GetCategories": "sql/get_categories.sql",
            "GetPage": "sql/get_product_page.sql",
            "GetPageCount": "sql/get_page_count.sql",
            "GetCategoryPage": "sql/get_category_product_page.sql",
                                                 "GetCategoryPageCount":
"sql/get_category_product_page_count.sql",
            "GetOrder": "sql/get_order.sql",
            "GetOrderLines": "sql/get_order_lines.sql",
            "GetOrders": "sql/get_orders.sql",
            "GetOrdersLines": "sql/get_orders_lines.sql",
            "SaveOrder": "sql/save_order.sql",
            "SaveOrderLine": "sql/save_order_line.sql",
            "SaveProduct":          "sql/save_product.sql",
            "UpdateProduct":        "sql/update_product.sql",
            "SaveCategory":         "sql/save_category.sql",
            "UpdateCategory":       "sql/update_category.sql",
            "UpdateOrder":          "sql/update_order.sql"
        }
    },
    "authorization": {
        "failUrl": "/signin"
    },
    "http": {
        "enableHttp": false,
        "enableHttps": true,
```

```
        "httpsPort": 5500,
        "httpsCert": "certificate.cer",
        "httpsKey": "certificate.key"
    }
}
```

Листинг 38-24 Изменение настроек в файле config.json в папке sportsstore

Убедитесь, что значения, указанные вами для свойств `httpsCert` и `httpsKey`, соответствуют именам ваших файлов сертификатов и что файлы сертификатов находятся в папке `sportsstore`.

Сборка приложения

Контейнеры Docker работают под управлением Linux. Если вы используете Windows, вы должны выбрать Linux в качестве цели сборки, выполнив команды, показанные в листинге 38-25, в окне PowerShell, чтобы настроить инструменты сборки Go. Это не требуется, если вы используете Linux.

```
$Env:GOOS = "linux"; $Env:GOARCH = "amd64"
```

Листинг 38-25 Установка Linux в качестве цели сборки

Запустите команду, показанную в листинге 38-26, в папке `sportsstore`, чтобы собрать приложение.

```
go build
```

Листинг 38-26 Сборка приложения

> **Примечание**
> Если вы пользователь Windows, вы можете вернуться к обычной сборке Windows с помощью следующей команды: `$Env:GOOS = "windows"; $Env:GOARCH = "amd64"`. Но не запускайте эту команду, пока не завершите процесс развертывания.

Установка рабочего стола Docker

Перейдите на `docker.com`, загрузите и установите пакет Docker Desktop. Следуйте процессу установки, перезагрузите компьютер и выполните команду, показанную в листинге 38-27, чтобы убедиться, что Docker установлен и находится на вашем пути. (Похоже, что процесс установки Docker часто меняется, поэтому я не буду подробно рассказывать об этом процессе.)

> **Примечание**

> Вам нужно будет создать учетную запись на docker.com, чтобы загрузить установщик.

```
docker --version
```

Листинг 38-27 Проверка установки Docker Desktop

Creating the Docker Configuration Files

Чтобы создать конфигурацию Docker для приложения, создайте файл с именем Dockerfile в папке sportsstore с содержимым, показанным в листинге 38-28.

```
FROM alpine:latest

COPY sportsstore /app/
COPY templates /app/templates
COPY sql/* /app/sql/
COPY files/* /app/files/
COPY config.json /app/
COPY certificate.* /app/

EXPOSE 5500
WORKDIR /app
ENTRYPOINT ["./sportsstore"]
```

Листинг 38-28 Содержимое файла Dockerfile в папке sportsstore

Эти инструкции копируют приложение и его вспомогательные файлы в образ Docker и настраивают его выполнение. Следующим шагом является создание образа с помощью инструкций, определенных в листинге 38-28. Запустите команду, показанную в листинге 38-29, в папке sportsstore, чтобы создать образ Docker.

```
docker build  --tag go_sportsstore .
```

Листинг 38-29 Создание образа

Убедитесь, что вы остановили все другие экземпляры приложения, и выполните команду, показанную в листинге 38-30, чтобы создать новый контейнер из образа и выполнить его.

```
docker run -p 5500:5500 go_sportsstore
```

Листинг 38-30 Создание и запуск контейнера

Дайте контейнеру некоторое время для запуска, а затем используйте браузер для запроса https://localhost:5500, что даст ответ, показанный на рисунке

38-4. Если вы использовали самозаверяющий сертификат, возможно, вам придется пройти через предупреждение системы безопасности.

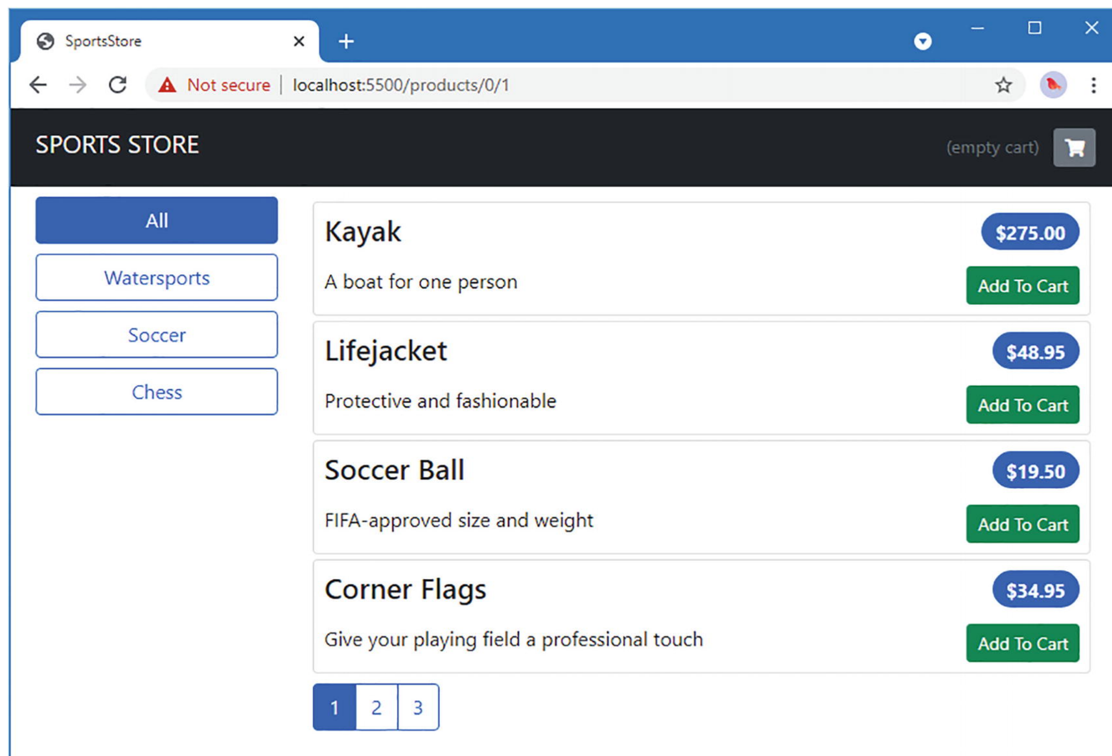

Рисунок 38-4 Запуск приложения в контейнере

Теперь приложение готово к развертыванию. Чтобы остановить контейнер — и любой другой работающий контейнер — выполните команду, показанную в листинге 38-31.

```
docker kill $(docker ps -q)
```

Листинг 38-31 Остановка контейнеров

Резюме

В этой главе я завершил приложение SportsStore, завершив функции администрирования, настроив авторизацию и создав базовый веб-сервис, прежде чем подготовить приложение для развертывания с использованием контейнера Docker.

Это все, что я могу рассказать вам о Go. Я могу только надеяться, что вам понравилось читать эту книгу так же, как мне понравилось ее писать, и я желаю вам всяческих успехов в ваших проектах на Go.