

Адам Фриман

**Pro Go**

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

****

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)**

**}**

}

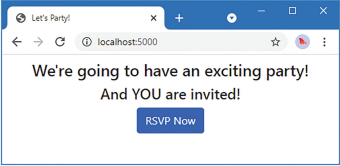
***Листинг 1-17*** Создание HTTP-сервера в файле main.go в папке partyinvites

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 {},

})

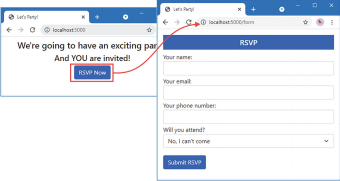
...

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.

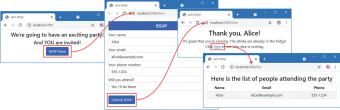
Остальные операторы используют значение поля WillAttend для выбора шаблона, который будет представлен пользователю.

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

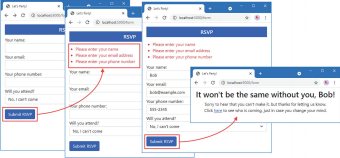
Приложение получит пустую строку ("") из запроса, если пользователь не предоставит значение для поля формы. Новые операторы в листинге 1-23 проверяют поля Name, EMail и Phone и добавляют сообщение к срезу строк для каждого поля, не имеющего значения. Я использую встроенную функцию len, чтобы получить 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) {**

category, price) **} else {**

Printfln("Row: %v %v %v %v", id, name,

**}**

}

} else {

**Printfln("Scan error: %v", scanErr) break**

}

}...

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()**

**tx.Stmt(changeProductCategory)**

**preparedStatement :=**

**for \_, id := range productIDs {**

**preparedStatement.Exec(newID, id) if (err == nil) {**

**changeResult, err :=**

**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

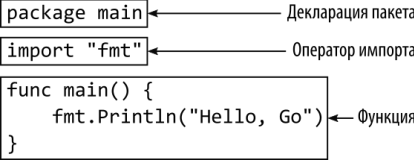
import "fmt"

func main() {

fmt.Println("Hello, Go") }

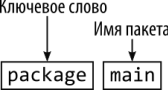
***Листинг 3-1*** Содержимое файла main.go в папке tools

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**

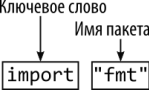
Первый оператор — это объявление пакета. Пакеты используются для группировки связанных функций, и каждый файл кода должен объявлять пакет, к которому принадлежит его содержимое. В объявлении пакета используется ключевое слово package, за которым следует имя пакета, как показано на рисунке 3-2. Оператор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



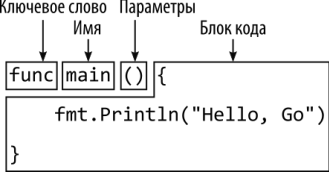
***Рисунок 3-3*** Объявление зависимости пакета

**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.

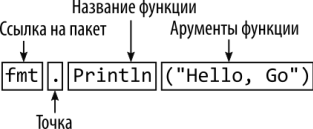
Функция в листинге 3-1 не определяет никаких параметров, что обозначено пустыми скобками и не дает результата. Я опишу более

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

012

34

**Использование отладчика**

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

dlv debug main.go

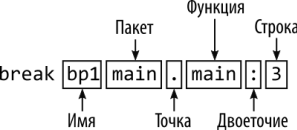
***Листинг 3-12*** Запуск отладчика

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:

23

4Process 3160 has exited with status 0

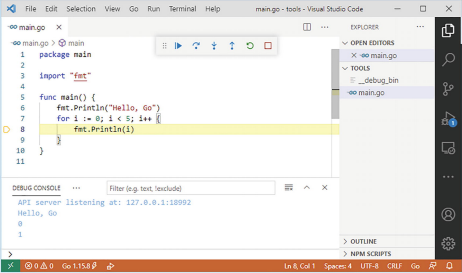
Условие, которое я указал для точки останова, больше не выполняется, поэтому программа работает до тех пор, пока не завершится. Используйте команду exit, чтобы выйти из отладчика и вернуться в командную строку.

**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).