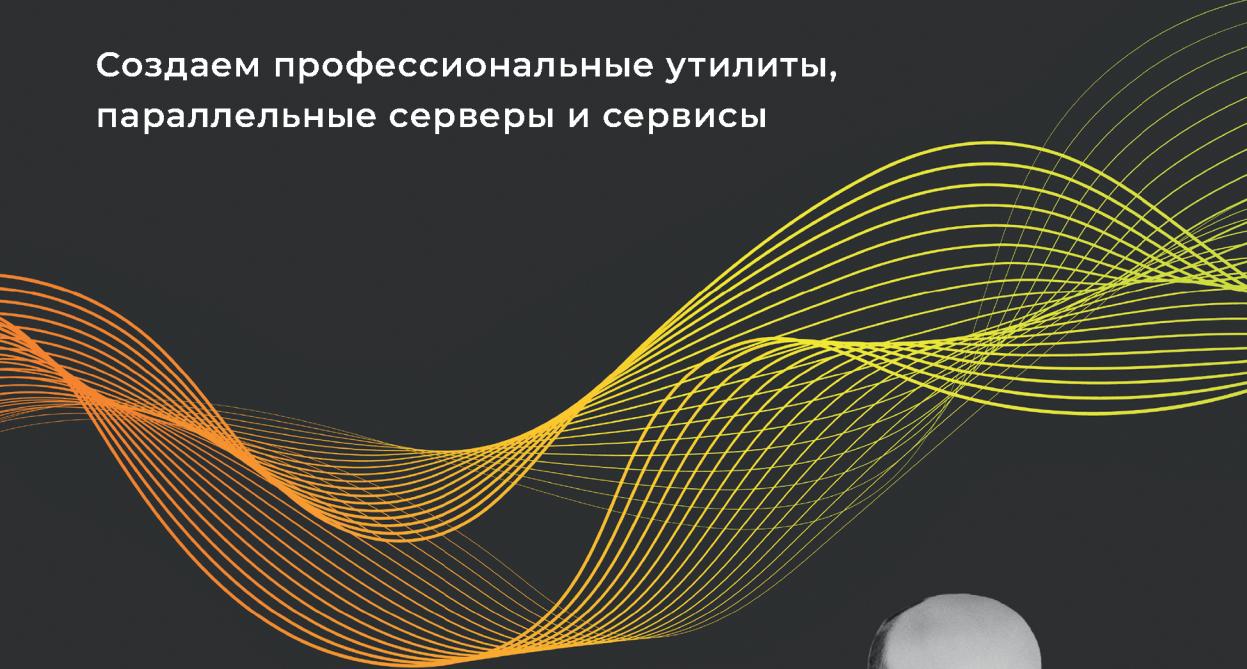




EXPERT INSIGHT

Golang для профи

Создаем профессиональные утилиты,
параллельные серверы и сервисы



Третье издание



Михалис Цукалос

Packt

Mastering Go

Third Edition

Harness the power of Go to build professional utilities
and concurrent servers and services

Mihalis Tsoukalos

Packt

BIRMINGHAM - MUMBAI

Golang для профи

Третье издание

Создаем профессиональные утилиты,
параллельные серверы и сервисы

Михалис Цукалос



Санкт-Петербург · Москва · Минск

2024

ББК 32.988.02-018.1

УДК 004.43

Ц85

Цукалос Михалис

- Ц85 *Golang для профи: Создаем профессиональные утилиты, параллельные серверы и сервисы.* 3-е изд. — СПб.: Питер, 2024. — 624 с.: ил. — (Серия «Для профессионалов»).

ISBN 978-5-4461-1999-8

Язык Go — это простой и понятный язык для создания высокопроизводительных систем будущего. Используйте Go в реальных производственных системах. В новое издание включены такие темы, как создание серверов и клиентов RESTful, знакомство с дженериками Go и разработка серверов и клиентов gRPC.

Третье издание «Golang для профи» исследует практические возможности Go и описывает такие продвинутые темы, как параллелизм и работа сборщика мусора Go, использование Go с Docker, разработка мощных утилит командной строки, обработка данных в формате JSON (JavaScript Object Notation) и взаимодействие с базами данных. Кроме того, книга дает дополнительные сведения о работе внутренних механизмов Go, знание которых позволит оптимизировать код на Go и использовать типы и структуры данных новыми и необычными способами.

Также охватываются некоторые нюансы и идиомы языка Go, предлагаются упражнения и приводятся ссылки на ресурсы для закрепления полученных знаний.

Станьте опытным программистом на Go, создавая системы и внедряя передовые методы программирования на Go в свои проекты!

16+ (В соответствии с Федеральным законом от 29 декабря 2010 г. № 436-ФЗ.)

ББК 32.988.02-018.1

УДК 004.43

Права на издание получены по соглашению с Packt Publishing. Все права защищены. Никакая часть данной книги не может быть воспроизведена в какой бы то ни было форме без письменного разрешения владельцев авторских прав.

Информация, содержащаяся в данной книге, получена из источников, рассматриваемых издательством как надежные. Тем не менее, имея в виду возможные человеческие или технические ошибки, издательство не может гарантировать абсолютную точность и полноту приводимых сведений и не несет ответственности за возможные ошибки, связанные с использованием книги. В книге возможны упоминания организаций, деятельность которых запрещена на территории Российской Федерации, таких как Meta Platforms Inc., Facebook, Instagram и др. Издательство не несет ответственности за доступность материалов, ссылки на которые вы можете найти в этой книге. На момент подготовки книги к изданию все ссылки на интернет-ресурсы были действующими.

ISBN 978-1801079310 англ.

© Packt Publishing 2021.

First published in the English language under the title 'Mastering Go — Third Edition — (9781801079310)'

ISBN 978-5-4461-1999-8

© Перевод на русский язык ООО «Прогресс книга», 2023

© Издание на русском языке, оформление ООО «Прогресс книга», 2023

© Серия «Для профессионалов», 2023

Краткое содержание

Об авторе	17
О научном редакторе	18
Предисловие	19
Глава 1. Краткое введение в Go.....	24
Глава 2. Основные типы данных Go	68
Глава 3. Составные типы данных	118
Глава 4. Рефлексия и интерфейсы	148
Глава 5. Пакеты и функции Go.....	194
Глава 6. Даём указания системе UNIX.....	254
Глава 7. Параллельное выполнение в Go	313
Глава 8. Создание веб-сервисов	370
Глава 9. Работа с TCP/IP и WebSocket.....	423
Глава 10. Работа с REST API	461
Глава 11. Тестирование и профилирование кода.....	523
Глава 12. Работа с gRPC	575
Глава 13. Джениерики Go	590
Приложение. Сборщик мусора Go	607

Оглавление

Об авторе	17
О научном редакторе	18
Предисловие	19
Для кого эта книга	19
Структура издания	19
Как сделать книгу максимально полезной	21
Файлы с примерами кода	22
Цветные иллюстрации	22
Условные обозначения	22
От издательства	23
Глава 1. Краткое введение в Go	24
Введение в Go	25
История Go	26
Почему UNIX, а не Windows	27
Преимущества Go	28
Утилиты go doc и godoc	30
Hello World!	31
Введение в функции	32
Введение в пакеты	32
Запуск Go-кода	33
Компиляция Go-кода	33
Использование Go в качестве языка скриптов	33
Важные правила форматирования и кодирования	34
Важные особенности Go	35
Определение и использование переменных	36
Управление ходом выполнения программы	39
Итерации с помощью циклов for и range	41

Получение пользовательского ввода	43
Использование переменных ошибок для различения типов входных данных	48
Знакомство с моделью параллелизма в Go	49
Разработка утилиты <code>which(1)</code> на Go	51
Вывод информации в лог	54
Функции <code>log.Fatal()</code> и <code>log.Panic()</code>	56
Запись в пользовательский файл журнала	58
Вывод номеров строк в записях журнала	59
Обзор Go-дженериков	61
Разработка базового приложения телефонной книги	62
Упражнения	66
Резюме	66
Дополнительные ресурсы	67
Глава 2. Основные типы данных Go	68
Тип данных <code>error</code>	69
Числовые типы данных	72
Нечисловые типы данных	75
Строки, символы и руны	75
Время и даты	82
Go-константы	87
Генератор констант <code>iota</code>	88
Группировка схожих данных	90
Массивы	90
Срезы	91
Указатели	108
Генерация случайных чисел	112
Генерация случайных строк	113
Генерация безопасных случайных чисел	114
Обновление приложения телефонной книги	115
Упражнения	117
Резюме	117
Дополнительные ресурсы	117

Глава 3. Составные типы данных	118
Карты	119
Сохранение в карту nil	120
Перебор карт	122
Структуры	123
Определение новых структур	123
Использование ключевого слова new	124
Срезы структур	126
Регулярные выражения и сопоставление с образцом	128
О регулярных выражениях Go	129
Сопоставление имен и фамилий	130
Сопоставление целых чисел	131
Сопоставление полей записи	132
Улучшение приложения телефонной книги	133
Работа с CSV-файлами	133
Добавление индекса	137
Улучшенная версия приложения для телефонной книги	138
Упражнения	146
Резюме	147
Дополнительные ресурсы	147
Глава 4. Рефлексия и интерфейсы	148
Рефлексия	149
Изучение внутренней структуры Go-структур	151
Изменение значений структуры с использованием рефлексии	153
Три недостатка рефлексии	154
Методы типа	155
Создание методов типа	155
Использование методов типа	156
Интерфейсы	159
Интерфейс sort.Interface	162
Пустой интерфейс	164
Утверждения типа и переключатели типов	166
Карта map[string]interface{}	170

Тип данных <code>eggog</code>	173
Написание собственных интерфейсов	176
Работа с двумя различными форматами файлов CSV	182
Объектно-ориентированное программирование в Go	185
Обновление приложения телефонной книги.	189
Настройка значения CSV-файла	189
Использование пакета <code>sort</code>	190
Упражнения	192
Резюме	192
Дополнительные ресурсы	193
Глава 5. Пакеты и функции Go	194
Go-пакеты	195
Скачивание Go-пакетов	195
Функции	198
Анонимные функции	198
Функции, возвращающие несколько значений	199
Возвращаемые значения функции могут иметь имя	200
Функции, которые принимают другие функции в качестве параметров	201
Функции могут возвращать другие функции	202
Функции с переменным количеством параметров	204
Ключевое слово <code>defer</code>	208
Разработка собственных пакетов	210
Функция <code>init()</code>	211
Порядок исполнения	212
Использование GitHub для хранения Go-пакетов	213
Пакет для работы с базой данных	215
Знакомство с базой данных	216
Хранение Go-пакета	220
Дизайн Go-пакета	221
Реализация Go-пакета	223
Тестирование Go-пакета	230
Модули	233
Создание более качественных пакетов	234

Создание документации	235
GitLab Runners и Go	242
Начальная версия файла конфигурации	243
Окончательная версия конфигурационного файла	245
GitHub Actions и Go	246
Хранение секретов в GitHub	248
Окончательная версия конфигурационного файла	248
Утилиты управления версиями	250
Упражнения	252
Резюме	252
Дополнительные ресурсы	253
Глава 6. Даем указания системе UNIX	254
stdin, stdout и stderr	255
Процессы UNIX	256
Обработка сигналов UNIX	256
Файловый ввод-вывод	260
Интерфейсы io.Reader и io.Writer	260
Правильное и неправильное использование io.Reader и io.Writer	261
Буферизованный и небуферизованный файловый ввод-вывод	265
Чтение текстовых файлов	266
Чтение текстового файла построчно	266
Чтение текстового файла слово за словом	267
Чтение текстового файла символ за символом	269
Чтение из /dev/random	270
Считывание определенного объема данных из файла	271
Запись в файл	272
Работа с JSON	275
Использование Marshal() и Unmarshal()	275
Структуры и JSON	277
Чтение и запись данных JSON в виде потоков	278
Структурный вывод записи JSON	279
Работа с XML	281
Преобразование JSON в XML и обратно	282

Работа с YAML	283
Пакет <code>viper</code>	285
Использование флагов командной строки	286
Чтение конфигурационных файлов JSON	289
Пакет <code>cobra</code>	292
Утилита с тремя командами	294
Добавление флагов командной строки	294
Создание псевдонимов команд	295
Создание подкоманд	296
Поиск циклов в файловой системе UNIX	297
Новое в Go 1.16	300
Встраивание файлов	300
<code>ReadDir</code> и <code>DirEntry</code>	303
Пакет <code>io/fs</code>	304
Обновление приложения телефонной книги	306
Использование <code>cobra</code>	307
Хранение и загрузка данных в формате JSON	308
Реализация команды <code>delete</code>	308
Реализация команды <code>insert</code>	309
Реализация команды <code>list</code>	309
Реализация команды <code>search</code>	310
Упражнения	311
Резюме	312
Дополнительные ресурсы	312
Глава 7. Параллельное выполнение в Go	313
Процессы, потоки и горутины	314
Планировщик Go	315
Переменная среды <code>GOMAXPROCS</code>	317
Параллелизм и распараллеливание	319
Горутины	319
Создание горутины	320
Создание нескольких горутин	321

Ожидание завершенияgorутин	321
ЧтоДелать, если количество вызовов Add() и Done() разное	323
Создание нескольких файлов с помощью горутин	325
Каналы	326
Запись в канал и чтение из него	326
Прием из закрытого канала	329
Каналы как параметры функций	330
Состояния гонки	331
Ключевое слово select	333
Установка тайм-аута горутины	335
Ограничение времени выполнения горутины — внутри main()	335
Ограничение времени выполнения горутины — вне main()	337
Еще раз о каналах в Go	338
Буферизованные каналы	339
Nil-каналы	340
Пулы рабочих процессов	342
Сигнальные каналы	345
Общая память и общие переменные	348
Тип sync.Mutex	349
Тип sync.RWMutex	351
Пакет atomic	354
Совместное использование памяти с помощью горутин	356
Закрытые переменные и оператор go	358
Пакет context	360
Пакет semaphore	365
Упражнения	368
Резюме	368
Дополнительные ресурсы	369
Глава 8. Создание веб-сервисов	370
Пакет net/http	371
Тип http.Response	371
Тип http.Request	372
Тип http.Transport	372

Создание веб-сервера	373
Обновление приложения телефонной книги	377
Определение API	377
Реализация обработчиков	378
Предоставление метрик для Prometheus	386
Пакет runtime/metrics	387
Предоставление метрик	389
Чтение метрик	396
Ввод метрик в Prometheus	397
Визуализация метрик Prometheus в Grafana	401
Разработка веб-клиентов	403
Использование http.NewRequest() для улучшения работы клиента	404
Создание клиента для сервиса телефонной книги	407
Создание файловых серверов	412
Загрузка содержимого приложения телефонной книги	414
Время ожидания HTTP-соединений	416
Использование функции SetDeadline()	416
Установка периода ожидания на стороне клиента	417
Установка времени ожидания на стороне сервера	420
Упражнения	421
Резюме	421
Дополнительные ресурсы	422
Глава 9. Работа с TCP/IP и WebSocket	423
TCP/IP	424
Пакет net	426
Разработка TCP-клиента	426
Разработка TCP-клиента с помощью net.Dial()	426
Разработка TCP-клиента, использующего net.DialTCP()	428
Разработка TCP-сервера	430
Разработка TCP-сервера с помощью net.Listen()	430
Разработка TCP-сервера, использующего net.ListenTCP()	433
Разработка UDP-клиента	435

Разработка UDP-сервера	437
Разработка параллельных TCP-серверов.....	440
Работа с доменными сокетами UNIX	442
Сервер на сокетах домена UNIX	442
Клиент сокета домена UNIX.....	444
Создание сервера WebSocket	447
Реализация сервера	448
Создание клиента WebSocket	455
Упражнения	459
Резюме	460
Дополнительные ресурсы	460
Глава 10. Работа с REST API.....	461
Введение в REST	462
Разработка серверов и клиентов RESTful	464
Сервер RESTful	465
Клиент RESTful	473
Создание функционального сервера RESTful	481
REST API	482
Использование пакета gorilla/mux	483
Использование подмаршрутизаторов	484
Работа с базой данных	484
Тестирование пакета restdb	490
Реализация сервера RESTful	491
Тестирование сервера RESTful	495
Создание клиента RESTful	499
Создание структуры клиента командной строки	500
Реализация клиентских команд RESTful	501
Использование клиента RESTful	505
Работа с несколькими версиями REST API	507
Загрузка и скачивание двоичных файлов	507
Использование Swagger для документации REST API	512
Документирование REST API	514

Создание файла документации	517
Обслуживание файла документации	518
Упражнения	521
Резюме	521
Дополнительные ресурсы	521
Глава 11. Тестирование и профилирование кода	523
Оптимизация кода	524
Оценка производительности	525
Переписывание функции <code>main()</code> для более качественного тестирования...	526
Анализ производительности буферизованной записи и чтения	527
Утилита <code>benchstat</code>	531
Неправильно определенные бенчмарк-функции	532
Профилирование кода	533
Профилирование приложения командной строки	534
Профилирование HTTP-сервера	537
Веб-интерфейс профилировщика Go	539
Утилита <code>go tool trace</code>	540
Трассировка веб-сервера со стороны клиента	542
Посещение всех маршрутов веб-сервера	544
Тестирование Go-кода	548
Написание тестов для <code>./ch03/intRE.go</code>	549
Функция <code>TempDir()</code>	551
Функция <code>Cleanup()</code>	551
Пакет <code>testing/quick</code>	553
Тайм-аут тестов	555
Покрытие тестового кода	556
Поиск недостигшего Go-кода	559
Тестирование HTTP-сервера с помощью серверной части базы данных	561
Фаззинг	566
Кросс-компиляция	567
Использование директивы <code>go:generate</code>	569
Создание примеров функций	571

Упражнения	573
Резюме	573
Дополнительные ресурсы	574
Глава 12. Работа с gRPC	575
Введение в gRPC	575
Буферы протокола	576
Определение файла языка определения интерфейса	577
Разработка сервера gRPC	581
Разработка клиента gRPC	584
Упражнения	588
Резюме	588
Дополнительные ресурсы	589
Глава 13. Джениерики Go	590
Введение в дженерики	591
Ограничения	593
Определение новых типов данных с помощью дженериков	596
Интерфейсы и дженерики	600
Рефлексия и дженерики	602
Упражнения	604
Резюме	605
Дополнительные ресурсы	605
Приложение. Сборщик мусора Go	607
Куча и стек	607
Сборка мусора	611
Алгоритм трех цветов	614
Подробнее о работе сборщика мусора в Go	617
Карты, срезы и сборщик мусора Go	618
Сравнение эффективности представленных техник	621
Дополнительные ресурсы	622

Об авторе

Михалис Цукалос — системный инженер UNIX, который увлекается написанием технических текстов. Автор книг *Go Systems Programming* и *Mastering Go*¹, как первого, так и второго изданий. Получил степень бакалавра математики в Университете Патр и степень магистра информационных технологий в Университетском колледже Лондона. Написал более 300 технических статей для различных журналов, включая *Sys Admin*, *MacTech*, *Linux User and Developer*, *Usenix login*; *Linux Format* и *Linux Journal*. В круг научных интересов Михалиса входят временные ряды, базы данных и индексирование.

Вы можете связаться с автором по адресу <https://www.mtsoukalos.eu/> и @mactsouk.

Поскольку написание книги — командная работа, я хотел бы поблагодарить за помощь сотрудников издательства Packt Publishing. Спасибо Амиту Рамадасу за ответы на все мои вопросы, Шайлешу Джайну — за то, что убедил начать работу над третьим изданием Mastering Go, Эдварду Докси — за полезные комментарии и предложения и Дереку Паркеру — за его хорошую работу.

Наконец, я хотел бы поблагодарить вас, читателя, за выбор этой книги. Надеюсь, она окажется вам полезной.

¹ Цукалос M. Golang для профи. Работа с сетью, многопоточность, структуры данных и машинное обучение с Go. — СПб.: Питер, 2020.

○ научном редакторе

Дерек Паркер — инженер-программист в Red Hat. Создатель отладчика Delve для Go и автор компилятора Go, компоновщика и стандартной библиотеки. Является автором проектов с открытым исходным кодом и специалистом по сопровождению ПО, работал над множеством проектов — от интерфейсного JavaScript до низкоуровневого ассемблерного кода.

Я хотел бы поблагодарить мою жену Эрику и двух наших замечательных детей за то, что они дали мне возможность работать над этим проектом.

Предисловие

Книга, которую вы сейчас читаете, называется «Golang для профи: Создаем профессиональные утилиты, параллельные серверы и сервисы» (3-е издание), и она вся о том, как стать лучшим разработчиком на Go! Если у вас есть второе издание, то не выбрасывайте его — Go не так уж сильно изменился, и второе издание по-прежнему полезно. Однако третье издание во многих аспектах лучше!

Добавлено много интересных новых тем, включая написание сервисов RESTful, работу с протоколом WebSocket и использование GitHub Actions и GitLab Actions для проектов Go, а также совершенно новая глава о дженериках и разработке множества полезных утилит. Кроме того, я постарался сделать это издание меньше, чем второе, и улучшить структуру книги, чтобы облегчить вам ее чтение и ускорить его.

Я также постарался включить нужное количество теории и практики — но только вы, читатель, можете сказать, насколько мне это удалось! Попробуйте выполнить упражнения в конце каждой главы и не стесняйтесь обращаться ко мне с идеями, которые могут еще больше улучшить будущие издания этой книги!

Для кого эта книга

Книга предназначена для программистов на Go среднего уровня, которые хотят улучшить свои навыки и перевести их на новый уровень. Она также будет полезна опытным разработчикам на других языках программирования, которые хотят изучить Go, не углубляясь в основы программирования.

Структура издания

Глава 1 начинается с рассказа об истории возникновения, важных свойствах и преимуществах Go. После этого описываются утилиты `godoc` и `go doc` и объясняется, как компилировать и исполнять программы на Go. Далее в главе рассказывается о выводе данных и получении пользовательского ввода, работе

с аргументами командной строки и использовании файлов журнала. Наконец, мы разработаем базовую версию приложения для телефонной книги, которую будем совершенствовать в следующих главах.

В главе 2 рассматриваются основные типы данных Go, как числовые, так и нечисловые, а также массивы и срезы, позволяющие группировать данные одного типа. В ней также рассматриваются указатели Go, константы и работа с датами и временем. Последняя часть главы посвящена генерации случайных чисел и заполнению приложения телефонной книги случайными данными.

Глава 3 начинается с карт, после чего переходит к структурам и ключевому слову `struct`. Кроме того, в ней вы найдете информацию о регулярных выражениях, сопоставлении с образцами и работе с CSV-файлами. Наконец, мы улучшим приложение телефонной книги, добавив в него постоянное хранение данных.

Глава 4 посвящена рефлексии, интерфейсам и методам типов — функциям, прикрепленным к типам данных. В ней также описывается использование интерфейса `sort.Interface` для сортировки фрагментов, использование пустого интерфейса, а также представлены утверждения типа, переключатели типа и тип данных `error`. Дополнительно, прежде чем улучшать приложение телефонной книги, мы обсудим, как Go может имитировать некоторые объектно-ориентированные концепции.

Глава 5 посвящена пакетам, модулям и функциям, которые являются основными элементами пакетов. Среди прочего мы создадим пакет Go, позволяющий взаимодействовать с базой данных PostgreSQL, документацию для него, а также объясним не всегда простое использование ключевого слова `defer`. В этой главе также содержится информация о том, как использовать GitLab Runners и GitHub Actions для автоматизации, и о том, как создать образ Docker для двоичного файла Go.

Глава 6 посвящена системному программированию и содержит такие темы, как работа с аргументами командной строки, обработка сигналов UNIX, ввод и вывод файлов, интерфейсы `io.Reader` и `io.Writer` и использование пакетов `viper` и `cobra`. Кроме того, мы поговорим о работе с файлами JSON, XML и YAML, создадим удобную утилиту командной строки, позволяющую обнаруживать циклы в файловой системе UNIX, и обсудим встраивание файлов в двоичные файлы Go, а также функцию `os.ReadDir()`, тип `os.DirEntry` и пакет `io/fs`. Наконец, мы обновим приложение телефонной книги, чтобы можно было использовать данные JSON, и преобразуем его в соответствующую утилиту командной строки с помощью пакета `cobra`.

В главе 7 обсуждаются горутины, каналы и конвейеры. Мы рассмотрим различия между процессами, потоками и горутиными, пакет `sync` и то, как работает

планировщик Go. Кроме того, мы исследуем использование ключевого слова `select` и обсудим различные «типы» каналов Go, а также общую память, мьютексы, типы `sync.Mutex` и `sync.RWMutex`. В остальной части главы мы рассмотрим пакеты `context` и `semaphore`, рабочие пулы и выясним, как устанавливать тайм-аут для горутин и выявлять состояние гонки.

В главе 8 обсуждаются пакет `net/http`, разработка веб-серверов и веб-сервисов, предоставление метрик в Prometheus, визуализация метрик в Grafana, создание веб-клиентов и файловых серверов. Мы также преобразуем приложение телефонной книги в веб-сервис и создадим для него клиент командной строки.

Глава 9 посвящена пакету `net`, TCP/IP и протоколам TCP и UDP, а также сокетам UNIX и протоколу WebSocket. В этой главе мы разработаем множество сетевых серверов и клиентов.

В главе 10 описана работа с REST API и сервисами RESTful. Мы выясним, как определять REST API и разрабатывать мощные параллельные серверы RESTful, а также утилиты командной строки, которые действуют как клиенты сервисов RESTful. Наконец, мы познакомимся со Swagger, позволяющим создавать документацию для REST API, и выясним, как загружать двоичные файлы.

В главе 11 обсуждаются тестирование, оптимизация и профилирование кода, а также кросс-компиляция, сравнительный анализ кода Go, создание примеров функций, использование директивы `go:generate` и поиск недоступного кода Go.

Глава 12 посвящена работе с gRPC в Go. gRPC — это альтернатива сервисам RESTful, разработанная Google. В этой главе вы узнаете, как определить методы и сообщения сервиса gRPC, как перевести их в код Go и как разработать сервер и клиент для этого сервиса gRPC.

Глава 13 посвящена дженерикам и тому, как использовать новый синтаксис для написания универсальных функций и определения универсальных типов данных. Дженерики появились в версии Go 1.18, которая, согласно циклу разработки Go, была официально выпущена в марте 2022 года.

В приложении рассказывается о работе сборщика мусора Go и показано, как этот компонент Go может повлиять на производительность вашего кода.

Как сделать книгу максимально полезной

Для работы с книгой требуется компьютер UNIX с относительно недавно установленной версией Go, то есть это может быть любая машина, работающая под управлением macOS X, macOS или Linux. Большая часть представленного

кода также будет выполняться на компьютерах с Microsoft Windows без каких-либо изменений.

Чтобы извлечь из книги максимальную пользу, попробуйте как можно скорее применить знания из каждой главы в собственных программах и посмотреть, что работает, а что нет! Как я уже говорил, попытайтесь выполнить упражнения, приведенные в конце каждой главы, или реализуйте свои программные задачи.

Файлы с примерами кода

Пакет кода для книги размещен на GitHub по адресу <https://github.com/mactsovuk/mastering-Go-3rd>. У нас есть и другие пакеты кода из нашего богатого каталога книг и видео, доступные по адресу <https://github.com/PacktPublishing/>. Не забудьте в них заглянуть!

Цветные иллюстрации

Мы также предоставляем PDF-файл с цветными оригинальными снимками экрана и схемами из этой книги. Вы можете скачать его здесь: https://static.packt-cdn.com/downloads/9781801079310_ColorImages.pdf.

Условные обозначения

В книге используются следующие условные обозначения.

Кодовые слова в тексте, имена папок, имена файлов и их расширения, пути и пользовательский ввод оформляются моноширинным шрифтом. Например: «Звезда этой главы — пакет `net/http`, который содержит функции, позволяющие разрабатывать мощные веб-серверы и веб-клиенты».

Блок кода оформляется следующим образом:

```
package main

import (
    "fmt"
    "math/rand"
    "os"
    "path/filepath"
    "strconv"
    "time"
)
```

Когда мы хотим привлечь ваше внимание к определенной части блока кода, соответствующие строки или элементы выделяются **моноширинным жирным шрифтом**:

```
package main

import (
    "fmt"
    "math/rand"
    "os"
    "path/filepath"
    "strconv"
    "time"
)
```

Любой ввод или вывод из командной строки оформляется следующим образом:

```
$ go run www.go
Using default port number: :8001
Served: localhost:8001
```

Шрифтом без засечек оформляются URL или слова, которые вы видите на экране, например в меню или диалоговых окнах.

Новые термины и важные слова выделены *курсивом*.



Этот рисунок указывает на предупреждения или важные примечания.



Этот рисунок указывает на советы и рекомендации.

От издательства

Ваши замечания, предложения, вопросы отправляйте по адресу comp@piter.com (издательство «Питер», компьютерная редакция).

Мы будем рады узнать ваше мнение!

На веб-сайте издательства www.piter.com вы найдете подробную информацию о наших книгах.

1

Краткое введение в Go

Представьте, что вы разработчик, которому нужно создать утилиту командной строки. Или что у вас есть REST API и вы хотите создать сервер RESTful, который реализует этот REST API. Первый вопрос, который придет вам в голову, скорее всего, будет в том, какой язык программирования выбрать.

Рекомендуется использовать тот язык, который вы знаете лучше всего. Однако эта книга предназначена для того, чтобы дать вам возможность использовать Go для решения всех этих и многих других задач и проектов. Главу мы начнем с объяснения того, что такое Go, далее расскажем его историю и покажем, как запускать код Go. Мы объясним некоторые основные характеристики Go, например, как определять переменные, управлять потоком ваших программ и получать пользовательский ввод, после чего применим некоторые из этих концепций, создав приложение телефонной книги для командной строки.

В этой главе:

- введение в Go;
- Hello World;
- запуск Go-кода;
- важные особенности Go;
- разработка утилиты `which(1)` в Go;
- вывод информации в лог;
- обзор Go-дженериков;
- разработка базового приложения телефонной книги.

Введение в Go

Go — это язык системного программирования с открытым исходным кодом, первоначально разработанный как внутренний проект Google и ставший общедоступным еще в 2009 году. Духовными отцами Go являются Роберт Гриземер, Кен Томсон и Роб Пайк.



Официальное название языка — Go, однако его иногда называют Golang. Официальная причина в том, что домен go.org был недоступен для регистрации и вместо него был выбран golang.org. Практическая же причина в том, что, когда вы запрашиваете в поисковой системе информацию, связанную с Go, слово Go обычно интерпретируется как глагол. Кроме того, официальный хештег Go в Twitter — #golang.

Go является языком программирования общего назначения, но в основном используется для написания системных инструментов, утилит командной строки, веб-сервисов и программного обеспечения, которое работает в сетях. С помощью Go также можно обучаться программированию, плюс он является хорошим кандидатом на первый язык программирования благодаря своей немногословности, четким идеям и принципам. Go может помочь в разработке следующих видов приложений:

- профессиональные веб-сервисы;
- сетевые инструменты и серверы, такие как Kubernetes и Istio;
- серверные системы;
- системные утилиты;
- эффективные утилиты командной строки, такие как docker и hugo;
- приложения, которые обмениваются данными в формате JSON;
- приложения, обрабатывающие данные из реляционных баз данных, баз данных NoSQL или других популярных систем хранения;
- компиляторы и интерпретаторы для разрабатываемых вами языков программирования;
- системы баз данных, такие как CockroachDB, и хранилища ключей/значений, такие как etcd.

Есть много вещей, которые Go делает лучше, чем другие языки программирования, например:

- компилятор Go по умолчанию может обнаруживать большой набор глупых ошибок, которые могут привести к ошибкам в программном коде;

- Go использует меньше круглых скобок, чем C, C++ или Java, и не использует точки с запятой, что делает внешний вид исходного кода Go более удобочитаемым и менее подверженным ошибкам;
- Go поставляется с богатой и надежной стандартной библиотекой;
- Go поддерживает параллелизм «из коробки» через горутины и каналы;
- горутины действительно легковесные. Вы можете с легкостью запустить тысячи горутин на любой современной машине, обойдясь без каких-либо проблем с производительностью;
- в отличие от C, Go поддерживает функциональное программирование;
- Go-код имеет обратную совместимость, то есть более новые версии компилятора Go принимают программы, созданные с использованием предыдущей версии языка, без каких-либо изменений. Эта гарантия совместимости распространяется только на основные версии Go. Например, нет никакой гарантии, что программа на Go 1.x будет скомпилирована в Go 2.x.

Теперь, когда вы знаете, что может Go и чем он хорош, вспомним его историю.

История Go

Как упоминалось ранее, Go начинался как внутренний проект Google, который стал общедоступным еще в 2009 году. Гризмер, Томсон и Пайк разработали Go как язык для профессиональных программистов, желающих создавать надежное, стабильное и эффективное программное обеспечение, которым легко управлять. Они разрабатывали Go с прицелом на его простоту, даже если она означала, что Go не суждено стать языком программирования для всех.

На рис. 1.1 показано, какие языки программирования прямо или косвенно повлияли на Go. К примеру, синтаксис Go выглядит как синтаксис C, в то время как концепция пакетов была создана под влиянием Modula-2.

Результатами стали язык программирования, инструменты и стандартная библиотека. Помимо синтаксиса и инструментов Go, вы получаете довольно богатую стандартную библиотеку и систему типов, которая призвана избавить вас от простых ошибок, таких как неявные преобразования типов, неиспользуемые переменные и пакеты. Go-компилятор отлавливает большинство этих простых ошибок и останавливает компиляцию, пока вы как-то их не исправите. Кроме того, Go-компилятор с трудом обнаруживает сложные ошибки, такие как состояние гонки.

Если вы устанавливаете Go впервые, то можете начать со страницы <https://golang.org/dl/>. Однако есть большая вероятность, что в вашем варианте UNIX уже

имеется готовый к установке пакет для языка программирования Go, так что, возможно, у вас получится запустить Go с помощью вашего любимого менеджера пакетов.

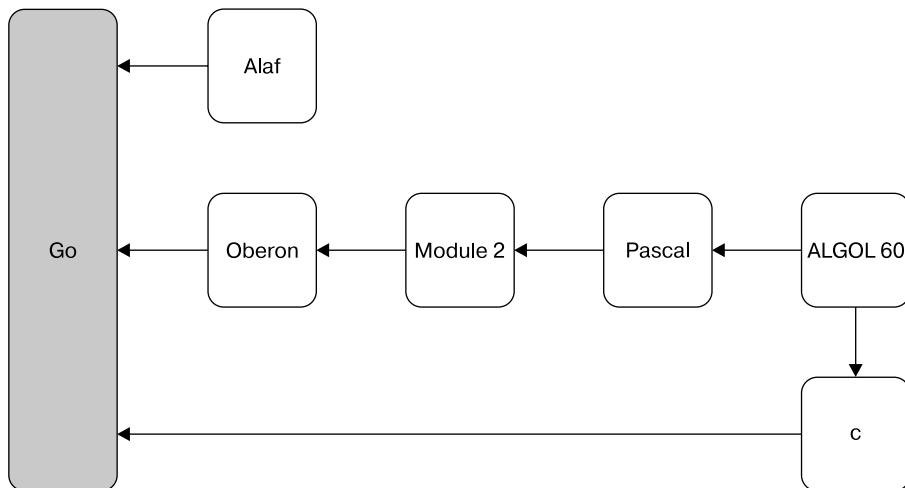


Рис. 1.1. Языки программирования, повлиявшие на Go

Почему UNIX, а не Windows

Вы можете задаться вопросом, почему мы все время говорим о UNIX и даже не обсуждаем Microsoft Windows. Для этого есть две основные причины. Первая состоит в том, что большинство Go-программ будут работать на компьютерах с Windows без каких-либо изменений, поскольку *Go переносим по сути*, и это означает, что вам не следует беспокоиться об используемой операционной системе.

Однако вам может потребоваться внести небольшие изменения в код некоторых системных утилит, чтобы они заработали в Windows. Кроме того, по-прежнему будут существовать библиотеки, работающие только на компьютерах с Windows или только на компьютерах, отличных от Windows. Вторая причина заключается в том, что многие сервисы, написанные на Go, выполняются в среде Docker — образы Docker используют операционную систему Linux, а это означает, что вы должны программировать свои утилиты с учетом операционной системы Linux.



Что же касается пользовательского опыта, то UNIX и Linux очень похожи. Основное отличие состоит в том, что Linux — программное обеспечение с открытым исходным кодом, тогда как UNIX — проприетарное ПО.

Преимущества Go

Go обладает рядом важных преимуществ для разработчиков, начиная с того факта, что он был разработан и поддерживается настоящими программистами. Помимо этого, Go прост в освоении, особенно если вы уже знакомы с такими языками программирования, как C, Python или Java. Вдобавок ко всему код на Go выглядит красиво (по крайней мере, на мой взгляд), и это замечательно, особенно когда вы зарабатываете на жизнь программированием и вам приходится работать с кодом ежедневно. Go-код также легко читается и имеет поддержку Unicode «из коробки», а это означает, что вы можете легко вносить изменения в существующий код Go. Наконец, Go резервирует лишь 25 ключевых слов, что делает его очень простым для запоминания. Возможно ли такое в случае C++?

Go также поставляется с поддержкой простой модели параллелизма, которая реализована с использованием *горутин* и *каналов*. Go не только управляет потоками операционной системы за вас, но и имеет эффективную среду выполнения, позволяющую создавать облегченные рабочие модули (*горутины*), которые взаимодействуют друг с другом с помощью *каналов*. Хотя Go поставляется с богатой стандартной библиотекой, существуют действительно удобные пакеты Go, такие как *cobra* и *viper*. Они позволяют разрабатывать на Go сложные утилиты командной строки, такие как *docker* и *hugo*. Это в значительной степени подтверждается тем фактом, что в исполняемых двоичных файлах Go используется *статическая компоновка*. Иными словами, после создания они уже не зависят от каких-либо совместно используемых библиотек и содержат всю необходимую информацию.

Благодаря своей простоте Go-код предсказуем и не имеет странных побочных эффектов. Хотя Go и поддерживает *указатели*, он не поддерживает подобную С арифметику указателей, если, конечно, вы не используете пакет *unsafe*, который зачастую служит причиной множества ошибок и дыр в безопасности. Язык Go не является объектно-ориентированным, тем не менее Go-интерфейсы очень универсальны и позволяют имитировать некоторые возможности объектно-ориентированных языков, такие как *полиморфизм*, *инкапсуляция* и *композиция*.

Кроме того, последние версии Go предлагают поддержку *универсальных паттернов* (или дженериков), благодаря чему упрощается код при работе с несколькими типами данных. И последнее, но не менее важное: Go поставляется с поддержкой *сборки мусора*, а это означает, что ручное управление памятью не требуется.

Go — очень практичный и надежный язык программирования, однако он далеко не идеален.

- Хотя это скорее личное предпочтение, нежели фактический технический недостаток, но Go не имеет прямой поддержки объектно-ориентированного программирования, которое является распространенной парадигмой программирования.
- Горутины достаточно легковесны, однако не так эффективны, как потоки операционной системы. В зависимости от реализуемого приложения вполне допустимы некоторые редкие случаи, когда горутины не подойдут для решения задачи. Однако в большинстве случаев разработка вашего приложения с помощью горутин и каналов решит поставленные задачи.
- Сборка мусора выполняется достаточно быстро большую часть времени и почти для всех видов приложений. Тем не менее бывают случаи, когда вам требуется работать над распределением памяти вручную, но Go так не может. На практике это означает, что Go не позволит осуществить какое-либо управление памятью вручную.

Однако существует множество сценариев, когда вы можете выбрать Go, например:

- создание сложных утилит командной строки со множеством команд, подкоманд и параметров командной строки;
- создание приложений с высокой степенью параллелизма;
- разработка серверов, работающих с API, и клиентов, которые взаимодействуют путем обмена данными во множестве форматов, включая JSON, XML и CSV;
- разработка серверов и клиентов WebSocket;
- разработка серверов и клиентов gRPC;
- разработка надежных системных инструментов для UNIX и Windows;
- изучение программирования.

Далее мы рассмотрим ряд концепций и утилит, что послужит прочной основой, после чего создадим упрощенную версию утилиты `which(1)`. В конце главы мы разработаем простое приложение для телефонной книги, которое будет развиваться по мере того, как в последующих главах мы будем продолжать знакомиться с особенностями Go.

Но сначала мы представим команду `go doc`, которая позволяет вам искать информацию о стандартной библиотеке Go, ее пакетах и их функциях. Затем, на примере программы `Hello World!`, мы покажем, как выполнить Go-код.

Утилиты `go doc` и `godoc`

Дистрибутив Go поставляется со множеством инструментов, которые облегчают вашу жизнь как программиста. Двумя такими инструментами являются подкоманда `go doc` и утилита `godoc`, которые позволяют вам просматривать документацию имеющихся функций и пакетов Go, не подключаясь к Интернету. Однако если вы предпочитаете просматривать документацию Go онлайн, то можете посетить <https://pkg.go.dev/>. Поскольку `godoc` не установлена по умолчанию, вам может потребоваться установить ее. Для этого выполните `go get golang.org/x/tools/cmd/godoc`.

Команда `go doc` может быть выполнена как обычное приложение командной строки, которое отображает свои выходные данные на терминале, а `godoc` — как приложение командной строки, которое запускает веб-сервер. В последнем случае вам понадобится браузер, чтобы просматривать документацию Go. Первая утилита аналогична команде UNIX `man(1)`, но для функций и пакетов Go.



Число после названия программы или системного вызова UNIX указывает на раздел руководства, к которому относится данная страница. Большинство имен встречаются на страницах руководства только один раз (это значит, указывать номер раздела не требуется), однако существуют имена, которые можно найти в нескольких разделах, поскольку они имеют несколько значений, например `crontab(1)` и `crontab(5)`. Так что если вы попытаетесь получить страницу руководства с многозначным названием, не указав номер раздела, то получите запись с наименьшим номером раздела.

Итак, чтобы найти информацию о функции `Printf()` пакета `fmt`, вам нужно выполнить следующую команду:

```
$ go doc fmt.Printf
```

Аналогичным образом вы можете найти информацию обо всем пакете `fmt`, выполнив следующую команду:

```
$ go doc fmt
```

Вторая утилита требует выполнения `godoc` с параметром `-http`:

```
$ godoc -http=:8001
```

Числовое значение в этой команде, которое в данном случае равно `8001`, является номером порта, который будет прослушиваться HTTP-сервером. Поскольку мы опустили IP-адрес, `godoc` будет прослушивать все сетевые интерфейсы.



Вы можете выбрать любой доступный номер порта при условии, что у вас есть соответствующие привилегии. Однако обратите внимание, что порты под номерами 0–1023 предназначены для служебного пользования и могут быть задействованы только пользователем root, поэтому лучше выбрать другой порт при условии, что он не используется другим процессом.

Вы можете опустить знак равенства в представленной команде и поставить вместо него пробел. Итак, следующая команда полностью эквивалентна предыдущей:

```
$ godoc -http :8001
```

После этого вы должны перейти в своем браузере по адресу <http://localhost:8001/>, что даст возможность получить список доступных Go-пакетов и просмотреть их документацию. Если вы используете Go впервые, то документация по Go позволит изучить параметры и возвращаемые значения используемых функций. По мере своего путешествия по Go вы будете применять документацию Go для детального изучения функций и переменных, которые хотите использовать.

Hello World!

Ниже приведена версия программы Hello World на Go. Введите ее и сохраните как `hw.go`:

```
package main

import (
    "fmt"
)

func main() {
    fmt.Println("Hello World!")
}
```

Любой исходный код на Go начинается с объявления пакета `package`. В нашем случае название пакета — `main`, что имеет особое значение в Go. Ключевое слово `import` позволяет включать функционал из существующих пакетов. В нашем случае нам нужна только часть функций пакета `fmt`, который принадлежит стандартной библиотеке Go. Пакеты, которые не являются ее частью, импортируются с использованием их полного интернет-пути. Следующая важная вещь при создании исполняемого приложения — это функция `main()`. Go считает ее точкой входа в приложение и начинает выполнение приложения с кода, обнаруженного в функции `main()` пакета `main`.

`hw.go` — это Go-программа, которая работает сама по себе. Две характеристики делают ее автономным исходным файлом, который может генерировать

исполняемый двоичный файл: имя пакета, которое должно быть `main`, и наличие функции `main()`. Более подробно мы обсудим Go-функции в следующем подразделе, но еще больше о функциях и методах (функциях, привязанных к конкретным типам данных) вы узнаете в главе 5.

Введение в функции

Каждое определение Go-функции начинается с ключевого слова `func`, за которым следуют название, сигнатура и реализация. Как и в случае пакета `main`, вы можете называть свои функции как угодно — существует глобальное правило Go, которое также применяется к именам функций и переменных и действует для всех пакетов, кроме `main`: *все, что начинается со строчной буквы, считается закрытым и доступно только в текущем пакете*. Больше информации о данном правиле мы узнаем в главе 5. Единственным исключением из него являются имена пакетов, которые могут начинаться как со строчных, так и с прописных букв. Хотя я это и сказал, но не помню ни одного Go-пакета, который начинался бы с заглавной буквы!

Теперь вы можете спросить, как функции организуются и предоставляются. Ответ: с помощью пакетов и в следующем подразделе мы немного поговорим об этом.

Введение в пакеты

Go-программы организованы в пакеты, и даже самая маленькая Go-программа должна поставляться в виде пакета. Ключевое слово `package` помогает задать имя нового пакета, которое может быть любым, за одним исключением: если вы создаете исполняемое приложение, а не просто пакет, который будет совместно использоваться другими приложениями или пакетами, то должны назвать свой пакет `main`. Больше информации о разработке Go-пакетов вы получите в главе 5.



Пакеты могут использоваться другими пакетами. Фактически повторное использование существующих пакетов — хорошая практика, которая избавляет от необходимости писать много кода или реализовывать существующую функциональность с нуля.

Ключевое слово `import` используется для импорта других Go-пакетов в ваши Go-программы и дает возможность использовать некоторые или все их функциональные возможности. Go-пакет может либо быть частью богатой стандартной библиотеки Go, либо поступать извне. Пакеты стандартной библиотеки Go импортируются по имени (`os`) без необходимости указывать имя хоста и путь, в то время как внешние пакеты импортируются с использованием их полных интернет-путей, таких как github.com/spf13/cobra.

Запуск Go-кода

Теперь нам нужно понять, как же запустить `hw.go` или любое другое Go-приложение. В двух следующих подразделах мы выясним, что есть два способа выполнения Go-кода: в виде скомпилированного языка с использованием `go build` или в виде языка сценариев с использованием `go run`. Подробно поговорим об этих двух способах запуска Go-кода.

Компиляция Go-кода

Чтобы скомпилировать Go-код и создать двоичный исполняемый файл, вам необходимо использовать команду `go build`. Она создает исполняемый файл, который вы можете распространять и выполнять вручную. Это означает, что команда `go build` требует дополнительного шага для запуска вашего кода.

Сгенерированный исполняемый файл автоматически называется по имени файла исходного кода за вычетом расширения `.go`. Следовательно, из исходного файла `hw.go` получится исполняемый файл `hw`. Если это не то, что вам нужно, то `go build` поддерживает параметр `-o`, который позволяет изменить имя файла и путь к сгенерированному исполняемому файлу. Например, если вы хотите назвать исполняемый файл `HelloWorld`, то вам следует выполнить `go build -o HelloWorld hw.go`. Если исходные файлы не предоставлены, то `go build` ищет пакет `main` в текущем каталоге.

После этого вам необходимо самостоятельно запустить сгенерированный исполняемый двоичный файл. В нашем случае это означает выполнение либо `hw`, либо `HelloWorld`. Это показано в следующем выводе:

```
$ go build hw.go
$ ./hw
Hello World!
```

Теперь, когда мы выяснили, как компилировать Go-код, перейдем к использованию Go в качестве языка скриптов.

Использование Go в качестве языка скриптов

Команда `go run` создает именованный Go-пакет (который в нашем случае является пакетом `main`, реализованным в единственном файле), временный исполняемый файл, выполняет его и удаляет после завершения — для нас это похоже на использование языка скриптов. В нашем случае мы можем сделать следующее:

```
$ go run hw.go
Hello World!
```

Если вы хотите протестировать свой код, то лучше использовать команду `go run`. Однако если вы хотите создать и распространить исполняемый двоичный файл, то лучше использовать команду `go build`.

Важные правила форматирования и кодирования

Вы должны знать, что Go подразумевает строгие правила форматирования и кодирования, которые помогают разработчикам избежать ошибок начинающих. Изучив эти несколько правил и характерных особенностей Go и поняв, какое значение они имеют для вашего кода, вы сможете сосредоточиться на фактической функциональности вашего кода. Кроме того, компилятор Go помогает вам следовать этим правилам, выдавая выразительные сообщения об ошибках и предупреждения. Наконец, Go предлагает стандартный инструментарий (`gofmt`), который умеет форматировать ваш код за вас, так что вам не придется думать об этом.

Ниже приведен список важных правил Go, которые помогут вам при чтении этой главы.

- Go-код поставляется в пакетах, и вы можете свободно использовать функционал существующих пакетов. Однако если вы собираетесь импортировать пакет, то вам следует использовать некоторые из этих функций. Из этого правила есть несколько исключений, но в основном они связаны с инициализацией подключений и сейчас не важны.
- Вы либо используете переменную, либо вообще ее не объявляете. Это правило поможет избежать ошибок, таких как неправильное написание существующей переменной или имени функции.
- В Go есть только один способ форматирования фигурных скобок.
- Блоки кода в Go заключаются в фигурные скобки, даже если содержат лишь один оператор или вообще их не содержат.
- Go-функции могут возвращать несколько значений.
- Вы не можете автоматически конвертировать различные типы данных, даже если они из одного семейства. Например, вы не можете неявно преобразовать целое число в число с плавающей запятой.

В Go гораздо больше правил, но эти — самые важные. Вы увидите их в действии не только в текущей главе, но и в других. Пока мы рассмотрим единственный способ форматирования фигурных скобок в Go, так как это правило применяется повсеместно.

Взгляните на следующую Go-программу `curly.go`:

```
package main

import (
```

```
        "fmt"
)

func main()
{
    fmt.Println("Go has strict rules for curly braces!")
}
```

Код выглядит просто замечательно, но при запуске вас ждет разочарование, так как он не будет компилироваться. В итоге вы получите следующее сообщение о синтаксической ошибке:

```
$ go run curly.go
# command-line-arguments
./curly.go:7:6: missing function body
./curly.go:8:1: syntax error: unexpected semicolon or newline before {
```

Официальное объяснение этого сообщения об ошибке заключается в том, что Go требует использования точек с запятой в качестве ограничителей оператора во многих контекстах и компилятор автоматически вставляет требуемые точки с запятой, когда считает, что они необходимы. Следовательно, из-за открывающей фигурной скобки ({), помещенной в отдельную строку, компилятор Go будет вынужден вставить точку с запятой в конце предыдущей строки (`func main()`), что является основной причиной сообщения об ошибке. Правильный способ написания кода, который был приведен выше, выглядит так:

```
package main

import (
    "fmt"
)

func main() {
    fmt.Println("Go has strict rules for curly braces!")
}
```

Теперь, когда мы познакомились с этим глобальным правилом, перейдем к некоторым важным особенностям Go.

Важные особенности Go

В этом большом разделе мы обсудим важные и незаменимые функции Go, включая переменные, управление потоком программы, итерации, получение пользовательского ввода и параллелизм в Go. Мы начнем с обсуждения переменных, их объявления и использования.

Определение и использование переменных

Предположим, нам требуется с помощью Go выполнить некоторые базовые математические вычисления. В этом случае придется определить переменные, чтобы сохранить как входные данные, так и результаты.

Чтобы сделать процесс объявления переменных более естественным и удобным, Go предоставляет несколько способов объявления новых переменных. Вы можете объявить новую переменную, используя ключевое слово `var`, за которым следуют имя переменной и желаемый тип данных (мы подробно рассмотрим типы данных в главе 2). При желании вы можете дополнить объявление знаком `=` и начальным значением для вашей переменной. Если задано начальное значение, то вы можете опустить тип данных, и компилятор подберет его за вас.

Это подводит нас к очень важному правилу Go: *если переменной не задано начальное значение, то компилятор Go автоматически инициализирует ее нулевым значением ее типа данных.*

Существует также нотация `:=`, которую можно использовать вместо объявления `var`. Команда `:=` определяет новую переменную, делая вывод о данных из следующего за ней значения. Официальное название для `:=` звучит так: *короткое присваивание*, и этот оператор очень часто используется в Go, особенно для получения возвращаемых значений из функций и для циклов `for` с ключевым словом `range`.

Оператор короткого присваивания может применяться вместо объявления `var` с неявным типом. В Go вы будете редко встречать `var`. Это ключевое слово в основном служит для объявления глобальных или локальных переменных без начального значения. Причина первого заключается в том, что каждый оператор, существующий вне кода функции, должен начинаться с ключевого слова, такого как `func` или `var`.

Это означает, что оператор короткого присваивания не получится использовать вне функциональной среды, поскольку там он недоступен. Наконец, вам может понадобиться `var`, когда вы хотите четко указать тип данных. Например, когда вам нужен `int8` или `int32` вместо `int`.

Поэтому, даже если вы и можете объявлять локальные переменные через `var` или `:=`, только `const` (когда значение переменной не будет меняться) и `var` работают для *глобальных переменных*, которые являются переменными, определенными вне функции и не заключенными в фигурные скобки. К глобальным переменным можно получить доступ из любого места пакета, не прибегая к необходимости явно передавать их в функцию, и они могут меняться, если только не были определены как константы с использованием ключевого слова `const`.

Вывод переменных

Программы, как правило, отображают информацию, а это значит, что им часто нужно вывести данные или отправить их куда-нибудь, чтобы другая программа могла их сохранить или обработать. Для вывода данных на экран Go использует функции пакета `fmt`. Если вы хотите, чтобы Go взял на себя вывод, то можете воспользоваться `fmt.Println()`. Однако бывают ситуации, когда нужен полный контроль над тем, как будут выводиться данные. В таких случаях можно использовать функцию `fmt.Printf()`.

Она аналогична функции C `printf()` и требует использования управляющих последовательностей, которые определяют тип данных переменной, которая будет выведена. Кроме того, `fmt.Printf()` позволяет форматировать сгенерированный вывод, что особенно удобно для значений с плавающей запятой, поскольку позволяет указать цифры, которые будут отображаться в выводе (`.2f` отображает две цифры после десятичной точки). Наконец, символ `\n` используется для вывода символа новой строки и, следовательно, ее создания, так как `fmt.Printf()` не вставляет автоматически новую строку (это не относится к `fmt.Println()`, которая вставляет новую строку).

В следующей программе показано, как объявлять, использовать и выводить новые переменные. Введите следующий код в обычный текстовый файл `variables.go`:

```
package main

import (
    "fmt"
    "math"
)

var Global int = 1234
var AnotherGlobal = -5678

func main() {
    var j int
    i := Global + AnotherGlobal
    fmt.Println("Initial j value:", j)
    j = Global
    // math.Abs() требует параметр float64
    // соответственно, мы приводим тип
    k := math.Abs(float64(AnotherGlobal))
    fmt.Printf("Global=%d, i=%d, j=%d k=%.2f.\n", Global, i, j, k)
}
```



Я предпочитаю выделять глобальные переменные, либо начиная их с заглавной буквы, либо используя для названия только заглавные буквы.

В этой программе мы видим:

- глобальную переменную `int` с именем `Global`;
- вторую глобальную переменную с именем `AnotherGlobal` — Go автоматически выводит ее тип данных из ее значения, которое в данном случае является целым числом;
- локальную переменную с именем `j` и типом `int`, который, как вы узнаете в следующей главе, является особым типом данных. Переменная `j` не имеет начального значения; это значит, Go автоматически присваивает ей нулевое значение ее типа данных, что в данном случае равно `0`;
- еще одну локальную переменную с именем `i` — Go выводит ее тип данных из значения. Поскольку это сумма двух значений `int`, то и тип тоже `int`;
- поскольку функция `math.Abs()` требует параметра `float64`, мы не можем передать туда `AnotherGlobal`, так как это переменная типа `int`. Приведение типа `float64()` преобразует значение `AnotherGlobal` в `float64`. Обратите внимание, что `AnotherGlobal` сохраняет свой тип `int`;
- наконец, `fmt.Printf()` форматирует и выводит результат.

При выполнении `variables.go` мы получаем такой вывод:

```
Initial j value: 0
Global=1234, i=-4444, j=1234 k=5678.00.
```

В этом примере показано еще одно важное правило Go, которое уже упоминалось ранее: Go не допускает *неявных преобразований данных*, подобно C.

В `variables.go` мы увидели, что при использовании функции `math.Abs()`, которая ожидает значение `float64`, значение `int` не получится использовать вместо `float64`, даже если это конкретное преобразование простое и безошибочное. Компилятор Go отказывается компилировать подобные операторы. Чтобы все работало правильно, придется преобразовать значение `int` в `float64` явно, используя `float64()`.

Для преобразований, которые не являются прямыми (например, `string` в `int`), существуют специализированные функции, позволяющие обнаруживать проблемы с преобразованием и возвращающие их в виде переменной `error`.

Управление ходом выполнения программы

Мы разобрались с Go-переменными, но как мы можем менять поток Go-программы на основе значения переменной или какого-либо другого условия? Go поддерживает структуры управления `if/else` и `switch`. Обе эти структуры можно найти в большинстве современных языков программирования, так что если вы уже программируали на другом языке, то должны быть знакомы с `if` и `switch`. Оператор `if` не использует круглые скобки для встраивания проверяемых условий, потому что в Go вообще не используются круглые скобки. Кроме того, `if` ожидаемо поддерживает операторы `else` и `else if`.

Продемонстрируем использование `if` с помощью очень распространенного паттерна, который повсеместно применяется в Go. Он гласит, что если значение переменной `error`, возвращаемой из функции, равно `nil`, то с выполнением функции все в порядке. В противном случае где-то возникла ошибка, требующая особого внимания. Этот паттерн обычно реализуется следующим образом:

```
err := anyFunctionCall()
if err != nil {
    // сделать что-нибудь, если возникла ошибка
}
```

`err` — это переменная, которая содержит значение `error`, возвращаемое функцией, а `!=` говорит о том, что значение переменной `err` не равно `nil`. Подобный код вы встретите в Go-программах множество раз.



Строки, начинающиеся с `//`, представляют собой однострочные комментарии. Если вы ставите `//` в середине строки, то все, что после `//`, считается комментарием. Это правило не применяется, если `//` находится внутри строкового значения.

Оператор `switch` имеет две разные формы. В первой он содержит вычисляемое выражение, тогда как во второй не имеет выражения для вычисления. В этом случае выражения вычисляются в каждом операторе `case`, что повышает гибкость `switch`. Основное преимущество `switch` заключается в том, что при правильном использовании он упрощает сложные и трудночитаемые блоки `if-else`.

Как `if`, так и `switch` показаны в следующем коде, который предназначен для обработки пользовательского ввода, заданного в качестве аргумента командной строки. Пожалуйста, введите его и сохраните как `control.go`. В учебных целях мы представляем код `control.go` по частям, чтобы его было удобнее объяснять:

```
package main

import (
```

```

    "fmt"
    "os"
    "strconv"
)

```

Эта первая часть содержит ожидаемую преамбулу с импортом пакетов. Реализация функции `main()` начинается следующим образом:

```

func main() {
    if len(os.Args) != 2 {
        fmt.Println("Please provide a command line argument")
        return
    }
    argument := os.Args[1]

```

Эта часть программы гарантирует, что у вас есть единственный аргумент командной строки для обработки, доступ к которому осуществляется по имени `os.Args[1]`. Позже мы расскажем об этом более подробно, но вы можете обратиться к рис. 1.2, чтобы получить дополнительную информацию о срезе `os.Args`.

```

// с выражением после switch
switch argument {
case "0":
    fmt.Println("Zero!")
case "1":
    fmt.Println("One!")
case "2", "3", "4":
    fmt.Println("2 or 3 or 4")
    fallthrough
default:
    fmt.Println("Value:", argument)
}

```

Здесь мы видим блок `switch` с четырьмя ветвлениями. Первые три требуют точного совпадения `string`, а последнее соответствует всему остальному. Порядок операторов `case` важен, поскольку выполняется только первое совпадение. Ключевое слово `fallthrough` сообщает Go, что после выполнения этой ветви необходимо перейти на следующую, которая в данном случае является веткой `default`:

```

value, err := strconv.Atoi(argument)
if err != nil {
    fmt.Println("Cannot convert to int:", argument)
    return
}

```

Поскольку аргументы командной строки инициализируются как строковые значения, нам нужно преобразовать пользовательский ввод в целочисленное значение с помощью отдельного вызова, который в данном случае будет вызовом `strconv.Atoi()`. Если значение переменной `err` равно `nil`, то преобразование

прошло успешно и мы можем продолжать. В противном случае на экране выводится сообщение об ошибке, и программа завершает работу.

Следующий код показывает вторую форму `switch`, где условие вычисляется в каждой ветви `case`:

```
// без выражения после switch
switch {
    case value == 0:
        fmt.Println("Zero!")
    case value > 0:
        fmt.Println("Positive integer")
    case value < 0:
        fmt.Println("Negative integer")
    default:
        fmt.Println("This should not happen:", value)
}
```

Мы получаем больше гибкости, но должны приложить больше усилий, чтобы вникнуть в код. В этом случае ветка `default` не должна выполниться главным образом потому, что любое допустимое целочисленное значение будет перехвачено тремя другими. Тем не менее ветка `default` присутствует, что является хорошей практикой, поскольку она может перехватывать неожиданные значения.

При выполнении `control.go` мы получаем такой вывод:

```
$ go run control.go 10
Value: 10
Positive integer
$ go run control.go 0
Zero!
Zero!
```

Каждый из двух блоков `switch` в `control.go` создает одну строку вывода.

Итерации с помощью циклов `for` и `range`

Данный подраздел посвящен итерациям в Go. Этот язык поддерживает циклы `for`, а также ключевое слово `range` для перебора всех элементов массивов, срезов и (как мы увидим в главе 3) карт. Примером простоты Go служит тот факт, что язык обеспечивает поддержку только ключевого слова `for`, вместо того чтобы включать прямую поддержку циклов `while`. Однако в зависимости от записи цикл `for` может функционировать как `while` или бесконечный цикл. Более того, в сочетании с ключевым словом `range` циклы `for` могут реализовать функциональность `forEach` из JavaScript.



Цикл `for` нужно заключать в фигурные скобки, даже если он содержит один оператор или вообще не содержит операторы.

Вы также можете создавать циклы `for` с переменными и условиями. Цикл `for` можно завершить с помощью ключевого слова `break` или пропустить текущую итерацию, применив ключевое слово `continue`. При использовании с `range` циклы `for` позволяют просматривать все элементы среза или массива, не зная размер структуры данных. Как вы увидите в главе 3, `for` и `range` аналогичным образом позволяют выполнять итерации по элементам карты.

В следующей программе показано использование `for` как отдельно, так и с ключевым словом `range`. Введите ее и сохраните как `forLoops.go`, чтобы выполнить в дальнейшем:

```
package main

import "fmt"

func main() {
    // традиционный цикл for
    for i := 0; i < 10; i++ {
        fmt.Println(i*i, " ")
    }
    fmt.Println()
}
```

В этом коде показан традиционный цикл `for`, который использует локальную переменную `i`. Код выведет на экран квадраты 0, 1, 2, 3, 4, 5, 6, 7, 8 и 9. Квадрат 10 не выводится, поскольку не удовлетворяет условию `10 < 10`.

Следующий код является идиоматическим Go:

```
i := 0
for ok := true; ok; ok = (i != 10) {
    fmt.Println(i*i, " ")
    i++
}
fmt.Println()
```

Можно использовать и его, но зачастую его трудно читать, особенно новичкам. Следующий код показывает, как цикл `for` может имитировать цикл `while`, который напрямую не поддерживается:

```
// цикл for, используемый как цикл while
i := 0
for {
    if i == 10 {
        break
    }
}
```

```
    fmt.Println(i*i, " ")
    i++
}
fmt.Println()
```

Ключевое слово `break` в условии `if` досрочно завершает цикл и действует как условие выхода из цикла.

Наконец, с помощью `range` мы выполняем итерацию по всем элементам среза `aSlice`, что возвращает два упорядоченных значения: индекс текущего элемента в срезе и его значение. Если вы хотите проигнорировать любое из этих возвращаемых значений, что в нашем случае не нужно, то можете использовать `_` вместо значения, которое хотите проигнорировать. Если вам нужен только индекс, то можете полностью исключить второе значение из `range`, не используя `_`.

```
// это срез, но range работает и с массивами
aSlice := []int{-1, 2, 1, -1, 2, -2}
for i, v := range aSlice {
    fmt.Println("index:", i, "value: ", v)
}
```

Если вы запустите `forLoops.go`, то получите следующий вывод:

```
$ go run forLoops.go
0 1 4 9 16 25 36 49 64 81
0 1 4 9 16 25 36 49 64 81
0 1 4 9 16 25 36 49 64 81
index: 0 value: -1
index: 1 value: 2
index: 2 value: 1
index: 3 value: -1
index: 4 value: 2
index: 5 value: -2
```

В этом выводе показано, что первые три цикла `for` эквивалентны и, следовательно, выдают один и тот же результат. Последние шесть строк показывают индекс и значение каждого элемента, найденного в `aSlice`.

Теперь, когда мы познакомились с циклами `for`, посмотрим, как получить пользовательский ввод.

Получение пользовательского ввода

Получение пользовательского ввода — важная часть любой программы. В этом подразделе представлены два способа получения пользовательского ввода, которые заключаются в чтении из стандартного ввода и использовании аргументов командной строки программы.

Чтение стандартного ввода

Функция `fmt.Scanln()` помогает прочитать пользовательский ввод, когда программа уже запущена, и сохранить его в переменной `string`, которая передается как указатель в `fmt.Scanln()`. Пакет `fmt` содержит дополнительные функции для чтения пользовательского ввода из консоли (`os.Stdin`), из файлов или списков аргументов.

В следующем коде показано чтение из стандартного ввода — введите его и сохраните как `input.go`:

```
package main

import (
    "fmt"
)

func main() {
    // получить пользовательский ввод
    fmt.Printf("Please give me your name: ")
    var name string
    fmt.Scanln(&name)
    fmt.Println("Your name is", name)
}
```

В ожидании ввода данных полезно сообщить пользователю, какую информацию он должен предоставить, что и является целью вызова `fmt.Printf()`. Мы не используем здесь `fmt.Println()`, поскольку `fmt.Println()` автоматически добавляет символ новой строки в конце, а здесь это не нужно.

Выполнение `input.go` дает следующий вывод и взаимодействие с пользователем:

```
$ go run input.go
Please give me your name: Mihalis
Your name is Mihalis
```

Работа с аргументами командной строки

Использование пользовательского ввода при необходимости может показаться хорошей идеей, но обычно реальное программное обеспечение так не работает. Как правило, пользовательский ввод предоставляется исполняемому файлу в виде аргументов командной строки. По умолчанию аргументы командной строки в Go хранятся в срезе `os.Args`. Go также предлагает для анализа аргументов командной строки пакет `flag`, но есть лучшие и более эффективные альтернативы.

На рис. 1.2 показано, как работают аргументы командной строки в Go. Этот принцип повторяет работу в языке программирования С. Важно понимать, что срез `os.Args` должным образом инициализируется Go и программа может к нему обращаться. `os.Args` содержит значения `string`.

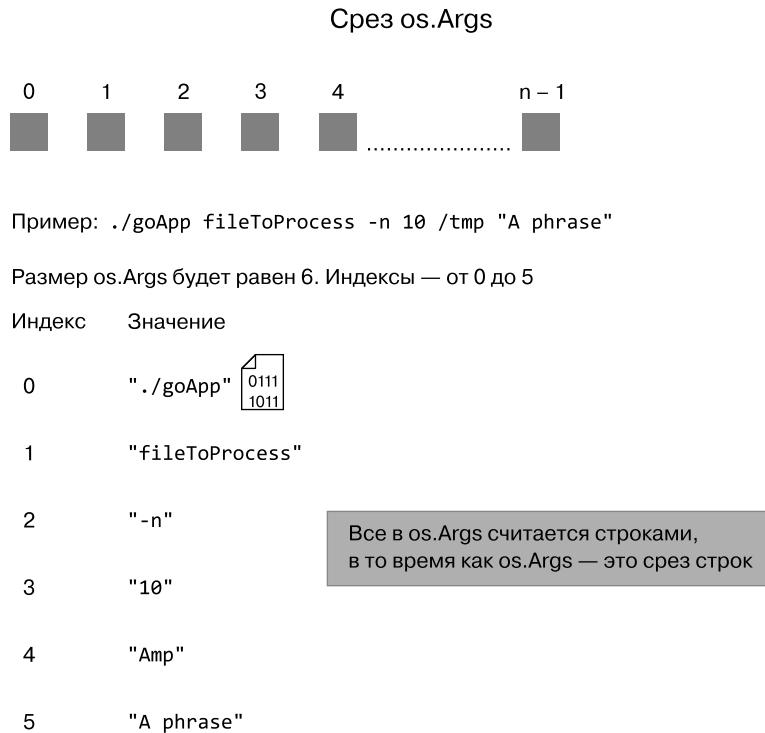


Рис. 1.2. Как работает срез `os.Args`

Первый аргумент командной строки в срезе `os.Args` — это всегда имя исполняемого файла. Если вы используете `go run`, то получите временное имя и путь. В противном случае это будет путь к исполняемому файлу, указанный пользователем. Остальные аргументы — это то, что следует за именем исполняемого файла, а именно различные аргументы командной строки, автоматически разделенные пробелами, если только они не заключены в двойные или одинарные кавычки.

Использование `os.Args` показано в следующем коде. Его задача состоит в поиске минимального и максимального числовых значений входных данных, при этом недопустимые входные данные, такие как символы и строки, игнорируются.

Ведите код и сохраните как `cla.go` (или под любым другим именем, которое вам по душе):

```
package main

import (
    "fmt"
    "os"
    "strconv"
)
```

Файл `cla.go` ожидаемо начинается со своей преамбулы. Пакет `fmt` используется для вывода данных, а пакет `os` нужен потому, что `os.Args` является его частью. Наконец, пакет `strconv` содержит функции для преобразования строк в числовые значения. Далее мы хотим убедиться, что у нас есть хотя бы один аргумент командной строки:

```
func main() {
    arguments := os.Args
    if len(arguments) == 1 {
        fmt.Println("Need one or more arguments!")
        return
}
```

Помните, что первый элемент в `os.Args` — это всегда путь к исполняемому файлу. Поэтому `os.Args` никогда не бывает полностью пустым. Далее программа проверяет наличие ошибок тем же способом, который мы встречали в предыдущих примерах. Больше информации об ошибках и их обработке вы узнаете в главе 2.

```
var min, max float64
for i := 1; i < len(arguments); i++ {
    n, err := strconv.ParseFloat(arguments[i], 64)
    if err != nil {
        continue
}}
```

В этом случае мы используем переменную `error`, возвращаемую `strconv.parseFloat()`, чтобы убедиться, что вызов `strconv.parseFloat()` прошел успешно и мы получили допустимое числовое значение для дальнейшей обработки. В противном случае нужно перейти к следующему аргументу командной строки. Цикл `for` используется для перебора всех доступных аргументов командной строки, кроме первого, который использует значение индекса `0`. Это еще один распространенный метод работы со всеми аргументами командной строки.

Следующий код используется для правильной инициализации значений `min` и `max` после того, как был обработан первый аргумент командной строки:

```
if i == 1 {
    min = n
    max = n
    continue
}
```

Мы используем `i == 1` в качестве проверки того, является ли итерация первой. В данном случае это именно так, поэтому мы обрабатываем первый аргумент командной строки. Следующий код проверяет, является ли текущее значение нашим новым минимумом или максимумом — именно здесь реализуется логика программы:

```
if n < min {  
    min = n  
}  
if n > max {  
    max = n  
}  
}  
fmt.Println("Min:", min)  
fmt.Println("Max:", max)  
}
```

Последняя часть программы предназначена для вывода результатов, которые представляют собой минимальное и максимальное числовые значения из всех допустимых аргументов командной строки. Результат, который вы получите после запуска `cla.go`, зависит от входных данных:

```
$ go run cla.go a b 2 -1  
Min: -1  
Max: 2
```

В этом случае `a` и `b` — недопустимые данные, и единственными допустимыми входными данными являются `-1` и `2`, которые служат минимальным и максимальным значениями соответственно.

```
$ go run cla.go a 0 b -1.2 10.32  
Min: -1.2  
Max: 10.32
```

В этом случае `a` и `b` являются недопустимыми входными данными и поэтому игнорируются.

```
$ go run cla.go  
Need one or more arguments!
```

В последнем случае, поскольку `cla.go` не имеет входных данных для обработки, программа выводит информационное сообщение. Если вы запустите программу без допустимых входных значений, например `go run cla.go a b c`, то значения `min` и `max` будут равны нулю.

В следующем подразделе показан метод различия разных типов данных с использованием переменных `error`.

Использование переменных ошибок для различия типов входных данных

Теперь позвольте продемонстрировать вам метод, который использует переменные `error` для различия видов пользовательского ввода. Чтобы метод сработал, необходимо перейти от конкретных случаев к более общим. Если мы говорим о числовых значениях, то необходимо сначала проверить, является ли строка допустимым целым числом, после чего уже проверять, является ли та же самая строка значением с плавающей запятой. Именно в такой последовательности, поскольку каждое допустимое целое число также является допустимым значением с плавающей запятой.

Это показано в следующем фрагменте кода:

```
var total, nInts, nFloats int
invalid := make([]string, 0)
for _, k := range arguments[1:] {
    // Это целое число?
    _, err := strconv.Atoi(k)
    if err == nil {
        total++
        nInts++
        continue
    }
}
```

Сначала мы создаем три переменные для подсчета общего количества проверенных допустимых значений, найденных целочисленных значений и найденных значений с плавающей запятой соответственно. Переменная `invalid`, представляющая собой срез, используется для сохранения всех нечисловых значений.

Повторюсь, нам нужно перебрать все аргументы командной строки, кроме первого с индексом `0`, поскольку это путь к исполняемому файлу. Мы игнорируем путь к исполняемому файлу, используя `arguments[1:]` вместо просто `arguments`, — выбор непрерывной части среза мы обсудим в следующей главе.

Вызов `strconv.Atoi()` определяет, обрабатываем ли мы допустимое значение `int`. Если это так, то мы увеличиваем счетчики `total` и `nInts`:

```
// число с плавающей запятой
_, err = strconv.ParseFloat(k, 64)
if err == nil {
    total++
    nFloats++
    continue
}
```

Аналогично, если проверяемая строка представляет допустимое значение с плавающей запятой, то вызов `strconv.ParseFloat()` будет успешным, а программа

обновит соответствующие счетчики. Наконец, если значение вообще не является числовым, оно добавляется к срезу `invalid` с помощью вызова `append()`:

```
// значит, недопустимое значение
invalid = append(invalid, k)
}
```

Это обычная методика сохранения неожиданных входных данных в приложениях. Вышеприведенный код можно найти в `process.go` в репозитории книги на GitHub. Здесь же представлен дополнительный код, который предупреждает вас, если ваш недопустимый ввод превышает допустимый. При выполнении `process.go` мы получаем такой вывод:

```
$ go run process.go 1 2 3
#read: 3 #ints: 3 #floats: 0
```

В этом случае мы обрабатываем 1, 2 и 3. Все они являются допустимыми целочисленными значениями.

```
$ go run process.go 1 2.1 a
#read: 2 #ints: 1 #floats: 1
```

В этом случае у нас есть допустимое целое число 1, значение с плавающей запятой 2.1 и недопустимое значение a.

```
$ go run process.go a 1 b
#read: 1 #ints: 1 #floats: 0
Too much invalid input: 2
a
b
```

Если недопустимых входных данных больше, чем допустимых, то `process.go` выводит дополнительное сообщение об ошибке.

В следующем подразделе мы обсудим модель параллелизма Go.

Знакомство с моделью параллелизма в Go

Этот подраздел представляет собой краткое введение в модель параллелизма в Go. Она реализована с использованием горутин и каналов. *Горутина* — это наименьшая исполняемая сущность в Go. Чтобы создать новую горутину, вы должны использовать ключевое слово `go`, за которым следует предопределенная или анонимная функция — оба метода эквивалентны, если речь идет о Go.



Обратите внимание, что вы можете выполнять функции или анонимные функции только как горутинны.

Канал в Go — это механизм, который, помимо прочего, позволяет горутинам взаимодействовать и обмениваться данными. Если вы программист-любитель или впервые слышите о горутинах и каналах, то не паникуйте. Горутины и каналы, а также конвейеры и совместное использование данных горутиными мы рассмотрим гораздо более подробно в главе 7.

Несмотря на то что создавать горутины легко, при параллельном программировании мы сталкиваемся с другими сложностями, включая синхронизацию и обмен данными между горутинаами (это механизм Go, позволяющий избежать побочных эффектов при запуске горутина). Поскольку `main()` также выполняется как горутина, нам не нужно, чтобы она завершалась раньше других горутинах программы, поскольку при выходе из `main()` завершится вся программа вместе с любыми еще не завершенными горутинаами. Горутины не имеют общих переменных, но могут совместно использовать память. Хорошо то, что существуют различные методы, заставляющие `main()` ожидать, пока горутины будут обмениваться данными по каналам или, что реже встречается в Go, используя общую память.

Ведите следующую Go-программу, которая синхронизирует горутины с помощью вызовов `time.Sleep()` (это неправильный способ синхронизации горутина, а правильный мы обсудим в главе 7), в ваш любимый редактор и сохраните как `goRoutines.go`:

```
package main

import (
    "fmt"
    "time"
)

func myPrint(start, finish int) {
    for i := start; i <= finish; i++ {
        fmt.Print(i, " ")
    }
    fmt.Println()
    time.Sleep(100 * time.Microsecond)
}

func main() {
    for i := 0; i < 5; i++ {
        go myPrint(i, 5)
    }
    time.Sleep(time.Second)
}
```

Здесь в наивно реализованном примере мы создаем четыре горутины и выводим на экран определенные значения с помощью функции `myPrint()` — для создания

горутин используется ключевое слово `go`. При выполнении `goRoutines.go` мы получаем такой вывод:

```
$ go run goRoutines.go
2 3 4 5
0 4 1 2 3 1 2 3 4 4 5
5
3 4 5
5
```

Однако если вы запустите программу несколько раз, то, скорее всего, будете каждый раз получать разные выходные данные:

```
1 2 3 4 5
4 2 5 3 4 5
3 0 1 2 3 4 5

4 5
```

Это происходит потому, что горутины инициализируются и начинают выполняться в случайном порядке. Планировщик Go отвечает за выполнение горутин точно так же, как планировщик ОС — за выполнение потоков ОС. В главе 7 мы обсудим это более подробно. Кроме того, мы представим и решение данной проблемы случайности с помощью переменной `sync.WaitGroup`. Однако имейте в виду, что параллелизм в Go присутствует повсюду и это является основной причиной добавления данного подраздела. Поэтому, поскольку в некоторых сообщениях об ошибках компиляции говорится о горутинах, не следует думать, что эти горутины были созданы именно вами.

В следующем разделе приводится практический пример, который заключается в разработке на Go версии утилиты `which(1)`, находящей программный файл в пользовательском значении PATH.

Разработка утилиты `which(1)` на Go

Go может работать с вашей операционной системой с помощью набора пакетов. Хорошим способом изучения нового языка программирования неизменно является попытка реализовать упрощенные версии традиционных утилит UNIX. В этом разделе вы увидите реализованную на Go версию утилиты `which(1)`, которая поможет вам разобраться в том, как Go взаимодействует с базовой ОС и получает переменные среды.

Представленный код, реализующий функционал `which(1)`, можно разделить на три логические части. Первая посвящена получению входного аргумента, представляющего собой имя исполняемого файла, который утилита будет искать.

Вторая отвечает за чтение переменной среды PATH, ее разделение и перебор каталогов переменной PATH. Третья посвящена поиску в этих каталогах нужного двоичного файла и выяснению того, возможно ли его найти в принципе, а также является ли он обычным файлом или исполняемым. Если нужный исполняемый файл найден, то программа прекращает работу с помощью оператора `return`. В противном случае она завершается после окончания цикла `for` и прекращения работы функции `main()`.

Теперь взглянем на сам код, начиная с логической преамбулы, которая обычно включает имя пакета, оператор `import` и другие определения с глобальной областью действия:

```
package main

import (
    "fmt"
    "os"
    "path/filepath"
)
```

Пакет `fmt` используется для вывода на экран, пакет `os` — для взаимодействия с базовой операционной системой, а пакет `path/filepath` — для работы с содержимым переменной PATH, которая считывается как строка с длиной, зависящей от количества содержащихся в ней каталогов.

Вторая логическая часть утилиты выглядит так:

```
func main() {
    arguments := os.Args
    if len(arguments) == 1 {
        fmt.Println("Please provide an argument!")
        return
    }
    file := arguments[1]

    path := os.Getenv("PATH")
    pathSplit := filepath.SplitList(path)
    for _, directory := range pathSplit {
```

Сначала мы считываем аргументы командной строки программы (`os.Args`) и сохраняем первый аргумент в переменной `file`. Затем получаем содержимое переменной среды PATH и разделяем его с помощью функции `filepath.SplitList()`, которая предлагает *переносимый* способ разделения списка путей. Наконец, перебираем все каталоги переменной PATH, используя цикл `for` вместе с `range`, поскольку `filepath.SplitList()` возвращает срез.

Оставшаяся часть утилиты содержит следующий код:

```
 fullPath := filepath.Join(directory, file)
// Существует ли путь?
fileInfo, err := os.Stat(fullPath)
if err == nil {
    mode := fileInfo.Mode()
    // Это обычный файл?
    if mode.IsRegular() {
        // Является ли он исполняемым?
        if mode&0111 != 0 {
            fmt.Println(fullPath)
            return
        }
    }
}
}
```

Мы создаем полный путь и изучаем его с помощью функции `filepath.Join()`, которая используется для объединения различных частей пути с помощью разделителя, *характерного для конкретной ОС*, что позволяет `filepath.Join()` работать во всех поддерживаемых операционных системах. В этой части мы также получаем некоторую информацию о файле более низкого уровня. Нужно помнить, что в UNIX все представляет собой файл, а это значит, необходимо убедиться в том, что мы имеем дело с обычным файлом, который также является исполняемым.



В этой главе мы приводим весь код представленных исходных файлов. Однако начиная с главы 2 ситуация изменится. Этим мы убиваем сразу двух зайцев: вы видите только действительно важный код, а мы экономим место в книге.

Выполнение `which.go` генерирует следующий результат:

```
$ go run which.go which
/usr/bin/which
$ go run which.go doesNotExist
```

Последней команде не удалось найти исполняемый файл `DoesNotExist`. В соответствии с философией UNIX и тем, как работают конвейеры UNIX, утилиты ничего не выводят на экран, если им нечего сказать. Однако код выхода, равный 0, означает успех, тогда как *ненулевой* код выхода обычно означает неудачу.

Хотя вывод сообщений об ошибках на экран довольно полезен, бывают случаи, когда требуется сохранить все сообщения об ошибках и иметь возможность выполнять по ним поиск по мере необходимости. В таком случае вам нужно использовать один или несколько файлов журнала.

Вывод информации в лог

Все системы UNIX имеют собственные файлы журналов (логов) для протоколирования информации, поступающей от запущенных серверов и программ. Обычно большинство файлов системного журнала системы UNIX можно найти в каталоге `/var/log`. Однако файлы журналов многих популярных сервисов, таких как Apache и Nginx, расположены в другом месте в зависимости от их конфигурации.



Ведение журнала и внесение информации в файлы журналов — это практический способ асинхронного анализа данных и информации из вашего программного обеспечения либо локально, либо на центральном сервере журналов, а также с помощью серверного программного обеспечения, такого как Elasticsearch, Beats и Grafana Loki.

Вообще говоря, использование файла журнала для записи некоторой информации раньше считалось лучшей практикой, нежели вывод той же информации на экран, по двум причинам: во-первых, вывод не теряется, поскольку хранится в файле, и, во-вторых, вы можете осуществлять поиск и обрабатывать файлы журнала с помощью инструментов UNIX, таких как `grep(1)`, `awk(1)` и `sed(1)`, что не получится сделать, если сообщения выводятся в окне терминала.

Однако сейчас это уже не так.

Поскольку мы обычно запускаем наши сервисы через `systemd`, программы должны выводить лог в `stdout`, чтобы `systemd` имел возможность поместить данные в журнал. Страница <https://12factor.net/logs> содержит дополнительную информацию о журналах приложений. Кроме того, в облачных приложениях рекомендуется просто выводить лог в `stderr` и позволить контейнерной системе перенаправить поток `stderr` в нужное место.

Сервис журналирования UNIX поддерживает два свойства, названные *уровнем журналирования* (logging level) и *средством журналирования* (logging facility). Уровень журналирования — это значение, которое определяет серьезность записи в журнале. Существуют различные уровни журналирования, включающие в себя `debug`, `info`, `notice`, `warning`, `err`, `crit`, `alert` и `emerg` (в обратном порядке важности). Пакет `log` стандартной библиотеки Go не поддерживает работу с уровнями журналирования. Средство журналирования напоминает категорию, при-

сваиваемую записи в журнале, и может принимать значения auth, authpriv, cron, daemon, kern, lpr, mail, mark, news, syslog, user, UUCP, local0, local1, local2, local3, local4, local5, local6 или local7 и определяется внутри `/etc/syslog.conf`, `/etc/rsyslog.conf` либо другого подходящего файла в зависимости от серверного процесса, используемого для ведения системного журнала на вашем компьютере UNIX. Это означает, что если средство журналирования определено некорректно, то запись не будет обработана. Таким образом, сообщения журнала, которые вы в него отправляете, могут быть проигнорированы и, следовательно, утрачены.

Пакет `log` отправляет сообщения журнала в стандартную ошибку. Частью пакета `log` является `log/syslog`, который позволяет отправлять сообщения журнала на сервер `syslog` вашего компьютера. Хотя по умолчанию `log` пишет в стандартную ошибку, использование `log.SetOutput()` меняет это поведение. В список функций журналирования входят `log.Printf()`, `log.Print()`, `log.Println()`, `log.Fatalf()`, `log.Fatalln()`, `log.Panic()`, `log.Panicln()` и `log.Panicf()`.



Ведение журнала предназначено для кода приложения, а не библиотеки. Если вы разрабатываете библиотеки, то не включайте в них журналирование.

Чтобы осуществить запись в системные журналы, вам необходимо вызвать функцию `syslog.New()` с соответствующими параметрами. Запись в основной файл системного журнала осуществляется путем простого вызова `syslog.New()` с параметром `syslog.LOG_SYSLOG`. После этого нужно сообщить вашей Go-программе, что вся информация о ведении журнала поступает в новое средство ведения журнала — это реализуется с помощью вызова функции `log.SetOutput()`. Процесс показан в следующем коде — введите его в текстовом редакторе и сохраните как `systemLog.go`:

```
package main

import (
    "log"
    "log/syslog"
)

func main() {
    sysLog, err := syslog.New(syslog.LOG_SYSLOG, "systemLog.go")

    if err != nil {
        log.Println(err)
        return
    } else {
        log.SetOutput(sysLog)
        log.Print("Everything is fine!")
    }
}
```

После вызова `log.SetOutput()` вся информация журнала поступает в переменную журналирования `syslog`, которая перенаправляет ее в `syslog.LOG_SYSLOG`. Пользовательский текст для записей журнала, поступающих из этой программы, указывается в качестве второго параметра при вызове `syslog.New()`.



Обычно требуется сохранить данные журнала в пользовательских файлах, поскольку они группируют соответствующую информацию, что облегчает обработку и проверку.

При выполнении `systemLog.go` данные не выводятся. Однако если вы посмотрите на системные журналы, например, компьютера с macOS Big Sur, то внутри `/var/log/system.log` обнаружите записи, подобные следующим:

```
Dec 5 16:20:10 iMac systemLog.go[35397]: 2020/12/05 16:20:10
Everything is fine!
Dec 5 16:43:18 iMac systemLog.go[35641]: 2020/12/05 16:43:18
Everything is fine!
```

Число в скобках — это идентификатор процесса, который внес запись в журнал, — в нашем случае это 35397 и 35641.

Аналогично, если вы запустите `journalctl -xe` на компьютере с Linux, то можете увидеть записи, подобные этим:

```
Dec 05 16:33:43 thinkpad systemLog.go[12682]: 2020/12/05 16:33:43
Everything is fine!
Dec 05 16:46:01 thinkpad systemLog.go[12917]: 2020/12/05 16:46:01
Everything is fine!
```

Выходные данные в вашей операционной системе могут слегка различаться, но общая идея сохранится.

Плохие вещи происходят постоянно, даже с хорошими людьми и хорошим ПО. Так что следующий подраздел мы посвятим способам, с помощью которых Go справляется с плохими ситуациями в ваших программах.

Функции `log.Fatal()` и `log.Panic()`

`log.Fatal()` используется, когда произошло что-то неправильное и вам просто требуется выйти из программы как можно скорее после сообщения об этой плохой ситуации. Вызов `log.Fatal()` завершает Go-программу в точке, где `log.Fatal()` была вызвана после вывода сообщения об ошибке. В большинстве случаев оно может содержать `Not enough arguments`, `Cannot access file` или что-то подобное. Кроме того, функция возвращает ненулевой код выхода, который в UNIX указывает на ошибку.

Бывают ситуации, когда программа вот-вот аварийно завершится и вы хотите получить как можно больше информации о сбое — `log.Panic()` подразумевает нечто действительно неожиданное и неизвестное, например невозможность найти файл, который был ранее доступен, или нехватку места на диске. Аналогично функции `log.Fatal()`, `log.Panic()` выводит пользовательское сообщение и немедленно завершает работу Go-программы.

Имейте в виду, что, с одной стороны, `log.Panic()` эквивалентна вызову `log.Print()`, за которым следует вызов `panic()` — встроенной функции, которая останавливает выполнение текущей функции и начинает паниковать. После этого происходит возврат в вызывающую функцию. С другой стороны, `log.Fatal()` вызывает `log.Print()`, а затем `os.Exit(1)`, что является способом немедленного завершения текущей программы.

Как `log.Fatal()`, так и `log.Panic()` показаны в файле `logs.go`, который содержит следующий Go-код:

```
package main

import (
    "log"
    "os"
)

func main() {
    if len(os.Args) != 1 {
        log.Fatal("Fatal: Hello World!")
    }
    log.Panic("Panic: Hello World!")
}
```

Если запустить `logs.go` без каких-либо аргументов командной строки, то он вызовет `log.Panic()`. В противном случае — `log.Fatal()`. Это показано в следующем выводе из системы Arch Linux:

```
$ go run logs.go
2020/12/03 18:39:26 Panic: Hello World!
panic: Panic: Hello World!

goroutine 1 [running]:
log.Panic(0xc00009ef68, 0x1, 0x1)
    /usr/lib/go/src/log/log.go:351 +0xae
main.main()
    /home/mtsouk/Desktop/mGo3rd/code/ch01/logs.go:12 +0x6b
exit status 2
$ go run logs.go 1
2020/12/03 18:39:30 Fatal: Hello World!
exit status 1
```

Таким образом, вывод `log.Panic()` включает в себя дополнительную низкоуровневую информацию, которая, как мы надеемся, поможет вам разрешить сложные ситуации, возникшие в вашем Go-коде.

Запись в пользовательский файл журнала

В большинстве случаев, и особенно в приложениях и сервисах, развернутых в рабочей среде, вам часто нужно просто записать данные протокола в выбранный файл журнала. Для этого может быть множество причин, включая запись отладочных данных без вмешательства в файлы системного журнала или хранение ваших собственных данных журнала отдельно от системных журналов в целях передачи его или сохранения в базе данных либо программном обеспечении, таком как Elasticsearch. В этом подразделе рассказывается, как выполнять запись в пользовательский файл журнала, который обычно относится к конкретному приложению.



Запись в файлы и файловый ввод/вывод вы изучите в главе 6, однако сохранять информацию в файлах очень удобно при устранении ошибок и отладке Go-кода, поэтому данной темы мы коснемся и здесь.

Путь к используемому файлу журнала жестко закодирован в коде с использованием глобальной переменной `LOGFILE`. Для целей этой главы и для предотвращения переполнения вашей файловой системы в случае, если что-то пойдет не так, этот файл журнала находится в каталоге `/tmp`, который не является обычным местом для хранения данных, поскольку обычно `/tmp` очищается после каждой перезагрузки.

Кроме того, на данный момент это избавит вас от необходимости запускать `CustomLog.go` с правами root и от размещения ненужных файлов в ваших драгоценных системных каталогах.

Ведите следующий код и сохраните его как `CustomLog.go`:

```
package main

import (
    "fmt"
    "log"
    "os"
    "path"
)
```

```
func main() {
    LOGFILE := path.Join(os.TempDir(), "mGo.log")
    f, err := os.OpenFile(LOGFILE, os.O_APPEND|os.O_CREATE|os.O_WRONLY, 0644)
    // вызов os.OpenFile() создает файл журнала для записи,
    // если он еще не существует, или же открывает его для записи
    // путем добавления новых данных в конце (os.O_APPEND)

    if err != nil {
        fmt.Println(err)
        return
    }
    defer f.Close()
```

Ключевое слово `defer` сообщает Go, что нужно выполнить оператор непосредственно перед возвратом текущей функции. Это означает, что `f.Close()` будет выполнена непосредственно перед возвратом `main()`. Мы более подробно рассмотрим `defer` в главе 5.

```
iLog := log.New(f, "iLog ", log.LstdFlags)
iLog.Println("Hello there!")
iLog.Println("Mastering Go 3rd edition!")
}
```

Последние три оператора создают новый файл журнала на основе открытого (`f`) и записывают в него два сообщения с помощью `Println()`.



Если вы когда-нибудь решите использовать код `CustomLog.go` в реальном приложении, то вам следует изменить путь, хранящийся в файле `LOGFILE`, на что-то более осмысленное.

При выполнении `CustomLog.go` данные не выводятся. Однако что действительно важно, так это то, что при этом записывается в пользовательский журнал:

```
$ cat /tmp/mGo.log
iLog 2020/12/05 17:31:07 Hello there!
iLog 2020/12/05 17:31:07 Mastering Go 3rd edition!
```

Вывод номеров строк в записях журнала

В этом подразделе вы узнаете, как вывести имя файла, а также номер строки, в которой находится оператор, оставивший запись в журнале.

Эта функциональность реализована с использованием `log.Lshortfile` в параметрах `log.New()` или `setFlags()`. Флаг `log.Lshortfile` добавляет имя файла,

а также номер строки с Go-оператором, который внес запись журнала, в саму эту запись. Если вы используете `log.Llongfile` вместо `log.Lshortfile`, то получите полный путь к исходному файлу Go. Обычно в этом нет необходимости, особенно если у вас действительно длинный путь.

Ведите следующий код и сохраните его как `customLogLineNumber.go`:

```
package main

import (
    "fmt"
    "log"
    "os"
    "path"
)

func main() {
    LOGFILE := path.Join(os.TempDir(), "mGo.log")
    f, err := os.OpenFile(LOGFILE, os.O_APPEND|os.O_CREATE|os.O_WRONLY, 0644)

    if err != nil {
        fmt.Println(err)
        return
    }
    defer f.Close()

    LstdFlags := log.Ldate | log.Lshortfile
    iLog := log.New(f, "LNum ", LstdFlags)
    iLog.Println("Mastering Go, 3rd edition!")

    iLog.SetFlags(log.Lshortfile | log.LstdFlags)
    iLog.Println("Another log entry!")
}
```

Вам разрешено изменять формат записей журнала во время выполнения программы. Иными словами, при необходимости вы можете вывести в запись журнала больше аналитической информации. Это реализуется с помощью нескольких вызовов `iLog.setFlags()`.

При выполнении `customLogLineNumber.go` не генерирует выходных данных, но вносит следующие записи в путь к файлу, указанный в глобальной переменной `LOGFILE`:

```
$ cat /tmp/mGo.log
LNum 2020/12/05 customLogLineNumber.go:24: Mastering Go, 3rd edition!
LNum 2020/12/05 17:33:23 customLogLineNumber.go:27: Another log entry!
```

Скорее всего, на своем компьютере вы получите другой результат, что является вполне ожидаемым поведением.

Обзор Go-дженериков

В этом разделе обсуждаются Go-дженерики, которые являются функцией Go. В настоящее время тема дженериков в Go обсуждается сообществом Go. Так или иначе, полезно знать, как они работают, их философию и о чем конкретно ведутся споры в случае дженериков.



Go-дженерики — одно из наиболее востребованных дополнений к языку программирования Go.

Основная идея дженериков в Go, как и в любом другом поддерживающем их языке программирования, заключается в том, что при выполнении одной и той же задачи не приходится писать специальный код для поддержки нескольких типов данных.

В настоящее время Go поддерживает несколько типов данных в таких функциях, как `fmt.Println()`, путем использования пустого интерфейса и рефлексии (как интерфейсы, так и рефлексия обсуждаются в главе 4).

Однако требовать от каждого программиста написания большого количества кода и реализации множества функций и методов для поддержки нескольких пользовательских типов данных — не самое оптимальное решение. Здесь и вступают в дело дженерики, предоставляя альтернативу использованию интерфейсов и рефлексии для поддержки нескольких типов данных. В следующем коде показано, как и где могут быть полезны дженерики:

```
package main

import (
    "fmt"
)

func Print[T any](s []T) {
    for _, v := range s {
        fmt.Print(v, " ")
    }
    fmt.Println()
}

func main() {
    Ints := []int{1, 2, 3}
    Strings := []string{"One", "Two", "Three"}
    Print(Ints)
    Print(Strings)
}
```

Здесь мы видим функцию `Print()`, использующую дженерики через переменную дженериков, которая задается с помощью `[T any]` после имени функции и перед параметрами функции. Благодаря использованию `[T any]` функция `Print()` может принимать любой срез любого типа данных и работать с ним. Однако `Print()` не работает ни с чем, кроме срезов. Это вполне нормально: если ваше приложение поддерживает срезы разных типов, то данная функция все равно избавляет вас от необходимости реализовывать *несколько* функций для поддержки каждого отдельного среза. Это и есть основная идея, лежащая в основе дженериков.



В главе 4 вы узнаете о пустом интерфейсе и о том, как его можно использовать для приема данных любого типа. Однако пустой интерфейс потребует дополнительного кода для работы с определенными типами данных.

Мы заканчиваем этот раздел изложением некоторых полезных фактов о дженериках.

- Вам не нужно постоянно использовать дженерики в своих программах.
- Вы можете продолжать работать с Go, как и раньше, даже если используете дженерики.
- Вы можете полностью заменить код дженериков кодом без использования дженериков. Вопрос в том, готовы ли вы писать дополнительный код, который необходим для этого?
- Я считаю, что дженерики следуют использовать тогда, когда они помогают создавать более простой код и дизайн. Лучше иметь *повторяющийся простой* код, чем оптимальные абстракции, которые замедляют работу приложений.
- Бывают случаи, когда вам необходимо ограничить типы данных, поддерживаемые функцией, которая использует дженерики. Это неплохо, поскольку не все типы данных обладают одними и теми же возможностями. Вообще говоря, дженерики могут пригодиться при обработке типов данных, которые имеют какие-либо общие характеристики.

Вам потребуется время, чтобы освоить дженерики и использовать их в полной мере. Не торопитесь. Более подробно мы рассмотрим их в главе 13.

Разработка базового приложения телефонной книги

В этом разделе мы разработаем базовое приложение для телефонной книги в Go, что позволит нам применить полученные навыки на практике. Несмотря на свои ограничения, это приложение представляет собой утилиту командной строки, которая выполняет поиск среза структур, статически определенного (*жестко*

закодированного) в Go-коде. Утилита поддерживает две команды: `search` и `list`. Первая выполняет поиск по заданной фамилии и возвращает полную запись, если фамилия найдена, а вторая перечисляет все доступные записи.

Наша реализация имеет ряд недостатков, в том числе такие:

- если вы хотите добавить или удалить какие-либо данные, то вам необходимо изменить исходный код;
- вы не можете представить данные в отсортированной форме, что не так страшно, когда у вас три записи, но может не работать с более чем 40 записями;
- вы не можете экспорттировать свои данные или загрузить их извне;
- вы не можете распространять приложение телефонной книги в виде двоичного файла, поскольку в нем используются жестко закодированные данные.



В последующих главах мы расширим функциональность приложения телефонной книги, чтобы оно стало полностью функциональным, универсальным и эффективным.

Код `phoneBook.go` можно кратко описать следующим образом.

- Существует новый определяемый пользователем тип данных для хранения записей телефонной книги, который представляет собой структуру Go с тремя полями: `Name`, `Surname` и `Tel`. Структуры группируют набор значений в единый тип данных, позволяя передавать и получать этот набор значений как единый объект.
- Существует глобальная переменная, которая содержит данные телефонной книги и представляет собой срез структур `data`.
- Имеются две функции, которые помогут вам реализовать функциональность команд `search` и `list`.
- Содержимое глобальной переменной `data` определяется в функции `main()` с помощью нескольких вызовов `append()`. Вы можете изменять, добавлять или удалять содержимое среза `data` по мере необходимости.
- Наконец, программа может выполнять лишь одну задачу одновременно. Это означает, что для выполнения нескольких запросов вам придется запускать программу несколько раз.

Теперь рассмотрим `phoneBook.go` более подробно, начав с преамбулы:

```
package main

import (
    "fmt"
    "os"
)
```

Далее у нас есть раздел, в котором мы объявляем Go-структурку `Entry`, а также глобальную переменную `data`:

```
type Entry struct {
    Name    string
    Surname string
    Tel     string
}

var data = []Entry{}
```

После этого определяем и реализуем две функции, обеспечивающие функциональность телефонной книги:

```
func search(key string) *Entry {
    for i, v := range data {
        if v.Surname == key {
            return &data[i]
        }
    }
    return nil
}

func list() {
    for _, v := range data {
        fmt.Println(v)
    }
}
```

Функция `search()` выполняет линейный поиск по срезу `data`. Этот поиск выполняется медленно, но пока справляется с задачей, учитывая, что телефонная книга не содержит большого количества записей. Функция `list()` просто выводит содержимое среза `data`, используя цикл `for` совместно с `range`. Поскольку не требуется отображать индекс элемента, который мы выводим, то мы игнорируем его, используя символ `_`, и просто выводим структуру, содержащую фактические данные.

Наконец, в коде имеется реализация функции `main()`. Первая ее часть выглядит так:

```
func main() {
    arguments := os.Args
    if len(arguments) == 1 {
        exe := path.Base(arguments[0])
        fmt.Printf("Usage: %s search|list <arguments>\n", exe)
        return
    }
```

Переменная `exe` содержит путь к исполняемому файлу — это аккуратный и профессиональный способ напечатать имя исполняемого двоичного файла в инструкциях программы.

```
data = append(data, Entry{"Mihalis", "Tsoukalos", "2109416471"})
data = append(data, Entry{"Mary", "Doe", "2109416871"})
data = append(data, Entry{"John", "Black", "2109416123"})
```

В этой части мы проверяем, были ли нам предоставлены какие-либо командные аргументы. Если нет (`len(аргументы) == 1`), то программа выводит сообщение и завершает работу путем вызова `return`. В противном случае, прежде чем продолжить, она помещает нужные данные в срез `data`.

Оставшаяся часть реализации функции `main()` выглядит следующим образом:

```
// различаем команды
switch arguments[1] {
    // команда поиска
    case "search":
        if len(arguments) != 3 {
            fmt.Println("Usage: search Surname")
            return
        }
        result := search(arguments[2])
        if result == nil {
            fmt.Println("Entry not found:", arguments[2])
            return
        }
        fmt.Println(*result)
    // команда списка
    case "list":
        list()
    // ответ на все остальное
    default:
        fmt.Println("Not a valid option")
}
```

В этом коде используется блок `case`, который очень удобен, когда требуется написать читаемый код и избежать использования нескольких вложенных блоков `if`. Блок `case` различает две поддерживаемые команды, проверяя значение `arguments[1]`. Если команда не распознана, то вместо нее выполняется ветка `default`. В случае команды `search` также рассматривается `arguments[2]`.

Работа с `phoneBook.go` выглядит следующим образом:

```
$ go build phoneBook.go
$ ./phoneBook list
```

```
{Mihalis Tsoukalos 2109416471}
{Mary Doe 2109416871}
{John Black 2109416123}
$ ./phoneBook search Tsoukalos
{Mihalis Tsoukalos 2109416471}
$ ./phoneBook search Tsouk
Entry not found: Tsouk
$ ./phoneBook
Usage: ./phoneBook search|list <arguments>
```

Первая команда перечисляет содержимое телефонной книги, тогда как вторая выполняет поиск по заданной фамилии (`Tsoukalos`). Третья команда ищет что-то, чего нет в телефонной книге, а последняя компилирует `phoneBook.go` и запускает сгенерированный исполняемый файл без каких-либо аргументов, что влечет за собой вывод инструкций.

Несмотря на недостатки, приложение `phoneBook.go` имеет понятный дизайн, который можно легко расширять, и работает ожидаемым образом, что служит отличной отправной точкой. Мы будем совершенствовать это приложение в следующих главах по мере того, как будем изучать более расширенные концепции.

Упражнения

- Наша версия `which(1)` останавливается после нахождения первого вхождения нужного исполняемого файла. Внесите необходимые изменения в `which.go`, чтобы найти все возможные вхождения нужного исполняемого файла.
- Текущая версия `which.go` обрабатывает только первый аргумент командной строки. Внесите необходимые изменения в `which.go`, чтобы принять переменную `PATH` и выполнить поиск нескольких исполняемых двоичных файлов.
- Ознакомьтесь с документацией пакета `fmt` по адресу <https://golang.org/pkg/fmt/>.

Резюме

Если вы столкнулись с Go впервые, то благодаря информации из этой главы получите представление о преимуществах языка, внешнем виде Go-кода и некоторых важных характеристиках Go, таких как переменные, итерации, управление потоком и модель параллелизма Go. Если вы уже знакомы с этим языком, то глава поможет освежить ваши знания о том, чем он отличается от других и для каких видов программного обеспечения его рекомендуется использовать. Наконец, мы создали базовое приложение для телефонной книги, используя техники, которые успели изучить.

В следующей главе мы более подробно рассмотрим основные типы данных Go.

Дополнительные ресурсы

- Официальный сайт Go: <https://golang.org/>.
- The Go Playground: <https://play.golang.org/>.
- Пакет log: <https://golang.org/pkg/log/>.
- Elasticsearch Beats: <https://www.elastic.co/beats/>.
- Grafana Loki: <https://grafana.com/oss/loki/>.
- Microsoft Visual Studio: <https://visualstudio.microsoft.com/>.
- Стандартная библиотека Go: <https://golang.org/pkg/>.

2

Основные типы данных Go

Данные хранятся и используются в переменных, и все переменные Go должны иметь тип данных, который определяется явно или неявно. Знание встроенных типов данных Go позволяет вам понять, как манипулировать простыми значениями данных и создавать более сложные структуры данных, когда простых типов недостаточно или они неэффективны в рамках данной задачи.

В текущей главе рассказывается об основных типах Go и структурах данных, которые позволяют *группировать данные одного типа*. Однако начнем с чего-то более практического: представьте, что нам нужно прочитать данные в качестве аргументов командной строки утилиты. Как можно гарантировать, что прочитанное окажется именно тем, что мы ожидали? Как справляться с ошибочными ситуациями? Как насчет чтения из командной строки не только чисел и строк, но и дат и времени? Придется ли писать собственный синтаксический анализатор для работы с датами и временем?

В этой главе мы ответим на эти и многие другие вопросы, реализовав следующие три утилиты:

- утилиту командной строки, которая анализирует даты и времена;
- утилиту, которая генерирует случайные числа и случайные строки;
- новую версию приложения телефонной книги, содержащую случайно сгенерированные данные.

В этой главе:

- тип данных `error`;
- числовые типы данных;
- нечисловые типы данных;
- Go-константы;
- группировка схожих данных;
- указатели;
- генерация случайных чисел;
- обновление приложения телефонной книги.

Начнем эту главу с типа данных `error`, поскольку ошибки в Go играют ключевую роль.

Тип данных `error`

Для представления условий ошибки и сообщений Go содержит специальный тип данных `error`. На практике это означает, что этот язык обрабатывает ошибки как значения. Чтобы успешно программировать на Go, вы должны иметь представление об ошибках, которые могут возникнуть по мере использования функций и методов, а также соответствующим образом их обрабатывать.

Как вы уже знаете из предыдущей главы, Go следует такому соглашению о значениях `error`: если значение переменной `error` равно `nil`, то ошибки не было. В качестве примера мы рассмотрим `strconv.Atoi()`, который используется для преобразования значения `string` в `int` (`Atoi` расшифровывается как ASCII to Int). Как и указано в сигнатуре, `strconv.Atoi()` возвращает `(int, error)`. Значение `error`, равное `nil`, означает, что преобразование прошло успешно и что значение `int` можно использовать. Если значение `error` отлично от `nil`, это означает, что преобразование было неудачным и значение в `string` не является допустимым значением `int`.



Если вы хотите узнать больше о `strconv.Atoi()`, выполните `go doc strconv.Atoi` в окне вашего терминала.

Вы можете задаться вопросом, как быть, если необходимо создать собственные сообщения об ошибках. Возможно ли это? Если требуется вернуть пользовательскую ошибку, то вы можете использовать `errors.New()` из пакета `errors`. Обычно это делается внутри функций, отличных от `main()`, поскольку `main()` ничего

не возвращает никакой другой функции. Кроме того, хорошее место для определения ваших пользовательских ошибок — это создаваемые вами пакеты Go.



Скорее всего, для работы с ошибками в своих программах вам не понадобится функциональность пакета `errors`. Кроме того, вам не придется определять пользовательские сообщения об ошибках, если только вы не создаете комплексные приложения или пакеты.

Если вы хотите отформатировать свои сообщения об ошибках (как это делает `fmt.Sprintf()`), можете использовать функцию `fmt.Errorf()`, которая упрощает создание пользовательских сообщений об ошибках — функция `fmt.Errorf()` возвращает значение `error` точно так же, как это делает `errors.New()`.

А теперь мы должны поговорить кое о чем важном: нам необходима общая тактика обработки ошибок в каждом приложении, которая не должна меняться. На практике это означает следующее.

- Все сообщения об ошибках должны обрабатываться на одном уровне, то есть все ошибки должны либо возвращаться в вызывающую функцию, либо обрабатываться на месте.
- Должно быть четко задокументировано, как обрабатываются критические ошибки. Это означает, что будут ситуации, когда критическая ошибка в одних случаях приведет к завершению работы программы, а в других — лишь к выводу на экран предупреждающего сообщения.
- Считается хорошей практикой отправлять все сообщения об ошибках в *сервис журнализации* вашего компьютера, поскольку таким образом сообщения об ошибках могут быть просмотрены позже. Однако это не всегда так, поэтому будьте осторожны — например, облачные приложения работают иначе.



Тип данных `error` фактически определяется как интерфейс — интерфейсы мы рассмотрим в главе 4.

Ведите следующий код в текстовом редакторе и сохраните как `error.go` в каталоге, куда вы помещаете весь код для этой главы. Хорошей идеей будет назвать этот каталог `ch02`.

```
package main  
  
import (
```

```
    "errors"
    "fmt"
    "os"
    "strconv"
)
```

Первая часть — это преамбула программы. Файл `error.go` использует пакеты `fmt`, `os`, `strconv` и `errors`.

```
// пользовательское сообщение об ошибке с помощью errors.New()
func check(a, b int) error {
    if a == 0 && b == 0 {
        return errors.New("this is a custom error message")
    }
    return nil
}
```

Здесь мы реализуем функцию `check()`, которая возвращает значение `error`. Если оба входных параметра `check()` равны 0, то функция возвращает пользовательское сообщение об ошибке, используя `errors.New()`, — в противном случае возвращает `nil`, и это означает, что все в порядке.

```
// пользовательское сообщение об ошибке с помощью fmt.Errorf()
func formattedError(a, b int) error {
    if a == 0 && b == 0 {
        return fmt.Errorf("a %d and b %d. UserID: %d", a, b, os.Getuid())
    }
    return nil
}
```

Здесь мы реализуем `formattedError()`, функцию, которая возвращает форматированное сообщение об ошибке, используя `fmt.Errorf()`. Среди прочего, с помощью `os.Getuid()` в сообщении об ошибке выводится идентификатор пользователя, который выполнил программу. Если вы хотите создать пользовательское сообщение об ошибке, то с помощью `fmt.Errorf()` получите больше контроля над выводом.

```
func main() {
    err := check(0, 10)
    if err == nil {
        fmt.Println("check() ended normally!")
    } else {
        fmt.Println(err)
    }

    err = check(0, 0)
    if err.Error() == "this is a custom error message" {
```

```

        fmt.Println("Custom error detected!")
    }

    err = formattedError(0, 0)
    if err != nil {
        fmt.Println(err)
    }

    i, err := strconv.Atoi("-123")
    if err == nil {
        fmt.Println("Int value is", i)
    }

    i, err = strconv.Atoi("Y123")
    if err != nil {
        fmt.Println(err)
    }
}

```

Этот код представляет собой реализацию функции `main()`, где вы можете видеть многократное использование оператора `if err != nil`, а также `if err == nil`, который позволяет перед выполнением нужного кода убедиться, что все в порядке.

При выполнении `error.go` мы получаем такой вывод:

```
$ go run error.go
check() ended normally!
Custom error detected!
a 0 and b 0. UserID: 501
Int value is -123
strconv.Atoi: parsing "Y123": invalid syntax
```

Теперь, когда вы узнали о типе данных `error`, о том, как создавать пользовательские ошибки и использовать значения `error`, мы вернемся к основным типам данных в Go, которые можно логически разделить на две основные категории: числовые и нечисловые. Go также поддерживает тип данных `bool`, который может принимать только значение `true` или `false`.

Числовые типы данных

Go поддерживает целочисленные значения, значения с плавающей запятой и комплексные числа в различных вариациях в зависимости от занимаемого ими объема памяти. Такой подход экономит память и вычислительное время. Целочисленные типы могут быть как со знаком, так и без знака, чего нельзя сказать о числах с плавающей запятой.

В табл. 2.1 перечислены числовые типы данных Go.

Таблица 2.1

Тип данных	Описание
<code>int8</code>	8-разрядное целое число со знаком
<code>int16</code>	16-разрядное целое число со знаком
<code>int32</code>	32-разрядное целое число со знаком
<code>int64</code>	64-разрядное целое число со знаком
<code>int</code>	32- или 64-разрядное целое число со знаком
<code>uint8</code>	8-разрядное целое число без знака
<code>uint16</code>	16-разрядное целое число без знака
<code>uint32</code>	32-разрядное целое число без знака
<code>uint64</code>	64-разрядное целое число без знака
<code>uint</code>	32- или 64-разрядное целое число без знака
<code>float32</code>	32-разрядное число с плавающей запятой
<code>float64</code>	64-разрядное число с плавающей запятой
<code>complex64</code>	Комплексное число с частями <code>float32</code>
<code>complex128</code>	Комплексное число с частями <code>float64</code>

Типы данных `int` и `uint` особенные, поскольку являются наиболее эффективными размерами для целых чисел со знаком и без знака на данной платформе и могут быть либо 32, либо 64 бита каждый — их размер определяется самим Go. Тип данных `int` — это наиболее широко используемый тип данных в Go из-за своей универсальности.

В коде далее показано использование числовых типов данных — всю программу `numbers.go` вы можете найти в каталоге `ch02` репозитория книги на GitHub.

```
func main() {
    c1 := 12 + 1i
    c2 := complex(5, 7)
    fmt.Printf("Type of c1: %T\n", c1)
    fmt.Printf("Type of c2: %T\n", c2)
```

Здесь мы создаем две комплексные переменные двумя различными способами — оба допустимы и эквивалентны. Если вы не увлекаетесь математикой, то, скорее

всего, не будете использовать комплексные числа в своих программах. Однако существование комплексных чисел показывает, насколько современен Go.

```
var c3 complex64 = complex64(c1 + c2)
fmt.Println("c3:", c3)
fmt.Printf("Type of c3: %T\n", c3)
cZero := c3 - c3
fmt.Println("cZero:", cZero)
```

Далее мы продолжаем работу с комплексными числами путем сложения и вычитания двух их пар. Хотя `CZero` равно нулю, это все еще комплексное число и переменная типа `complex64`.

```
x := 12
k := 5
fmt.Println(x)
fmt.Printf("Type of x: %T\n", x)

div := x / k
fmt.Println("div", div)
```

В этой части мы определяем две целочисленные переменные `x` и `k` — их тип данных Go определяет на основе их начальных значений. Обе получают тип `int`, который Go предпочитает использовать для хранения целочисленных значений. Кроме того, когда вы делите два целых значения, то получаете целочисленный результат, даже если деление не является идеальным. Это означает, что вам следует проявить особую осторожность, если это не то, что вам нужно. Пример можно увидеть в следующем фрагменте кода:

```
var m, n float64
m = 1.223
fmt.Println("m, n:", m, n)

y := 4 / 2.3
fmt.Println("y:", y)

divFloat := float64(x) / float64(k)
fmt.Println("divFloat", divFloat)
fmt.Printf("Type of divFloat: %T\n", divFloat)
}
```

Здесь мы работаем со значениями и переменными `float64`. Поскольку `n` не имеет начального значения, ему *автоматически присваивается* нулевое значение его типа данных (`0` для типа данных `float64`).

Кроме того, в коде представлен метод деления целых значений и получения результата с плавающей запятой, который заключается в использовании `float64():` `divFloat := float64(x) / float64(k)`. Это преобразование типа, при котором два целых числа (`x` и `k`) преобразуются в значения `float64`. Поскольку

при делении двух значений `float64` результатом будет `float64`, мы получаем результат в виде желаемого типа данных.

При выполнении `numbers.go` мы получаем такой вывод:

```
$ go run numbers.go
Type of c1: complex128
Type of c2: complex128
c3: (17+8i)
Type of c3: complex64
cZero: (0+0i)
12
Type of x: int
div 2
m, n: 1.223 0
y: 1.7391304347826086
divFloat 2.4
Type of divFloat: float64
```

Выходные данные показывают `c1` и `c2` — это значения `complex128`, что является предпочтительным комплексным типом данных для машины, на которой был запущен код. Однако `c3` является значением `complex64`, поскольку переменная была создана с помощью `complex64()`. Значение `n` равно `0`, поскольку переменная `n` не была инициализирована; это значит, Go автоматически присвоил `n` нулевое значение ее типа данных.

Рассмотрев числовые типы данных, перейдем к нечисловым.

Нечисловые типы данных

Go поддерживает *строки*, *символы*, *руны*, *даты* и *время*. Однако в Go нет выделенного типа данных `char`. Мы начнем с изучения типов данных, имеющих отношение к строкам.



Для Go даты и время — одно и то же, поэтому они представлены одним и тем же типом данных. Однако вам решать, содержит ли переменная времени и даты достоверную информацию.

Строки, символы и руны

Для представления строк в Go поддерживается тип данных `string`. Стока Go — это просто набор байтов, и доступ можно получить как к строке, так и к массиву. В одном байте может храниться любой символ ASCII, однако для хранения одного символа Unicode обычно требуется несколько байтов.

В настоящее время поддержка символов Unicode является распространенным требованием — Go разработан с учетом поддержки Unicode, что является основной причиной наличия типа данных `rune`. Руна — это значение `int32`, которое используется для представления одного кодового пункта Unicode. Руна представляет собой целое значение и используется для представления отдельных символов Unicode или, реже, для предоставления информации о форматировании.



Хотя руна и является значением `int32`, вы не можете сравнивать руну со значением `int32`. Go относится к этим типам данных как к совершенно разным.

Вы можете создать новый байтовый срез из заданной строки, используя оператор `[]byte("A String")`. Располагая переменной `b` байтового среза, вы можете преобразовать ее в строку, используя оператор `string(b)`. При работе с байтовыми срезами, содержащими символы Unicode, количество байтов в байтовом срезе не всегда связано с количеством символов, поскольку для представления большинства символов Unicode требуется несколько байтов. В результате, когда вы пытаетесь вывести каждый отдельный байт байтового среза с помощью `fmt.Println()` или `fmt.Print()`, на выходе получается не текст, представленный в виде символов, а целочисленные значения. Если требуется вывести содержимое байтового среза в виде текста, то следует распечатать его, используя либо `string(byteSliceVar)`, либо `fmt.Printf()` и `%s`, чтобы сообщить `fmt.Printf()`, что вы хотите вывести строку. Вы можете инициализировать новый байтовый срез с помощью нужной строки, используя такой оператор, как `[]byte("My Initialization String")`.



Мы рассмотрим байтовые срезы более подробно в соответствующем подразделе.

Вы можете определить руну, используя одинарные кавычки: `r := '€'`, а вывести целое значение составляющих ее байтов с помощью `fmt.Println(r)` — в этом случае целочисленное значение равно `8364`. Для вывода в виде одного символа Unicode потребуется использование управляющей строки `%c` в `fmt.Printf()`.

Поскольку доступ к строкам возможен в виде массивов, вы можете перебирать руны строки с помощью цикла `for` или указывать на определенный символ, если знаете его положение в строке. Длина строки совпадает с количеством символов в строке, что обычно неверно для байтовых срезов, поскольку для символов Unicode обычно требуется несколько байтов.

В следующем коде Go показано использование строк и рун, а также работа со строками в собственном коде. Вы можете найти всю программу в файле `text.go` в каталоге `ch02` репозитория книги на GitHub.

Первая часть программы определяет строковый литерал, содержащий символ Unicode. Затем она обращается к первому символу, как если бы строка была массивом.

```
func main() {
    aString := "Hello World! €"
    fmt.Println("First character", string(aString[0]))
```

Следующая часть посвящена работе с рунами.

```
// руны
// руна
r := '€'
fmt.Println("As an int32 value:", r)
// преобразование рун в текст
fmt.Printf("As a string: %s and as a character: %c\n", r, r)

// вывести существующую строку в виде рун
for _, v := range aString {
    fmt.Printf("%x ", v)
}
fmt.Println()
```

Сначала мы определяем руну `r`. Что делает ее руной, так это использование одинарных кавычек вокруг `€`. Руна представляет собой значение `int32` и выводится как таковая с помощью `fmt.Println()`. Управляющая строка `%c` в `fmt.Printf()` выведет руну в виде символа.

Затем мы перебираем `aString` как срез или массив, используя цикл `for` с `range`, и выводим содержимое `aString` в виде рун.

```
// вывести существующую строку в виде символов
for _, v := range aString {
    fmt.Printf("%c", v)
}
fmt.Println()
```

Наконец, мы выполняем итерацию по `aString` как по срезу или массиву, используя цикл `for` с `range`, и выводим содержимое `aString` в виде символов.

При выполнении `text.go` мы получаем такой вывод:

```
$ go run text.go
First character H
As an int32 value: 8364
As a string: %!s(int32=8364) and as a character: €
48 65 6c 6c 6f 20 57 6f 72 6c 64 21 20 20ac
Hello World! €
```

Первая строка выходных данных показывает, что мы можем получить доступ к `string` в виде массива, тогда как вторая строка проверяет, что руна является

целочисленным значением. Третья строка показывает, чего ожидать, когда вы выводите `rune` как `string` и как символ. Правильный способ — напечатать ее как символ. Пятая строка показывает, как вывести строку в виде рун, а последняя демонстрирует результат обработки строки в виде символов с использованием `range` и цикла `for`.

Преобразование из `int` в `string`

Вы можете преобразовать целое значение в строку двумя основными способами: с помощью `string()` и функции из пакета `strconv`. Однако эти два метода принципиально различны. Функция `string()` преобразует целое значение в кодовый пункт Unicode, который представляет собой один символ. А такие функции, как `strconv.FormatInt()` и `strconv.Itoa()`, преобразуют целочисленное значение в строковое с тем же представлением и тем же количеством символов.

Это показано в программе `intString.go` — ее наиболее важные операторы представлены ниже. Всю программу можно увидеть в репозитории книги на GitHub.

```
input := strconv.Itoa(n)
input = strconv.FormatInt(int64(n), 10)
input = string(n)
```

При выполнении `intString.go` мы получаем такой вывод:

```
$ go run intString.go 100
strconv.Itoa() 100 of type string
strconv.FormatInt() 100 of type string
string() d of type string
```

Тип выходных данных всегда `string`, однако `string()` преобразовал `100` в `d`, поскольку ASCII-представление `d` — это `100`.

Пакет `unicode`

Стандартный Go-пакет `unicode` содержит различные удобные функции для работы с кодовыми пунктами Unicode. Один из них, `unicode.IsPrint()`, помогает определить части строки, которые можно выводить с помощью рун.

В следующем фрагменте кода показана функциональность пакета `unicode`:

```
for i := 0; i < len(sL); i++ {
    if unicode.IsPrint(rune(sL[i])) {
        fmt.Printf("%c\n", sL[i])
    } else {
        fmt.Println("Not printable!")
    }
}
```

Цикл `for` проходит по содержимому строки, определенной как список рун ("`\x99\x00ab\x50\x00\x23\x50\x29\x9c`"), в то время как `unicode.IsPrint()` проверяет, доступен ли символ для вывода. Если функция возвращает значение `true`, то руна доступна для вывода.

Вы можете найти этот фрагмент кода внутри исходного файла `unicode.go` в каталоге `ch02` в репозитории книги на GitHub. При выполнении `unicode.go` мы получаем такой вывод:

```
Not printable!
Not printable!
a
b
P
Not printable!
#
P
)
Not printable!
```

С помощью этой утилиты очень удобно фильтровать ваши входные данные или данные перед выводом на экран, сохранением в файлах журналов, передачей по сети или сохранением в базе данных.

Пакет `strings`

Стандартный Go-пакет `strings` позволяет манипулировать строками UTF-8 в Go и содержит множество эффективных функций. Многие из этих функций показаны в файле `useStrings.go`, который можно найти в каталоге `ch02` репозитория книги на GitHub.



Если вы работаете с текстом и его обработкой, то вам определенно придется подробно изучить функции пакета `strings`. Поэтому убедитесь, что поэкспериментировали со всеми этими функциями и проработали множество примеров, которые помогут вам прояснить тот или иной аспект.

Наиболее важными частями `useStrings.go` являются:

```
import (
    "fmt"
    s "strings"
    "unicode"
)

var f = fmt.Printf
```

Пакет `strings` мы будем использовать многократно, поэтому создадим для него удобный псевдоним `s`. Мы делаем то же самое для функции `fmt.Printf()`, где создаем глобальный псевдоним, используя переменную `f`. Две эти краткие формы сделают код менее заполненным длинными повторяющимися строками. Вы можете использовать это при изучении Go, но так не рекомендуется делать в производственном программном обеспечении, поскольку подобный подход ухудшает читабельность кода.

Первый фрагмент кода выглядит следующим образом:

```
f("EqualFold: %v\n", s.EqualFold("Mihalis", "MIHALis"))
f("EqualFold: %v\n", s.EqualFold("Mihalis", "MIHALi"))
```

Функция `strings.EqualFold()` сравнивает две строки без учета их регистра и возвращает `true`, если они совпадают, и `false` в противном случае.

```
f("Index: %v\n", s.Index("Mihalis", "ha"))
f("Index: %v\n", s.Index("Mihalis", "Ha"))
```

Функция `strings.Index()` проверяет, можно ли найти строку во втором параметре в строке, заданной в качестве первого параметра, и возвращает индекс, в котором она впервые была найдена. При неудачном поиске она возвращает значение `-1`.

```
f("Prefix: %v\n", s.HasPrefix("Mihalis", "Mi"))
f("Prefix: %v\n", s.HasPrefix("Mihalis", "mi"))
f("Suffix: %v\n", s.HasSuffix("Mihalis", "is"))
f("Suffix: %v\n", s.HasSuffix("Mihalis", "IS"))
```

Функция `strings.HasPrefix()` проверяет, начинается ли данная строка (первый параметр) со строки, которая задана в качестве второго параметра. В этом коде первый вызов `strings.HasPrefix()` возвращает `true`, тогда как второй — `false`.

Аналогично, функция `strings.HasSuffix()` проверяет, заканчивается ли данная строка второй строкой. Обе функции учитывают как регистр входной строки, так и регистр второго параметра.

```
t := s.Fields("This is a string!")
f("Fields: %v\n", len(t))
t = s.Fields("ThisIs a\tstring!")
f("Fields: %v\n", len(t))
```

Весьма удобная функция `strings.Fields()` разбивает заданную строку, используя один или несколько символов пробела, как определено в `unicode.IsSpace()`, и возвращает срез подстрок, найденных во входной строке. Если входная строка содержит только пробелы, то функция возвращает пустой срез.

```
f("%s\n", s.Split("abcd efg", ""))
f("%s\n", s.Replace("abcd efg", "", "_", -1))
f("%s\n", s.Replace("abcd efg", "", "_", 4))
f("%s\n", s.Replace("abcd efg", "", "_", 2))
```

Функция `strings.Split()` позволяет разделить заданную строку в соответствии с желаемой строкой-разделителем. Функция `strings.Split()` возвращает *срез строк*. Использование "" в качестве второго параметра позволяет с помощью `strings.Split()` обрабатывать строку символ за символом.

Функция `strings.Replace()` принимает четыре параметра. Первый — это строка, которую нужно обработать. Второй содержит строку, которая, если найдена, будет заменена третьим параметром `strings.Replace()`. Последний параметр — это максимальное количество замен, которые могут быть выполнены. Если этот параметр имеет отрицательное значение, то количество выполняемых замен неограниченно.

```
f("SplitAfter: %s\n", s.SplitAfter("123++432++", "++"))

trimFunction := func(c rune) bool {
    return !unicode.IsLetter(c)
}
f("TrimFunc: %s\n", s.TrimFunc("123 abc ABC \t .", trimFunction))
```

Функция `strings.SplitAfter()` разбивает строку из первого параметра на подстроки на основе строки-разделителя, которая выступает в качестве второго параметра. Стока-разделитель включается в возвращаемый срез.

Последние строки кода определяют *функцию обрезки* `trimFunction`, которая используется в качестве второго параметра в `strings.TrimFunc()` для фильтрации заданных входных данных на основе возвращаемого значения функции обрезки, — в этом случае функция обрезки сохраняет только буквы и ничего больше на основе вызова `unicode.isLetter()`.

При выполнении `useStrings.go` мы получаем такой вывод:

```
To Upper: HELLO THERE!
To Lower: hello there
THis WILL Be A Title!
EqualFold: true
EqualFold: false
Prefix: true
Prefix: false
Suffix: true
Suffix: false
Index: 2
Index: -1
Count: 2
Count: 0
Repeat: ababababab
TrimSpace: This is a line.
TrimLeft: This is a      line.
TrimRight:      This is a      line.
Compare: 1
```

```
Compare: 0
Compare: -1
Fields: 4
Fields: 3
[a b c d e f g]
_a_b_c_d_ _e_f_g_
_a_b_c_d efg
_a_bcd efg
Join: Line 1+++Line 2+++Line 3
SplitAfter: [123++ 432++ ]
TrimFunc: abc ABC
```

Посетите страницу документации пакета `strings` по адресу <https://golang.org/pkg/strings/>, чтобы получить полный список доступных функций. Вы встретитесь с использованием этого пакета и далее в книге.

Однако хватит о строках и тексте; следующий подраздел посвящен работе с датами и временем в Go.

Время и даты

Часто нам нужно работать с информацией о дате и времени. Например, чтобы сохранить время последнего использования записи в базе данных или время вставки записи в базу данных. Это подводит нас к новой интересной теме: работа с датами и временем в Go.

Главное при работе с временем и датами в Go — тип данных `time.Time`, который представляет момент времени *с точностью до наносекунды*. Каждое значение `time.Time` связано с местоположением (часовым поясом).

Если вы являетесь пользователем UNIX, то, вероятно, уже знаете о времени эпохи UNIX и задаетесь вопросом, как получить его в Go. Функция `time.Now().Unix()` возвращает популярное время эпохи UNIX, которое представляет собой количество секунд, прошедших с 00:00:00 UTC 1 января 1970 года. Если вы хотите преобразовать время UNIX в эквивалентное значение `time.Time`, то можете использовать функцию `time.Unix()`. Если вы не пользуетесь UNIX, то, возможно, раньше не слышали о времени эпохи UNIX. Однако теперь вы знаете, что это такое!



Функция `time.Since()` вычисляет время, прошедшее с момента заданного времени, и возвращает переменную `time.Duration`. Тип данных длительности определяется как type `Duration int64`. Хотя `Duration` на самом деле является значением `int64`, вы не можете сравнивать или преобразовывать длительность в значение `int64` неявно, поскольку Go не допускает неявных преобразований типов данных.

Что касается дат и времени в Go, то необходимо упомянуть один очень важный момент: способ, с помощью которого Go анализирует строку, чтобы преобразовать ее в дату и время. Причина того, почему это так важно, заключается в том, что обычно подобные входные данные задаются в виде строки, а не допустимой переменной даты. Для синтаксического анализа используется функция `time.Parse()`, и ее полная сигнатура выглядит так: `Parse(layout, value string) (Time, error)`, где `layout` — это формат, а `value` — входные данные для анализа. Возвращаемое значение `Time` представляет собой момент времени с точностью до наносекунды и содержит информацию как о дате, так и о времени.

В табл. 2.2 показаны строки, наиболее широко используемые для анализа дат и времени.

Таблица 2.2

Константа	Значение (примеры)
05	12-часовое значение (12 вечера, 07 утра)
15	24-часовое значение (23, 07)
04	Минуты (55, 15)
05	Секунды (5, 23)
Mon	Сокращенный день недели (Tue, Fri)
Monday	День недели (Tuesday, Friday)
02	День месяца (15, 31)
2006	Год с четырьмя цифрами (2020, 2004)
06	Год с последними двумя цифрами (20, 04)
Jan	Сокращенное название месяца (Feb, Mar)
January	Полное название месяца (July, August)
MST	Часовой пояс (EST, UTC)

В таблице показано, что если вы хотите проанализировать строку `30 January 2020` и преобразовать ее в переменную даты Go, то должны сопоставить ее со строкой `02 January 2006` — вы не можете использовать что-либо другое при сопоставлении со строкой формата `30 January 2020`. Аналогично, если вы хотите проанализировать строку `15 August 2020 10:00`, то должны сопоставить ее со строкой `02 January 2006 15:04`. Документация пакета `time` (<https://golang.org/pkg/time/>) содержит еще более подробную информацию об анализе дат и времени. Тем не менее представленной здесь информации должно быть более чем достаточно для решения обычных задач.

Утилита для анализа дат и времени

В редких случаях может возникнуть ситуация, когда мы ничего не знаем о нашем вводе. Если вы не знаете точный формат ваших входных данных, вам придется сопоставлять их с несколькими строками Go, при этом не имея каких-либо гарантий успеха. Это подход, который используется в примере. Строки соответствия для дат и времени можно использовать в любом порядке.

Если вы сопоставляете строку, содержащую только дату, то ваше время будет установлено на `00:00` и, скорее всего, будет неверным. Аналогично при сопоставлении только времени ваша дата будет неверной и ее не следует использовать.



Строки форматирования можно использовать для вывода дат и времени в желаемом формате. Так, чтобы вывести текущую дату в виде `01-02-2006`, необходимо использовать `time.Now().Format("01-02-2006")`.

В приведенном ниже коде показано, как работать с временем эпохи в Go, и представлен сам процесс анализа. Создайте текстовый файл, введите следующий код и сохраните его как `dates.go`:

```
package main

import (
    "fmt"
    "os"
    "time"
)
```

Это ожидаемая преамбула любого исходного файла Go:

```
func main() {
    start := time.Now()

    if len(os.Args) != 2 {
        fmt.Println("Usage: dates parse_string")
        return
    }
    dateString := os.Args[1]
```

Так мы получаем пользовательский ввод, который хранится в переменной `dateString`. Если утилита не получает входных данных, то нет смысла продолжать работу.

```
// Это лишь дата?
d, err := time.Parse("02 January 2006", dateString)
if err == nil {
    fmt.Println("Full:", d)
    fmt.Println("Time:", d.Day(), d.Month(), d.Year())
}
```

Первый тест предназначен для сопоставления даты только с форматом `02 January 2006`. Если совпадение есть, можно получить доступ к отдельным полям переменной, содержащей действительную дату, используя `Day()`, `Month()` и `Year()`.

```
// Это дата + время?
d, err = time.Parse("02 January 2006 15:04", dateString)
if err == nil {
    fmt.Println("Full:", d)
    fmt.Println("Date:", d.Day(), d.Month(), d.Year())
    fmt.Println("Time:", d.Hour(), d.Minute())
}
```

На этот раз мы пытаемся сопоставить строку, используя шаблон `"02 January 2006 15:04"`, который содержит значение даты и времени. Если совпадение обнаружено, можно получить доступ к полям действительного времени, используя `Hour()` и `Minute()`.

```
// Это дата + время с месяцем, представленным в виде числа?
d, err = time.Parse("02-01-2006 15:04", dateString)
if err == nil {
    fmt.Println("Full:", d)
    fmt.Println("Date:", d.Day(), d.Month(), d.Year())
    fmt.Println("Time:", d.Hour(), d.Minute())
}
```

На этот раз мы пытаемся сопоставить строку с форматом `"02-01-2006 15:04"`, который содержит как дату, так и время. Обратите внимание на то, что строка должна содержать символы – и :, как указано в вызове `time.Parse()`, и что `"02-01-2006 15:04"` отличается от `"02/01/2006 1504"`.

```
// Это лишь время?
d, err = time.Parse("15:04", dateString)
if err == nil {
    fmt.Println("Full:", d)
    fmt.Println("Time:", d.Hour(), d.Minute())
}
```

Последняя проверка проводится только по времени в формате `"15:04"`. Обратите внимание, что символ : должен присутствовать в проверяемой строке.

```
t := time.Now().Unix()
fmt.Println("Epoch time:", t)
// преобразовать время эпохи в time.Time
d = time.Unix(t, 0)
fmt.Println("Date:", d.Day(), d.Month(), d.Year())
fmt.Printf("Time: %d:%d\n", d.Hour(), d.Minute())

duration := time.Since(start)
fmt.Println("Execution time:", duration)
}
```

В последней части `dates.go` показано, как работать со временем эпохи UNIX. Текущую дату и время в формате времени эпохи можно получить, используя `time.Now().Unix()`, после чего преобразовать ее в значение `time.Time` с помощью вызова `time.Unix()`.

Наконец, вы можете рассчитать продолжительность времени между текущим временем и временем в прошлом, используя вызов `time.Since()`.

При выполнении `dates.go` мы получаем такой вывод в зависимости от входных данных:

```
$ go run dates.go
Usage: dates parse_string
$ go run dates.go 14:10
Full: 0000-01-01 14:10:00 +0000 UTC
Time: 14 10
Epoch time: 1607964956
Date: 14 December 2020
Time: 18:55
Execution time: 163.032µs
$ go run dates.go "14 December 2020"
Full: 2020-12-14 00:00:00 +0000 UTC
Time: 14 December 2020
Epoch time: 1607964985
Date: 14 December 2020
Time: 18:56
Execution time: 180.029µs
```



Если аргумент командной строки, такой как `14 December 2020`, содержит пробелы, то его следует заключить в двойные кавычки, чтобы оболочка UNIX рассматривала его как единый аргумент. Запуск `go run dates.go 14 December 2020` не сработает.

Теперь, когда мы выяснили, как работать с датами и временем, пришло время узнать больше о часовых поясах.

Работа с часовыми поясами

Представленная утилита принимает дату и время и преобразует их в разные часовые пояса. Это может оказаться особенно удобно, когда нужно предварительно обработать файлы журналов из разных источников, использующих разные часовые пояса, чтобы привести эти пояса к одному.

Напомню, что перед конвертированием корректного ввода в `time.Time` вам понадобится использовать `time.Parse()` до выполнения преобразований. На этот раз входная строка содержит часовой пояс и анализируется с помощью строки `"02 January 2006 15:04 MST"`.

Чтобы преобразовать проанализированные дату и время в нью-йоркское время, программа использует следующий код:

```
loc, _ = time.LoadLocation("America/New_York")
fmt.Printf("New York Time: %s\n", now.In(loc))
```

В программе `convertTimes.go` этот метод используется несколько раз.

Ее запуск дает следующий вывод:

```
$ go run convertTimes.go "14 December 2020 19:20 EET"
Current Location: 2020-12-14 19:20:00 +0200 EET
New York Time: 2020-12-14 12:20:00 -0500 EST
London Time: 2020-12-14 17:20:00 +0000 GMT
Tokyo Time: 2020-12-15 02:20:00 +0900 JST
$ go run convertTimes.go "14 December 2020 20:00 UTC"
Current Location: 2020-12-14 22:00:00 +0200 EET
New York Time: 2020-12-14 15:00:00 -0500 EST
London Time: 2020-12-14 20:00:00 +0000 GMT
Tokyo Time: 2020-12-15 05:00:00 +0900 JST
$ go run convertTimes.go "14 December 2020 25:00 EET"
parsing time "14 December 2020 25:00": hour out of range
```

При последнем выполнении программы код анализирует 25 как час дня, что неверно, и генерирует сообщение об ошибке `hour out of range` (час выходит за границы диапазона).

Go-константы

Go поддерживает *константы*, которые представляют собой неизменяемые переменные. Константы в Go определяются с помощью ключевого слова `const`. Вообще говоря, константы могут быть *как глобальными, так и локальными переменными*.

Однако вам, возможно, стоит пересмотреть свой подход, если обнаружится, что вы определяете слишком много констант с локальной областью видимости. Основным преимуществом использования констант в ваших программах является гарантия того, что их значение не изменится во время выполнения программы. Строго говоря, значение постоянной определяется во время компиляции, а не во время выполнения — это означает, что она включена в двоичный исполняемый файл. «Под капотом» Go использует `Boolean`, `string` или `number` в качестве типа для хранения постоянных значений, поскольку это позволяет Go быть более гибким при работе с константами.

В следующем подразделе обсуждается генератор констант `iota`, который является удобным способом создания последовательностей констант.

Генератор констант `iota`

Генератор констант iota предназначен для объявления последовательности связанных значений, которые используют увеличивающиеся числа, не прибегая к необходимости явно вводить каждое из них.

Концепции, связанные с ключевым словом `const`, включая генератор констант `iota`, показаны в файле `constants.go`:

```
package main

import (
    "fmt"
)

type Digit int
type Power2 int

const PI = 3.1415926

const (
    C1 = "C1C1C1"
    C2 = "C2C2C2"
    C3 = "C3C3C3"
)
```

В этой части мы объявляем два новых типа: `Digit` и `Power2`, которые будут использованы вскоре, и четыре новые константы: `PI`, `C1`, `C2` и `C3`.



В Go `type` — это способ определения нового именованного типа, который использует тот же базовый тип, что и некий существующий. Данный способ служит в основном для различия разных типов, которые могут использовать один и тот же тип данных. С помощью ключевого слова `type` можно определять структуры и интерфейсы.

```
func main() {
    const s1 = 123
    var v1 float32 = s1 * 12
    fmt.Println(v1)
    fmt.Println(PI)

    const (
        Zero Digit = iota
        One
        Two
        Three
        Four
    )
}
```

Здесь мы задаем константу `s1`. И в этом же коде вы найдете определение *генератора констант iota* на основе `Digit`, что эквивалентно следующему объявлению четырех констант:

```
const (
    Zero = 0
    One = 1
    Two = 2
    Three = 3
    Four = 4
)
```



Хотя мы определяем константы внутри `main()`, их обычно можно найти вне `main()` или любой другой функции или метода.

Последняя часть `constants.go` содержит следующий код:

```
fmt.Println(One)
fmt.Println(Two)

const (
    p2_0 Power2 = 1 << iota
    -
    p2_2
    -
    p2_4
    -
    p2_6
)

fmt.Println("2^0:", p2_0)
fmt.Println("2^2:", p2_2)
fmt.Println("2^4:", p2_4)
fmt.Println("2^6:", p2_6)
}
```

Здесь присутствует еще один генератор констант `iota`, который слегка отличается от приведенного выше. Во-первых, вы могли заметить использование символа подчеркивания в блоке `const` с генератором констант `iota`, что позволяет пропускать нежелательные значения. Во-вторых, значение `iota` всегда увеличивается и может быть использовано в выражениях, что и делается в данном случае.

Теперь посмотрим, что на самом деле происходит внутри блока `const`. Для `p2_0` `iota` имеет значение `0`, а `p2_0` определяется как `1`. Для `p2_2` `iota` имеет значение `2`, а `p2_2` определяется как результат выражения `1 << 2`, что равно `00000100` в двоичном представлении. Десятичное значение `00000100` равно `4`, что является результатом и значением `p2_2`. Аналогично, значение `p2_4` равно `16`, а значение `p2_6` равно `64`.

При выполнении `constants.go` мы получаем такой вывод:

```
$ go run constants.go
1476
3.1415926
1
2
2^0: 1
2^2: 4
2^4: 16
2^6: 64
```

Наличие данных — это хорошо, но что происходит, когда имеется много похожих данных? Нужно ли иметь множество переменных для их хранения или есть способ лучше? Go отвечает на эти вопросы поддержкой массивов и срезов.

Группировка схожих данных

Бывают случаи, когда вы хотите сохранить несколько значений одного и того же типа данных в одной переменной и получать к ним доступ, используя индекс. Самый простой способ сделать это в Go — использовать массивы или срезы.

Массивы являются наиболее широко используемыми структурами данных и поддерживаются практически во всех языках программирования благодаря своей простоте и скорости доступа. Go предоставляет альтернативу массивам, которая называется срезом. Материал следующих подразделов поможет вам понять разницу между массивами и срезами, чтобы вы знали, когда и какую структуру данных использовать.



Быстрый ответ состоит в том, что срезы вместо массивов можно использовать практически в любом месте Go. Тем не менее мы продемонстрируем и массивы, поскольку они все еще могут быть полезны и в добавок срезы в Go реализованы с помощью массивов!

Массивы

Массивы в Go имеют следующие характеристики и ограничения.

- При определении переменной массива вы должны задать ее размер. В противном случае необходимо поместить в объявление массива [...] и позволить компилятору Go определить длину для вас. Таким образом, вы можете создать массив с четырьмя элементами `string` либо как `[4]string{"Zero", "One", "Two", "Three"}`, либо как `[...]string{"Zero", "One", "Two", "Three"}`. Если ничего не заключить в квадратные скобки, то вместо массива будет создан

срез. (Допустимыми) индексами для этого конкретного массива являются `0, 1, 2` и `3`.

- Вы не можете изменить размер массива после того, как он уже создан.
- Когда вы передаете массив функции, Go создает его копию и передает ее в функцию — поэтому любые изменения, которые вы вносите в массив внутри функции, теряются при возврате.

В результате массивы в Go получились не очень эффективными. Это основная причина того, что в языке появилась дополнительная структура данных, называемая *срезом*, которая похожа на массив, но динамична по своей природе. О ней мы расскажем в следующем подразделе. Тем не менее доступ к данным как в массивах, так и в срезах осуществляется одинаково.

Срезы

Срезы в Go эффективнее массивов главным образом потому, что динамичны, а это значит, что при необходимости они могут увеличиваться или уменьшаться после создания. Кроме того, любые изменения, которые вы вносите в срез внутри функции, влияют и на исходный срез. Но как это происходит? Строго говоря, *все параметры в Go передаются по значению* — другого способа передать параметры в этом языке нет.

В действительности значение среза — это *заголовок*, содержащий *указатель на базовый массив*, в котором фактически хранятся элементы, длину массива и его емкость (о емкости среза поговорим в следующем подразделе). Обратите внимание, что значение среза не включает его элементы, а лишь указывает на базовый массив. Следовательно, когда вы передаете срез в Go-функцию, язык создает копию этого заголовка и передает его в функцию. Эта копия заголовка среза включает указатель на базовый массив. Данный заголовок среза определен в пакете `reflect` (<https://golang.org/pkg/reflect/#SliceHeader>) следующим образом:

```
type SliceHeader struct {
    Data uintptr
    Len  int
    Cap  int
}
```

Побочным эффектом передачи заголовка среза является то, что срез передается быстрее, поскольку Go нужно создавать не копию среза и его элементов, а лишь его заголовок.

Вы можете создать срез с помощью `make()` или как массив, не указывая его размера или используя `[...]`. Если инициализировать срез не требуется, то лучше и быстрее использовать `make()`. Однако если вы хотите инициализировать его во время создания, то `make()` уже не подойдет. Как итог, вы можете создать срез

с тремя элементами `float64` с помощью `aSlice := []float64{1.2, 3.2, -4.5}`. Создать срез, вмещающий три элемента `float64`, с помощью `make()` можно так: `make([]float64, 3)`. Каждый элемент этого среза имеет значение 0, что является нулевым значением типа данных `float64`.

Как срезы, так и массивы могут иметь много измерений — создать срез с двумя измерениями с помощью `make()` так же просто: `make([][]int, 2)`. Оператор возвращает срез двумя измерениями, где первое равно 2 (строки), а второе (столбцы) не определено и должно быть явно указано при добавлении в него данных.

Если вы хотите определить и инициализировать срез с двумя измерениями одновременно, то должны выполнить что-то похожее на `twoD := [][]int{{1, 2, 3}, {4, 5, 6}}`.

Длину массива или среза можно получить с помощью `len()`. Как вы узнаете далее, срезы обладают дополнительным свойством, названным *емкостью* (capacity). Вы можете добавлять новые элементы к полному срезу с помощью функции `append()`. Она автоматически выделяет необходимое пространство памяти.

Приведенный ниже пример многое проясняет касательно срезов — можете с ним поэкспериментировать. Введите следующий код и сохраните его как `goSlices.go`:

```
package main

import "fmt"

func main() {
    // создать пустой срез
    aSlice := []float64{}
    // и длина, и емкость равны 0, поскольку aSlice пуст
    fmt.Println(aSlice, len(aSlice), cap(aSlice))

    // добавить элементы в срез
    aSlice = append(aSlice, 1234.56)
    aSlice = append(aSlice, -34.0)
    fmt.Println(aSlice, "with length", len(aSlice))
}
```

Команды `append()` добавляют два новых элемента в `aSlice`. Вы должны сохранить возвращаемое значение `append()` в существующую переменную или в новую.

```
// срез длиной 4
t := make([]int, 4)
t[0] = -1
t[1] = -2
t[2] = -3
t[3] = -4
// теперь нужно использовать append
t = append(t, -5)
fmt.Println(t)
```

Как только в срезе не останется места для дополнительных элементов, вы должны добавлять в него новые элементы с помощью `append()`.

```
// двумерный срез
// у вас может быть столько измерений, сколько необходимо
twoD := [][]int{{1, 2, 3}, {4, 5, 6}}
// просмотр всех элементов 2D-среза
// с помощью двойного цикла for
for _, i := range twoD {
    for _, k := range i {
        fmt.Println(k, " ")
    }
    fmt.Println()
}
```

Здесь показано, как создать переменную 2D-среза `twoD` и одновременно ее инициализировать.

```
make2D := make([][]int, 2)
fmt.Println(make2D)
make2D[0] = []int{1, 2, 3, 4}
make2D[1] = []int{-1, -2, -3, -4}
fmt.Println(make2D)
}
```

В этой части показано, как создать 2D-срез с помощью `make()`. Что делает `make2D` 2D-срезом, так это использование `[][]int` в `make()`.

При выполнении `goSlices.go` мы получаем такой вывод:

```
$ go run goSlices.go
[] 0 0
[1234.56 -34] with length 2
[-1 -2 -3 -4 -5]
1 2 3
4 5 6
[][] []
[[1 2 3 4] [-1 -2 -3 -4]]
```

О длине и емкости среза

Как массивы, так и срезы поддерживают функцию `len()`.

Однако у срезов также есть и дополнительное свойство — *емкость* (*capacity*), которое можно получить с помощью функции `cap()`.



Емкость среза действительно важна, когда необходимо выделить его часть или требуется ссылаться на массив с помощью среза. Обе эти темы мы обсудим далее.

Емкость показывает, насколько можно расширить срез, не выделяя большего объема памяти и не изменяя базового массива. Хотя после создания среза его емкость обрабатывается Go, разработчику позволяет задать емкость среза во время создания с помощью функции `make()` — после этого емкость среза удваивается каждый раз, когда длина среза становится больше его текущей емкости. Первый аргумент `make()` — это тип среза и его размеры, второй — его начальная длина, а третий (необязательный) — емкость среза. Тип данных среза уже не может изменяться после создания, в отличие от двух других свойств.



Запись наподобие `make([]int, 3, 2)` генерирует сообщение об ошибке, поскольку в любой момент времени емкость среза (2) не может быть меньше его длины (3).

Но что происходит, когда вы хотите добавить срез или массив к существующему срезу? Нужно ли это делать поэлементно? Go поддерживает операцию `...`, которая используется для разбиения среза или массива на несколько аргументов перед добавлением его к существующему срезу.

На рис. 2.1 наглядно показано, как длина и емкость работают в случае срезов.

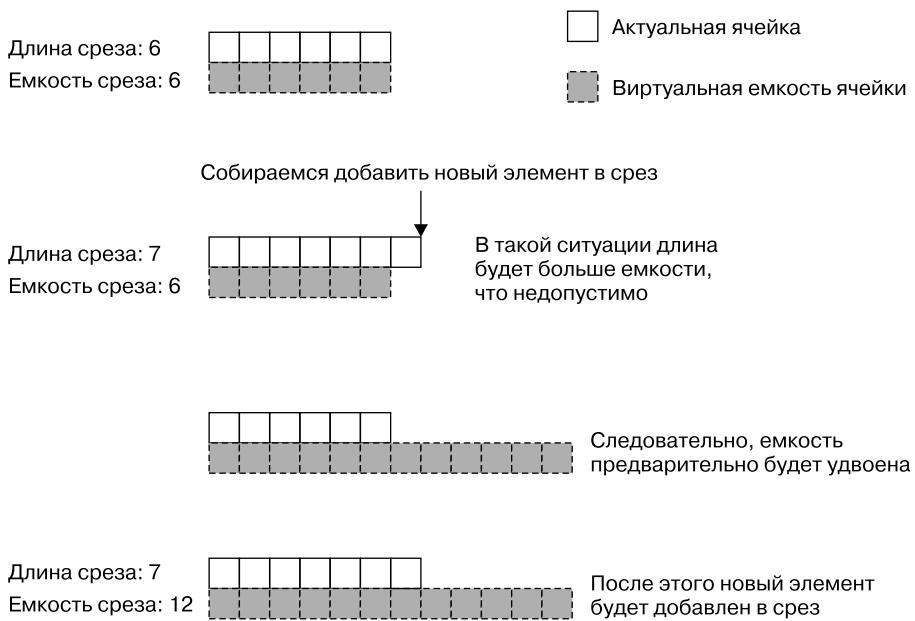


Рис. 2.1. Как связаны длина среза и емкость

Для тех из вас, кто предпочитает код, вот небольшая Go-программа, в которой показаны свойства длины и емкости срезов. Введите ее и сохраните как `capLen.go`:

```
package main

import "fmt"

func main() {
    // Определена только длина. Capacity = length
    a := make([]int, 4)
```

В этом случае емкость `a` совпадает с длиной, которая равна 4.

```
fmt.Println("L:", len(a), "C:", cap(a))
// Инициализировать срез. Capacity = length
b := []int{0, 1, 2, 3, 4}
fmt.Println("L:", len(b), "C:", cap(b))
```

И снова емкость среза `b` совпадает с его длиной, которая равна 5.

```
// одинаковая длина и емкость
aSlice := make([]int, 4, 4)
fmt.Println(aSlice)
```

На этот раз емкость среза `aSlice` совпадает с его длиной не потому, что Go решил так сделать, а потому, что мы это указали.

```
// добавить элемент
aSlice = append(aSlice, 5)
```

Когда вы добавляете новый элемент в срез `aSlice`, его емкость удваивается и становится 8.

```
fmt.Println(aSlice)
// емкость увеличивается вдвое
fmt.Println("L:", len(aSlice), "C:", cap(aSlice))
// теперь добавим четыре элемента
aSlice = append(aSlice, []int{-1, -2, -3, -4}...)
```

Операция `...` разворачивает `[]int{-1, -2, -3, -4}` в несколько аргументов, и `append()` по отдельности добавляет каждый аргумент в `aSlice`.

```
fmt.Println(aSlice)
// емкость увеличивается вдвое
fmt.Println("L:", len(aSlice), "C:", cap(aSlice))
}
```

При выполнении `capLen.go` мы получаем такой вывод:

```
$ go run capLen.go
L: 4 C: 4
```

```
L: 5 C: 5
[0 0 0 0]
[0 0 0 0 5]
L: 5 C: 8
[0 0 0 5 -1 -2 -3 -4]
L: 9 C: 16
```



Установка правильной емкости среза, если она известна заранее, ускорит ваши программы, так как Go не придется выделять новый базовый массив и копировать все данные.

Работать со срезами удобно, но что происходит, когда нужно использовать непрерывную часть существующего среза? Есть ли практический способ выбрать часть среза? К счастью, ответ положительный, и далее мы немного поговорим о том, как выбрать *непрерывную часть* среза.

Выбор части среза

Go позволяет выбирать части среза при условии, что все нужные элементы расположены рядом друг с другом. Это довольно удобно, когда вы выбираете диапазон элементов и не хотите указывать их индексы один за другим. В Go вы выбираете часть среза, определяя два индекса, первый из которых является началом выделения, а второй — концом (без включения элемента с этим индексом), разделенные символом `:`.



Если вы хотите обработать все аргументы командной строки, кроме первого (который является ее именем утилиты), то можете назначить их новой переменной (`arguments := os.Args`) для простоты использования и с помощью нотации `arguments[1:]` пропустить первый аргумент командной строки.

Однако существует вариация, в которой вы можете добавить третий параметр, управляющий емкостью результирующего среза. Так, с помощью `aSlice[0:2:4]` выбираются первые два элемента среза (с индексами `0` и `1`) и создается новый срез с максимальной емкостью `4`. Итоговая емкость определяется как результат вычитания `4-0`, где `4` — максимальная емкость, а `0` — первый индекс. Если он опущен, то автоматически устанавливается равным `0`. В данном случае емкость итогового среза будет равна `4`, поскольку `4-0` равно `4`.

Если бы мы использовали `aSlice[2:4:4]`, то создали бы новый срез с элементами `aSlice[2]` и `aSlice[3]` и емкостью `4-2`. Наконец, *итоговая емкость не может быть больше емкости исходного среза*, поскольку в этом случае вам понадобится другой базовый массив.

Ведите следующий код в редакторе и сохраните его как `partSlice.go`:

```
package main
import "fmt"

func main() {
    aSlice := []int{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
    fmt.Println(aSlice)
    l := len(aSlice)

    // первые пять элементов
    fmt.Println(aSlice[0:5])
    // первые пять элементов
    fmt.Println(aSlice[:5])
```

В этой первой части мы определяем новый срез `aSlice`, содержащий десять элементов. Его емкость такая же, как и длина. Нотации `0:5` и `:5` выбирают первые пять элементов среза, которые являются элементами с индексами `0, 1, 2, 3` и `4`.

```
// последние два элемента
fmt.Println(aSlice[1-2 : 1])
// Последние два элемента
fmt.Println(aSlice[1-2:])
```

Учитывая длину среза (`1`), мы можем выбрать последние два элемента среза либо как `1-2 : 1`, либо как `1-2::`.

```
// первые пять элементов
t := aSlice[0:5:10]
fmt.Println(len(t), cap(t))
// элементы с индексами 2,3,4
// емкость составит 10-2
t = aSlice[2:5:10]
fmt.Println(len(t), cap(t))
```

Первоначально емкость `t` будет равна `10-0`, что дает `10`. Во втором случае емкость `t` будет равна `10-2`.

```
// элементы с индексами 0,1,2,3,4
// новая емкость составит 6-0
t = aSlice[:5:6]
fmt.Println(len(t), cap(t))
}
```

Емкость `t` теперь равна `6-0`, а его длина будет равна `5`, поскольку мы выбрали первые пять элементов среза `aSlice`.

При выполнении `partSlice.go` мы получаем такой вывод:

```
$ go run partSlice.go
[0 1 2 3 4 5 6 7 8 9]
```

Предыдущая строка — это выходные данные `fmt.Println(aSlice)`.

```
[0 1 2 3 4]  
[0 1 2 3 4]
```

Предыдущие две строки генерируются из `fmt.Println(aSlice[0:5])` и `fmt.Println(aSlice[:5])`.

```
[8 9]  
[8 9]
```

Аналогично предыдущие две строки генерируются из `fmt.Println(aSlice[1-2 : 1])` и `fmt.Println(aSlice[1-2:])`.

```
5 10  
3 8  
5 6
```

В последних трех строках выводятся длина и емкость `aSlice[0:5:10]`, `aSlice[2:5:10]` и `aSlice[:5:6]`.

Байтовые срезы

Байтовый срез — это срез с типом данных `byte([]byte)`. Go знает, что большинство срезов типа `byte` используются для хранения строк, и поэтому позволяет легко переключаться между этим типом и типом `string`. Получение доступа к байтовому срезу ничем не отличается от работы с другими типами срезов. Особенность заключается в том, что Go использует *байтовые срезы* для выполнения операций ввода-вывода в файлы, поскольку они позволяют с точностью определить объем данных, которые необходимо прочитать или записать в файл. Это происходит потому, что байты — универсальная единица измерения в компьютерном мире.



Go не имеет типа данных `char`, поэтому использует `byte` и `rune` для хранения символьных значений. Один байт может хранить только один символ ASCII, тогда как руна — символы Unicode. Однако руна может занимать несколько байтов.

В приведенной ниже небольшой программе показано, как можно преобразовать срез типа `byte` в `string` и, наоборот, что необходимо для большинства операций ввода/вывода в файлы. Введите ее и сохраните как `byteSlices.go`:

```
package main  
  
import "fmt"
```

```
func main() {
    // байтовый срез
    b := make([]byte, 12)
    fmt.Println("Byte slice:", b)
```

Пустой байтовый срез содержит нули — в данном случае 12 нулей.

```
b = []byte("Byte slice €")
fmt.Println("Byte slice:", b)
```

Здесь размер `b` равен размеру строки "Byte slice €", без двойных кавычек — `b` теперь указывает на другую ячейку памяти, нежели ту, где хранится "Byte slice €". Именно так `string` конвертируется в срез `byte`.

Для представления символов Unicode, таких как €, требуется несколько байтов, поэтому длина среза `byte` может отличаться от длины строки, которую он хранит.

```
// вывести содержимое байтового среза в виде текста
fmt.Printf("Byte slice as text: %s\n", b)
fmt.Println("Byte slice as text:", string(b))
```

Здесь показано, как вывести содержимое байтового среза в виде текста, используя два подхода. В первом это делается с помощью управляющей строки `%s`, а во втором — с помощью `string()`.

```
// длина b
fmt.Println("Length of b:", len(b))
}
```

Здесь мы выводим реальную длину байтового среза.

При выполнении `byteSlices.go` мы получаем такой вывод:

```
$ go run byteSlices.go
Byte slice: [0 0 0 0 0 0 0 0 0 0 0 0]
Byte slice: [66 121 116 101 32 115 108 105 99 101 32 226 130 172]
Byte slice as text: Byte slice €
Byte slice as text: Byte slice €
Length of b: 14
```

Последняя строка вывода доказывает, что, хотя байтовый срез `b` содержит 12 символов, его размер равен 14.

Удаление элемента из среза

Функции удаления элемента из среза по умолчанию не существует; это значит, если вам нужно удалить элемент из среза, то придется написать для этого собственный код. Удаление элемента из среза может оказаться непростой задачей, поэтому здесь мы сделаем это с помощью двух методов. Первый фактически

делит исходный срез на два, разделенных по индексу элемента, который необходимо удалить. Ни один из двух срезов не включает элемент, который нужно удалить. После этого мы объединяем эти два среза и создаем новый. Второй метод копирует последний элемент на место элемента, который будет удален, и создает новый срез, исключая последний элемент исходного.

На рис. 2.2 наглядно показаны оба метода удаления элемента из среза.

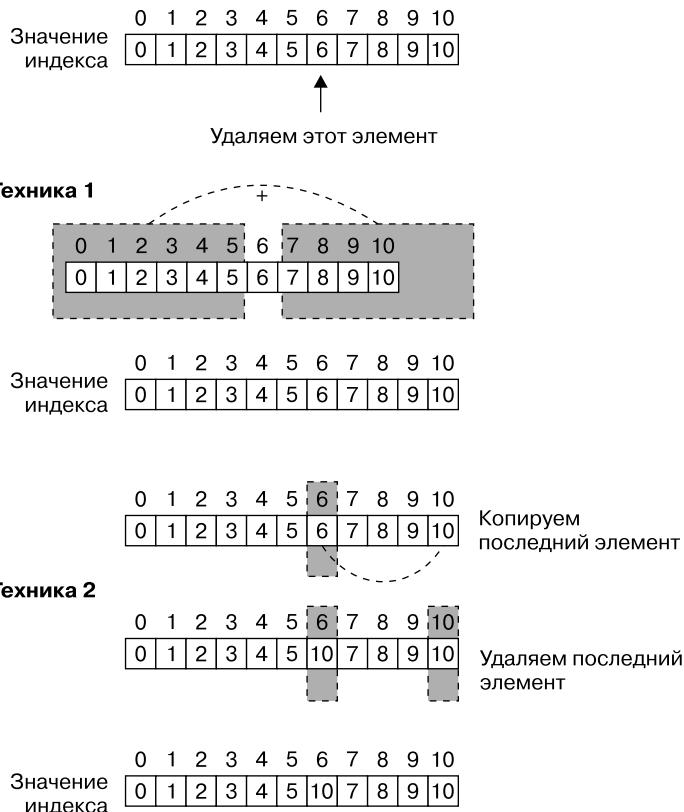


Рис. 2.2. Удаление элемента из среза

В следующей программе показаны два метода, которые можно использовать при удалении элемента из среза. Создайте текстовый файл, введя следующий код, и сохраните его как `deleteSlice.go`:

```
package main

import (
    "fmt"
)
```

```

    "os"
    "strconv"
)

func main() {
    arguments := os.Args
    if len(arguments) == 1 {
        fmt.Println("Need an integer value.")
        return
    }

    index := arguments[1]
    i, err := strconv.Atoi(index)
    if err != nil {
        fmt.Println(err)
        return
    }
    fmt.Println("Using index", i)

    aSlice := []int{0, 1, 2, 3, 4, 5, 6, 7, 8}
    fmt.Println("Original slice:", aSlice)

    // удалить элемент с индексом i
    if i > len(aSlice)-1 {
        fmt.Println("Cannot delete element", i)
        return
    }

    // Операция ... автоматически развертывает aSlice[i+1:] так,
    // что его элементы можно добавить к aSlice[:i] последовательно
    aSlice = append(aSlice[:i], aSlice[i+1:]...)
    fmt.Println("After 1st deletion:", aSlice)
}

```

Здесь мы логически разделяем исходный срез на два. Два среза разделяются по индексу элемента, который необходимо удалить. После этого мы объединяем эти два среза с помощью операции Далее мы увидим второй метод в действии.

```

// удалить элемент с индексом i
if i > len(aSlice)-1 {
    fmt.Println("Cannot delete element", i)
    return
}

// заменить элемент с индексом i на последний
aSlice[i] = aSlice[len(aSlice)-1]
// удалить последний элемент
aSlice = aSlice[:len(aSlice)-1]
fmt.Println("After 2nd deletion:", aSlice)
}

```

Мы заменяем элемент, который хотим удалить, на последний, используя оператор `aSlice[i] = aSlice[len(aSlice)-1]`, а затем удаляем последний элемент с помощью оператора `aSlice = aSlice[:len(aSlice)-1]`.

При выполнении `deleteSlice.go` мы получаем такой вывод в зависимости от входных данных:

```
$ go run deleteSlice.go 1
Using index 1
Original slice: [0 1 2 3 4 5 6 7 8]
After 1st deletion: [0 2 3 4 5 6 7 8]
After 2nd deletion: [0 8 3 4 5 6 7]
```

Поскольку срез состоит из девяти элементов, вы можете удалить элемент со значением индекса 1.

```
$ go run deleteSlice.go 10
Using index 10
Original slice: [0 1 2 3 4 5 6 7 8]
Cannot delete element 10
```

Поскольку срез содержит только девять элементов, вы не сможете удалить элемент со значением индекса 10.

Как срезы связаны с массивами

Как упоминалось ранее, «под капотом» каждый срез реализуется с помощью *базового массива*. Длина массива совпадает с емкостью среза, и существуют указатели, которые соединяют элементы среза с соответствующими элементами массива.

Из этого понятно, что, связывая существующий массив со срезом, Go с помощью среза позволяет вам ссылаться на массив или его часть. Это дает довольно странные возможности, включая тот факт, что изменения в срезе влияют на соответствующий массив. Однако, когда емкость среза изменяется, связь с массивом перестает существовать. Это происходит потому, что при изменении емкости среза изменяется и базовый массив и связь между срезом и исходным массивом теряет смысл.

Ведите следующий код и сохраните его как `sliceArrays.go`:

```
package main

import (
    "fmt"
)
```

```
func change(s []string) {
    s[0] = "Change_function"
}
```

Эта функция изменяет первый элемент среза.

```
func main() {
    a := [4]string{"Zero", "One", "Two", "Three"}
    fmt.Println("a:", a)
```

Здесь мы определяем массив `a`, состоящий из четырех элементов.

```
var S0 = a[0:1]
fmt.Println(S0)
S0[0] = "S0"
```

Здесь мы связываем `S0` с первым элементом массива `a` и выводим его. Затем меняем значение `S0[0]`.

```
var S12 = a[1:3]
fmt.Println(S12)
S12[0] = "S12_0"
S12[1] = "S12_1"
```

В этой части мы связываем `S12` с `a[1]` и `a[2]`. Следовательно, `S12[0] = a[1]` и `S12[1] = a[2]`. Затем мы меняем значения как `S12[0]`, так и `S12[1]`. Эти два изменения также изменят содержимое `a`. Проще говоря, `a[1]` принимает новое значение `S12[0]`, а `a[2]` принимает новое значение `S12[1]`.

```
fmt.Println("a:", a)
```

И выводим переменную `a`, которая напрямую не изменялась. Однако из-за связываний `a` с `S0` и `S12` содержимое `a` изменилось!

```
// изменения в срезе -> изменения в массиве
change(S12)
fmt.Println("a:", a)
```

Поскольку срез и массив связаны, любые изменения, которые вы вносите в срез, также повлияют на массив, даже если изменения происходят внутри функции.

```
// емкость S0
fmt.Println("Capacity of S0:", cap(S0), "Length of S0:", len(S0))

// добавление четырех элементов в S0
S0 = append(S0, "N1")
S0 = append(S0, "N2")
S0 = append(S0, "N3")
a[0] = "-N1"
```

По мере изменения емкости `S0` он больше не связан с тем же базовым массивом (`a`).

```
// изменение емкости S0
// Уже другой базовый массив!
S0 = append(S0, "N4")

fmt.Println("Capacity of S0:", cap(S0), "Length of S0:", len(S0))
// это изменение не относится к S0
a[0] = "-N1-"

// это изменение не относится к S12
a[1] = "-N2-"
```

Однако массив `a` и срез `S12` по-прежнему связаны, поскольку емкость `S12` не изменилась.

```
fmt.Println("S0:", S0)
fmt.Println("a: ", a)
fmt.Println("S12:", S12)
}
```

Наконец, мы выводим окончательные версии `a`, `S0` и `S12`.

При выполнении `sliceArrays.go` мы получаем такой вывод:

```
$ go run sliceArrays.go
a: [Zero One Two Three]
[Zero]
[One Two]
a: [S0 S12_0 S12_1 Three]
a: [S0 Change_function S12_1 Three]
Capacity of S0: 4 Length of S0: 1
Capacity of S0: 8 Length of S0: 5
S0: [-N1 N1 N2 N3 N4]
a: [-N1- -N2- N2 N3]
S12: [-N2- N2]
```

Далее мы обсудим использование функции `copy()`.

Функция `copy()`

Go содержит функцию `copy()`, предназначенную для копирования существующего массива в срез или существующего среза в другой срез. Однако при использовании `copy()` возможны нюансы, поскольку целевой срез не расширяется автоматически, если меньше исходного. Кроме того, если целевой срез больше исходного, то функция `copy()` не очищает элементы целевого, которые не были скопированы. Это наглядно показано на рис. 2.3.

**Рис. 2.3.** Использование функции copy()

В следующей программе показано использование функции `copy()`. Введите ее в текстовом редакторе и сохраните как `copySlice.go`:

```
package main

import "fmt"

func main() {
    a1 := []int{1}
    a2 := []int{-1, -2}
    a5 := []int{10, 11, 12, 13, 14}
    fmt.Println("a1", a1)
    fmt.Println("a2", a2)
    fmt.Println("a5", a5)
    // copy(destination, input)
    // len(a2) > len(a1)
    copy(a1, a2)
    fmt.Println("a1", a1)
    fmt.Println("a2", a2)
}
```

Здесь мы выполняем команду `copy(a1, a2)`. В этом случае срез `a2` больше, чем `a1`. После `copy(a1, a2)` `a2` остается неизменным, что понятно, ведь `a2` является входным срезом, тогда как первый элемент `a2` копируется в первый элемент `a1`, поскольку в `a1` есть место лишь для одного элемента.

```
// len(a2) > len(a1)
copy(a1, a5)
fmt.Println("a1", a1)
fmt.Println("a5", a5)
```

В этом случае `a5` больше, чем `a1`. Опять же, после `copy(a1, a5)` `a5` остается прежним, тогда как `a5[0]` копируется в `a1[0]`.

```
// len(a2) < len(a5) -> OK
copy(a5, a2)
fmt.Println("a2", a2)
fmt.Println("a5", a5)
}
```

В последнем случае `a2` короче, чем `a5`. Это означает, что `a2` целиком копируется в `a5`. Поскольку длина `a2` равна 2, изменяются только первые два элемента `a5`.

При выполнении `copySlice.go` мы получаем такой вывод:

```
$ go run copySlice.go
a1 [1]
a2 [-1 -2]
a5 [10 11 12 13 14]
a1 [-1]
a2 [-1 -2]
```

Оператор `copy(a1, a2)` не изменяет срез `a2`, а только `a1`. Размер `a1` равен 1, поэтому копируется только первый элемент из `a2`.

```
a1 [10]
a5 [10 11 12 13 14]
```

Аналогично `copy(a1, a5)` изменяет только `a1`. Размер `a1` равен 1, поэтому в `a1` копируется лишь первый элемент из `a5`.

```
a2 [-1 -2]
a5 [-1 -2 12 13 14]
```

И наконец, `copy(a5, a2)` изменяет только `a5`. Размер `a5` равен 5, поэтому только первые два элемента из `a5` изменяются и становятся такими же, как первые два элемента `a2`, размер которого равен 2.

Сортировка срезов

Бывают ситуации, когда информацию необходимо отсортировать, и желательно, чтобы Go проделал эту работу за вас. Далее мы увидим, как сортируются срезы различных стандартных типов данных, используя функциональность, предлагаемую пакетом `sort`.

Пакет `sort` может сортировать срезы встроенных типов данных, при этом вам не придется писать дополнительный код. Кроме того, Go предоставляет функцию `sort.Reverse()` для сортировки в порядке, обратном порядку по умолчанию. Однако интересен тот факт, что `sort` позволяет писать собственные функции сортировки для пользовательских типов данных, реализуя интерфейс `sort.Interface`. Больше информации о `sort.Interface` и интерфейсах в целом вы узнаете в главе 4.

Итак, вы можете отсортировать срез целых чисел, сохраненных как `sInts`, набрав команду `sort.Ints(sInts)`. При сортировке среза целых чисел в обратном порядке с помощью `sort.Reverse()` вам необходимо передать нужный срез в `sort.Reverse()`, используя `sort.IntSlice(sInts)`, поскольку тип `IntSlice`

внутренне реализует `sort.Interface`, что позволяет сортировать способом, отличным от обычного. То же самое относится и к другим стандартным типам данных Go.

Создайте текстовый файл с кодом, показывающим использование `sort`, и назовите его `sortSlice.go`:

```
package main

import (
    "fmt"
    "sort"
)

func main() {
    sInts := []int{1, 0, 2, -3, 4, -20}
    sFloats := []float64{1.0, 0.2, 0.22, -3, 4.1, -0.1}
    sStrings := []string{"aa", "a", "A", "Aa", "aab", "AAa"}

    fmt.Println("sInts original:", sInts)
    sort.Ints(sInts)
    fmt.Println("sInts:", sInts)
    sort.Sort(sort.Reverse(sort.IntSlice(sInts)))
    fmt.Println("Reverse:", sInts)
}
```

Поскольку `sort.Interface` знает, как сортировать целые числа, отсортировать их в обратном порядке — тривиальная задача. Такая сортировка осуществляется путем простого вызова функции `sort.Reverse()`.

```
fmt.Println("sFloats original:", sFloats)
sort.Float64s(sFloats)
fmt.Println("sFloats:", sFloats)
sort.Sort(sort.Reverse(sort.Float64Slice(sFloats)))
fmt.Println("Reverse:", sFloats)

fmt.Println("sStrings original:", sStrings)
sort.Strings(sStrings)
fmt.Println("sStrings:", sStrings)
sort.Sort(sort.Reverse(sort.StringSlice(sStrings)))
fmt.Println("Reverse:", sStrings)
}
```

Те же правила применяются при сортировке чисел и строк с плавающей запятой.

При выполнении `sortSlice.go` мы получаем такой вывод:

```
$ go run sortSlice.go
sInts original: [1 0 2 -3 4 -20]
sInts: [-20 -3 0 1 2 4]
Reverse: [4 2 1 0 -3 -20]
sFloats original: [1 0.2 0.22 -3 4.1 -0.1]
```

```
sFloats: [-3 -0.1 0.2 0.22 1 4.1]
Reverse: [4.1 1 0.22 0.2 -0.1 -3]
sStrings original: [aa a A Aa aab AAa]
sStrings: [A AAa Aa a aa aab]
Reverse: [aab aa a Aa AAa A]
```

Выходные данные показывают, что исходные срезы были отсортированы как в обычном, так и в обратном порядке.

Указатели

Go поддерживает указатели, но не их арифметику, что является причиной многих ошибок в языках программирования, таких как С. Указатель — это адрес переменной в памяти. Вам необходимо *разыменовать* указатель, чтобы получить его значение. Разыменование выполняется с помощью символа * перед переменной указателя. Кроме того, вы можете получить адрес обычной переменной, использовав перед ней &.

На рис. 2.4 показана разница между указателем на `int` и переменная `int`.

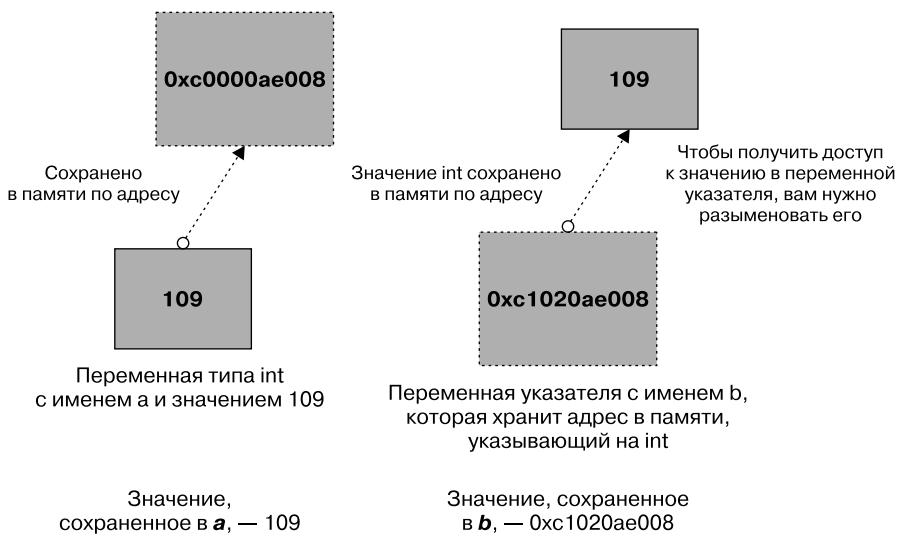


Рис. 2.4. Переменная `int` и указатель на переменную `int`

Если переменная-указатель указывает на существующую обычную переменную, то любые изменения, которые вы вносите в сохраненное значение с помощью переменной-указателя, изменят обычную переменную.



Формат и значения адресов памяти могут различаться на разных компьютерах, в разных операционных системах и архитектурах.

Вы можете спросить, а зачем вообще использовать указатели, если нет поддержки их арифметики. Основное преимущество, которое вы получаете благодаря указателям, заключается в том, что передача переменной в функцию в виде указателя (это называется *по ссылке*) не отменяет никаких изменений, которые вы вносите в значение этой переменной внутри данной функции, когда последняя возвращается. Бывают случаи, когда подобная функциональность может пригодиться. Она упрощает код, но цена, которую вы платите за эту простоту, заключается в том, что приходится проявлять особую осторожность при работе с переменной-указателем. Помните, что срезы можно передавать функциям, не используя указатель, — Go самостоятельно передает указатель на базовый массив среза, и изменить это поведение невозможно.

Помимо соображений простоты, существуют еще три причины для использования указателей.

- Указатели позволяют функциям обмениваться данными. Однако при обмене данными между функциями и горутинами следует быть особенно осторожным, чтобы не вызывать состояние гонки.
- Указатели также очень удобны, когда требуется определить разницу между нулевым значением переменной и значением, которое не задано (`nil`). Это особенно полезно в случае структур, поскольку указатели (и, следовательно, *указатели на структуры*, которые в полном объеме рассматриваются в следующей главе) могут иметь значение `nil`. Это означает, что вы можете сравнить указатель на структуру со значением `nil`, что недопустимо для обычных переменных структур.
- Наличие поддержки указателей и, как следствие, указателей на структуры позволяет Go поддерживать структуры данных, такие как связанные списки и двоичные деревья, которые широко используются в информатике. Следовательно, вам разрешено определять структурное поле структуры `Node` как `Next *Node`, что является указателем на другую структуру `Node`. Без указателей это было бы трудно реализовать и работала бы такая реализация слишком медленно.

В следующем коде показано использование указателей в Go. Создайте текстовый файл `pointers.go` и введите представленный код:

```
package main

import "fmt"
```

```
type aStructure struct {
    field1 complex128
    field2 int
}
```

Это структура с двумя полями: `field1` и `field2`.

```
func processPointer(x *float64) {
    *x = *x * *x
}
```

Это функция, которая получает указатель на переменную `float64` в качестве входных данных. Поскольку мы используем указатель, все изменения параметра функции внутри функции являются постоянными. Кроме того, нет необходимости что-то возвращать.

```
func returnPointer(x float64) *float64 {
    temp := 2 * x
    return &temp
}
```

Это функция, которая требует параметра `float64` в качестве входных данных и возвращает указатель на `float64`. Чтобы вернуть адрес памяти обычной переменной, вам нужно использовать `&` (`&temp`).

```
func bothPointers(x *float64) *float64 {
    temp := 2 * *x
    return &temp
}
```

Это функция, которая требует указатель на `float64` в качестве входных данных и возвращает указатель на `float64` в качестве выходных. Нотация `*x` используется для получения значения, хранящегося по адресу в памяти, который хранится в `x`.

```
func main() {
    var f float64 = 12.123
    fmt.Println("Memory address of f:", &f)
```

Чтобы получить адрес в памяти для обычной переменной `f`, следует использовать нотацию `&f`.

```
// указатель на f
fp := &f
fmt.Println("Memory address of f:", fp)
fmt.Println("Value of f:", *fp)
// значение f изменяется
processPointer(fp)
fmt.Printf("Value of f: %.2f\n", f)
```

`fP` теперь является указателем на адрес в памяти переменной `f`. Любые изменения значения, хранящегося в адресе памяти `fP`, также влияют на значение `f`.

Однако это верно только до тех пор, пока `fP` указывает на адрес в памяти переменной `f`.

```
// значение f не изменяется
x := returnPointer(f)
fmt.Printf("Value of x: %.2f\n", *x)
```

Значение `f` не изменяется, поскольку функция использует только ее значение.

```
// значение f не изменяется
xx := bothPointers(fP)
fmt.Printf("Value of xx: %.2f\n", *xx)
```

В этом случае значение `f`, а также значение, сохраненное в адресе памяти `fP`, не изменяется, поскольку функция `bothPointers()` не вносит никаких изменений в значение, сохраненное в адресе памяти `fP`.

```
// проверить наличие пустой структуры
var k *aStructure
```

Переменная `k` – это указатель на структуру `aStructure`. Поскольку `k` указывает в никуда, Go заставляет его указывать на `nil`, что является *нулевым значением для указателей*.

```
// равно nil, поскольку в настоящее время k указывает в никуда
fmt.Println(k)
// следовательно, вам разрешено сделать следующее:
if k == nil {
    k = new(aStructure)
}
```

Поскольку переменная `k` равна `nil`, нам разрешено присвоить ей значение `aStructure` с помощью `new(aStructure)` без потери каких-либо данных. Теперь `k` больше не равна `nil`, но оба поля `aStructure` содержат нулевые значения своих типов данных.

```
fmt.Printf("%+v\n", k)
if k != nil {
    fmt.Println("k is not nil!")
}
}
```

Просто убеждаемся, что `k` не равно `nil`. Вы можете счесть эту проверку избыточной, но перепроверить не помешает.

При выполнении `pointers.go` мы получаем такой вывод:

```
Memory address of f: 0xc000014090
Memory address of f: 0xc000014090
Value of f: 12.123
Value of f: 146.97
Value of x: 293.93
Value of xx: 293.93
<nil>
&{field1:(0+0i) field2:0}
k is not nil!
```

Мы вернемся к указателям в следующей главе, когда будем обсуждать структуры. Далее обсудим генерацию случайных чисел и случайных строк.

Генерация случайных чисел

Генерация случайных чисел — не только искусство, но и область исследований в сфере компьютерных наук. А все потому, что компьютеры — это чисто логические машины, и оказывается, что использовать их для генерации случайных чисел чрезвычайно сложно! Go может помочь вам в этом, предоставляя функциональность пакета `math/rand`. Каждому генератору случайных чисел требуется *начальное значение*, чтобы начать их генерировать. Начальное значение используется для инициализации всего процесса и является чрезвычайно важным. Если вы всегда начинаете с одного и того же начального значения, то всегда будете получать одну и ту же последовательность псевдослучайных чисел. Это означает, что любой может ее восстановить и эта конкретная последовательность в итоге не будет случайной. Однако данная функция действительно полезна для целей тестирования. В Go для инициализации генератора случайных чисел используется функция `rand.Seed()`.



Если вас по-настоящему интересует генерация случайных чисел, то вам следует начать с чтения второго тома книги Дональда Э. Кнута *The Art of Computer Programming* (Addison-Wesley Professional, 2011)¹.

Следующая функция, которая является частью `randomNumbers.go` из каталога `ch02` в репозитории GitHub книги, генерирует случайные числа в диапазоне `[min, max]`:

```
func random(min, max int) int {
    return rand.Intn(max-min) + min
}
```

Функция `random()` выполняет всю работу, которая заключается в генерации псевдослучайных чисел в заданном диапазоне от `min` до `max-1` путем вызова

¹ Кнут Д. Э. Искусство программирования.

`rand.Intn()`. Функция `rand.Intn()` генерирует неотрицательные случайные целые числа от 0 до значения его единственного параметра минус 1.

Утилита `randomNumbers.go` принимает четыре параметра командной строки, но может также работать с меньшим количеством параметров, используя значения по умолчанию. По умолчанию `randomNumbers.go` выдает 100 случайных целых чисел от 0 до 99 включительно.

```
$ go run randomNumbers.go
Using default values!
39 75 78 89 39 28 37 96 93 42 60 69 50 9 69 27 22 63 4 68 56 23 54 14
93 61 19 13 83 72 87 29 4 45 75 53 41 76 84 51 62 68 37 11 83 20 63 58
12 50 8 31 14 87 13 97 17 60 51 56 21 68 32 41 79 13 79 59 95 56 24 83
53 62 97 88 67 59 49 65 79 10 51 73 48 58 48 27 30 88 19 16 16 11 35 45
72 51 41 28
```

В следующем выводе мы определяем каждый из параметров вручную (последний параметр утилиты — это начальное значение):

```
$ go run randomNumbers.go 1 5 10 10
3 1 4 4 1 1 4 4 4 3
$ go run randomNumbers.go 1 5 10 10
3 1 4 4 1 1 4 4 4 3
$ go run randomNumbers.go 1 5 10 11
1 4 2 1 3 2 2 4 1 3
```

Первые два раза начальное значение было равно 10, поэтому мы получили одинаковый результат. В третий раз значение начального значения составило 11, что привело к другому результату.

Генерация случайных строк

Представьте, что вы хотите сгенерировать случайные строки, которые можно использовать в качестве трудно угадываемых паролей или в целях тестирования. Мы создадим утилиту, которая на основе генерации случайных чисел генерирует случайные строки. Утилита реализована как `genPass.go`, и ее можно найти в каталоге `ch02` репозитория книги на GitHub. Основная функциональность `genPass.go` содержится в следующей функции:

```
func getString(len int64) string {
    temp := ""
    startChar := "!"
    var i int64 = 1
    for {
        myRand := random(MIN, MAX)
        newChar := string(startChar[0] + byte(myRand))
        temp = temp + newChar
        if i == len {
            break
        }
    }
    return temp
}
```

```

    }
    i++
}
return temp
}

```

Мы хотим получить только печатные символы ASCII, поэтому ограничиваем диапазон генерируемых псевдослучайных чисел. Общее количество печатаемых символов в таблице ASCII равно 94. Это означает, что диапазон псевдослучайных чисел, которые может генерировать программа, должен составлять от 0 до 94, не включая 94. Следовательно, значения глобальных переменных MIN и MAX, которые здесь не показаны, равны 0 и 94 соответственно.

Переменная `startChar` содержит первый символ ASCII, который может быть сгенерирован утилитой, и им в данном случае является восклицательный знак, имеющий десятичное значение ASCII, равное 33. Учитывая, что программа может генерировать псевдослучайные числа до 94, максимальное значение ASCII, которое может быть сгенерировано, равно $93 + 33$, то есть 126, что является значением ASCII ~. Все сгенерированные символы сохраняются в переменной `temp`, которая возвращается после завершения цикла `for`. Оператор `string(startChar[0] + byte(myRand))` преобразует случайные целые числа в символы в желаемом диапазоне.

Утилита `genPass.go` принимает единственный параметр, который представляет собой длину сгенерированного пароля. Если параметр не задан, то `genPass.go` выдает пароль из восьми символов, который является значением переменной `LENGTH` по умолчанию.

При выполнении `genPass.go` мы получаем такой вывод:

```

$ go run genPass.go
Using default values...
!QrNq@;R
$ go run genPass.go 20
sZL>{F~"hQqY>r_>TX?0

```

Первый запуск программы использует значение по умолчанию для длины сгенерированной строки, тогда как второй создает случайную строку из 20 символов.

Генерация безопасных случайных чисел

Если вы намерены применять эти псевдослучайные числа для работы, связанной с безопасностью, то вам важно использовать пакет `crypto/rand`, который реализует криптографически защищенный генератор псевдослучайных чисел. При использовании данного пакета не нужно задавать начальное значение.

В следующей функции, которая является частью исходного кода `CryptoRand.go`, показано, как с помощью функциональности `crypto/rand` генерируются безопасные случайные числа:

```
func generateBytes(n int64) ([]byte, error) {
    b := make([]byte, n)
    _, err := rand.Read(b)
    if err != nil {
        return nil, err
    }
    return b, nil
}
```

Функция `rand.Read()` случайным образом генерирует числа, которые заполняют весь байтовый срез `b`. Вам придется декодировать этот байтовый срез, используя `base64.URLEncoding.EncodeToString(b)`, чтобы получить допустимую строку без каких-либо управляющих или непечатаемых символов. Преобразование выполняется в функции `generatePass()`, которая здесь не представлена.

При выполнении `CryptoRand.go` мы получаем такой вывод:

```
$ go run cryptoRand.go
Using default values!
Ce30g--D
$ go run cryptoRand.go 20
AEIePSYb13KwkDn05Xk_
```

Результат не отличается от того, который генерируется `genPass.go`, просто случайные числа генерируются более надежно, а это значит, что их можно использовать в приложениях, где важна безопасность.

Теперь, когда мы выяснили, как генерировать случайные числа и строки, вернемся к приложению телефонной книги и используем эти методы, чтобы заполнить ее случайными данными.

Обновление приложения телефонной книги

В этом последнем разделе мы напишем функцию, которая заполняет приложение телефонной книги, представленное в предыдущей главе, случайными данными. Это довольно удобно, когда нужно поместить в приложение большой объем данных для целей тестирования.



Я использовал этот весьма удобный способ ранее, чтобы поместить образцы данных в темы Kafka.

Ключевое изменение в этой версии приложения телефонной книги состоит в том, что поиск будет основан на телефонном номере, поскольку проще искать случайные числа вместо случайных строк. Но это лишь небольшое изменение кода в функции `search()`. На этот раз `search()` использует `v.Tel == key` вместо `v.Surname == key` для сопоставления поля `Tel`.

Функция `populate()` из `phoneBook.go` (можно найти в каталоге `ch02`) выполняет всю работу. Реализация `populate()` заключается в следующем:

```
func populate(n int, s []Entry) {
    for i := 0; i < n; i++ {
        name := getString(4)
        surname := getString(5)
        n := strconv.Itoa(random(100, 199))
        data = append(data, Entry{name, surname, n})
    }
}
```

Функция `getString()` генерирует буквы от A до Z и ничего более, что делает сгенерированные строки более удобочитаемыми. Нет смысла использовать специальные символы в именах и фамилиях. Сгенерированные телефонные номера находятся в диапазоне от 100 до 198, который реализуется путем вызова `random(100, 199)`. Причина этого заключается в том, что трехзначное число искать легче. Не забудьте поэкспериментировать со сгенерированными именами, фамилиями и номерами телефонов.

Работа с `phoneBook.go` генерирует следующий вид выходных данных:

```
$ go run phoneBook.go search 123
Data has 100 entries.
{BVHA QEEQL 123}
$ go run phoneBook.go search 1234
Data has 100 entries.
Entry not found: 1234
$ go run phoneBook.go list
Data has 100 entries.
{DGTB GNQKI 169}
{BQNU ZUQFP 120}
...
```

Эти случайно сгенерированные имена и фамилии неидеальны, однако их более чем достаточно для целей тестирования. В следующей главе мы узнаем, как работать с данными CSV.

Упражнения

- Создайте функцию, которая объединяет два массива в новый срез.
- Создайте функцию, которая объединяет два массива в новый массив.
- Создайте функцию, которая объединяет два среза в новый массив.

Резюме

В этой главе мы познакомились с основными типами данных Go, включая числовые типы данных, строки и ошибки. Кроме того, мы выяснили, как группировать похожие значения с помощью массивов и срезов. Наконец, мы рассмотрели различия между массивами и срезами и обсудили, почему срезы более универсальны, чем массивы. Мы коснулись указателей и использовали генерацию случайных чисел и строк для предоставления случайных данных приложению телефонной книги.

В следующей главе мы обсудим пару более сложных составных типов данных Go, а именно *карты* и *структур*. Карты могут использовать ключи разных типов данных, тогда как структуры могут группировать несколько типов данных и создавать новые, к которым вы можете получать доступ как к отдельным объектам. Как вы увидите в последующих главах, структуры играют в Go ключевую роль.

Дополнительные ресурсы

- Документация по пакету `sort`: <https://golang.org/pkg/sort/>.
- Документация по пакету `time`: <https://golang.org/pkg/time/>.
- Документация по пакету `crypto/rand`: <https://golang.org/pkg/crypto/rand/>.
- Документация по пакету `math/rand`: <https://golang.org/pkg/math/rand/>.

3

Составные типы данных

В Go имеется поддержка карт и структур — составных типов данных, которые являются основной темой этой главы. Причина, по которой мы представляем их отдельно от массивов и срезов, заключается в том, что как карты, так и структуры являются более гибкими и эффективными, чем массивы и срезы. Общая идея состоит в том, что если массив или срез не в силах справиться с задачей, то вам, скорее всего, придется использовать карты. Если и карта не может вам помочь, то следует рассмотреть необходимость создания и использования структуры.

Мы уже встречали структуры в главе 1, где создавали первую версию приложения телефонной книги. Однако в текущей главе мы собираемся более пристально рассмотреть как структуры, так и карты. Эти знания позволят нам читать и сохранять данные в формате CSV, используя структуры, и *создавать индекс* для быстрого поиска среза структур на основе заданного ключа с помощью карты.

Наконец, мы применим часть этих функций Go для улучшения приложения телефонной книги, разработку которого начали в главе 1. Новая версия этого приложения загружает и сохраняет свои данные с диска, а это значит, больше не нужно жестко кодировать данные.

В этой главе:

- карты;
- структуры;
- указатели и структуры;

- регулярные выражения и сопоставление с образцом;
- улучшение приложения телефонной книги.

Карты могут использовать ключи разных типов данных, тогда как структуры могут группировать несколько типов данных и создавать новые. Итак, без лишних слов перейдем к картам.

Карты

Как массивы, так и срезы позволяют использовать в качестве индексов только целые положительные числа. Карты (или хеш-таблицы) — эффективные структуры данных, поскольку позволяют использовать индексы различных типов данных в качестве ключей для поиска ваших данных, если эти ключи *можно сравнивать*. Практическое эмпирическое правило заключается в том, что вы должны использовать карту, когда вам нужны индексы, не являющиеся целыми положительными числами, или когда целочисленные индексы разделены большими интервалами.



Хотя переменные `bool` сравниваются, нет смысла использовать `bool` в качестве ключа к Go-карте, поскольку этот тип допускает только два различных значения. Кроме того, значения с плавающей запятой тоже сравниваются, однако проблемы с точностью, вызванные внутренним представлением таких значений, могут привести к ошибкам и сбоям, поэтому лучше избегать использования значений с плавающей запятой в качестве ключей для Go-карт.

Вы можете спросить, зачем вообще нужны карты и в чем их преимущества. Следующий список поможет прояснить ситуацию.

- Карты очень универсальны. Позже в этой главе мы с помощью карты создадим *индекс базы данных*, который позволит искать элементы среза и получать к ним доступ на основе заданного ключа или, в более сложных ситуациях, комбинации ключей.
- Хотя это не всегда так, но работа с картами в Go происходит быстро, поскольку вы можете получать доступ ко всем элементам карты за *линейное время*. Вставка и извлечение элементов из карты выполняются оперативно и не зависят от количества элементов карты.
- Карты просты для понимания, что приводит к понятному дизайну.

Вы можете создать новую переменную `map`, используя либо `make()`, либо литерал карты. С одной стороны, чтобы создать новую карту с ключами типа `string` и значениями `int` с помощью `make()`, достаточно просто вызвать

`make(map[string]int)` и присвоить возвращаемое значение переменной. С другой стороны, если вы решите создать карту, используя литерал карты, то вам нужно написать нечто наподобие этого:

```
m := map[string]int {  
    "key1": -1  
    "key2": 123  
}
```

Версия с литералом карты работает быстрее, если нужно добавить данные в карту во время создания.



Вы не должны делать никаких предположений о порядке расположения элементов внутри карты. Go рандомизирует ключи при проходе по карте — это делается специально и является преднамеренной частью дизайна языка.

Вы можете получить длину карты, которая представляет собой количество ключей в карте, используя функцию `len()`, работающую также с массивами и срезами. Вы можете удалить пару «ключ — значение» из карты, используя функцию `delete()`, которая принимает два аргумента: имя карты и название ключа, именно в таком порядке.

Кроме того, вы можете определить, существует ли ключ `k` в карте `aMap` по второму возвращаемому значению оператора `v, ok := aMap[k]`. Если `ok` имеет значение `true`, то `k` существует и его значение равно `v`. Если он не существует, то для `v` будет установлено нулевое значение его типа данных, зависящее от определения карты. Если вы попытаетесь получить значение ключа, которого нет в карте, то Go не будет на это жаловаться и вернет нулевое значение типа данных.

Теперь обсудим особый случай, при котором переменная карты имеет значение `nil`.

Сохранение в карту `nil`

Вам разрешается присваивать переменной карты значение `nil`. В этом случае вы не сможете использовать ее до тех пор, пока не назначите ее новой переменной карты. Проще говоря, если вы попытаетесь сохранить данные в карту `nil`, то ваша программа аварийно завершится. Это показано в следующем фрагменте кода, который представляет собой реализацию функции `main()` исходного файла `nilMap.go`, который можно найти в каталоге `ch03` репозитория книги на GitHub:

```
func main() {  
    aMap := map[string]int{}  
    aMap["test"] = 1
```

Это работает, поскольку `aMap` указывает на что-то, что является возвращаемым значением `map[string]int{}`.

```
fmt.Println("aMap:", aMap)
aMap = nil
```

На данный момент `aMap` указывает на `nil`, что является синонимом «ничто».

```
fmt.Println("aMap:", aMap)
if aMap == nil {
    fmt.Println("nil map!")
    aMap = map[string]int{}
}
```

Перед использованием хорошо проверять, не указывает ли карта на `nil`. Оператор `if aMap == nil` позволяет нам определить, можем ли мы сохранить значение ключа/пары в `aMap`. В данном случае не можем, и попытка это сделать приведет к сбою в программе. Мы исправляем это с помощью оператора `aMap = map[string]int{}`.

```
aMap["test"] = 1
// Это не сработает!
aMap = nil
aMap["test"] = 1
}
```

В последней части программы мы показываем, как ваша программа аварийно завершится, если вы попытаетесь выполнить сохранение в карту со значением `nil`, — никогда не используйте такой код в рабочих версиях!



В реальных приложениях если функция принимает аргумент карты, то до работы с ним она должна проверить, что карта не равна `nil`.

При выполнении `nilMap.go` мы получаем такой вывод:

```
$ go run nilMap.go
aMap: map[test:1]
aMap: map[]
nil map!
panic: assignment to entry in nil map

goroutine 1 [running]:
main.main()
    /Users/mtsouk/Desktop/mGo3rd/code/ch03/nilMap.go:21 +0x225
```

Причина сбоя программы указана в выводе программы: `panic: assignment to entry in nil map`.

Перебор карт

Объединяясь с ключевым словом `range`, цикл `for` реализует функциональность циклов `foreach` из других языков программирования и позволяет выполнять итерации по всем элементам карты, не зная ее размера или ключей. Применяясь к карте, `range` возвращает *пары «ключ — значение»* именно в таком порядке.

Ведите следующий код и сохраните его как `forMaps.go`:

```
package main

import "fmt"

func main() {
    aMap := make(map[string]string)
    aMap["123"] = "456"
    aMap["key"] = "A value"

    // range также работает с картами
    for key, v := range aMap {
        fmt.Println("key:", key, "value:", v)
    }
}
```

В этом случае мы используем как ключ, так и значение, возвращенное из `range`.

```
for _, v := range aMap {
    fmt.Print(" # ", v)
}
fmt.Println()
```

В данном случае нас интересуют только значения, возвращаемые картой, поэтому мы игнорируем ключи.



Как вы уже знаете, не стоит делать никаких предположений о порядке, в котором пары ключей и значений карты будут возвращены циклом `for` и `range`.

При выполнении `forMaps.go` мы получаем такой вывод:

```
$ go run forMaps.go
key:1 key value: A value
key: 123 value: 456
# 456 # A value
```

После того как мы познакомились с картами, пришло время поговорить о Go-структурах.

Структуры

Структуры в Go одновременно и эффективны, и очень популярны. Они используются для организации и группировки различных типов данных под одним именем. Структуры – более универсальный тип данных в Go и даже могут быть связаны с функциями, которые называются методами.



Структуры, а также другие определяемые пользователем типы данных обычно определяются вне функции `main()` или любой другой функции пакета, так что получают глобальную область видимости и доступны для всего Go-пакета. Поэтому если вы не хотите четко указывать, что тип нужен только в пределах текущей локальной области видимости, то вам следует помещать определения новых типов данных вне функций.

Определение новых структур

Определяя новую структуру, вы группируете набор значений в единый тип данных, который позволяет вам передавать и получать этот набор как единый объект. Структура имеет *поля*, и у каждого поля есть собственный тип данных, который даже может быть другой структурой или срезом структур. Кроме того, поскольку структура является новым типом данных, она определяется с помощью ключевого слова `type`, за которым следует название структуры и ключевое слово `struct`, которое означает, что мы определяем новую структуру.

Следующий код определяет новую структуру `Entry`:

```
type Entry struct {
    Name     string
    Surname string
    Year     int
}
```



Ключевое слово `type` позволяет определять новые типы данных или создавать псевдонимы для существующих. Следовательно, вам разрешено написать `type myInt int` и тем самым определить новый тип данных `myInt`, который является псевдонимом для `int`. Однако Go рассматривает `myInt` и `int` как совершенно разные типы данных, которые нельзя сравнивать напрямую, даже если они содержат одинаковые значения. Каждая структура определяет новый тип данных, отсюда и использование ключевого слова `type`.

По причинам, которые станут очевидны в главе 5, поля структуры обычно пишутся с заглавной буквы — это зависит от того, что вы хотите сделать с ними.

Структура `Entry` состоит из трех полей: `Name`, `Surname` и `Year`. Первые два поля имеют тип `string`, тогда как последнее содержит значение `int`.

К этим трем полям можно получить доступ через точечную нотацию вида `V.Name`, `V.Surname` и `V.Year`, где `V` — имя переменной, содержащей экземпляр структуры `Entry`. *Структурный литерал* `p1` может быть определен как `p1 := aStruct{ "fmt", 12, -2 }`.

Есть два способа работы со структурными переменными. Первый — как с *обычными переменными*, а второй — через *переменные-указатели*, которые указывают на адрес структуры в памяти. Оба способа одинаково хороши и обычно внедрены в отдельные функции, поскольку позволяют правильно инициализировать некоторые либо все поля переменной структуры и/или выполнить любые другие нужные задачи, прежде чем использовать структурную переменную. В результате есть два основных способа создания новой структурной переменной с помощью функции. Первый возвращает обычную структурную переменную, тогда как второй — указатель на структуру. Каждый из способов имеет по два варианта. Первый возвращает экземпляр структуры, инициализируемый компилятором Go, тогда как второй возвращает экземпляр структуры, инициализируемый пользователем.

Порядок, в котором вы размещаете поля в определении типа структуры, имеет значение для *идентификации типа* определенной структуры. Проще говоря, две структуры с одинаковыми полями не будут считаться идентичными в Go, если их поля расположены не в одинаковом порядке.

Использование ключевого слова `new`

Кроме того, вы можете создавать новые экземпляры структуры, используя ключевое слово `new()`, например: `pS := new(Entry)`. Оно обладает следующими свойствами:

- выделяет необходимое пространство памяти, которое зависит от типа данных, а затем обнуляет его;
- всегда *возвращает указатель* на выделенную память;
- работает для всех типов данных, кроме *канала* и *карты*.

Все эти техники показаны в следующем коде. Введите его и сохраните как `structures.go`:

```
package main
```

```
import "fmt"
```

```

type Entry struct {
    Name      string
    Surname   string
    Year      int
}

// инициализируется Go
func zeroS() Entry {
    return Entry{}
}

```

Сейчас самое время вспомнить важное правило Go: *если переменной не задано начальное значение, то компилятор Go автоматически инициализирует ее нулевым значением ее типа данных*. Для структур это означает, что структурная переменная без начального значения инициализируется нулевыми значениями каждого из типов данных ее полей. Следовательно, функция `zeroS()` возвращает инициализированную нулями структуру `Entry`.

```

// инициализируется пользователем
func initS(N, S string, Y int) Entry {
    if Y < 2000 {
        return Entry{Name: N, Surname: S, Year: 2000}
    }
    return Entry{Name: N, Surname: S, Year: Y}
}

```

В этом случае пользователь сам инициализирует новую структурную переменную. Кроме того, функция `initS()` сверяет значение поля `Year` с `2000` и действует соответствующим образом. Если оно меньше `2000`, то значение поля `Year` становится `2000`. Это условие характерно для требований разрабатываемого приложения. Таким образом видно, что место, где вы инициализируете структуру, подходит для проверки вашего ввода.

```

// инициализируется Go – возвращает указатель
func zeroPtoS() *Entry {
    t := &Entry{}
    return t
}

```

Функция `zeroPtoS()` возвращает указатель на структуру, инициализированную нулями.

```

// инициализируется пользователем – возвращает указатель
func initPtoS(N, S string, Y int) *Entry {
    if len(S) == 0 {
        return &Entry{Name: N, Surname: "Unknown", Year: Y}
    }
    return &Entry{Name: N, Surname: S, Year: Y}
}

```

Функция `initPtoS()` также возвращает указатель на структуру, но дополнительно проверяет длину пользовательского ввода. Опять же, эта проверка зависит от конкретного приложения.

```
func main() {
    s1 := zeroS()
    p1 := zeroPtoS()
    fmt.Println("s1:", s1, "p1:", *p1)

    s2 := initS("Mihalis", "Tsoukalos", 2020)
    p2 := initPtoS("Mihalis", "Tsoukalos", 2020)
    fmt.Println("s2:", s2, "p2:", *p2)

    fmt.Println("Year:", s1.Year, s2.Year, p1.Year, p2.Year)

    pS := new(Entry)
    fmt.Println("pS:", pS)
}
```

Вызов `new(Entry)` возвращает *указатель* на структуру `Entry`. Вообще говоря, когда вам приходится инициализировать множество структурных переменных, считается хорошим тоном создать для этого функцию, благодаря чему уменьшится количество потенциальных ошибок.

При выполнении `structures.go` мы получаем такой вывод:

```
s1: { 0} p1: { 0}
s2: {Mihalis Tsoukalos 2020} p2: {Mihalis Tsoukalos 2020}
Year: 0 2020 0 2020
pS: &{ 0}
```

Поскольку нулевое значение строки — это пустая строка, `s1`, `p1` и `pS` не отображают никаких данных из полей `Name` и `Surname`.

В следующем подразделе показано, как группировать структуры одного и того же типа данных и использовать их в качестве элементов среза.

Срезы структур

Вы можете создавать срезы структур, чтобы группировать и обрабатывать несколько структур под одним именем. Однако для доступа к полю заданной структуры требуется знать точное положение структуры в срезе.

А пока взгляните на рис. 3.1, чтобы лучше понять работу среза структур и то, как можно получить доступ к полям определенного элемента среза.

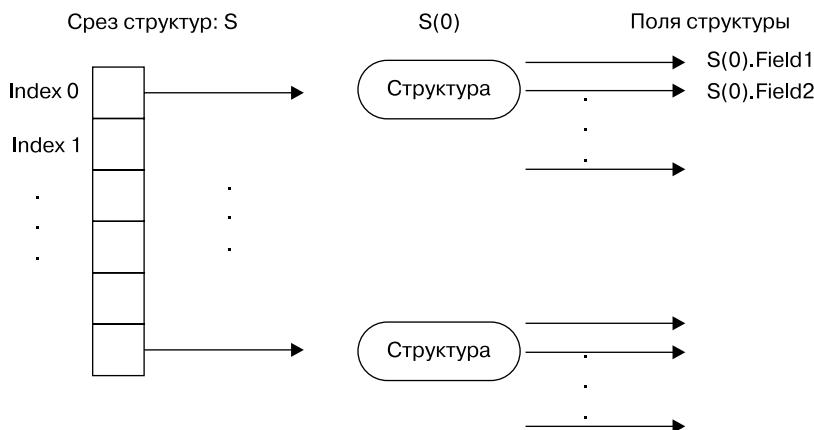


Рис. 3.1. Срез структур

Таким образом, каждый элемент среза представляет собой структуру, доступ к которой осуществляется с помощью индекса. Выбрав нужный элемент среза, мы сможем выбрать его поле.

Поскольку весь процесс может показаться немного запутанным, код этого подраздела призван прояснить ситуацию. Введите следующий код и сохраните его как `sliceStruct.go`. Вы также можете найти его под тем же именем в каталоге `ch03` в репозитории книги на GitHub.

```
package main

import (
    "fmt"
    "strconv"
)

type record struct {
    Field1 int
    Field2 string
}

func main() {
    S := []record{}
    for i := 0; i < 10; i++ {
        text := "text" + strconv.Itoa(i)
        temp := record{Field1: i, Field2: text}
        S = append(S, temp)
    }
}
```

Вам по-прежнему придется использовать `append()`, чтобы добавить новую структуру к срезу.

```
// доступ к полям первого элемента
fmt.Println("Index 0:", S[0].Field1, S[0].Field2)
fmt.Println("Number of structures:", len(S))
sum := 0
for _, k := range S {
    sum += k.Field1
}
fmt.Println("Sum:", sum)
```

При выполнении `sliceStruct.go` мы получаем такой вывод:

```
Index 0: 0 text0
Number of structures: 10
Sum: 45
```

Мы вернемся к структурам в следующей главе, где обсудим рефлексию, а также в главе 6, где выясним, как работать с данными JSON, используя структуры. Пока же мы обсудим регулярные выражения и сопоставление с образцом.

Регулярные выражения и сопоставление с образцом

Сопоставление с образцом — это метод поиска в строке некоторого набора символов на основе определенного паттерна поиска, основанного на регулярных выражениях и грамматиках.

Регулярное выражение — это последовательность символов, которая задает паттерн поиска. Каждое регулярное выражение компилируется в распознаватель путем построения обобщенной диаграммы перехода, называемой *конечным автоматом*. Он может быть как детерминированным, так и недетерминированным. Недетерминированность означает, что для одного и того же входного сигнала может быть несколько переходов из состояния. *Распознаватель* — это программа, которая принимает строку *x* в качестве входных данных и может определить, является ли *x* предложением данного языка.

Грамматика — это набор производственных правил для строк на формальном языке. Производственные правила описывают, как создавать строки из алфавита языка, которые допустимы с точки зрения его синтаксиса. Грамматика не описывает значение строки или то, что с ней можно сделать в любом контексте, — она описывает лишь ее форму. Здесь важно понимать, что грамматики

лежат в основе регулярных выражений, поскольку без грамматики вы не сможете определить или использовать регулярное выражение.

Вас, наверное, уже мучит вопрос, почему в этой главе мы говорим о регулярных выражениях и сопоставлении с образцом. Причина проста. Через некоторое время вы узнаете, как хранить и считывать данные CSV из обычных текстовых файлов. Следовательно, вы должны быть в состоянии определить, являются ли читаемые данные действительными.

О регулярных выражениях Go

Мы начнем этот подраздел с представления некоторых распространенных *пatterнов сопоставления*, используемых для построения регулярных выражений (табл. 3.1).

Таблица 3.1

Выражение	Описание
.	Соответствует любому символу
*	Означает любое количество раз — не может использоваться сам по себе
?	Ноль или один раз — не может использоваться сам по себе
+	Означает один или несколько раз — не может использоваться сам по себе
^	Обозначает начало строки
^	Обозначает конец строки
[]	Предназначен для группировки символов
[A-Z]	Означает все символы от заглавной буквы A до заглавной буквы Z
\d	Любая цифра в интервале 0-9
\D	Не цифра
\w	Любой словообразующий символ: [0-9A-Za-z_]
\W	Любой несловесный символ
\s	Символ пробела
\S	Символ, не являющийся пробелом

Символы, представленные в этой таблице, используются для построения и определения грамматики регулярного выражения. Пакет Go, отвечающий

за определение регулярных выражений и выполнение сопоставления с образцом, называется `regexp`. Для создания регулярного выражения мы используем функцию `regexp.MustCompile()`, а чтобы увидеть, соответствует ли данная строка паттерну, — функцию `Match()`.

Функция `regexp.MustCompile()` анализирует данное регулярное выражение и возвращает переменную `regexp.Regexp`, которую можно использовать для сопоставления. Данная переменная является представлением *скомпилированного регулярного выражения*. Функция выдает состояние паники (`panic`), если выражение не может быть проанализировано, и это хорошо, поскольку уже на ранней стадии процесса вы будете знать, что ваше выражение недопустимо. Метод `re.Match()` возвращает значение `true`, если данный *байтовый срез* соответствует регулярному выражению `re`, которое является переменной `regexp.Regexp`, и значение `false` в противном случае.



Создание отдельных функций для сопоставления с образцом может оказаться удобным, поскольку позволяет повторно использовать функции, не беспокоясь о контексте программы.

Имейте в виду: регулярные выражения и сопоставление с образцом на первый взгляд выглядят удобными и полезными, однако служат причиной множества ошибок. Мой совет — использовать самое простое регулярное выражение, которое может решить вашу задачу. Но если вы сможете вообще избежать использования регулярных выражений, то в долгосрочной перспективе это будет намного лучше!

Сопоставление имен и фамилий

Представленная утилита сопоставляет имена и фамилии, то есть строки, которые начинаются с заглавной буквы и продолжаются строчными. Входные данные не должны содержать никаких цифр или других символов.

Исходный код утилиты можно найти в файле `nameSurRE.go`, который расположен в каталоге `ch03`. Функция, поддерживающая желаемую функциональность, называется `matchNameSur()` и реализована следующим образом:

```
func matchNameSur(s string) bool {
    t := []byte(s)
    re := regexp.MustCompile(`^[A-Z][a-z]*$`)
    return re.Match(t)
}
```

Логика функции заключается в регулярном выражении `^[A-Z][a-z]*\$`, где ^ обозначает начало строки, а \$ — ее конец. Данное регулярное выражение сопоставляет все, что начинается с заглавной буквы ([A-Z]), и продолжается любым количеством строчных ([a-z]*). Это означает, что Z будет совпадением, но ZA уже не будет из-за второй буквы в верхнем регистре. Аналогично, Jo+ не покажет совпадение, поскольку содержит символ +.

При выполнении `nameSurRE.go` с различными типами входных данных мы получаем такой вывод:

```
$ go run nameSurRE.go Z
true
$ go run nameSurRE.go ZA
false
$ go run nameSurRE.go Mihalis
True
```

Этот метод поможет вам проверить вводимые пользователем данные.

Сопоставление целых чисел

Представленная утилита сопоставляет целые числа как со знаком, так и без него. Реализация зависит от определения регулярного выражения. Если нам нужны только целые числа без знака, то мы должны удалить из регулярного выражения [-+]? или заменить его на [+]?.

Исходный код утилиты можно найти в файле `intRE.go` в каталоге `ch03`. Функция `matchInt()`, поддерживающая желаемую функциональность, реализована следующим образом:

```
func matchInt(s string) bool {
    t := []byte(s)
    re := regexp.MustCompile(`^[-+]?[\\d]+$`)
    return re.Match(t)
}
```

Как и ранее, логика функции содержится в регулярном выражении, которое используется для сопоставления целых чисел, а именно `^[-+]?[\\d]+\$`. На языке людей мы говорим здесь о том, что хотим сопоставить что-то, что начинается с — или +, является необязательным (?) и заканчивается любым количеством цифр (\d+). Кроме того, требуется, чтобы у нас была хотя бы одна цифра до конца проверяемой строки (\$).

При выполнении `intRE.go` с различными типами входных данных мы получаем такой вывод:

```
$ go run intRE.go 123
true
$ go run intRE.go /123
false
$ go run intRE.go +123.2
false
$ go run intRE.go +
false
$ go run intRE.go -123.2
false
```

Позже в этой книге вы узнаете, как тестировать Go-код с помощью тестирующих функций, но сейчас мы будем проводить большую часть тестирования вручную.

Сопоставление полей записи

В этом примере используется другой подход, поскольку мы читаем запись целиком и разделяем ее перед выполнением проверки. Кроме того, мы проводим дополнительную проверку, чтобы убедиться в том, что обрабатываемая запись содержит нужное количество полей. Каждая запись должна содержать три поля: имя, фамилию и номер телефона.

Полный код утилиты можно найти в `fieldsRE.go` в каталоге `ch03`. Функция с желаемой функциональностью реализована следующим образом:

```
func matchRecord(s string) bool {
    fields := strings.Split(s, ",")
    if len(fields) != 3 {
        return false
    }

    if !matchNameSur(fields[0]) {
        return false
    }

    if !matchNameSur(fields[1]) {
        return false
    }
    return matchTel(fields[2])
}
```

Сначала функция `matchRecord()` разделяет поля записи на основе символа запятой, после чего отправляет каждое отдельное поле в соответствующую функцию для дальнейшей проверки, предварительно убедившись в том, что запись содержит нужное количество полей, что является обычной практикой. Разде-

ление полей выполняется с помощью `strings.Split(s, ",")`, которая возвращает срез с таким количеством элементов, сколько есть полей записи.

Если проверки первых двух полей успешны, то функция возвращает возвращаемое значение `matchTel(fields[2])`, поскольку именно эта последняя проверка и определяет конечный результат.

При выполнении `fieldsRE.go` с различными типами входных данных мы получаем такой вывод:

```
$ go run fieldsRE.go Name,Surname,2109416471
true
$ go run fieldsRE.go Name,Surname,OtherName
false
$ go run fieldsRE.go One,Two,Three,Four
false
```

Первая запись верна, и поэтому возвращается значение `true`. Второй запуск, где поле номера телефона неверно, дает противоположный результат. Последний запуск заканчивается неудачей из-за наличия четырех полей вместо трех.

Улучшение приложения телефонной книги

Пришло время обновить приложение телефонной книги. Новая версия соответствующей утилиты получит следующие улучшения:

- поддержку команд `insert` и `delete`;
- возможность считывать данные из файла и записывать их перед завершением;
- у каждой записи появится поле последнего посещения, которое обновляется;
- появится индекс базы данных, реализованный с использованием Go-карты;
- для проверки прочитанных телефонных номеров будут использоваться регулярные выражения.

Работа с CSV-файлами

В большинстве случаев вы не захотите терять все данные или начинать работу без каких-либо данных вообще каждый раз, когда запускаете свое приложение. Существует множество способов это исправить, и самый простой из них — сохранить данные локально. Очень подходящим форматом для этого является CSV, о котором мы расскажем далее, а потом используем в приложении телефонной книги. Хорошая новость в том, что Go предоставляет специальный пакет для работы с данными CSV, называемый `encoding/csv` (<https://golang.org/pkg/encoding/csv/>). В представленной утилите как входные, так и выходные файлы задаются в качестве аргументов командной строки.



Существуют два очень популярных Go-интерфейса: `io.Reader` и `io.Writer`, которые связаны с чтением и записью в файлы. Почти все операции чтения и записи в Go задействуют именно эти интерфейсы. Использование одного и того же интерфейса для считывателей позволяет им иметь некоторые общие характеристики, и, что наиболее важно, дает возможность создавать собственные считыватели и использовать их везде, где Go ожидает увидеть `io.Reader`. То же самое относится и к записывателям, которые удовлетворяют требованиям интерфейса `io.Writer`. Больше информации об интерфейсах вы узнаете в главе 4.

Вот основные задачи, которые нам необходимо реализовать:

- загрузка CSV-данных с диска и помещение их в срез структур;
- сохранение данных на диск в формате CSV.

Пакет `encoding/csv` содержит функции, которые помогут вам читать и записывать CSV-файлы. Мы имеем дело с небольшими CSV-файлами, поэтому используем `csv.NewReader(f).ReadAll()` для чтения всего входного файла сразу. В случае больших файлов данных или когда необходимо проверить либо внести какие-то изменения во входные данные по мере чтения, было бы лучше получать их построчно, используя `Read()` вместо `ReadAll()`.

Go предполагает, что CSV-файл использует символ запятой (,) для разделения различных полей каждой строки. Если мы захотим изменить это поведение, то нам придется изменить значение переменной `Comma` в программе чтения или записи CSV в зависимости от поставленной задачи. Мы изменим это поведение в выходном CSV-файле, где поля будут разделены с помощью символа табуляции.



По соображениям совместимости для входных и выходных CSV-файлов лучше использовать один и тот же разделитель. Мы используем символ табуляции в качестве разделителя полей в выходном файле, чтобы просто проиллюстрировать использование переменной `Comma`.

Поскольку работа с CSV-файлами — новая тема, в каталоге `ch03` репозитория книги на GitHub есть отдельная утилита `csvData.go`, которая иллюстрирует методы чтения и записи CSV-файлов. Исходный код `csvData.go` представлен в виде фрагментов. Сначала мы представляем преамбулу `csvData.go`, содержащую раздел `import`, а также структуру `Record` и глобальную переменную `myData`, которая представляет собой срез `Record`.

```
package main

import (
    "encoding/csv"
    "fmt"
```

```
        "os"
    )

type Record struct {
    Name      string
    Surname   string
    Number    string
    LastAccess string
}

var myData = []Record{}
```

Затем мы вводим функцию `readCSVFile()`, которая считывает обычный текстовый файл с данными CSV.

```
func readCSVFile(filepath string) ([][]string, error) {
    _, err := os.Stat(filepath)
    if err != nil {
        return nil, err
    }

    f, err := os.Open(filepath)
    if err != nil {
        return nil, err
    }
    defer f.Close()

    // CSV-файл читается весь сразу
    // тип данных lines – [][]string
    lines, err := csv.NewReader(f).ReadAll()
    if err != nil {
        return [][]string{}, err
    }

    return lines, nil
}
```

Обратите внимание: мы проверяем, существует ли указанный путь к файлу и связан ли он с обычным файлом внутри функции. Нет правильного или неправильного места, где выполнять эту проверку, — вы просто должны быть последовательны. Функция `readCSVFile()` возвращает срез `string[][]`, содержащий все прочитанные нами строки. Кроме того, имейте в виду: `csv.NewReader()` разделяет поля каждой строки ввода, что является основной причиной необходимости использования двумерного среза для хранения входных данных.

После этого мы иллюстрируем технику записи в CSV-файл с помощью функции `saveCSVFile()`:

```
func saveCSVFile(filepath string) error {
    csvfile, err := os.Create(filepath)
    if err != nil {
```

```

        return err
    }
    defer csvfile.Close()

    csvwriter := csv.NewWriter(csvfile)
    // изменение разделителя полей по умолчанию на табуляцию
    csvwriter.Comma = '\t'
    for _, row := range myData {
        temp := []string{row.Name, row.Surname, row.Number, row.LastAccess}
        _ = csvwriter.Write(temp)
    }
    csvwriter.Flush()
    return nil
}

```

Обратите внимание на изменение значения по умолчанию для `csvwriter.Comma`.

Наконец, мы можем взглянуть на реализацию функции `main()`:

```

func main() {
    if len(os.Args) != 3 {
        fmt.Println("csvData input output!")
        return
    }

    input := os.Args[1]
    output := os.Args[2]
    lines, err := readCSVFile(input)
    if err != nil {
        fmt.Println(err)
        return
    }

    // данные CSVчитываются по столбцам – каждая строка
    // представляет собой срез
    for _, line := range lines {
        temp := Record{
            Name:      line[0],
            Surname:   line[1],
            Number:    line[2],
            LastAccess: line[3],
        }
        myData = append(myData, temp)
        fmt.Println(temp)
    }

    err = saveCSVFile(output)
    if err != nil {
        fmt.Println(err)
        return
    }
}

```

Функция `main()` с помощью `readCSVFile()` помещает прочитанное в срез `myData`. Помните, что `lines` — это срез с двумя измерениями и что каждая строка в `lines` уже разделена на поля.

В данном случае каждая строка ввода содержит четыре поля. Содержимое файла данных CSV, используемого в качестве входных данных, выглядит следующим образом:

```
$ cat ~/csv.data
Dimitris,Tsoukalos,2101112223,1600665563
Mihalis,Tsoukalos,2109416471,1600665563
Jane,Doe,0800123456,1608559903
```

При выполнении `csvData.go` мы получаем такой вывод:

```
$ go run csvData.go ~/csv.data /tmp/output.data
{Dimitris Tsoukalos 2101112223 1600665563}
{Mihalis Tsoukalos 2109416471 1600665563}
{Jane Doe 0800123456 1608559903}
```

Содержимое выходного CSV-файла будет таким:

```
$ cat /tmp/output.data
Dimitris      Tsoukalos      2101112223      1600665563
Mihalis      Tsoukalos      2109416471      1600665563
Jane        Doe      0800123456      1608559903
```

В файле `output.data` для разделения различных полей каждой записи используются символы табуляции. Утилита `csvData.go` может пригодиться для преобразования между различными типами CSV-файлов.

Добавление индекса

В этом подразделе мы посмотрим, как с помощью карты реализуется индекс базы данных. Индексация в базах основана на одном или нескольких уникальных ключах. Обычно мы индексируем нечто уникальное, то, к чему хотим получить быстрый доступ. В случае базы данных первичные ключи уникальны по умолчанию и не могут присутствовать в нескольких записях. В нашем случае номера телефонов используются в качестве первичных ключей; это значит, индекс строится на основе поля номера телефона структуры.



Как правило, индексируется поле, которое будет использоваться для поиска. Нет смысла создавать индекс, который не будет использоваться для запросов.

Теперь посмотрим, что все это означает на практике. Представьте, что у нас есть срез `S` со следующим типом данных:

```
S[0]={0800123123, ...}  
S[1]={0800123000, ...}  
S[2]={2109416471, ...}  
. . .
```

Таким образом, каждый элемент среза представляет собой структуру, которая может содержать гораздо больше данных, помимо телефонного номера. Как мы можем создать для него индекс? Индекс `Index` будет иметь следующие данные и формат:

```
Index["0800123123"] = 0  
Index["0800123000"] = 1  
Index["2109416471"] = 2  
. . .
```

Это означает, что если мы хотим найти телефонный номер `0800123000`, то должны проверить, существует ли `0800123000` в качестве ключа в `Index`. Если существует, то мы понимаем, что значение `0800123000`, то есть `Index["0800123000"]`, является индексом элемента среза, который содержит нужную запись. Теперь мы знаем, к какому элементу среза нужен доступ, поэтому нам не требуется выполнять поиск по всему срезу. Вооружившись этими знаниями, обновим приложение.

Улучшенная версия приложения для телефонной книги

Было бы прискорбно создать приложение телефонной книги, записи в котором жестко закодированы. На этот раз записи адресной книгичитываются из внешнего файла, содержащего данные в формате CSV. Аналогичным образом новая версия сохраняет свои данные в тот же CSV-файл, который можно прочитать позже.

Каждая запись в приложении телефонной книги основана на следующей структуре:

```
type Entry struct {  
    Name      string  
    Surname   string  
    Tel       string  
    LastAccess string  
}
```

Ключом к записям служит поле `Te1` и, следовательно, его значения. На практике это означает, что если вы попытаетесь добавить запись, использующую существующее значение `Te1`, то процесс завершится неудачей.

Это также означает, что приложение выполняет поиск в телефонной книге, используя значения `Te1`. Базы данных используют первичные ключи для идентификации уникальных записей. Наше же приложение телефонной книги имеет небольшую базу данных, реализованную в виде среза структур `Entry`. Наконец, телефонные номера сохраняются без каких-либо символов – внутри. Перед их сохранением утилита удаляет все символы – из телефонных номеров, если таковые там имеются.



Я предпоchitaю разбираться с различными частями более крупного приложения, создавая программы меньшего размера, которые в сочетании реализуют некоторые или все функциональные возможности более крупной программы. Это помогает понять, как должно быть реализовано более крупное приложение, и значительно облегчает последующее объединение функционала и разработку конечного продукта.

Поскольку это реальное приложение, реализованное в виде утилиты командной строки, оно должно поддерживать команды для манипулирования данными и поиска. Обновленная функциональность утилиты содержит:

- вставку данных с помощью команды `insert`;
- удаление данных с помощью команды `delete`;
- поиск данных с помощью команды `search`;
- получение списка записей с помощью команды `list`.

Для упрощения кода путь к файлу данных CSV жестко закодирован. Кроме того, файл CSV автоматически считывается при запуске утилиты и автоматически же обновляется при выполнении команд `insert` и `delete`.



Go поддерживает CSV, но гораздо более популярным форматом, который используется для обмена данными в веб-сервисах, является JSON. Однако работать с данными в CSV проще, чем с данными в JSON. Работа с данными JSON рассматривается в главе 6.

Как я уже говорил, данная версия приложения для телефонной книги поддерживает *индексацию*, что позволяет быстрее находить нужные записи, не выполняя линейный поиск по всему срезу. Используемый метод индексирования не очень сложный, но действительно ускоряет поиск. При условии,

что процесс поиска основан на телефонных номерах, мы создадим карту, которая связывает каждый номер телефона с индексом записи, содержащей этот номер в срезе структур. Таким образом, простой и быстрый поиск по карте сообщит нам, существует ли такой номер телефона. Если номер есть, то мы можем получить доступ к его записи напрямую, обходясь без необходимости искать его по всему срезу структур. Единственным недостатком этого метода, как и любого другого метода индексации, является то, что вы должны постоянно обновлять карту.

Вышеописанный процесс называется высокоуровневым проектированием приложения. Для такого простого приложения вам не нужно углубляться в аналитику его возможностей, а достаточно указать поддерживаемые команды и местоположение файла данных CSV. Однако для сервера RESTful, который реализует REST API, этапы проектирования или зависимости программы так же важны, как и сам этап разработки.

Весь код обновленной утилиты телефонной книги можно найти в каталоге `ch03` в файле `phoneBook.go` — как всегда, мы ссылаемся на репозиторий книги на GitHub. Это последний раз, когда мы делаем подобное уточнение. Далее мы будем сообщать только имя исходного файла, если нет конкретной причины поступить иначе.

Ниже представлены наиболее интересные части файла `phoneBook.go`, начиная с реализации функции `main()`, которая имеет две части. Первая посвящена получению команды для выполнения и наличию действительного CSV-файла для работы.

```
func main() {
    arguments := os.Args
    if len(arguments) == 1 {
        fmt.Println("Usage: insert|delete|search|list <arguments>")
        return
    }

    // если CSVFILE не существует, создаем пустой
    _, err := os.Stat(CSVFILE)
    // если ошибка не равна nil, то файл не существует
    if err != nil {
        fmt.Println("Creating", CSVFILE)
        f, err := os.Create(CSVFILE)
        if err != nil {
            f.Close()
            fmt.Println(err)
            return
        }
        f.Close()
    }
}
```

Если путь к файлу, указанный в глобальной переменной `CSVFILE`, еще не существует, то мы должны создать его, чтобы далее программа могла его использовать. Это определяется с помощью возвращаемого значения вызова `os.Stat(CSVFILE)`.

```
fileInfo, err := os.Stat(CSVFILE)
// Это обычный файл?
mode := fileInfo.Mode()
if !mode.IsRegular() {
    fmt.Println(CSVFILE, "not a regular file!")
    return
}
```

Файл `CSVFILE` должен не только существовать, но и быть обычным файлом UNIX, что выясняется путем вызова `mode.IsRegular()`. Если это не обычный файл, то утилита выводит сообщение об ошибке и завершает работу.

```
err = readCSVFile(CSVFILE)
if err != nil {
    fmt.Println(err)
    return
}
```

Это место, где мы читаем `CSVFILE`, даже если он пуст. Содержимое файла хранится в глобальной переменной `data`, которая определяется как `[]Entry{}` и представляет собой срез переменных `Entry`.

```
err = createIndex()
if err != nil {
    fmt.Println("Cannot create index.")
    return
}
```

Здесь мы создаем индекс, вызывая `createIndex()`. Индекс хранится в глобальной переменной `index`, которая определена как `map[string]int`.

Вторая часть функции `main()` посвящена запуску нужной команды и выяснению того, была ли команда выполнена успешно:

```
// различие между командами
switch arguments[1] {
case "insert":
    if len(arguments) != 5 {
        fmt.Println("Usage: insert Name Surname Telephone")
        return
    }
    t := strings.ReplaceAll(arguments[4], "-", "")
    if !matchTel(t) {
        fmt.Println("Not a valid telephone number:", t)
        return
    }
}
```

Необходимо с помощью `strings.ReplaceAll()` удалить из телефонного номера все символы `-`, прежде чем сохранять его. Если символов `-` нет, то замены не происходит.

```

temp := initS(arguments[2], arguments[3], t)
// если nil, то произошла ошибка
if temp != nil {
    err := insert(temp)
    if err != nil {
        fmt.Println(err)
        return
    }
}
case "delete":
    if len(arguments) != 3 {
        fmt.Println("Usage: delete Number")
        return
    }
    t := strings.ReplaceAll(arguments[2], "-", "")
    if !matchTel(t) {
        fmt.Println("Not a valid telephone number:", t)
        return
    }
    err := deleteEntry(t)
    if err != nil {
        fmt.Println(err)
    }
}
case "search":
    if len(arguments) != 3 {
        fmt.Println("Usage: search Number")
        return
    }
    t := strings.ReplaceAll(arguments[2], "-", "")
    if !matchTel(t) {
        fmt.Println("Not a valid telephone number:", t)
        return
    }
    temp := search(t)
    if temp == nil {
        fmt.Println("Number not found:", t)
        return
    }
    fmt.Println(*temp)
}
case "list":
    list()
default:
    fmt.Println("Not a valid option")
}
}

```

В этом относительно большом блоке `switch` мы можем наблюдать, что именно выполняется для каждой заданной команды. Итак, мы имеем следующее:

- для команды `insert` выполняем функцию `insert()`;
- для команды `list` выполняем функцию `list()`, которая является единственной функцией, требующей каких-либо аргументов;
- для команды `delete` выполняем функцию `deleteEntry()`;
- для команды `search` выполняем функцию `search()`.

Все остальное обрабатывается веткой `default`. Индекс создается и обновляется с помощью функции `createIndex()`, которая реализована следующим образом:

```
func createIndex() error {
    index = make(map[string]int)
    for i, k := range data {
        key := k.Tel
        index[key] = i
    }
    return nil
}
```

Проще говоря, вы получаете доступ ко всему срезу `data` и помещаете пары индекса и значения среза на карту, используя значение в качестве ключа для карты и индекс среза в качестве значения карты.

Команда `delete` реализована следующим образом:

```
func deleteEntry(key string) error {
    i, ok := index[key]
    if !ok {
        return fmt.Errorf("%s cannot be found!", key)
    }
    data = append(data[:i], data[i+1:]...)
    // обновить индекс – ключ больше не существует
    delete(index, key)

    err := saveCSVFile(CSVFILE)
    if err != nil {
        return err
    }
    return nil
}
```

Функция `deleteEntry()` работает просто. Сначала вы выполняете поиск по индексу телефонного номера, чтобы найти место записи в срезе с данными. Если его нет, то вы просто создаете сообщение об ошибке с помощью `fmt.Errorf("%s cannot be found!", key)` и функция возвращается. Если номер телефона найден,

то вы удаляете соответствующую запись из среза `data` с помощью `append(data[:i], data[i+1:]...)`.

Затем вы должны обновить индекс, поскольку забота о нем — та цена, которую вы платите за дополнительную скорость, возникающую благодаря ему. Кроме того, после удаления записи следует сохранить обновленные данные, вызвав `saveCSVFile(CSVFILE)`, чтобы изменения вступили в силу.



Строго говоря, поскольку текущая версия приложения телефонной книги обрабатывает по одному запросу за раз, вам не нужно обновлять индекс, так как он создается с нуля каждый раз, когда вы используете приложение. В системах управления базами данных индексы в добавок сохраняются на диске, чтобы избежать больших затрат на их создание с нуля.

Команда `insert` реализована следующим образом:

```
func insert(pS *Entry) error {
    // если уже существует, то не добавлять
    _, ok := index[(*pS).Tel]
    if ok {
        return fmt.Errorf("%s already exists", pS.Tel)
    }
    data = append(data, *pS)
    // обновить индекс
    _ = createIndex()

    err := saveCSVFile(CSVFILE)
    if err != nil {
        return err
    }
    return nil
}
```

Индекс здесь помогает определить, существует ли номер телефона, который вы пытаетесь добавить. Как говорилось ранее, если вы попытаетесь добавить запись, использующую существующее значение `Tel`, то процесс завершится неудачей. Если данная проверка пройдена, то мы добавляем новую запись в срез `data`, обновляем индекс и сохраняем данные в CSV-файле.

Команда `search`, которая использует индекс, реализована следующим образом:

```
func search(key string) *Entry {
    i, ok := index[key]
    if !ok {
        return nil
    }
    data[i].LastAccess = strconv.FormatInt(time.Now().Unix(), 10)
    return &data[i]
}
```

Благодаря индексу поиск телефонного номера очень прост — код просто ищет в `index` нужный телефонный номер. Если он есть, то код возвращает эту запись. В противном случае возвращает `nil`, что возможно, поскольку функция возвращает указатель на переменную `Entry`. Перед возвратом записи функция `search()` обновляет поле `LastAccess` структуры, которая должна быть возвращена, чтобы зафиксировать время, когда к ней обращались в последний раз.

Исходное содержимое файла данных CSV, используемого в качестве входных данных, выглядит следующим образом:

```
$ cat ~/csv.data
Dimitris,Tsoukalos,2101112223,1600665563
Mihalis,Tsoukalos,2109416471,1600665563
Mihalis,Tsoukalos,2109416771,1600665563
Efipanios,Savva,2101231234,1600665582
```

Поскольку номер телефона унikalен, поля имени и фамилии могут встречаться несколько раз. При выполнении `phoneBook.go` мы получаем такой вывод:

```
$ go run phoneBook.go list
{Dimitris Tsoukalos 2101112223 1600665563}
{Mihalis Tsoukalos 2109416471 1600665563}
{Mihalis Tsoukalos 2109416771 1600665563}
{Efipanios Savva 2101231234 1600665582}
$ go run phoneBook.go delete 2109416771
$ go run phoneBook.go search 2101231234
{Efipanios Savva 2101231234 1608559833}
$ go run phoneBook.go search 210-1231-234
{Efipanios Savva 2101231234 1608559840}
```

В нашем коде `210-1231-234` преобразуется в `2101231234`.

```
$ go run phoneBook.go delete 210-1231-234
$ go run phoneBook.go search 210-1231-234
Number not found: 2101231234
$ go run phoneBook.go insert Jane Doe 0800-123-456
$ go run phoneBook.go insert Jane Doe 0800-123-456
0800123456 already exists
$ go run phoneBook.go search 2101112223
{Dimitris Tsoukalos 2101112223 1608559928}
```

Содержимое CSV-файла после выполнения вышеприведенных команд выглядит следующим образом:

```
$ cat ~/csv.data
Dimitris,Tsoukalos,2101112223,1600665563
Mihalis,Tsoukalos,2109416471,1600665563
Jane,Doe,0800123456,1608559903
```

Вы можете видеть, что утилита преобразовала 0800-123-456 в 0800123456, и это является желаемым поведением.

Несмотря на то что новая версия утилиты телефонной книги превосходит предыдущую, она все еще неидеальна. Вот список возможностей, которые можно улучшить:

- сортировка выходных данных на основе телефонного номера;
- сортировка выходных данных на основе фамилии;
- использование записей JSON и срезов JSON для данных вместо файлов CSV.

Мы продолжим совершенствовать приложение телефонной книги уже в следующей главе, где реализуем сортировку срезов по элементам структуры.

Упражнения

- Напишите Go-программу, которая преобразует существующий массив в карту.
- Напишите Go-программу, которая преобразует существующую карту в два среза: первый должен содержать ключи карты, тогда как второй — значения. Значения с индексом *n* двух срезов должны соответствовать паре «ключ — значение», которую можно найти в исходной карте.
- Внесите необходимые изменения в `nameSurRE.go`, чтобы получить возможность обрабатывать несколько аргументов командной строки.
- Измените код `intRE.go` в целях обработки нескольких аргументов командной строки и отображения итоговой суммы результатов `true` и `false`.
- Внесите изменения в `csvData.go`, чтобы поля записи разделялись на основе символа #.
- Напишите Go-утилиту, которая преобразует `os.Args` в срез структур с полями для хранения индекса и значения каждого аргумента командной строки. Вы должны определить используемую структуру самостоятельно.
- Внесите необходимые изменения в `phoneBook.go`, чтобы индекс создавался на основе поля `LastAccess`. Может ли это пригодиться? Работает ли это? Почему?
- Внесите изменения в `csvData.go`, чтобы получить возможность разделять поля записи символом, который задается в качестве аргумента командной строки.

Резюме

В этой главе мы обсудили составные типы данных Go, а именно карты и структуры. Кроме того, мы поговорили о работе с CSV-файлами, а также об использовании регулярных выражений и сопоставлении с образцами в Go. Теперь мы можем хранить наши данные в надлежащих структурах, проверять их с помощью регулярных выражений, когда это возможно, и сохранять в CSV-файлах, чтобы обеспечить сохраняемость данных.

Следующая глава посвящена методам типов, которые представляют собой функции, привязанные к типу данных, рефлексии и интерфейсам. Все это позволит нам улучшить приложение телефонной книги.

Дополнительные ресурсы

- Документация по `encoding/csv`: <https://golang.org/pkg/encoding/csv/>.
- Документация по пакету `runtime`: <https://golang.org/pkg/runtime/>.

4

Рефлексия и интерфейсы

Помните приложение телефонной книги из предыдущей главы? Вы можете задаться вопросом, как сортировать пользовательские структуры данных, такие как записи телефонной книги, на основе ваших собственных критериев, например фамилии или имени. Что делать, когда нужно отсортировать разные наборы данных, имеющие какое-то общее поведение? Причем без необходимости реализовывать сортировку с нуля для каждого из различных типов данных, используя при этом несколько функций. Представьте утилиту, подобную приложению телефонной книги, которая может обрабатывать файлы данных CSV двух разных форматов на основе заданного входного файла. Каждый вид CSV-записи хранится в своей Go-структуре, а это означает, что каждый вид CSV-записи может быть отсортирован по-разному. Как же это реализовать без того, чтобы писать две разные утилиты командной строки? Наконец, представьте, что требуется создать утилиту, которая сортирует действительно необычные данные. Например, вы хотите отсортировать срез, содержащий различные виды 3D-фигур, в зависимости от их объема. Есть ли легкий и разумный способ выполнить подобную задачу?

Ответом на все эти вопросы и трудности будет использование *интерфейсов*. Однако это подразумевает не только манипулирование данными и их сортировку. Интерфейсы предназначены для выражения абстракций, а также идентификации и определения поведения, которое может быть общим для различных типов данных. Как только вы внедрили интерфейс для типа данных, переменным и значениям этого типа становится доступен целый мир функциональных возможностей, которые могут сэкономить ваше время и повысить производительность. Интерфейсы работают с *методами на типах* или *методами типов* типов, которые похожи на функции, прикрепленные к заданным типам данных, в роли

которых в Go обычно выступают структуры. Помните: как только вы реализуете требуемые методы типа интерфейса, этот интерфейс *реализуется неявно*, что также относится к *пустому интерфейсу*, о котором мы поговорим в текущей главе.

Еще одной удобной функцией Go является *рефлексия*, которая позволяет вам изучать структуру типа данных во время выполнения. Но рефлексия — расширенная функция Go, поэтому вам не нужно использовать ее на регулярной основе.

В этой главе:

- рефлексия;
- методы типа;
- интерфейсы;
- работа с двумя различными форматами файлов CSV;
- объектно-ориентированное программирование в Go;
- обновление приложения телефонной книги.

Рефлексия

Мы начинаем главу с рефлексии, расширенной функции Go, не потому, что это простая тема, а потому, что она поможет понять, как именно язык работает с различными типами данных, включая интерфейсы, и зачем они вообще нужны.

Возможно, вам интересно, как узнать имена полей структуры во время выполнения. В таких случаях вам нужно использовать рефлексию. Помимо того что рефлексия позволяет вам выводить поля и значения структуры, она также дает возможность изучать неизвестные структуры и манипулировать ими. Например, подобными тем, которые созданы на основе декодирования данных JSON.

Вот два главных вопроса, которые я задал себе, когда впервые познакомился с рефлексией.

- Почему рефлексия была включена в Go?
- Когда я должен использовать рефлексию?

Отвечу на первый вопрос: рефлексия позволяет *динамически* выяснить тип произвольного объекта вместе с информацией о его структуре. Для работы с рефлексией в Go представлен пакет `reflect`. Помните, в предыдущей главе мы упоминали, что `fmt.Println()` достаточно сообразителен, чтобы понимать типы данных своих параметров и действовать соответствующе? Так вот, «под капотом» пакет `fmt` использует для этого рефлексию.

Что касается второго вопроса, то рефлексия позволяет работать с типами данных, которые не существуют на момент написания кода, но могут существовать в будущем, когда мы используем существующий пакет с пользовательскими типами данных.

Кроме того, рефлексия может пригодиться, когда необходимо работать с типами данных, которые не реализуют общий интерфейс и, следовательно, имеют необычное или неизвестное поведение. Это не означает плохое или ошибочное поведение, а просто необычное, такое как определяемая пользователем структура.



Появление дженериков в Go может в ряде случаев снизить частоту использования рефлексии, поскольку с их помощью можно легко работать с различными типами данных, не зная заранее их точно. Однако ничто не сравнится с рефлексией, когда необходимо полностью изучить структуру и типы данных переменной. Мы сравним рефлексию с дженериками в главе 13.

Наиболее полезными частями пакета `reflect` являются два типа данных: `reflect.Value` и `reflect.Type`. В частности, `reflect.Value` используется для хранения значений любого типа, тогда как `reflect.Type` служит для представления Go-типов. Существуют две функции: `reflect.TypeOf()` и `reflect.valueOf()`, которые возвращают `reflect.Type` и `reflect.Value` соответственно. Обратите внимание, что `reflect.TypeOf()` возвращает фактический тип переменной, и если мы исследуем структуру, то она вернет имя структуры.

Поскольку структуры играют ключевую роль в Go, пакет `reflect` содержит метод `reflect.NumField()`, предназначенный для перечисления количества полей в структуре, а также метод `Field()`, позволяющий получать значение `reflect.Value` определенного поля структуры.

Пакет `reflect` также определяет тип данных `reflect.Kind`, который используется для представления *определенного* типа данных переменной: `int`, `struct` и т. д. В документации к пакету `reflect` перечислены все возможные значения типа данных `reflect.Kind`. Функция `Kind()` возвращает вид переменной.

Наконец, методы `Int()` и `String()` возвращают целое и строковое значения `reflect.Value` соответственно.



Код рефлексии иногда может выглядеть непривычно, и его трудно читать. Поэтому, согласно философии Go, рефлексию нужно использовать, только когда это абсолютно необходимо, поскольку, несмотря на функциональность, она не ведет к чистому коду.

Изучение внутренней структуры Go-структурь

Следующая утилита показывает, как использовать рефлексию для обнаружения внутренней структуры и полей переменной Go-структурь. Введите ее и сохраните как `reflection.go`:

```
package main

import (
    "fmt"
    "reflect"
)

type Secret struct {
    Username string
    Password string
}

type Record struct {
    Field1 string
    Field2 float64
    Field3 Secret
}

func main() {
    A := Record{"String value", -12.123, Secret{"Mihalis", "Tsoukalos"}}
}
```

Мы начинаем с определения структурной переменной `Record`, которая содержит другое структурное значение (`Secret{"Mihalis", "Tsoukalos"}`).

```
r := reflect.ValueOf(A)
fmt.Println("String value:", r.String())
```

Здесь возвращается значение `reflect.Value` переменной `A`.

```
iType := r.Type()
```

Используя `Type()`, мы получаем тип данных переменной — в данном случае переменной `A`.

```
fmt.Printf("i Type: %s\n", iType)
fmt.Printf("The %d fields of %s are\n", r.NumField(), iType)

for i := 0; i < r.NumField(); i++ {
```

Цикл `for` выше позволяет посетить все поля структуры и изучить их характеристики.

```
fmt.Printf("\t%s ", iType.Field(i).Name)
fmt.Printf("\twith type: %s ", r.Field(i).Type())
fmt.Printf("\tand value _%v_\n", r.Field(i).Interface())
```

Оператор `fmt.Sprintf()` выше возвращает имя, тип данных и значение полей.

```
// проверьте, есть ли другие структуры, встроенные в запись
k := reflect.TypeOf(r.Field(i).Interface()).Kind()
// нужно преобразование в строку, чтобы можно было сравнить
if k.String() == "struct" {
```

Чтобы проверить тип данных переменной с помощью строки, нам нужно сначала преобразовать тип данных в `string`.

```
    fmt.Println(r.Field(i).Type())
}

// то же, что и раньше, но с использованием внутреннего значения
if k == reflect.Struct {
```

Во время проверки вы также можете использовать внутреннее представление типа данных. Однако смысла в этом меньше, чем в использовании значения `string`.

```
    fmt.Println(r.Field(i).Type())
}
}
```

При выполнении `reflection.go` мы получаем такой вывод:

```
$ go run reflection.go
String value: <main.Record Value>
i Type: main.Record
The 3 fields of main.Record are
    Field1 with type: string      and value _String value_
    Field2 with type: float64     and value _-12.123_
    Field3 with type: main.Secret and value _{Mihalis Tsoukalos}_
main.Secret
main.Secret
```

Итак, `main.Record` — полное уникальное имя структуры, определенное Go, `main` — имя пакета, а `Record` — имя структуры. Это позволяет Go различать элементы разных пакетов.

Представленный код не изменяет никаких значений структуры. Если бы требовалось внести изменения в значения ее полей, то пришлось бы использовать метод `Elem()` и передавать ее в виде указателя в `valueOf()` — помните, что указатели позволяют вносить изменения в фактическую переменную. Есть методы, благодаря которым можно изменять существующее значение. В нашем случае мы собираемся с помощью `setString()` и `SetInt()` изменять поля `string` и `int` соответственно.

Данный метод описан в следующем подразделе.

Изменение значений структуры с использованием рефлексии

Изучать внутреннее строение Go-структур полезно само по себе, но более практично иметь возможность изменять значения в ней, что и является предметом обсуждения в этом подразделе.

Ведите следующий код Go и сохраните его как `setValues.go`. Его также можно найти в репозитории книги на GitHub:

```
package main

import (
    "fmt"
    "reflect"
)

type T struct {
    F1 int
    F2 string
    F3 float64
}

func main() {
    A := T{1, "F2", 3.0}
```

`A` — это переменная, которая исследуется в данной программе.

```
    fmt.Println("A:", A)
    r := reflect.ValueOf(&A).Elem()
```

С помощью `Elem()` и указателя на переменную `A` она может быть изменена при необходимости.

```
    fmt.Println("String value:", r.String())
    typeOfA := r.Type()
    for i := 0; i < r.NumField(); i++ {
        f := r.Field(i)
        tOfA := typeOfA.Field(i).Name
        fmt.Printf("%d: %s %s = %v\n", i, tOfA, f.Type(), f.Interface())

        k := reflect.TypeOf(r.Field(i).Interface()).Kind()
        if k == reflect.Int {
            r.Field(i)..SetInt(-100)
        } else if k == reflect.String {
            r.Field(i).SetString("Changed!")
        }
    }
```

Мы используем `SetInt()` для изменения целочисленного значения и `setString()` для изменения значения `string`. Целочисленные значения устанавливаются равными `-100`, а строковые значения — `Changed!`.

```
    fmt.Println("A:", A)
}
```

При выполнении `setValues.go` мы получаем такой вывод:

```
$ go run setValues.go
A: {1 F2 3}
String value: <main.T Value>
0: F1 int = 1
1: F2 string = F2
2: F3 float64 = 3
A: {-100 Changed! 3}
```

Первая строка выходных данных показывает начальную версию `A`, тогда как последняя — окончательную версию `A` с измененными полями. Основное применение такого кода заключается в *динамическом* изменении значений полей структуры.

Три недостатка рефлексии

Без сомнения, рефлексия — эффективная функция Go. Однако, как и все инструменты, ее следует использовать осторожно по трем основным причинам.

- Первая заключается в том, что широкое использование рефлексии затрудняет чтение и поддержку ваших программ. Потенциальным решением этой проблемы служит хорошее документирование, но разработчики печально известны тем, что не находят времени на написание надлежащей документации.
- Вторая причина состоит в том, что Go-код, использующий рефлексию, замедляет ваши программы. Вообще говоря, Go-код, который работает с определенным типом данных, всегда быстрее, чем Go-код, который использует рефлексию для динамической работы с любым типом данных Go. Кроме того, такой динамический код затрудняет рефакторинг или анализ вашего кода с помощью специальных инструментов.
- Последняя причина заключается в том, что ошибки рефлексии не могут быть обнаружены во время сборки и появляются в виде сообщений об ошибке (`panic`) уже во время выполнения. Это означает, что ошибки рефлексии потенциально могут привести к аварийному завершению ваших программ. Это может произойти через месяцы или даже годы после разработки Go-программы! Одним из решений проблемы будут тщательные проверки перед вызовом опасной функции. Однако это добавляет еще больше Go-кода в ваши программы, что делает их еще медленнее.

Теперь, когда мы поговорили о рефлексии и о том, что она может, пришло время обсудить методы типов, которые необходимы для использования интерфейсов.

Методы типа

Метод типа — это функция, которая привязана к определенному типу данных. Методы типов (или методы на типах) на самом деле являются функциями, однако определяются и используются немного по-другому.



Функционал методов типа добавляет в Go некоторые возможности объектно-ориентированного программирования, что очень удобно и широко используется в Go. Кроме того, методы типа нужны для работы интерфейсов.

Определять новые методы типа так же просто, как создавать новые функции, при условии, что вы следите определенным правилам, которые связывают функцию с типом данных.

Создание методов типа

Итак, представьте, что вы хотите выполнить вычисления с помощью матриц 2×2 . Самым естественным способом реализации этого станет определение нового типа данных и методов типа для сложения, вычитания и умножения матриц 2×2 с использованием этого нового типа данных. Чтобы сделать его еще более интересным и универсальным, мы создадим утилиту командной строки, которая принимает элементы двух матриц 2×2 в качестве аргументов командной строки (в общей сложности восемь целых значений) и выполняет все три вычисления, используя определенные методы типа.

Имея тип данных `ar2x2`, вы можете создать для него метод типа `functionName` следующим образом:

```
func (a ar2x2) functionName(parameters) <return values> {  
    ...  
}
```

Часть (`a ar2x2`) — это то, что делает функцию `functionName()` методом типа, поскольку связывает ее с типом данных `ar2x2`. *Никакой другой тип данных не сможет использовать эту функцию*. Однако вы можете без проблем реализовать `functionName()` для других типов данных или в виде обычной функции. Если у вас есть переменная `ar2x2` с именем `varAr`, вы можете вызвать

`functionName()` как `varAr.functionName(...)`, что выглядит как выбор поля структуры.

Вы не обязаны использовать методы типа, если не хотите этого. Фактически любой метод типа может быть *переписан как обычная функция*. Следовательно, `functionName()` можно переписать так:

```
func FunctionName(a ar2x2, parameters...) <return values> {  
    ...  
}
```

Имейте в виду, что «под капотом» компилятор Go действительно превращает методы в обычные вызовы функций со значением `self` в качестве первого параметра. Однако для работы интерфейсов требуется использование методов *типа*.



Выражения, используемые для выбора поля структуры или метода типа типа данных, которые заменяют многоточие после имени переменной выше, называются селекторами.

Выполнение вычислений между матрицами заданного размера — один из редких случаев, когда использование массива вместо среза имеет больше смысла, поскольку вам не требуется менять размер матриц. Здесь можно возразить, что использование среза вместо указателя массива будет более эффективным. Вам разрешено использовать то, что выглядит для вас более разумным.

Большую часть времени и когда есть такая необходимость, результаты метода типа сохраняются в переменной, которая вызвала метод типа. Для реализации этого в случае типа данных `ar2x2` мы передаем указатель на массив, который вызвал метод типа, например, так: `func (a *ar2x2)`.

В следующем подразделе показаны методы типа в действии.

Использование методов типа

В этом подразделе показано использование методов типа с помощью типа данных `ar2x2` в качестве примера. Функция `Add()` и метод `Add()` применяют один и тот же алгоритм для сложения двух матриц. Единственное различие заключается в способе вызова и в том факте, что функция возвращает массив, тогда как метод сохраняет результат в вызывающей переменной.

Сложение и вычитание матриц — простая задача (вы просто поэлементно складываете или вычитаете элементы первой и второй матрицы, расположенные в одинаковой позиции), но умножение матриц — уже более сложный процесс.

Это основная причина, по которой как при сложении, так и при вычитании используют циклы `for`, а это значит, код может работать и с большими матрицами, тогда как умножение использует статический код, который неприменим к большим матрицам без внесения в него существенных изменений.



Если вы определяете методы типа для структуры, то должны убедиться, что имена методов типа не конфликтуют с каким-либо именем поля структуры, поскольку компилятор Go отклонит такие двусмысленности.

Ведите следующий код и сохраните его как `methods.go`:

```
package main

import (
    "fmt"
    "os"
    "strconv"
)

type ar2x2 [2][2]int

// традиционная функция Add()
func Add(a, b ar2x2) ar2x2 {
    c := ar2x2{}
    for i := 0; i < 2; i++ {
        for j := 0; j < 2; j++ {
            c[i][j] = a[i][j] + b[i][j]
        }
    }
    return c
}
```

Здесь у нас есть традиционная функция `Add()`, которая складывает две переменные `ar2x2` и возвращает результат.

```
// метод типа Add()
func (a *ar2x2) Add(b ar2x2) {
    for i := 0; i < 2; i++ {
        for j := 0; j < 2; j++ {
            a[i][j] = a[i][j] + b[i][j]
        }
    }
}
```

А здесь метод типа `Add()`, который привязан к типу данных `ar2x2`. Результат сложения не возвращается. В данном случае переменная `ar2x2`, которая вызвала метод `Add()`, будет изменена и сохранит результат — это причина использования указателя при определении метода типа. Если такое поведение вам

не нужно, придется изменить сигнатуру и реализацию метода типа в соответствии с вашими потребностями.

```
// метод типа Subtract()
func (a *ar2x2) Subtract(b ar2x2) {
    for i := 0; i < 2; i++ {
        for j := 0; j < 2; j++ {
            a[i][j] = a[i][j] - b[i][j]
        }
    }
}
```

Представленный выше метод вычитает `ar2x2 b` из `ar2x2 a`, а результат сохраняется в `a`.

```
// метод типа Multiply()
func (a *ar2x2) Multiply(b ar2x2) {
    a[0][0] = a[0][0]*b[0][0] + a[0][1]*b[1][0]
    a[1][0] = a[1][0]*b[0][0] + a[1][1]*b[1][0]
    a[0][1] = a[0][0]*b[0][1] + a[0][1]*b[1][1]
    a[1][1] = a[1][0]*b[0][1] + a[1][1]*b[1][1]
}
```

Мы работаем с небольшими массивами, поэтому выполняем умножение, не используя циклы `for`.

```
func main() {
    if len(os.Args) != 9 {
        fmt.Println("Need 8 integers")
        return
    }

    k := [8]int{}
    for index, i := range os.Args[1:] {
        v, err := strconv.Atoi(i)
        if err != nil {
            fmt.Println(err)
            return
        }
        k[index] = v
    }
    a := ar2x2{{k[0]}, {k[1]}, {k[2]}, {k[3]}}
    b := ar2x2{{k[4]}, {k[5]}, {k[6]}, {k[7]}}
```

Функция `main()` получает входные данные и создает две матрицы 2×2 . После этого выполняются нужные вычисления с этими двумя матрицами.

```
fmt.Println("Traditional a+b:", Add(a, b))
a.Add(b)
```

```
fmt.Println("a+b:", a)
a.Subtract(a)
fmt.Println("a-a:", a)

a = ar2x2{{k[0], k[1]}, {k[2], k[3]}}
```

Мы вычисляем $a + b$ двумя различными способами: с помощью обычной функции и с помощью метода типа. Поскольку и `a.Add(b)`, и `a.Subtract(a)` изменяют значение `a`, мы должны инициализировать `a`, прежде чем использовать его повторно.

```
a.Multiply(b)
fmt.Println("a*b:", a)

a = ar2x2{{k[0], k[1]}, {k[2], k[3]}}
b.Multiply(a)
fmt.Println("b*a:", b)
}
```

Наконец, мы вычисляем $a * b$ и $b * a$, чтобы показать их различие, поскольку свойство коммутативности не применяется к умножению матриц.

При выполнении `methods.go` мы получаем такой вывод:

```
$ go run methods.go 1 2 0 0 2 1 1 1
Traditional a+b: [[3 3] [1 1]]
a+b: [[3 3] [1 1]]
a-a: [[0 0] [0 0]]
a*b: [[4 6] [0 0]]
b*a: [[2 4] [1 2]]
```

Входными данными здесь служат две матрицы 2×2 , $[[1\ 2]\ [0\ 0]]$ и $[[2\ 1]\ [1\ 1]]$, а результат — это их расчеты.

Теперь, когда мы знаем о методах типов, пришло время перейти к интерфейсам, поскольку они не могут быть реализованы без методов типа.

Интерфейсы

Интерфейс — это механизм Go для определения поведения, реализуемый с помощью набора методов. Интерфейсы играют ключевую роль в Go и могут упростить код ваших программ, когда приходится иметь дело с несколькими типами данных, выполняющими одну и ту же задачу. Например, вспомним, что `fmt.Println()` работает практически для всех типов данных. Но имейте в виду, что интерфейсы не должны быть излишне сложными. Если вы решите

создать собственные интерфейсы, то начать следует с общего поведения, которое вы хотите использовать для нескольких типов данных.

Интерфейсы работают с *методами на типах* (или *методами типов*), которые напоминают функции, прикрепленные к заданным типам данных. В Go такими типами обычно являются структуры (хотя мы можем использовать любой тип данных).

Как вы уже знаете, если требуемые методы типа интерфейса реализованы, считается, что тип *неявно удовлетворяет* интерфейсу.

Пустой интерфейс определяется просто как `interface{}`. Поскольку пустой интерфейс не имеет методов, он уже *реализован всеми типами данных*.



Как только вы реализуете методы интерфейса для типа данных, этот интерфейс автоматически считается удовлетворенным этим типом данных.

Говоря более формально, тип Go-интерфейса определяет (или описывает) *поведение* других типов, указывая набор *методов*, которые должны быть реализованы для поддержки этого поведения. Чтобы тип данных удовлетворял интерфейсу, он должен реализовать *все методы типа*, требуемые этим интерфейсом. Следовательно, интерфейсы являются *абстрактными типами*, которые определяют набор необходимых к реализации методов, чтобы другой тип мог считаться экземпляром интерфейса. Итак, интерфейс составляют два компонента: *набор методов и тип*. Имейте в виду, что небольшие и четко определенные интерфейсы обычно являются самыми популярными.



Возьмите за правило создавать новый интерфейс только в том случае, если вы хотите распространить общее поведение на два или более конкретных типа данных. По сути, это утиная (неявная) типизация.

Самое большое *преимущество* использования интерфейсов заключается в том, что при необходимости вы можете передать переменную типа данных, который реализует определенный интерфейс, любой функции, ожидающей параметр этого конкретного интерфейса. Таким образом вы избавляетесь от необходимости писать отдельные функции для каждого поддерживаемого типа данных. Однако с недавним появлением в Go дженериков появилась и альтернатива.

Интерфейсы также можно использовать для поддержки своего рода полиморфизма в Go, что является объектно-ориентированной концепцией. *Полиморфизм* предлагает способ доступа к объектам разных типов одним и тем же единственнообразным способом, а именно через общее поведение.

Наконец, интерфейсы могут быть использованы для *композиции*. На практике это означает, что вы можете комбинировать существующие интерфейсы и создавать новые, которые содержат комбинированное поведение объединяемых интерфейсов. На рис. 4.1 графически показана композиция интерфейса.

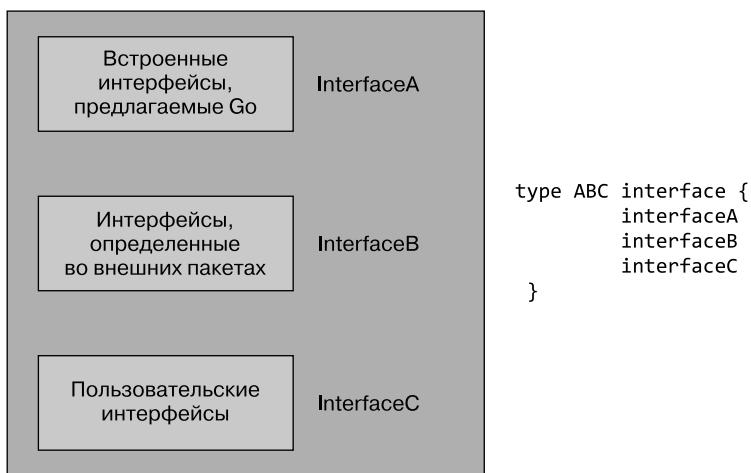


Рис. 4.1. Композиция интерфейса

Проще говоря, на этом рисунке показано следующее: для того чтобы тип удовлетворял интерфейсу ABC, необходимо, чтобы он удовлетворял также InterfaceA, InterfaceB и InterfaceC. Кроме того, любая переменная ABC может использоваться вместо переменных InterfaceA, InterfaceB или InterfaceC, поскольку поддерживает все три варианта поведения. Наконец, только переменные ABC могут использоваться там, где ожидается переменная ABC. Ничто не запрещает вам включать дополнительные методы в определение интерфейса ABC, если комбинация существующих интерфейсов не описывает желаемое поведение точно.



Когда вы объединяете существующие интерфейсы, лучше, чтобы они не содержали методы с одинаковым именем.

Всегда имейте в виду: нет необходимости в том, чтобы интерфейс был массивным и требовал реализации большого количества методов. Фактически чем меньше в нем методов, тем более универсальным и широко используемым он может стать, а это повышает его полезность и, следовательно, использование.

В следующем подразделе показано использование `sort.Interface`.

Интерфейс sort.Interface

Пакет `sort` содержит интерфейс `sort.Interface`, который позволяет сортировать срезы в соответствии с вашими потребностями и данными при условии, что вы реализуете `sort.Interface` для пользовательских типов данных, хранящихся в срезах. Пакет `sort` определяет `sort.Interface` так:

```
type Interface interface {
    // Len – это количество элементов в коллекции
    Len() int
    // Less сообщает, должен ли элемент с индексом i
    // сортироваться перед элементом с индексом j
    Less(i, j int) bool
    // Swap меняет местами элементы с индексами i и j
    Swap(i, j int)
}
```

Из определения `sort.Interface` видно, что для реализации `sort.Interface` нам нужно реализовать следующие три метода типа:

- `Len() int`;
- `Less(i, j int) bool`;
- `Swap(i, j int)`.

Метод `Len()` возвращает длину сортируемого среза и помогает интерфейсу обрабатывать все элементы среза, тогда как метод `Less()`, который сравнивает и сортирует элементы попарно, определяет, как именно элементы будут сравниваться и, следовательно, сортироваться. Возвращаемое значение `Less()` равно `bool`; это значит, `Less()` заботит только то, меньше ли элемент с индексом `i` элемента с индексом `j` при сравнении этих двух элементов. Наконец, метод `Swap()` позволяет обменивать два элемента среза, что необходимо для работы алгоритма сортировки.

В следующем коде, который можно найти в `sort.go`, показано использование `sort.Interface`:

```
package main

import (
    "fmt"
    "sort"
)

type S1 struct {
    F1 int
    F2 string
    F3 int
}
```

```
// мы хотим отсортировать записи S2 на основе значения F3.F1,
// что эквивалентно S1.F1, поскольку F3 является структурой S1
type S2 struct {
    F1 int
    F2 string
    F3 S1
}
```

Структура S2 включает в себя поле F3, относящееся к типу данных S1, который также является структурой.

```
type S2slice []S2
```

Вам нужно использовать срез, поскольку все операции сортировки работают со срезами. Именно для этого среза, который должен представлять собой новый тип данных, в этом случае S2slice, мы и будем реализовывать три метода типа `sort.Interface`.

```
// реализация sort.Interface для S2slice
func (a S2slice) Len() int {
    return len(a)
}
```

Так выглядит реализация `Len()` для типа данных `S2slice`. Обычно это так же просто.

```
// какое поле использовать при сравнении
func (a S2slice) Less(i, j int) bool {
    return a[i].F3.F1 < a[j].F3.F1
}
```

Так выглядит реализация `Less()` для типа данных `S2slice`. Этот метод определяет способ сортировки элементов. В данном случае — с помощью поля встроенной структуры данных (`F3.F1`).

```
func (a S2slice) Swap(i, j int) {
    a[i], a[j] = a[j], a[i]
}
```

Такой будет реализация метода типа `Swap()`, который определяет способ обмена элементов среза во время сортировки. Обычно это так же просто.

```
func main() {
    data := []S2{
        S2{1, "One", S1{1, "S1_1", 10}},
        S2{2, "Two", S1{2, "S1_1", 20}},
        S2{-1, "Two", S1{-1, "S1_1", -20}},
    }
    fmt.Println("Before:", data)
    sort.Sort(S2slice(data))
    fmt.Println("After:", data)
```

```
// обратная сортировка работает автоматически
sort.Sort(sort.Reverse(S2slice(data)))
fmt.Println("Reverse:", data)
}
```

Реализовав `sort.Interface`, вы увидите, что `sort.Reverse()`, который используется для обратной сортировки вашего среза, заработает автоматически.

При выполнении `sort.go` мы получаем такой вывод:

```
$ go run sort.go
Before: [{1 One {1 S1_1 10}} {2 Two {2 S1_1 20}} {-1 Two {-1 S1_1 -20}}]
After: [{-1 Two {-1 S1_1 -20}} {1 One {1 S1_1 10}} {2 Two {2 S1_1 20}}]
Reverse: [{2 Two {2 S1_1 20}} {1 One {1 S1_1 10}} {-1 Two {-1 S1_1 -20}}]
```

В первой строке показаны элементы среза в том виде, в каком они хранились изначально. Вторая строка показывает отсортированную версию, тогда как последняя — версию с обратной сортировкой.

Пустой интерфейс

Как упоминалось ранее, *пустой интерфейс* определяется просто как `interface{}` и уже реализован *всеми типами данных*. Следовательно, переменные любого типа данных могут быть помещены вместо параметра с типом данных пустого интерфейса. В свою очередь, функция с параметром `interface{}` может принимать в этом месте переменные любого типа данных. Но если вы собираетесь работать с параметрами функции `interface{}`, не изучая их тип данных внутри функции, то должны обрабатывать их с помощью операторов, которые работают со всеми типами данных. В противном случае ваш код может привести к сбою или неправильному поведению.

Следующая программа определяет две структуры `S1` и `S2`, но только одну функцию `Print()` для вывода обеих. Это разрешено, поскольку `Print()` требует параметра `interface{}`, который может принимать как переменные `S1`, так и `S2`. Оператор `Fmt.Println(s)` внутри `Print()` может работать как с `S1`, так и с `S2`.



Если создать функцию, которая принимает один или несколько параметров `interface{}`, и выполнить оператор, который может быть применен только к ограниченному количеству типов данных, то все может кончиться не очень хорошо. Так, например, не все параметры `interface{}` могут быть умножены на 5 или использоваться в `fmt.Printf()` с управляемой строкой `%d`.

Исходный код `empty.go` выглядит следующим образом:

```
package main

import "fmt"

type S1 struct {
    F1 int
    F2 string
}

type S2 struct {
    F1 int
    F2 S1
}

func Print(s interface{}) {
    fmt.Println(s)
}

func main() {
    v1 := S1{10, "Hello"}
    v2 := S2{F1: -1, F2: v1}
    Print(v1)
    Print(v2)
}
```

Хотя `v1` и `v2` относятся к разным типам данных, `Print()` может работать с обоими.

```
// вывод целого числа
Print(123)
// вывод строки
Print("Go is the best!")
}
```

Функция `Print()` может работать как с целыми числами, так и со строками.

При выполнении `empty.go` мы получаем такой вывод:

```
{10 Hello}
{-1 {10 Hello}}
123
Go is the best!
```

Использовать пустой интерфейс станет легко, как только вы поймете, что можно передать переменную любого типа вместо параметра `interface{}` и вернуть любой тип данных в качестве возвращаемого значения `interface{}`. Однако с большой властью приходит и большая ответственность. Нужно быть очень осторожными с параметрами `interface{}` и возвращаемыми значениями, поскольку использовать их реальные значения вы можете, только если уверены в их базовом типе данных. Мы обсудим это несколько позже.

Утверждения типа и переключатели типов

Утверждение типа — механизм для работы с базовым конкретным значением интерфейса. В основном это нужно потому, что интерфейсы представляют собой виртуальные типы данных без собственных значений — они просто определяют поведение и не содержат собственных данных. Но что, если вы не знаете тип данных перед попыткой утверждения типа? Как отличить поддерживаемые типы данных от неподдерживаемых? Как выбрать разные действия для каждого поддерживаемого типа данных? Ответ заключается в использовании *переключателей типа*. *Переключатели типа* используют блоки `switch` для типов данных и позволяют различать значения утверждения типа (которые являются типами данных) и обрабатывать каждый тип данных так, как вам требуется. Вместе с тем для того чтобы использовать пустой интерфейс в переключателях типа, вам необходимо использовать *утверждения типа*.



У вас могут быть переключатели типа для всех видов интерфейсов и типов данных в целом.

Таким образом, настоящая работа начинается после попадания в функцию, поскольку именно здесь вам нужно определить поддерживаемые типы данных и действия, которые выполняются для каждого из них.

Утверждения типа используют нотацию `x.(T)`, где `x` — тип интерфейса, а `T` — тип, и помогают извлечь значение, скрытое за пустым интерфейсом. Чтобы утверждение типа работало, `x` не должен быть равным `nil`, а динамический тип `x` должен быть идентичен типу `T`.

Следующий код можно найти в файле `typeSwitch.go`:

```
package main

import "fmt"

type Secret struct {
    SecretValue string
}

type Entry struct {
    F1 int
    F2 string
    F3 Secret
}

func Teststruct(x interface{}) {
    // переключатель типа
    switch T := x.(type) {
    case Secret:
```

```
    fmt.Println("Secret type")
    case Entry:
        fmt.Println("Entry type")
    default:
        fmt.Printf("Not supported type: %T\n", T)
    }
}
```

Это переключатель типа, который поддерживает типы `Secret` и `Entry`.

```
func Learn(x interface{}) {
    switch T := x.(type) {
    default:
        fmt.Printf("Data type: %T\n", T)
    }
}
```

Функция `Learn()` выводит тип данных своего входного параметра.

```
func main() {
    A := Entry{100, "F2", Secret{"myPassword"}}
    Teststruct(A)
    Teststruct(A.F3)
    Teststruct("A string")

    Learn(12.23)
    Learn('€')
}
```

Последняя часть кода вызывает нужные функции для изучения переменной `A`. При выполнении `typeSwitch.go` мы получаем такой вывод:

```
$ go run typeSwitch.go
Entry type
Secret type
Not supported type: string
Data type: float64
Data type: int32
```

Как вы можете видеть, на основе типа данных переменной, переданной в `TestStruct()` и `Learn()`, выполняется различный код.

Строго говоря, утверждения типа позволяют выполнять две основные задачи.

- Проверять, сохраняет ли значение интерфейса определенный тип. Будучи использованным таким образом, утверждение типа возвращает два значения: базовое и `bool`. Базовое значение — то, что вам, возможно, и нужно. Однако именно значение переменной `bool` сообщает, было ли утверждение типа успешным и, следовательно, можно ли использовать базовое значение. Проверка этого, относится ли переменная `aVar` к типу `int`, требует использования нотации `aVar.(int)`, которая возвращает два значения. В случае успеха вернется

реальное значение `int` для `aVar` и `true`. В противном случае будет возвращено значение `false` в качестве второго значения, а это значит, утверждение типа не было успешным и реальное значение извлечь не удалось.

- Использовать конкретное значение, хранящееся в интерфейсе, или присвоить его новой переменной. Это означает, что если в интерфейсе содержится переменная `float64`, то утверждение типа позволит вам получить это значение.



Функциональность, предлагаемая пакетом `reflect`, помогает Go идентифицировать базовый тип данных и реальное значение переменной `interface{}`.

До сих пор мы видели вариант только первого случая, когда мы извлекаем тип данных, хранящийся в переменной пустого интерфейса. Теперь же выясним, как извлечь реальное значение, хранящееся в переменной пустого интерфейса.

Как уже объяснялось ранее, попытка извлечь конкретное значение из интерфейса с помощью утверждения типа может иметь два результата:

- если вы используете правильный конкретный тип данных, то получите базовое значение без каких-либо проблем;
- если вы используете неправильный конкретный тип данных, то ваша программа выдаст сообщение об ошибке (`panic`).

Все это показано в `assertions.go`, который содержит следующий код, а также множество комментариев, объясняющих процесс:

```
package main

import (
    "fmt"
)

func returnNumber() interface{} {
    return 12
}

func main() {
    anInt := returnNumber()
```

Функция `returnNumber()` возвращает значение `int`, которое заключено в пустой интерфейс.

```
    number := anInt.(int)
    number++
    fmt.Println(number)
```

Здесь мы получаем значение `int`, заключенное в переменную пустого интерфейса (`anInt`).

```
// следующий оператор завершится неудачей,
// поскольку нет утверждения типа для получения
// значения: anInt++

// следующий оператор завершается с ошибкой,
// но это контролируемый сбой, поскольку
// переменная ok типа bool сообщает,
// было ли утверждение типа успешным
value, ok := anInt.(int64)
if ok {
    fmt.Println("Type assertion successful: ", value)
} else {
    fmt.Println("Type assertion failed!")
}

// следующий оператор выполняется успешно,
// но опасен тем, что не гарантирует
// успешного выполнения утверждения типа
// он успешен по стечению обстоятельств
i := anInt.(int)
fmt.Println("i:", i)

// следующий код вызовет панику, поскольку anInt не является bool
_ = anInt.(bool)
}
```

Последний оператор вызывает панику в программе, поскольку переменная `anInt` не содержит значения `bool`. При выполнении `assertions.go` мы получаем такой вывод:

```
$ go run assertions.go
13
Type assertion failed!
i: 12
panic: interface conversion: interface {} is int, not bool

goroutine 1 [running]:
main.main()
    /Users/mtsouk/Desktop/mGo3rd/code/ch04/assertions.go:39 +0x192
```

Причина паники на экране: `panic: interface conversion: interface {} is int, not bool`. Чем еще компилятор Go может вам помочь?

Далее мы обсудим карту `map[string]interface{}`` и ее использование.

Карта `map[string]interface{}`

У вас есть утилита, которая обрабатывает собственные аргументы командной строки; если все идет по плану, то вы получаете поддерживаемые типы аргументов командной строки. Но что будет, если произойдет нечто неожиданное? В этом случае поможет карта `map[string]interface{}`, и в данном подразделе показано, как именно!

Помните, что самое большое преимущество использования карты `map[string]interface{}` (или любой карты, которая хранит значение `interface{}`) заключается в том, что у вас сохраняются данные в их исходном состоянии и типе данных. Если использовать вместо этого `map[string]string` или что-то подобное, то любые имеющиеся у вас данные будут преобразованы в `string`, что означает потерю информации об исходном типе данных и структуре данных, которые вы храните в карте.

В настоящее время веб-сервисы работают, обмениваясь записями в формате JSON. Если вы получаете запись JSON в поддерживаемом формате, то можете ожидать ее обработать, и все будет хорошо. Однако бывают случаи, когда вы получаете ошибочную запись или запись в неподдерживаемом формате JSON. Тогда использование `map[string]interface{}` для хранения этих неизвестных записей JSON (произвольных данных) станет хорошим выбором, поскольку `map[string]interface{}` прекрасно подходит для хранения записей JSON неизвестного типа. Мы проиллюстрируем это с помощью утилиты `mapEmpty.go`, которая обрабатывает произвольные записи JSON, переданные в качестве аргументов командной строки. Мы обработаем входную запись JSON двумя способами, которые похожи, но не идентичны. Нет никакой реальной разницы между функциями `exploreMap()` и `typeSwitch()`, кроме того факта, что `typeSwitch()` генерирует гораздо более информативный результат. Код `mapEmpty.go` выглядит следующим образом:

```
package main

import (
    "encoding/json"
    "fmt"
    "os"
)

var JSONrecord = `{
    "Flag": true,
    "Array": ["a", "b", "c"],
    "Entity": {
        "a1": "b1",
        "a2": "b2"
    }
}`
```

```
    "a2": "b2",
    "Value": -456,
    "Null": null
},
"Message": "Hello Go!"
}`
```

Эта глобальная переменная содержит значение по умолчанию для `JSONrecord` на случай отсутствия пользовательского ввода.

```
func typeSwitch(m map[string]interface{}) {
    for k, v := range m {
        switch c := v.(type) {
        case string:
            fmt.Println("Is a string!", k, c)
        case float64:
            fmt.Println("Is a float64!", k, c)
        case bool:
            fmt.Println("Is a Boolean!", k, c)
        case map[string]interface{}:
            fmt.Println("Is a map!", k, c)
            typeSwitch(v.(map[string]interface{}))
        default:
            fmt.Printf "...Is %v: %T!\n", k, c
        }
    }
    return
}
```

Функция `typeSwitch()` использует переключатель типа для различения значений в своей входной карте. Если карта найдена, то мы *рекурсивно* вызываем `typeSwitch()` на новой карте, чтобы изучить ее еще глубже.

Цикл `for` позволяет изучить все элементы карты `map[string]interface{}`.

```
func exploreMap(m map[string]interface{}) {
    for k, v := range m {
        embMap, ok := v.(map[string]interface{})
        // если это карта, то исследовать глубже
        if ok {
            fmt.Printf("{ \"%v\": \n", k)
            exploreMap(embMap)
            fmt.Printf("}\n")
        } else {
            fmt.Printf("%v: %v\n", k, v)
        }
    }
}
```

Функция `exploreMap()` проверяет содержимое своей входной карты. Если карта найдена, то мы *рекурсивно* вызываем `exploreMap()` на новой карте, чтобы изучить ее саму.

```
func main() {
    if len(os.Args) == 1 {
        fmt.Println("*** Using default JSON record.")
    } else {
        JSONrecord = os.Args[1]
    }

    JSONMap := make(map[string]interface{})
    err := json.Unmarshal([]byte(JSONrecord), &JSONMap)
```

Как вы узнаете из главы 6, `json.Unmarshal()` обрабатывает данные JSON и преобразует их в Go-значение. Хотя этим значением обычно служит Go-структура, в данном случае мы используем карту, заданную переменной `map[string]interface{}`. Строго говоря, второй параметр `json.Unmarshal()` имеет тип данных пустого интерфейса, то есть его тип данных может быть любым.

```
if err != nil {
    fmt.Println(err)
    return
}
exploreMap(JSONMap)
typeSwitch(JSONMap)
}
```



Карта `map[string]interface{}` чрезвычайно удобна для хранения записей JSON, их схема неизвестна заранее. Другими словами, `map[string]interface{}` хороша для хранения произвольных данных JSON с неизвестной схемой.

При выполнении `mapEmpty.go` мы получаем такой вывод:

```
$ go run mapEmpty.go
*** Using default JSON record.
Message: Hello Go!
Flag: true
Array: [a b c]
{"Entity":
Value: -456
Null: <nil>
a1: b1
a2: b2
}
Is a Boolean! Flag true
...Is Array: []interface {}!
Is a map! Entity map[Null:<nil> Value:-456 a1:b1 a2:b2]
Is a string! a2 b2
Is a float64! Value -456
```

```
...Is Null: <nil>!
Is a string! a1 b1
Is a string! Message Hello Go!
$ go run mapEmpty.go '{"Array": [3, 4], "Null": null, "String": "Hello
Go!"}'
Array: [3 4]
Null: <nil>
String: Hello Go!
...Is Array: []interface {}!
...Is Null: <nil>!
Is a string! String Hello Go!
$ go run mapEmpty.go '{"Array":"Error"'  
unexpected end of JSON input
```

Первый запуск выполняется без каких-либо параметров командной строки, что означает использование значения `JSONrecord` по умолчанию, и, следовательно, выводит жестко закодированные данные. В двух других выполнениях нужны пользовательские данные. Сначала действительные, а затем данные, которые не представляют собой действительную запись JSON. Сообщение об ошибке при третьем выполнении генерируется функцией `json.Unmarshal()`, поскольку она не может понять схему записи JSON.

Тип данных `error`

Как и было обещано, мы возвращаемся к типу данных `error`, поскольку это интерфейс, определенный следующим образом:

```
type error interface {
    Error() string
}
```

Итак, для того чтобы реализовать интерфейс `error`, вам просто нужно реализовать метод типа `Error() string`. Это не меняет того, как мы используем ошибки с целью выяснить, было ли выполнение функции или метода успешным, но показывает важность интерфейсов в Go, поскольку они прозрачно применяются практически всегда. Однако ключевой вопрос заключается в том, *когда именно* вам следует реализовать интерфейс `error` самостоятельно вместо использования интерфейса по умолчанию. Ответ будет таким: когда нужно придать больше контекста условию ошибки.

Теперь поговорим об интерфейсе `error` с более практической точки зрения. Когда из файла больше нечего читать, Go возвращает ошибку `io.EOF`, которая, строго говоря, является не ошибочным состоянием, а логической частью чтения файла. Если файл полностью пуст, то вы все равно получите `io.EOF` при попытке его прочитать. Однако в ряде ситуаций это может вызвать проблемы, и вам понадобится способ отличить полностью пустой файл от файла, который был прочитан полностью. Один из способов решения этой задачи — использование интерфейса `error`.



Приведенный здесь пример кода связан с файловым вводом/выводом. Размещение его здесь может вызвать некоторые вопросы касательно чтения файлов в Go. Однако я чувствую, что это подходящее место, так как оно связано с ошибками и их обработкой больше, чем с чтением файлов.

Код `errorInt.go` без блоков `package` и `import` выглядит следующим образом:

```
type emptyFile struct {
    Ended bool
    Read int
}
```

Это новый тип данных, который используется в программе.

```
// реализуем интерфейс error
func (e emptyFile) Error() string {
    return fmt.Sprintf("Ended with io.EOF (%t) but read (%d) bytes",
        e.Ended, e.Read)
}
```

А это реализация интерфейса `error` для `emptyFile`.

```
// проверяем значения
func isEmptyFile(e error) bool {
    // Утверждение типа
    v, ok := e.(emptyFile)
```

Это утверждение типа для получения структуры `emptyFile` из переменной `error`.

```
if ok {
    if v.Read == 0 && v.Ended == true {
        return true
    }
}
return false
}
```

Это метод проверки того, является ли файл пустым. Условие `if` можно перевести так: если вы прочитали 0 байт (`v.Read == 0`) и достигли конца файла (`v.Ended == true`), то файл пуст.

Если вы имеете дело с несколькими переменными `error`, то вам следует добавить переключатель типа в функцию `isEmptyFile()` после утверждения типа.

```
func readFile(file string) error {
    var err error
    fd, err := os.Open(file)
    if err != nil {
        return err
    }
    defer fd.Close()
```

```

reader := bufio.NewReader(fd)
n := 0
for {
    line, err := reader.ReadString('\n')
    n += len(line)
}

```

Мы читаем входной файл построчно. Больше информации о файловом вводе-выводе вы получите в главе 6.

```

if err == io.EOF {
    // End of File: читать больше нечего
    if n == 0 {
        return emptyFile{true, n}
    }
}

```

Если мы достигли конца файла (`io.EOF`) и прочитали 0 символов, то имеем дело с пустым файлом. Такого рода контекст добавляется в структуру `emptyFile` и возвращается как значение `error`.

```

        break
    } else if err != nil {
        return err
    }
}
return nil
}

func main() {
    flag.Parse()
    if len(flag.Args()) == 0 {
        fmt.Println("usage: errorInt <file1> [<file2> ...]")
        return
    }

    for _, file := range flag.Args() {
        err := readFile(file)
        if isEmpty(err) {
            fmt.Println(file, err)
}

```

Здесь мы проверяем сообщение об ошибке функции `ReadFile()`. Порядок выполнения проверки важен, поскольку срабатывает только первое совпадение. Это означает, что мы должны переходить от более конкретных к более общим условиям.

```

    } else if err != nil {
        fmt.Println(file, err)
    } else {
        fmt.Println(file, "is OK.")
    }
}

```

При выполнении `errorInt.go` мы получаем такой вывод:

```
$ go run errorInt.go /etc/hosts /tmp/doesNotExist /tmp/empty /tmp /tmp/
Empty.txt
/etc/hosts is OK.
/tmp/doesNotExist open /tmp/doesNotExist: no such file or directory
/tmp/empty open /tmp/empty: permission denied
/tmp read /tmp: is a directory
/tmp/Empty.txt Ended with io.EOF (true) but read (0) bytes
```

Первый файл (`/etc/hosts`) был прочитан без каких-либо проблем, в то время как второй (`/tmp/DoesNotExist`) найти не удалось. Третий файл (`/tmp/empty`) на месте, но у нас не было необходимых прав доступа к файлу для его чтения, тогда как четвертый файл (`/tmp`) оказался каталогом. Последний файл (`/tmp/Empty.txt`) существовал, но оказался пуст, что и было ошибочной ситуацией, которую мы старались отловить.

Написание собственных интерфейсов

Выяснив, как использовать существующие интерфейсы, мы напишем еще одну утилиту командной строки, которая сортирует 3D-фигуры в соответствии с их объемами. Эта задача требует изучения следующих задач:

- создание новых интерфейсов;
- объединение существующих интерфейсов;
- реализация `csort.Interface` для 3D-фигур.

Создавать собственные интерфейсы несложно. Для простоты мы включим наш интерфейс в пакет `main`. Однако так бывает достаточно редко, ведь, как правило, мы хотим делиться нашими интерфейсами, а это означает, что интерфейсы обычно включаются в Go-пакеты, отличные от `main`.

Следующий фрагмент кода определяет новый интерфейс:

```
type Shape2D interface {
    Perimeter() float64
}
```

Интерфейс обладает следующими свойствами:

- он называется `Shape2D`;
- он требует реализации единственного метода `Perimeter()`, который возвращает значение `float64`.

Помимо того что этот интерфейс определяется пользователем, в нем нет ничего особенного по сравнению со встроенным Go-интерфейсами. Вы можете

использовать его так же, как и все другие существующие интерфейсы. Итак, для того чтобы тип данных удовлетворял интерфейсу `Shape2D`, ему необходимо реализовать метод типа `Perimeter()`, который возвращает значение `float64`.

Использование Go-интерфейса

Приведенный ниже код представляет собой простейший способ использования интерфейса, который заключается в прямом вызове его метода и получении результата, как если бы это была функция. Хотя это и разрешено, но происходит достаточно редко, поскольку мы обычно создаем функции, которые принимают интерфейсы как параметры, чтобы работать с несколькими типами данных.

Код использует удобный метод для быстрого определения того, относится ли данная переменная к заданному типу данных, который был представлен ранее в `assertions.go`. В нашем случае мы проверяем, относится ли переменная к интерфейсу `Shape2D`, используя нотацию `interface{()}(a).(Shape2D)`, где `a` — проверяемая переменная, а `Shape2D` — тип данных для проверки переменной.

Следующая программа называется `Shape2D.go`, и ее наиболее любопытными частями являются такие:

```
type Shape2D interface {
    Perimeter() float64
}
```

Это определение интерфейса `Shape2D`, которое требует реализации метода типа `Perimeter()`.

```
type circle struct {
    R float64
}

func (c circle) Perimeter() float64 {
    return 2 * math.Pi * c.R
}
```

Здесь тип `circle` реализует интерфейс `Shape2D` путем реализации метода типа `Perimeter()`.

```
func main() {
    a := circle{R: 1.5}
    fmt.Printf("R %.2f -> Perimeter %.3f \n", a.R, a.Perimeter())

    _, ok := interface{()}(a).(Shape2D)
    if ok {
        fmt.Println("a is a Shape2D!")
    }
}
```

Как указывалось ранее, нотация `interface{}(a).(Shape2D)` проверяет, удовлетворяет ли переменная `a` интерфейсу `Shape2D`, не используя ее базовое значение (`circle{R: 1.5}`).

При выполнении `Shape2D.go` мы получаем такой вывод:

```
R 1.50 -> Perimeter 9.425
a is a Shape2D!
```

Реализация интерфейса `sort.Interface` для 3D-фигур

Здесь мы создадим утилиту для сортировки различных 3D-фигур на основе объема, которая наглядно демонстрирует эффективность и универсальность Go-интерфейсов. На этот раз для хранения всех видов структур, которые удовлетворяют заданному интерфейсу, мы будем использовать единственный срез. Тот факт, что Go рассматривает интерфейсы как типы данных, позволяет нам создавать срезы с элементами, удовлетворяющими заданному интерфейсу, без получения каких-либо сообщений об ошибках.

Такого рода сценарий может быть полезен во множестве случаев, поскольку демонстрирует, как хранить элементы с различными типами данных, которые удовлетворяют общему интерфейсу, в одном срезе и сортировать их с помощью `sort.Interface`. Проще говоря, представленная утилита сортирует различные структуры с разными номерами и названиями полей, имеющие общее поведение, которое реализовано с помощью интерфейса. Размеры фигур создаются с использованием случайных чисел, и это означает, что при каждом новом запуске утилиты вы получите разные выходные данные.

Интерфейс называется `Shape3D` и требует реализации метода типа `Vol() float64`. Этому интерфейсу удовлетворяют типы данных `Cube`, `Cuboid` и `Sphere`. Интерфейс `sort.Interface` реализован для типа данных `shapes`, который определяется как срез элементов `Shape3D`.

Все числа с плавающей запятой генерируются случайным образом путем использования функции `rF64(min, max float64) float64`. Поскольку такие числа содержат много десятичных знаков, вывод реализуется с помощью отдельной функции `PrintShapes()`, которая задействует оператор `fmt.Printf("%.2f ", v)` для указания количества десятичных знаков при выводе. В нашем случае мы выводим первые два знака после десятичной точки для каждого значения с плавающей запятой.



Как вы, возможно, помните, после реализации `sort.Interface` вы также можете сортировать свои данные в обратном порядке, используя `sort.Reverse()`.

Ведите следующий код в редакторе и сохраните как `sortShapes.go`. Код показывает, как сортировать 3D-фигуры на основе их объема.

```
package main

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

const min = 1
const max = 5

func rF64(min, max float64) float64 {
    return min + rand.Float64()*(max-min)
}
```

Функция `rF64()` генерирует случайные значения `float64`.

```
type Shape3D interface {
    Vol() float64
}
```

Это определение интерфейса `Shape3D`.

```
type Cube struct {
    x float64
}

type Cuboid struct {
    x float64
    y float64
    z float64
}

type Sphere struct {
    r float64
}

func (c Cube) Vol() float64 {
    return c.x * c.x * c.x
}
```

Это тип `Cube`, реализующий интерфейс `Shape3D`.

```
func (c Cuboid) Vol() float64 {
    return c.x * c.y * c.z
}
```

Это тип `Cuboid`, реализующий интерфейс `Shape3D`.

```
func (c Sphere) Vol() float64 {
    return 4 / 3 * math.Pi * c.r * c.r * c.r
}
```

А это тип `Sphere`, реализующий интерфейс `Shape3D`.

```
type shapes []Shape3D
```

Это тип данных, использующий `sort.Interface`.

```
// реализация sort.Interface
func (a shapes) Len() int {
    return len(a)
}

func (a shapes) Less(i, j int) bool {
    return a[i].Vol() < a[j].Vol()
}

func (a shapes) Swap(i, j int) {
    a[i], a[j] = a[j], a[i]
}
```

Три вышеприведенные функции реализуют `sort.Interface`.

```
func PrintShapes(a shapes) {
    for _, v := range a {
        switch v.(type) {
        case Cube:
            fmt.Printf("Cube: volume %.2f\n", v.Vol())
        case Cuboid:
            fmt.Printf("Cuboid: volume %.2f\n", v.Vol())
        case Sphere:
            fmt.Printf("Sphere: volume %.2f\n", v.Vol())
        default:
            fmt.Println("Unknown data type!")
        }
    }
    fmt.Println()
}

func main() {
    data := shapes{}
    rand.Seed(time.Now().Unix())
```

Функция `PrintShapes()` используется для настройки выходных данных.

```
for i := 0; i < 3; i++ {
    cube := Cube{rF64(min, max)}
```

```
cuboid := Cuboid{rF64(min, max), rF64(min, max), rF64(min, max)}
sphere := Sphere{rF64(min, max)}
data = append(data, cube)
data = append(data, cuboid)
data = append(data, sphere)
}
PrintShapes(data)

// сортировка
sort.Sort(shapes(data))
PrintShapes(data)

// обратная сортировка
sort.Sort(sort.Reverse(shapes(data)))
PrintShapes(data)

}
```

При выполнении `sortShapes.go` мы получаем такой вывод:

```
Cube: volume 105.27
Cuboid: volume 34.88
Sphere: volume 212.31
Cube: volume 55.76
Cuboid: volume 28.84
Sphere: volume 46.50
Cube: volume 52.41
Cuboid: volume 36.90
Sphere: volume 257.03
```

Это несортированный вывод программы:

```
Cuboid: volume 28.84
Cuboid: volume 34.88
Cuboid: volume 36.90
Sphere: volume 46.50
Cube: volume 52.41
...
Sphere: volume 257.03
```

Это отсортированный вывод программы от меньших форм к большим:

```
Sphere: volume 257.03
...
Cuboid: volume 28.84
```

Это результат обратной сортировки программы от больших фигур к меньшим.

В следующем разделе показан метод различия в ваших программах двух форматов файлов CSV.

Работа с двумя различными форматами файлов CSV

В этом разделе мы собираемся реализовать отдельную утилиту командной строки, которая работает с двумя различными форматами CSV. Причина, по которой мы это делаем, заключается в том, что бывают случаи, когда вам нужны свои утилиты для работы с несколькими форматами данных.

Помните, что записи каждого формата CSV хранятся с использованием собственной Go-структурой и под разными именами переменной. В результате нам нужно реализовать `sort.Interface` для обоих форматов CSV и, следовательно, для обеих переменных среза.

Поддерживаются следующие два формата:

- *формат 1* – имя, фамилия, номер телефона, время последнего доступа;
- *формат 2* – имя, фамилия, код города, номер телефона, время последнего доступа.

Поскольку два формата CSV имеют разное количество полей, утилита определяет используемый формат по количеству полей в первой прочитанной записи и дальше действует соответствующим образом. После этого данные сортируются с помощью `sort.Sort()` — тип данных среза, в котором хранятся данные, помогает Go определить используемую реализацию сортировки без какой-либо помощи со стороны разработчика.



Основное преимущество функций, работающих с переменными пустого интерфейса, заключается в том, что вы можете легко добавить поддержку дополнительных типов данных позже, не реализуя дополнительных функций и не нарушая существующего кода.

Далее следует реализация наиболее важных функций утилиты, начиная с `readCSVFile()`, поскольку логика утилиты находится в функции `readCSVFile()`.

```
func readCSVFile(filepath string) error {
    .
    .
    .
```

Код, который имеет отношение к чтению входного файла и проверке его существования, для краткости опущен.

```
var firstLine bool = true
var format1 = true
```

Первая строка CSV-файла определяет его формат. Следовательно, нам нужна переменная флага для указания того, имеем ли мы дело с первой строкой (`firstLine`). Кроме того, нам нужна вторая переменная для указания формата, с которым мы работаем (`format1` — как раз та переменная).

```
for _, line := range lines {
    if firstLine {
        if len(line) == 4 {
            format1 = true
        } else if len(line) == 5 {
            format1 = false
        }
    }
}
```

Первый формат содержит четыре поля, в то время как второй — пять.

```
} else {
    return errors.New("Unknown File Format!")
}
firstLine = false
}
```

Если первая строка CSV-файла не содержит ни четырех, ни пяти полей, то возникает ошибка и функция возвращается с пользовательским сообщением об ошибке.

```
if format1 {
    if len(line) == 4 {
        temp := F1{
            Name:      line[0],
            Surname:   line[1],
            Tel:       line[2],
            LastAccess: line[3],
        }
        d1 = append(d1, temp)
    }
}
```

Если мы работаем с `format1`, то добавляем данные в глобальную переменную `d1`.

```
} else {
    if len(line) == 5 {
        temp := F2{
            Name:      line[0],
            Surname:   line[1],
            Areacode:  line[2],
            Tel:       line[3],
            LastAccess: line[4],
        }
        d2 = append(d2, temp)
    }
}
```

Если мы работаем с `format2`, то добавляем данные в глобальную переменную `d2`.

```
    }
}
return nil
}
```

Функция `sortData()` в качестве параметра принимает пустой интерфейс. Код функции определяет тип данных среза, который передается в качестве пустого интерфейса этой функции путем использования переключателя типа. После этого утверждение типа позволяет нам применять фактические данные, хранящиеся в параметре пустого интерфейса. Его полная реализация выглядит так:

```
func sortData(data interface{}) {
    // переключатель типа
    switch T := data.(type) {
        case Book1:
            d := data.(Book1)
            sort.Sort(Book1(d))
            list(d)
        case Book2:
            d := data.(Book2)
            sort.Sort(Book2(d))
            list(d)
        default:
            fmt.Printf("Not supported type: %T\n", T)
    }
}
```

Переключатель типа выполняет работу по определению типа данных, который может быть либо `Book1`, либо `Book2`. Если вы хотите взглянуть на реализацию `sort.Interface`, то обратитесь к файлу исходного кода `sortCSV.go`.

Наконец, `list()` выводит данные использованной переменной `data`, применяя метод, определенный в `sortData()`. Хотя код, обрабатывающий `Book1` и `Book2`, такой же, как в `sortData()`, вам все равно придется применить утверждение типа, чтобы получить данные из переменной пустого интерфейса.

```
func list(d interface{}) {
    switch T := d.(type) {
        case Book1:
            data := d.(Book1)
            for _, v := range data {
                fmt.Println(v)
            }
        case Book2:
            data := d.(Book2)
            for _, v := range data {
```

```
        fmt.Println(v)
    }
default:
    fmt.Printf("Not supported type: %T\n", T)
}
}
```

При выполнении `sortCSV.go` мы получаем такой вывод:

```
$ go run sortCSV.go /tmp/csv.file
{Jane Doe 0800123456 1609310777}
{Dimitris Tsoukalos 2109416871 1609310731}
{Dimitris Tsoukalos 2109416971 1609310734}
{Mihalis Tsoukalos 2109416471 1609310706}
{Mihalis Tsoukalos 2109416571 1609310717}
```

Программа правильно определила формат `/tmp/csv.file` и работала именно с ним, несмотря на то что поддерживает два формата CSV. Попытка работать с неподдерживаемым форматом дает следующий результат:

```
$ go run sortCSV.go /tmp/differentFormat.csv
Unknown File Format!
```

Это значит, код успешно определил, что мы имеем дело с неподдерживаемым форматом.

В следующем разделе рассматриваются ограниченные объектно-ориентированные возможности Go.

Объектно-ориентированное программирование в Go

Поскольку Go поддерживает не все объектно-ориентированные функции, он не в силах полноценно заменить объектно-ориентированный язык программирования. Однако он может *имитировать некоторые объектно-ориентированные концепции*.

Во-первых, Go-структура с ее методами типа подобна объекту с его методами. Во-вторых, интерфейсы подобны абстрактным типам данных, которые определяют поведение и объекты одного и того же класса, что аналогично *полиморфизму*. В-третьих, Go поддерживает *инкапсуляцию*, то есть скрытие данных и функций от пользователя, делая их закрытыми для структуры и текущего Go-пакета. Наконец, объединение интерфейсов и структур подобно *композиции* в объектно-ориентированной терминологии.



Если в действительности вы хотите разрабатывать приложения, используя объектно-ориентированную методологию, то выбор Go может оказаться не лучшим вариантом. Я не слишком большой фанат Java, поэтому посоветовал бы вместо Go обратиться к C++ или Python. Общее правило здесь заключается в выборе наилучшего инструмента для вашей задачи.

Вы уже сталкивались с некоторыми из этих моментов ранее в этой главе, так что в следующей мы обсудим, как определять скрытые поля и функции. В приведенном ниже примере `obj0.go` показаны композиция и полиморфизм, а также встраивание анонимной структуры в существующую для получения всех ее полей.

```
package main

import (
    "fmt"
)

type IntA interface {
    foo()
}

type IntB interface {
    bar()
}

type IntC interface {
    IntA
    IntB
}
```

Интерфейс `IntC` объединяет интерфейсы `IntA` и `IntB`. Если вы реализуете `IntA` и `IntB` для типа данных, то этот тип данных неявно удовлетворяет `IntC`.

```
func processA(s IntA) {
    fmt.Printf("%T\n", s)
}
```

Эта функция работает с типами данных, которые удовлетворяют интерфейсу `IntA`.

```
type a struct {
    XX int
    YY int
}

// реализация IntA
func (varC c) foo() {
    fmt.Println("Foo Processing", varC)
}
```

Структура с удовлетворяет `IntA`, поскольку реализует `foo()`.

```
// реализация IntB B
func (varC c) bar() {
    fmt.Println("Bar Processing", varC)
}
```

Структура с удовлетворяет `IntB`. Поскольку структура с удовлетворяет как `IntA`, так и `IntB`, она неявно удовлетворяет `IntC`, который представляет собой композицию интерфейсов `IntA` и `IntB`.

```
type b struct {
    AA string
    XX int
}

// структура с имеет два поля
type c struct {
    A a
    B b
}
```

Структура содержит два поля `A` и `B`, которые относятся к типам данных `a` и `b` соответственно.

```
// структура compose получает поля структуры a
type compose struct {
    field1 int
    a
}
```

Эта новая структура использует анонимную структуру (`a`), то есть получает ее поля.

```
// разные структуры могут иметь методы с одинаковыми именами
func (A a) A() {
    fmt.Println("Function A() for A")
}

func (B b) A() {
    fmt.Println("Function A() for B")
}

func main() {
    var iC c = c{a{120, 12}, b{"-12", -12}}
```

Здесь мы определяем переменную `c`, которая состоит из структуры `a` и структуры `b`.

```
iC.A.A()
iC.B.A()
```

Здесь мы получаем доступ к методу структуры `a` (`A.A()`) и методу структуры `b` (`B.B()`).

```
// следующий код работать не будет
// iComp := compose{field1: 123, a{456, 789}}
// iComp := compose{field1: 123, XX: 456, YY: 789}
iComp := compose{123, a{456, 789}}
fmt.Println(iComp.XX, iComp.YY, iComp.field1)
```

При использовании анонимной структуры внутри другой (как мы делаем с `a{456, 789}`) вы можете получить доступ к полям анонимной структуры (которая является структурой `a{456, 789}`) напрямую как `iComp.XX` и `iComp.YY`.

```
iC.bar()
processA(iC)
}
```

Хотя `ProcessA()` работает с переменными `IntA`, он также может работать с переменными `IntC`, поскольку интерфейс `IntC` удовлетворяет `IntA`!

Весь код в `obj0.go` выглядит значительно проще по сравнению с кодом реального объектно-ориентированного языка программирования, который поддерживает абстрактные классы и наследование. Однако этого более чем достаточно для генерации типов и элементов со структурой в них, а также для использования разных типов данных с одинаковыми именами методов.

При выполнении `obj0.go` мы получаем такой вывод:

```
$ go run obj0.go
Function A() for A
Function A() for B
456 789 123
Bar Processing {{120 12} {-12 -12}}
main.c
```

Первые две строки выходных данных показывают, что две разные структуры могут иметь метод с одинаковым именем. Третья строка показывает, что при использовании анонимной структуры внутри одной другой структуры вы можете получить прямой доступ к полям анонимной структуры. Четвертая строка — это выходные данные вызова `iC.bar()`, где `iC` — переменная `c`, обращающаяся к методу из интерфейса `IntB`. Последняя строка — это вывод `ProcessA(iC)`, который требует параметра `IntA` и выводит реальный тип данных своего параметра, который в данном случае является `main.c`.

Очевидно, что, хотя Go и не является объектно-ориентированным языком программирования, он может имитировать некоторые особенности этого вида программирования. Переходим к последнему разделу главы, посвященному тому, как обновить приложение телефонной книги путем добавления возможности чтения переменной среды и сортировки выходных данных.

Обновление приложения телефонной книги

Функциональность, добавленная в эту новую версию утилиты телефонной книги, заключается в следующем:

- путь к файлу CSV может быть дополнительно указан в качестве переменной среды PHONEBOOK;
- команда `list` сортирует выходные данные на основе поля `surname`.

Мы могли бы указывать путь к файлу CSV в качестве аргумента командной строки вместо значения переменной среды, но это усложнило бы код, особенно если бы данный аргумент был необязательным. Более расширенные Go-пакеты, такие как `viper`, представленный в главе 6, упрощают процесс анализа аргументов командной строки, использующий параметры командной строки, такие как `-f` (за которыми следует путь к файлу) или `--filepath`.



Текущее значение по умолчанию для `CSVFILE` указывает на мой домашний каталог на компьютере с macOS Big Sur. Вам же следует изменить это значение по умолчанию в соответствии с вашими потребностями или использовать правильное значение для переменной среды `PHONEBOOK`.

Наконец, если переменная среды `PHONEBOOK` не задана, то утилита для пути к файлу CSV использует значение по умолчанию. Вообще говоря, отсутствие необходимости перекомпилировать ваше программное обеспечение для пользовательских данных считается хорошей практикой.

Настройка значения CSV-файла

Значение CSV-файла задается в функции `setCSVFILE()`, которая определяется следующим образом:

```
func setCSVFILE() error {
    filepath := os.Getenv("PHONEBOOK")
    if filepath != "" {
        CSVFILE = filepath
    }
}
```

Здесь мы читаем переменную среды `PHONEBOOK`. Остальная часть кода позволяет убедиться, что мы можем использовать этот путь к файлу или путь по умолчанию, если `PHONEBOOK` не задана.

```
_ , err := os.Stat(CSVFILE)
if err != nil {
    fmt.Println("Creating", CSVFILE)
```

```

f, err := os.Create(CSVFILE)
if err != nil {
    f.Close()
    return err
}
f.Close()
}

```

Если указанный файл не существует, то создается с помощью `os.Create()`.

```

fileInfo, err := os.Stat(CSVFILE)
mode := fileInfo.Mode()
if !mode.IsRegular() {
    return fmt.Errorf("%s not a regular file", CSVFILE)
}

```

Далее мы убеждаемся, что указанный путь принадлежит обычному файлу, который можно использовать для сохранения данных.

```

return nil
}

```

Чтобы упростить реализацию функции `main()`, мы перенесли код, связанный с существованием пути к CSV-файлу и доступом к нему, в `setCSVFILE()`.

В первый раз, установив переменную среды `PHONEBOOK` и запустив приложение телефонной книги, мы получили следующий результат (вы должны получить что-то подобное):

```

$ export PHONEBOOK="/tmp/csv.file"
$ go run phoneBook.go list
Creating /tmp/csv.file

```

Поскольку файл `/tmp/csv.file` не существует, `phoneBook.go` создает его с нуля. Здесь проверяется, что Go-код функции `setCSVFILE()` работает должным образом.

Теперь, когда мы выяснили, откуда получать и куда записывать наши данные, пришло время научиться сортировать их с помощью `sort.Interface`, что и будет предметом обсуждения в следующем подразделе.

Использование пакета `sort`

Первое, что нужно решить, пытаясь отсортировать данные, — какое поле будет использоваться для сортировки. Затем нужно понять, что мы будем делать с двумя или более записями с одинаковым значением в основном поле, используемом для сортировки.

Это код, связанный с сортировкой при помощи `sort.Interface`:

```
type PhoneBook []Entry
```

Вам нужно иметь отдельный тип данных, для которого реализован `sort.Interface`.

```
var data = PhoneBook{}
```

Поскольку у вас имеется отдельный тип данных для реализации `sort.Interface`, тип данных переменной `data` должен быть изменен на `PhoneBook`. Затем `sort.Interface` реализуется для `PhoneBook`.

```
// реализуем sort.Interface
func (a PhoneBook) Len() int {
    return len(a)
}
```

Функция `Len()` имеет стандартную реализацию.

```
// Первый на основе фамилии. Если у них одинаковая
// фамилия, то учитывать имя
func (a PhoneBook) Less(i, j int) bool {
    if a[i].Surname == a[j].Surname {
        return a[i].Name < a[j].Name
    }
}
```

Функция `Less()` — это место для определения того, как вы собираетесь сортировать элементы среза. Здесь мы говорим, что если сравниваемые записи (Go-структуры) имеют одинаковое значение поля `Surname`, то сравнивать эти записи нужно, используя значения поля `Name`.

```
return a[i].Surname < a[j].Surname
}
```

Если записи имеют разные значения в поле `Surname`, то сравнивать их нужно, используя поле `Surname`.

```
func (a PhoneBook) Swap(i, j int) {
    a[i], a[j] = a[j], a[i]
}
```

Функция `Swap()` имеет стандартную реализацию. После реализации желаемого интерфейса нам нужно указать нашему коду сортировать данные, что происходит в реализации функции `list()`:

```
func list() {
    sort.Sort(PhoneBook(data))
    for _, v := range data {
        fmt.Println(v)
    }
}
```

Теперь, когда мы знаем, как реализована сортировка, пришло время воспользоваться утилитой. Сначала добавим несколько записей:

```
$ go run phoneBook.go insert Mihalis Tsoukalos 2109416471
$ go run phoneBook.go insert Mihalis Tsoukalos 2109416571
$ go run phoneBook.go insert Dimitris Tsoukalos 2109416871
$ go run phoneBook.go insert Dimitris Tsoukalos 2109416971
$ go run phoneBook.go insert Jane Doe 0800123456
```

Наконец, мы выводим содержимое телефонной книги с помощью команды `list`:

```
$ go run phoneBook.go list
{Jane Doe 0800123456 1609310777}
{Dimitris Tsoukalos 2109416871 1609310731}
{Dimitris Tsoukalos 2109416971 1609310734}
{Mihalis Tsoukalos 2109416471 1609310706}
{Mihalis Tsoukalos 2109416571 1609310717}
```

Поскольку по алфавиту `Dimitris` следует перед `Mihalis`, все соответствующие записи также показаны первыми, что означает должную работу нашей сортировки.

Упражнения

- Создайте срез структур, используя созданную вами структуру, и отсортируйте элементы среза с помощью ее поля.
- Интегрируйте функциональность `sortCSV.go` в `phonebook.go`.
- Добавьте поддержку команды `reverse` в `phonebook.go`, чтобы выводить ее записи в обратном порядке.
- Используйте пустой интерфейс и функцию, которая позволяет различать две разные создаваемые структуры.

Резюме

В этой главе вы узнали об *интерфейсах*, которые похожи на контракты, а также о *методах типа*, утверждении типов и рефлексии. Хотя рефлексия и является очень эффективной функцией Go, она может замедлить работу ваших программ Go, поскольку добавляет уровень сложности во время выполнения. Кроме того, при небрежном использовании рефлексии ваши Go-программы могут аварийно завершаться.

В последнем разделе этой главы мы обсуждали написание кода Go, который следует принципам объектно-ориентированного программирования. Если бы

вам нужно было запомнить из главы лишь одну вещь, то это должен быть тот факт, что Go не является объектно-ориентированным языком программирования, но может имитировать некоторые функциональные возможности, предлагаемые языками ООП, такими как Java, Python и C++.

В следующей главе мы рассмотрим Go-пакеты, функции и автоматизацию с использованием систем GitHub и GitLab CI/CD.

Дополнительные ресурсы

- Документация по пакету `reflect`: <https://golang.org/pkg/reflect/>.
- Документация по пакету `sort`: <https://golang.org/pkg/sort/>.
- Работа с ошибками в Go 1.13: <https://blog.golang.org/go1.13-errors>.
- Реализация пакета `sort`: <https://golang.org/src/sort/>.

5

Пакеты и функции Go

Основное внимание в этой главе уделяется *пакетам* Go, которые представляют собой способ организации, поставки и использования кода Go. Наиболее распространенный компонент пакетов — это *функции*, которые могут быть довольно гибкими и эффективными и используются для обработки данных и манипулирования ими. Go также поддерживает *модули*, которые представляют собой пакеты с номерами версий. В этой главе также будет объяснена работа ключевого слова `defer`, используемого для очистки и высвобождения ресурсов.

Что касается видимости элементов пакета, то Go следует простому правилу, которое гласит, что функции, переменные, типы данных, поля структуры и т. д., начинающиеся с заглавной буквы, *общедоступны* (public), тогда как начинающиеся со строчной буквы являются *закрытыми* (private). По этой причине `fmt.Println()` называется `Println()` вместо просто `println()`. То же правило применимо не только к названию переменной `struct`, но и к ее полям. На практике это означает, что у вас может быть `struct` как с закрытыми, так и с общедоступными полями. Однако это правило не влияет на имена пакетов, которые могут начинаться с любой буквы.

В этой главе представлено краткое описание следующих тем:

- Go-пакеты;
- функции;
- разработка собственных пакетов;
- использование GitHub для хранения Go-пакетов;
- пакет для работы с базой данных;
- модули;

- создание более качественных пакетов;
- создание документации;
- GitLab Runners и Go;
- GitHub Actions и Go;
- утилиты управления версиями.

Go-пакеты

Все в Go предоставляется в виде пакетов. Go-пакет — это исходный код Go, начинающийся с ключевого слова `package`, за которым следует имя пакета.



Обратите внимание, что пакеты могут иметь собственную структуру. Например, пакет `net` имеет несколько подкаталогов `http`, `mail`, `rpc`, `smtp`, `textproto` и `url`, которые следует импортировать как `net/http`, `net/mail`, `net/rpc`, `net/smtp`, `net/textproto` и `net/url` соответственно.

Помимо пакетов стандартной библиотеки Go, существуют внешние пакеты, которые можно импортировать, используя их полный адрес, и которые следует загружать на локальный компьютер перед первым использованием. Одним из таких примеров служит <https://github.com/spf13/cobra>, который хранится в GitHub.

Пакеты в основном используются для группировки связанных функций, переменных и констант, чтобы их можно было легко передавать и использовать в собственных программах Go. Обратите внимание, что, за исключением пакета `main`, Go-пакеты не являются автономными программами и не могут быть скомпилированы в самостоятельные исполняемые файлы. В результате, если вы попытаетесь выполнить Go-пакет так, как будто это автономная программа, вас ждет неудача:

```
$ go run aPackage.go
go run: cannot run non-main package
```

Вместо этого пакеты должны вызываться прямо или косвенно из пакета `main`. Тогда их можно будет использовать так, как было показано в предыдущих главах.

Скачивание Go-пакетов

В этом подразделе вы узнаете, как скачивать внешние Go-пакеты на примере <https://github.com/spf13/cobra>. Команда `go get` для загрузки пакета `cobra` выглядит следующим образом:

```
$ go get github.com/spf13/cobra
```

Обратите внимание, что вы можете загрузить пакет, не используя в его адресе `https://`. Результаты можно найти в каталоге `~/go`. Полный путь выглядит так: `~/go/src/github.com/spf13cobra`. Поскольку пакет `cobra` поставляется с двоичным файлом, который помогает вам структурировать и создавать утилиты командной строки, вы можете найти этот двоичный файл внутри `~/go/bin` под именем `cobra`.

Следующий вывод, который был создан с помощью утилиты `tree(1)`, показывает высокоуровневый вид с тремя уровнями детализации структуры `~/go` на моей машине:

```
$ tree ~/go -L 3
/Users/mtsouk/go
├── bin
│   ├── cobra
│   ├── go-outline
│   ├── gocode
│   ├── gocode-gomod
│   ├── godef
│   ├── golint
│   ├── gopkgs
│   └── goreturns
└── pkg
    ├── darwin_amd64
    │   ├── github.com
    │   ├── golang.org
    │   ├── gonum.org
    │   └── google.golang.org
    ├── mod
    │   ├── 9fans.net
    │   ├── cache
    │   ├── cloud.google.com
    │   ├── github.com
    │   ├── go.opencensus.io@v0.22.4
    │   ├── golang.org
    │   └── google.golang.org
    └── sumdb
        └── sum.golang.org
└── src
    ├── github.com
    │   ├── sirupsen
    │   └── spf13
    └── golang.org
        └── x

23 directories, 8 files
```



Путь `x`, который отображается последним, используется командой `Go`.

В принципе в `~/go` имеются три основных каталога с определенными свойствами.

- В каталоге `bin` размещаются двоичные инструменты.
- В каталог `pkg` помещаются повторно используемые пакеты. Каталог `darwin_amd64`, который можно найти только на компьютерах macOS, содержит скомпилированные версии установленных пакетов. На компьютере с Linux вместо `darwin_amd64` вы можете обнаружить `linux_amd64`.
- В каталоге `src` находится исходный код пакетов. Базовая структура основана на URL пакета, который вы ищете. Таким образом, URL для пакета `github.com/spf13/viper` — это `~/go/src/github.com/spf13/viper`. Если пакет загружен как модуль, то он будет расположжен в разделе `~/go/pkg/mod`.



Начиная с версии Go 1.16 `go install` является рекомендуемым способом сборки и установки пакетов в модульном режиме. Использование `go get` — устаревший метод, но в этой главе мы будем его использовать, поскольку он широко применяется в Интернете и о нем стоит знать. Однако в большинстве глав данной книги используются `go mod init` и `go mod tidy` для скачивания внешних зависимостей для ваших собственных файлов с исходным кодом.

Если требуется обновить существующий пакет, то вам следует выполнить `go get` с параметром `-u`. Кроме того, если вы хотите увидеть, что происходит «под капотом», то добавьте параметр `-v` в команду `go get` — в нашем случае мы используем пакет `viper` в качестве примера, но сократим выходные данные:

```
$ go get -v github.com/spf13/viper
github.com/spf13/viper (download)
...
github.com/spf13/afero (download)
get "golang.org/x/text/transform": found meta tag get.
metaImport{Prefix:"golang.org/x/text", VCS:"git", RepoRoot:"https://
go.googlesource.com/text"} at //golang.org/x/text/transform?go-get=1
get "golang.org/x/text/transform": verifying non-authoritative meta tag
...
github.com/fsnotify/fsnotify
github.com/spf13/viper
```

То, что вы в основном видите в выходных данных, — зависимости исходного пакета, скачиваемые до нужного пакета. В большинстве случаев это лишняя информация.

Далее мы рассмотрим наиболее важный элемент пакетов: *функции*.

Функции

Основными элементами пакетов служат функции, которые и выступают предметом обсуждения в этом разделе.



Методы и функции типов реализуются одинаковым образом, и иногда эти термины используются взаимозаменяюще.

Небольшой совет: функции должны быть настолько независимыми друг от друга, насколько это возможно, и должны хорошо выполнять одну (и только одну) задачу. Следовательно, если вы пишете функции, которые выполняют несколько задач, то вам стоило бы рассмотреть возможность замены их несколькими функциями.

Вы уже должны знать, что все определения функций начинаются с ключевого слова `func`, за которым следует сигнатура функции и ее реализация. Кроме того, функции могут не принимать ни одного аргумента или же принимать один или несколько аргументов, а также не возвращать ни одного значения или возвращать одно или несколько. Наиболее используемой Go-функцией является `main()`, которая включается в каждую исполняемую Go-программу. Она не принимает параметры и ничего не возвращает, но служит начальной точкой каждой программы Go. Кроме того, при завершении функции `main()` завершается и вся программа.

Анонимные функции

Анонимные функции можно встраивать, не указывая имени, и обычно они используются для реализации задач, требующих небольшого объема кода. В Go функция может возвращать или принимать анонимную функцию в качестве одного из аргументов. Кроме того, анонимные функции могут быть присоединены к Go-переменным. Обратите внимание, что в терминологии функционального программирования анонимные функции называются *лямбдами*. Аналогично этому, *замыкание* — особый тип анонимной функции, которая переносит или *замыкает* переменные, находящиеся в той же лексической области, что и определяемая анонимная функция.

Считается хорошей практикой, когда анонимные функции имеют небольшую реализацию и локальную направленность. Если анонимная функция не направлена на решение локальной задачи, то вам, вероятно, стоит рассмотреть возможность превращения ее в обычную. Когда анонимная функция подходит для нужной задачи, это чрезвычайно облегчает жизнь; просто не используйте слишком много анонимных функций в своих программах без веской на то причины. Через некоторое время мы рассмотрим анонимные функции в действии.

Функции, возвращающие несколько значений

Как вы уже знаете, такие функции, как `strconv.Atoi()`, могут возвращать несколько различных значений, что избавляет от необходимости создавать специальную структуру для возврата и получения из функции нескольких значений. Однако если у вас имеется функция, которая возвращает более трех значений, то вам следует еще раз обдумать это решение. Возможно, стоит перепроектировать ее, чтобы использовать одну структуру или срез для группировки и возврата желаемых значений в виде единого объекта. Это упростит обработку возвращаемых значений. Функции, анонимные функции и функции, возвращающие несколько значений, показаны в файле `functions.go`:

```
package main

import "fmt"

func doubleSquare(x int) (int, int) {
    return x * 2, x * x
}
```

Эта функция возвращает два значения `int`, не прибегая к необходимости содержать отдельные переменные для их хранения, — возвращаемые значения создаются динамически. Обратите внимание на обязательное использование *круглых скобок*, когда функция возвращает несколько значений.

```
// сортировка от меньшего значения к большему
func sortTwo(x, y int) (int, int) {
    if x > y {
        return y, x
    }
    return x, y
}
```

Вышеприведенная функция также возвращает два значения `int`.

```
func main() {
    n := 10
    d, s := doubleSquare(n)
```

Вышеприведенный оператор считывает два возвращаемых значения `doubleSquare()` и сохраняет их в `d` и `s`.

```
fmt.Println("Double of", n, "is", d)
fmt.Println("Square of", n, "is", s)

// анонимная функция
anF := func(param int) int {
    return param * param
}
```

Переменная `anF` содержит *анонимную функцию*, которая требует одного параметра в качестве входных данных и возвращает одно значение. Единственная разница между анонимной и регулярной функциями заключается в том, что имя анонимной функции — `func()` и мы не используем ключевое слово `func`.

```
fmt.Println("anF of", n, "is", anF(n))

fmt.Println(sortTwo(1, -3))
fmt.Println(sortTwo(-1, 0))
}
```

Последние два оператора выводят возвращаемые значения `sortTwo()`. При выполнении `functions.go` мы получаем такой вывод:

```
Double of 10 is 20
Square of 10 is 100
anF of 10 is 100
-3 1
-1 0
```

В следующем подразделе показаны функции, которые используют именованные возвращаемые значения.

Возвращаемые значения функции могут иметь имя

В отличие от C, Go позволяет называть возвращаемые значения Go-функции. Кроме того, когда такая функция имеет оператор `return` без каких-либо аргументов, она автоматически возвращает текущее значение каждого именованного возвращаемого значения в том порядке, в котором они были объявлены в сигнатуре.

Следующая функция включена в `namedReturn.go`:

```
func minMax(x, y int) (min, max int) {
    if x > y {
        min = y
        max = x
        return min, max
    }
}
```

Данный оператор `return` возвращает значения, сохраненные в переменных `min` и `max`. Обе переменные определены в *сигнатуре*, а не в теле функции.

```
min = x
max = y
return
```

Данный оператор `return` эквивалентен `return min, max`. В его основе — сигнатура функции и использование именованных возвращаемых значений.

При выполнении `namedReturn.go` мы получаем такой вывод:

```
$ go run namedReturn.go 1 -2
-2 1
-2 1
```

Функции, которые принимают другие функции в качестве параметров

Функции могут принимать в качестве параметров другие функции. Лучший пример функции, которая принимает другую функцию в качестве аргумента, можно обнаружить в пакете `sort`. В качестве аргумента вы можете предоставить функции `sort.Slice()` другую функцию, которая определяет способ реализации сортировки. Сигнатурой `sort.Slice()` служит `func Slice(slice interface{}, less func(i, j int) bool)`. Это означает следующее:

- функция `sort.Slice()` не возвращает никаких данных;
- функция `sort.Slice()` требует двух аргументов, среза типа `interface{}` и другой функции, при этом переменная среза изменяется внутри `sort.Slice()`;
- функциональный параметр `sort.Slice()` имеет имя `less` и должен обладать сигнатурой `func(i, j int) bool` — вам не нужно называть анонимную функцию. Имя `less` здесь требуется, поскольку все параметры функции должны иметь имя;
- параметры `less` (`i` и `j`) являются индексами параметра `slice`.

Аналогично в пакете `sort` содержится еще одна функция `sort.SliceIsSorted()`, которая определяется как `func SliceIsSorted(slice interface{}, less func(i, j int) bool) bool`. Функция `sort.SliceIsSorted()` возвращает `bool` и проверяет, отсортирован ли параметр `slice` в соответствии с правилами второго параметра, который представляет собой функцию.



Вы не обязаны применять анонимную функцию ни в одном из случаев использования `sort.Slice()` или `sort.SliceIsSorted()`. Вы можете определить обычную функцию с требуемой сигнатурой и работать с ней. Однако использовать анонимную функцию более удобно.

Использование как `sort.Slice()`, так и `sort.SliceIsSorted()` показано в следующей Go-программе (исходный файл `sorting.go`):

```
package main

import (
```

```

        "fmt"
        "sort"
    )

type Grades struct {
    Name     string
    Surname string
    Grade   int
}

func main() {
    data := []Grades{{"J.", "Lewis", 10}, {"M.", "Tsoukalos", 7},
                    {"D.", "Tsoukalos", 8}, {"J.", "Lewis", 9}}

    isSorted := sort.SliceIsSorted(data, func(i, j int) bool {
        return data[i].Grade < data[j].Grade
    })
}

```

Следующий блок `if else` проверяет значение `bool sort.SliceIsSorted()`, чтобы определить, отсортирован ли срез:

```

if isSorted {
    fmt.Println("It is sorted!")
} else {
    fmt.Println("It is NOT sorted!")
}

sort.Slice(data,
    func(i, j int) bool { return data[i].Grade < data[j].Grade })
fmt.Println("By Grade:", data)
}

```

Вызов `sort.Slice()` сортирует данные в соответствии с анонимной функцией, которая передается в качестве второго аргумента `sort.Slice()`.

При выполнении `sorting.go` мы получаем такой вывод:

```

It is NOT sorted!
By Grade: [{M. Tsoukalos 7} {D. Tsoukalos 8} {J. Lewis 9} {J. Lewis 10}]

```

Функции могут возвращать другие функции

Помимо принятия функций в качестве аргументов, функции также могут возвращать анонимные функции, что может пригодиться, когда возвращаемая функция не всегда одинакова, а зависит от входных данных или других внешних параметров. Это показано в файле `returnFunction.go`:

```

package main

import "fmt"

```

```

func funRet(i int) func(int) int {
    if i < 0 {
        return func(k int) int {
            k = -k
            return k + k
        }
    }

    return func(k int) int {
        return k * k
    }
}

```

В сигнатуре `funRet()` объявляется, что функция возвращает другую функцию с сигнатурой `func(int) int`. Реализация функции неизвестна, но будет определена во время выполнения. Функции возвращаются с помощью ключевого слова `return`. Разработчик должен сам позаботиться о сохранении возвращаемой функции.

```

func main() {
    n := 10
    i := funRet(n)
    j := funRet(-4)
}

```

Обратите внимание, что `n` и `-4` используются только для определения анонимных функций, которые будут возвращены из `funRet()`.

```

fmt.Printf("%T\n", i)
fmt.Printf("%T %v\n", j, j)
fmt.Println("j", j, j(-5))

```

Первый оператор выводит сигнатуру функции, тогда как второй — сигнатуру функции и ее адрес в памяти. Последний оператор также возвращает адрес памяти `j`, поскольку `j` является указателем на анонимную функцию, и значение `j(-5)`.

```

// тот же входной параметр, но РАЗНЫЕ
// анонимные функции, назначенные i и j
fmt.Println(i(10))
fmt.Println(j(10))
}

```

Хотя и `i`, и `j` вызываются с одним и тем же вводом (`10`), они будут возвращать разные значения, поскольку хранят различающиеся анонимные функции.

При выполнении `returnFunction.go` мы получаем такой вывод:

```

func(int) int
func(int) int 0x10a8d40
j 0x10a8d40 10
100
-20

```

Первая строка выходных данных показывает тип данных переменной `i`, содержащей возвращаемое значение `funRet(n)`, которое представляет собой `func(int) int`, поскольку содержит функцию. Вторая строка вывода показывает тип данных `j`, а также адрес в памяти, где хранится анонимная функция. Третья строка показывает адрес анонимной функции, хранящийся в переменной `j`, а также возвращаемое значение `j(-5)`. Последние две строки — это возвращаемые значения `i(10)` и `j(10)` соответственно.

Итак, в этом подразделе мы узнали о функциях, возвращающих функции. Это делает Go функциональным языком программирования (хотя и не чисто функциональным), что позволяет ему использовать парадигму функционального программирования.

Теперь обратимся к функциям с переменным количеством параметров.

Функции с переменным количеством параметров

Существуют функции, которые могут принимать переменное количество параметров, и вы уже знаете о двух таких широко используемых функциях: `fmt.Println()` и `append()`. Фактически большинство функций в пакете `fmt` являются функциями с переменным количеством параметров.

Общие идеи и правила, лежащие в основе таких функций, заключаются в следующем.

- Функции с переменным количеством параметров используют *операцию упаковки*, которая состоит из `...`, после чего следует тип данных. Итак, чтобы функция принимала переменное количество значений `int`, операция упаковки должна представлять собой `...int`.
- В любой заданной функции операцию упаковки можно использовать только один раз.
- Переменная, содержащая операцию упаковки, представляет собой срез `i`, следовательно, доступна как срез внутри функции с переменным количеством параметров.
- Имя переменной, связанное с операцией упаковки, всегда находится на последнем месте в списке параметров функции.
- При вызове функции вы должны поместить список значений, разделенных символом запятой, вместо переменной с помощью *операции упаковки* или срез с помощью *операции распаковки*.

Список содержит все правила, которые необходимо знать, чтобы определять и использовать функции с переменным количеством параметров.

Операция упаковки также может использоваться и с пустым интерфейсом. Фактически большинство функций в пакете `fmt` используют `...interface{}` для приема переменного количества аргументов всех типов данных. Вы можете найти исходный код последней реализации `fmt` по адресу <https://golang.org/src/fmt/>.

Однако возможна ситуация, требующая особого внимания (я допустил эту ошибку, когда учился Go). Если вы попытаетесь передать `os.Args`, которые представляют собой срез строк (`[]string`), как `...interface{}` в функцию с переменным количеством параметров, то ваш код не будет компилироваться, а вы получите примерно такое сообщение об ошибке: `cannot use os.Args (type []string) as type []interface{} in argument to <function_name>`. Это происходит потому, что два типа данных (`[]string` и `[]interface{}`) имеют разные представления в памяти — это относится ко всем типам данных. На практике это означает, что вы не можете написать `os.Args...`, чтобы передать каждое отдельное значение среза `os.Args` в функцию с переменным количеством параметров.

В то же время, если вы просто используете `os.Args`, это сработает, но срез будет передан как *единое целое*, а не в виде отдельных значений! Это означает, что оператор `everything(os.Args, os.Args)` сработает, но не сделает того, что вы хотите.

Решением проблемы будет преобразование среза строк (или любого другого среза) в срез `interface{}`. Один из способов проделать это — использовать следующий код:

```
empty := make([]interface{}, len(os.Args[1:]))
for i, v := range os.Args {
    empty[i] = v
}
```

Теперь вам можно использовать `empty...` в качестве аргумента функции с переменным количеством параметров. Это единственный нюанс, связанный с такими функциями и операцией упаковки.



Поскольку не существует стандартной библиотечной функции для выполнения подобного преобразования, вам придется написать собственный код. Обратите внимание, что такое преобразование требует времени, поскольку код должен пройти все элементы среза. Чем больше элементов содержится в срезе, тем больше времени займет преобразование. Эта тема также обсуждается здесь: <https://github.com/golang/go/wiki/InterfaceSlice>.

Теперь мы готовы увидеть действие функций с переменным количеством параметров. Введите следующий Go-код с помощью любого текстового редактора и сохраните как `variadic.go`:

```
package main

import (
    "fmt"
    "os"
)
```

Поскольку функции с переменным количеством параметров встроены в грамматику языка, вам не придется ничего дописывать для их поддержки.

```
func addFloats(message string, s ...float64) float64 {
```

Это функция, которая принимает `string` и некоторое количество значений `float64`. Она выводит переменную типа `string` и вычисляет сумму значений `float64`.

```
    fmt.Println(message)
    sum := float64(0)
    for _, a := range s {
        sum = sum + a
    }
```

Цикл `for` обращается к операции упаковки как к срезу, так что здесь нет ничего особенного.

```
    s[0] = -1000
    return sum
}
```

Вы также можете получить доступ к отдельным элементам среза `s`.

```
func everything(input ...interface{}) {
    fmt.Println(input)
}
```

Это еще одна функция, которая принимает неизвестное количество значений `interface{}`.

```
func main() {
    sum := addFloats("Adding numbers...", 1.1, 2.12, 3.14, 4, 5, -1, 10)
```

Вы можете встраивать аргументы функции.

```
    fmt.Println("Sum:", sum)
    s := []float64{1.1, 2.12, 3.14}
```

Но обычно переменная среза используется с операцией распаковки.

```
sum = addFloats("Adding numbers...", s...)
fmt.Println("Sum:", sum)
everything(s)
```

Вышеприведенный код работает, поскольку содержимое `s` не распаковано.

```
// не удается напрямую передать []string как []interface{}
// Сначала необходимо преобразовать!
empty := make([]interface{}, len(os.Args[1:]))
```

Вы можете преобразовать `[]string` в `[]interface{}`, чтобы использовать операцию распаковки.

```
for i, v := range os.Args[1:] {
    empty[i] = v
}
everything(empty...)
```

А теперь мы можем распаковать содержимое `empty`.

```
arguments := os.Args[1:]
empty = make([]interface{}, len(arguments))
for i := range arguments {
    empty[i] = arguments[i]
}
```

Это немного другой способ преобразования `[]string` в `[]interface{}`.

```
everything(empty...)
// Это работает!
str := []string{"One", "Two", "Three"}
everything(str, str, str)
}
```

Вышеприведенный оператор работает, поскольку вы трижды передаете всю переменную `str`, а не ее содержимое. Итак, срез содержит три элемента, и каждый из них равен содержимому переменной `str`.

При выполнении `variadic.go` мы получаем такой вывод:

```
$ go run variadic.go
Adding numbers...
Sum: 24.36
Adding numbers...
Sum: 6.36
[[-1000 2.12 3.14]]
[]
[]
[[One Two Three] [One Two Three] [One Two Three]]
```

Последняя строка выходных данных показывает, что мы трижды передавали переменную `str` функции `everything()` в виде трех отдельных объектов.

Функции с переменным количеством параметров очень удобны, когда функция должна принять неизвестное количество параметров. В следующем подразделе мы обсудим использование ключевого слова `defer` (которое уже использовали несколько раз).

Ключевое слово `defer`

Ранее мы встречали `defer` в `ch03/csvData.go`, а также в реализациях приложения телефонной книги. Но что именно делает это ключевое слово? Оно откладывает выполнение функции до тех пор, пока не вернется окружающая функция.

Обычно `defer` используется в операциях ввода-вывода файлов, чтобы сохранить вызов функции, закрывающий открытый файл, рядом с вызовом, который его открыл. Так что вам не придется помнить о закрытии файла, который вы открыли непосредственно перед завершением работы функции.

Очень важно помнить: *отложенные функции* выполняются в порядке «*последний вошел, первый вышел*» (last in, first out, LIFO) после того, как была возвращена окружающая функция. Проще говоря, это означает, что если вы отложите (`defer`) функцию `f1()` первой, функцию `f2()` второй и функцию `f3()` третьей в одной и той же окружающей функции, то перед самым возвратом окружающей функции функция `f3()` будет выполнена первой, функция `f2()` — второй, а функция `f1()` — последней.

В этом подразделе на примере простой программы мы обсудим опасности небрежного использования `defer`. Код для программы `defer.go` выглядит следующим образом:

```
package main

import (
    "fmt"
)

func d1() {
    for i := 3; i > 0; i-- {
        defer fmt.Println(i, " ")
    }
}
```

В `d1()` `defer` выполняется внутри тела функции с помощью вызова `fmt.Println()`. Не забывайте, что эти вызовы `fmt.Println()` выполняются непосредственно перед возвратом функции `d1()`.

```
func d2() {
    for i := 3; i > 0; i-- {
        defer func() {
            fmt.Println(i, " ")
        }()
    }
    fmt.Println()
}
```

В `d2()` `defer` привязано к анонимной функции, которая не принимает никаких параметров. На практике это означает, что анонимная функция должна получить значение `i` самостоятельно, что опасно, так как текущее значение `i` зависит от того, когда именно выполняется анонимная функция.



Анонимная функция является замыканием, поэтому имеет доступ к переменным, которые обычно находятся вне области видимости.

```
func d3() {
    for i := 3; i > 0; i-- {
        defer func(n int) {
            fmt.Println(n, " ")
        }(i)
    }
}
```

В этом случае текущее значение `i` передается анонимной функции в качестве параметра, который инициализирует параметр функции `n`. Это означает, что нет никаких двусмыслий относительно значения `i`.

```
func main() {
    d1()
    d2()
    fmt.Println()
    d3()
    fmt.Println()
}
```

Задача `main()` состоит в том, чтобы вызвать `d1()`, `d2()` и `d3()`.

При выполнении `defer.go` мы получаем такой вывод:

```
$ go run defer.go
1 2 3
0 0 0
1 2 3
```

Скорее всего, сгенерированный вывод покажется вам непростым для понимания; это доказывает, что действие и результаты использования `defer` могут быть

сложными, если ваш код неясный и двусмысленный. Позвольте мне объяснить полученный результат, чтобы вы лучше поняли, насколько сложным может оказаться `defer`, если не уделять пристального внимания своему коду.

Начнем с первой строки выходных данных (`1 2 3`), которая генерируется функцией `d1()`. Значения `i` в `d1()` равны `3`, `2` и `1`, именно в таком порядке. Функция, которая получает отсрочку в `d1()`, — это оператор `fmt.Println()`; в результате, когда функция `d1()` вот-вот вернется, вы получите три значения переменной `i` цикла `for` в обратном порядке. Это связано с тем, что отложенные функции выполняются в порядке LIFO.

Теперь что касается второй строки вывода, создаваемого функцией `d2()`. Здесь странно то, что на выходе мы получили три нуля вместо `1 2 3`; однако и для этого есть причина. После завершения цикла `for` значение `i` равно `0`, поскольку именно это значение `i` привело к завершению цикла `for`. Однако сложность здесь заключается в том, что отложенная анонимная функция вычисляется после завершения цикла `for` (поскольку у нее нет параметров). Это означает, что она вычисляется три раза для значения `i`, равного `0`, отсюда и полученный вывод. Такого рода запутанный код — как раз то, что может привести к появлению неприятных ошибок в ваших проектах, поэтому старайтесь его избегать.

Наконец, обратимся к третьей строке вывода, которая генерируется функцией `d3()`. Благодаря параметру анонимной функции каждый раз, когда анонимная функция откладывается, она получает `i`, следовательно, использует текущее значение `i`. В результате каждое выполнение анонимной функции имеет свое значение для обработки без каких-либо двусмысленностей, отсюда и сгенерированный вывод.

Все это говорит нам о том, что наилучшим подходом к использованию `defer` будет подход, показанный в функции `d3()`, поскольку вы намеренно передаете нужную переменную в анонимной функции в удобном для чтения виде. Теперь, когда мы узнали о `defer`, пришло время обсудить нечто совершенно другое: как разрабатывать собственные пакеты.

Разработка собственных пакетов

В какой-то момент вам понадобится разработать собственные пакеты, чтобы упорядочить свой код и иметь возможность при необходимости распространять его. Как указано в начале главы, все, что начинается с заглавной буквы, считается общедоступным и может использоваться извне, тогда как все остальные элементы считаются закрытыми. Единственным исключением из этого Go-правила служат имена пакетов — рекомендуется использовать имена в нижнем регистре (хотя и имена в верхнем регистре тоже разрешены).

Компиляцию Go-пакета можно выполнить вручную, если пакет существует на локальном компьютере, но это также выполняется автоматически после скачивания пакета из Интернета, так что беспокоиться об этом не нужно. Кроме того, если скачиваемый вами пакет содержит какие-либо ошибки, то вы узнаете о них во время скачивания.

Однако если вы хотите самостоятельно скомпилировать пакет, который был сохранен в файле `post05.go` (комбинация *PostgreSQL* и главы 05), можете использовать следующую команду:

```
$ go build -o post.a post05.go
```

Данная команда компилирует файл `post05.go` и сохраняет его выходные данные в файл `post.a`:

```
$ file post.a
post.a: current ar archive
```

Файл `post.a` представляет собой `ar`-архив.



Основная причина для самостоятельной компиляции Go-пакетов заключается в проверке кода на синтаксические и другие виды ошибок. Кроме того, вы можете создавать Go-пакеты в виде плагинов (<https://golang.org/pkg/plugin/>) или общих библиотек. Подробное обсуждение этих вопросов выходит за рамки данной книги.

Функция `init()`

Каждый Go-пакет может дополнительно иметь закрытую функцию `init()`, которая автоматически запускается, когда пакет инициализируется в начале выполнения программы. Эта функция обладает следующими характеристиками:

- не принимает аргументы;
- не возвращает никакие значения;
- является необязательной;
- вызывается Go неявно;
- у вас может быть функция `init()` в пакете `main`. В этом случае она выполнится перед функцией `main()`. Фактически все функции `init()` всегда выполняются перед функцией `main()`;
- исходный файл может содержать несколько функций `init()` — они выполняются в порядке объявления;

- функция `init()` или функции пакета выполняются только раз, даже если пакет импортируется неоднократно;
- Go-пакеты могут содержать несколько файлов. Каждый исходный файл может содержать одну или несколько функций `init()`.

Тот факт, что функция `init()` изначально является закрытой, означает, что она не может быть вызвана извне пакета, в котором содержится. Кроме того, пользователь пакета не имеет никакого контроля над функцией `init()`, поэтому вам следует тщательно подумать, прежде чем использовать ее в общедоступных пакетах или изменять в ней любое глобальное состояние.

Есть ряд исключений, когда можно использовать `init()`:

- для инициализации сетевых подключений, что может занять некоторое время до выполнения функций или методов пакета;
- для инициализации подключений к одному или нескольким серверам перед выполнением функций или методов пакета;
- для создания необходимых файлов и каталогов;
- для проверки доступности требуемых ресурсов.

Поскольку порядок выполнения иногда может вызывать недоумение, в следующем подразделе мы поговорим о нем более подробно.

Порядок исполнения

В этом подразделе показано, как выполняется Go-код. Например, если пакет `main` импортирует пакет `A`, а пакет `A` зависит от пакета `B`, то произойдет следующее:

- процесс начинается с пакета `main`;
- пакет `main` импортирует пакет `A`;
- пакет `A` импортирует пакет `B`;
- инициализируются глобальные переменные (если таковые имеются) в пакете `B`;
- выполняется функция `init()` или функции пакета `B`, если они существуют. Это первая функция `init()`, которая выполняется;
- глобальные переменные, если таковые имеются, в пакете `A` инициализируются;
- выполняется функция `init()` или функции пакета `A`, если таковые имеются;
- инициализируются глобальные переменные в пакете `main`;
- выполняется функция `init()` или функции пакета `main`, если они есть;
- функция `main()` пакета `main` начинает выполнение.



Обратите внимание: если пакет `main` импортирует пакет `B` самостоятельно, то ничего не произойдет, поскольку все, что связано с пакетом `B`, запускается пакетом `A`. Так происходит потому, что пакет `A` сначала импортирует пакет `B`.

На рис. 5.1 показано, как скрытым образом происходит упорядочивание выполнения Go-кода.

Вы можете узнать больше информации о порядке выполнения в Go, обратившись к документу спецификации языка Go по адресу https://golang.org/ref/spec#Order_of_evaluation, и об инициализации пакета — по адресу https://golang.org/ref/spec#Package_initialization.

Использование GitHub для хранения Go-пакетов

В этом разделе вы узнаете, как создать репозиторий GitHub, где можно сохранить свой Go-пакет, сделав его доступным для всего мира.

Прежде всего вам придется самостоятельно создать репозиторий GitHub. Самый простой способ сделать это — посетить сайт GitHub и перейти на вкладку **Repositories**, где вы можете просматривать существующие репозитории и создавать новые. Нажмите кнопку **New** и введите информацию, которая необходима для создания нового репозитория GitHub. Если вы сделали свой репозиторий общедоступным, то увидеть его смогут все желающие, а если ваш репозиторий закрытый, то лишь те люди, которых вы выберете.



Иметь в вашем репозитории GitHub понятный файл `README.md`, объясняющий, как работает Go-пакет, считается очень хорошей практикой.

Далее нужно клонировать репозиторий на вашем локальном компьютере. Обычно я делаю это с помощью утилиты `git(1)`. Поскольку имя репозитория — `post05`,



Рис. 5.1. Порядок выполнения в Go

а мое имя пользователя на GitHub — `mactsouk`, то команда `git clone` выглядит следующим образом:

```
$ git clone git@github.com:mactsouk/post05.git
```

Ведите `cd post05`, и дело в шляпе! После этого вам просто нужно написать код Go-пакета, не забыв выполнить `git commit` и `git push`.

Внешний вид такого репозитория после его использования в течение некоторого времени можно увидеть на рис. 5.2 (несколько позже вы узнаете больше о репозитории `post05`).

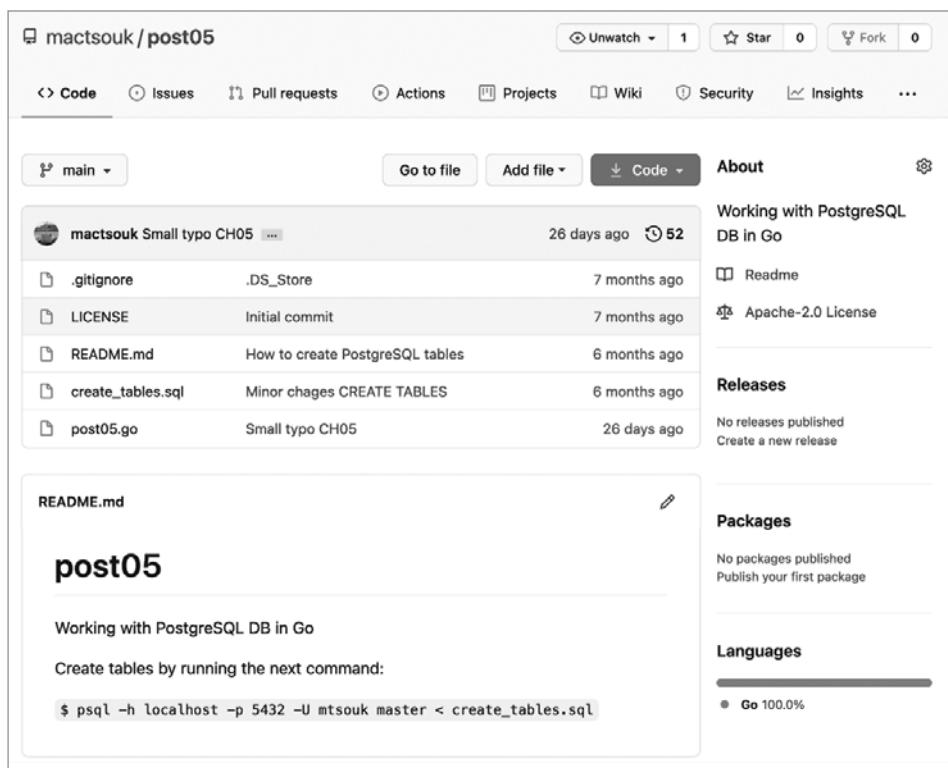


Рис. 5.2. Репозиторий GitHub с Go-пакетом



Использование GitLab вместо GitHub для размещения вашего кода не требует внесения каких-либо изменений в вашу работу.

Если вы хотите использовать этот пакет, вам просто нужно выполнить `go get` для пакета, используя его URL, и включить его в свой блок `import`. Это мы рассмотрим, когда на самом деле применим его в программе.

В следующем разделе представлен Go-пакет, который позволяет работать с базой данных.

Пакет для работы с базой данных

В этом разделе будет разработан пакет Go для работы с заданной схемой базы данных, хранящейся в базе данных Postgres. Конечной целью будет демонстрация того, как разрабатывать, хранить и использовать пакеты. При взаимодействии с конкретными схемами и таблицами в вашем приложении вы обычно создаете отдельные пакеты со всеми функциями, связанными с базой данных, — это относится и к базам данных NoSQL.

Go предлагает универсальный пакет (<https://golang.org/pkg/database/sql/>) для работы с базами данных. Однако для каждой базы данных требуется определенный пакет, который действует как драйвер и позволяет Go подключаться и работать с этой конкретной базой данных.

Шаги для создания нужного Go-пакета следующие.

- Загрузка необходимых внешних пакетов Go для работы с PostgreSQL.
- Создание файлов пакетов.
- Разработка необходимых функций.
- Использование Go-пакета для разработки утилит.
- Применение инструментов CI/CD для автоматизации (это не обязательно).

Возможно, вам интересно, почему мы создаем такой пакет для работы с базой данных, а не записываем сами команды в наши программы по мере необходимости. Причины таковы:

- Go-пакет может быть общим для всех членов команды, которые работают с приложением;
- Go-пакет позволяет людям использовать базу данных документированными способами;
- специализированные функции, которые вы включаете в свой Go-пакет, намного лучше соответствуют вашим потребностям;

- людям не нужен полный доступ к базе данных — они просто используют функции пакета и ту функциональность, которую он предлагает;
- если вы когда-либо внесете изменения в базу данных, то людям не нужно знать о них до тех пор, пока функции Go-пакета остаются неизменными.

С одной стороны, создаваемые вами функции могут взаимодействовать с определенной схемой базы данных со всеми ее таблицами и данными. С другой стороны, было бы практически невозможно работать с неизвестной схемой базы данных, не зная, как взаимосвязаны таблицы.



Помимо всех этих технических причин, очень интересно создавать Go-пакеты, которые являются общими для нескольких разработчиков!

Продолжим и узнаем больше о базе данных и ее таблицах.

Знакомство с базой данных

Скорее всего, вам потребуется загрузить дополнительный пакет для работы с сервером баз данных, таким как Postgres, MySQL или MongoDB. В данном случае мы используем *PostgreSQL* и поэтому должны загрузить Go-пакет, который позволит нам взаимодействовать с PostgreSQL. Существуют два основных пакета Go для подключения к PostgreSQL. Мы будем использовать github.com/lib/pq, но решать, какой пакет использовать, предстоит только вам.



Существует еще один Go-пакет для работы с PostgreSQL, называемый `jackc/pgx`, который можно найти по адресу <https://github.com/JackC/pgx>.

Вы можете загрузить его так:

```
$ go get github.com/lib/pq
```

Чтобы упростить задачу, сервер PostgreSQL запускается из образа Docker с помощью файла `docker-compose.yml`, который имеет следующее содержимое:

```
version: '3'

services:
  postgres:
    image: postgres
    container_name: postgres
    environment:
      - POSTGRES_USER=mtsouk
      - POSTGRES_PASSWORD=pass
```

```
- POSTGRES_DB=master
volumes:
- ./postgres:/var/lib/postgresql/data/
networks:
- psql
ports:
- "5432:5432"

volumes:
postgres:

networks:
pgsql:
driver: bridge
```

Номер порта по умолчанию, который прослушивает сервер PostgreSQL, равен 5432. Поскольку мы подключаемся к серверу PostgreSQL с того же компьютера, именем хоста будет `localhost` или, если вы предпочитаете IP-адрес, `127.0.0.1`. Если вы используете другой сервер PostgreSQL, то вам нужно соответствующим образом изменить сведения о подключении в следующем коде.



В PostgreSQL схема — это пространство имен, содержащее именованные объекты базы данных, такие как таблицы, представления и индексы. PostgreSQL автоматически создает схему `public` для каждой новой базы данных.

Следующая Go-утилита `GetSchema.go` помогает проверить, что вы можете подключиться к базе данных PostgreSQL и получить список доступных баз данных и таблиц в этой базе данных и `public` схеме — вся информация о подключении предоставляется в качестве аргументов командной строки:

```
package main

import (
    "database/sql"
    "fmt"
    "os"
    "strconv"

    _ "github.com/lib/pq"
)
```

Пакет `lib/pq`, который является интерфейсом к базе данных PostgreSQL, не используется непосредственно кодом. Следовательно, вам необходимо импортировать пакет `lib/pq` с помощью `_`, чтобы компилятор Go не выдавал сообщение об ошибке, связанное с импортом пакета, а не с его применением.

В большинстве случаев вам не нужно импортировать пакет с помощью `_`, но это одно из исключений. Данный вид импорта обычно вызван тем, что

импортированный пакет имеет побочные эффекты, такие как регистрация себя в качестве обработчика базы данных для пакета `sql`:

```
func main() {
    arguments := os.Args
    if len(arguments) != 6 {
        fmt.Println("Please provide: hostname port username password db")
        return
    }
```

Иметь хорошее справочное сообщение, чтобы утилита получила всю необходимую информацию, очень удобно.

```
host := arguments[1]
p := arguments[2]
user := arguments[3]
pass := arguments[4]
database := arguments[5]
```

Здесь мы собираем подробную информацию о подключении к базе данных.

```
// номер порта ДОЛЖЕН быть целым числом
if err != nil {
    fmt.Println("Not a valid port number:", err)
    return
}

// строка подключения
conn := fmt.Sprintf("host=%s port=%d user=%s password=%s dbname=%s
sslmode=disable", host, port, user, pass, database)
```

Вот как мы определяем строку подключения с подробными сведениями для подключения к серверу базы данных PostgreSQL. Стока подключения должна быть передана в функцию `sql.Open()` для установления соединения. Пока соединения у нас нет.

```
// открыть базу данных PostgreSQL
db, err := sql.Open("postgres", conn)
if err != nil {
    fmt.Println("Open():", err)
    return
}
defer db.Close()
```

Функция `sql.Open()` открывает соединение с базой данных и сохраняет его открытый до завершения программы или до тех пор, пока вы не выполните `Close()` для правильного закрытия соединения с базой данных.

```
// получить все базы данных
rows, err := db.Query(`SELECT "datname" FROM "pg_database"
WHERE datistemplate = false`)
if err != nil {
```

```

    fmt.Println("Query", err)
    return
}

```

Чтобы выполнить запрос `SELECT`, вам необходимо его создать. Поскольку предоставленный запрос `SELECT` не содержит параметров, то есть не изменяется в зависимости от переменных, вы можете передать его функции `Query()` и выполнить ее. *Текущий* результат запроса `SELECT` сохраняется в переменной `rows`, которая служит *курсором*. Вы не получите все результаты из базы данных сразу (так как запрос может возвращать миллионы записей), но получите их одну за другой — в этом весь смысл использования курсора.

```

for rows.Next() {
    var name string
    err = rows.Scan(&name)
    if err != nil {
        fmt.Println("Scan", err)
        return
    }
    fmt.Println("*", name)
}
defer rows.Close()

```

Здесь показано, как обрабатывать результаты запроса `SELECT`, который может как не содержать ни одной строки, так и включать множество строк. Поскольку переменная `rows` является курсором, вы переходите от строки к строке, вызывая `Next()`. После этого вам нужно присвоить значения, возвращаемые из запроса `SELECT`, Go-переменным, чтобы затем использовать их. Это происходит при вызове `Scan()`, для которого требуются параметры указателя. Если запрос `SELECT` возвращает несколько значений, то вам необходимо ввести несколько параметров в `Scan()`. Наконец, вы должны вызвать `Close()` с `defer` для переменной `rows`, чтобы закрыть оператор и освободить различные типы используемых ресурсов.

```

// получить все таблицы из текущей базы данных
query := `SELECT table_name FROM information_schema.tables WHERE
    table_schema = 'public' ORDER BY table_name`
rows, err = db.Query(query)
if err != nil {
    fmt.Println("Query", err)
    return
}

```

Мы собираемся выполнить еще один предоставленный пользователем запрос `SELECT` в текущей базе данных. Определение запроса `SELECT` сохраняется в переменной `query` в целях простоты и читаемости кода. Содержимое переменной `query` передается методу `db.Query()`.

```

// вот как вы обрабатываете строки, возвращаемые из SELECT
for rows.Next() {
    var name string

```

```

        err = rows.Scan(&name)
        if err != nil {
            fmt.Println("Scan", err)
            return
        }
        fmt.Println("+T", name)
    }
    defer rows.Close()
}

```

Повторюсь, нам нужно обработать строки, возвращаемые оператором `SELECT`, используя курсор `rows` и метод `Next()`.

При выполнении `GetSchema.go` мы получаем такой вывод:

```
$ go run getSchema.go localhost 5432 mtsouk pass go
* postgres
* master
* go
+T userdata
+T users
```

Но о чём они нам говорят? Строки, начинающиеся с `*`, показывают базы данных PostgreSQL, тогда как строки, начинающиеся с `+T`, показывают таблицы базы данных — это наше решение. Таким образом, эта конкретная установка PostgreSQL содержит три базы данных: `postgres`, `master` и `go`. Схема `public` базы данных `go`, которая задается последним аргументом командной строки, содержит две таблицы: `userdata` и `users`.

Главное преимущество утилиты `GetSchema.go` заключается в том, что она универсальна и может быть использована для получения дополнительной информации о серверах PostgreSQL. Это и является основной причиной того, что для ее работы требуется так много аргументов командной строки.

Теперь, когда мы знаем, как получить доступ к базе данных PostgreSQL и обратиться к ней с помощью Go, следующей задачей должно стать создание репозитория GitHub или GitLab для хранения и распространения Go-пакета, который мы собираемся разработать.

Хранение Go-пакета

Первое действие, которое мы должны предпринять, — это создать репозиторий для хранения Go-пакета. В нашем случае мы собираемся использовать для этого репозиторий GitHub. Неплохой идеей будет сохранить репозиторий GitHub *закрытым* во время разработки, прежде чем предоставлять его остальному миру, особенно когда вы создаете что-то критически важное.



Сохранение закрытого репозитория GitHub не влияет на процесс разработки, но это может затруднить совместное использование Go-пакета, поэтому в некоторых случаях его можно делать общедоступным.

Для простоты мы будем использовать общедоступный Go-репозиторий для Go-модуля, который называется `post05`, — его полный URL: <https://github.com/mactsovuk/post05>.

Чтобы использовать этот пакет на своих компьютерах, вы должны сначала выполнить `go get`. Однако во время разработки вы должны начать с `git clone git@github.com:mactsovuk/post05.git`, чтобы получить содержимое репозитория GitHub и внести в него изменения.

Дизайн Go-пакета

На рис. 5.3 показана схема базы данных, с которой работает Go-пакет. Помните, что при работе с определенной базой данных и схемой вам необходимо «включить» информацию о схеме в свой Go-код. Проще говоря, Go-код должен знать о схеме, по которой он работает.

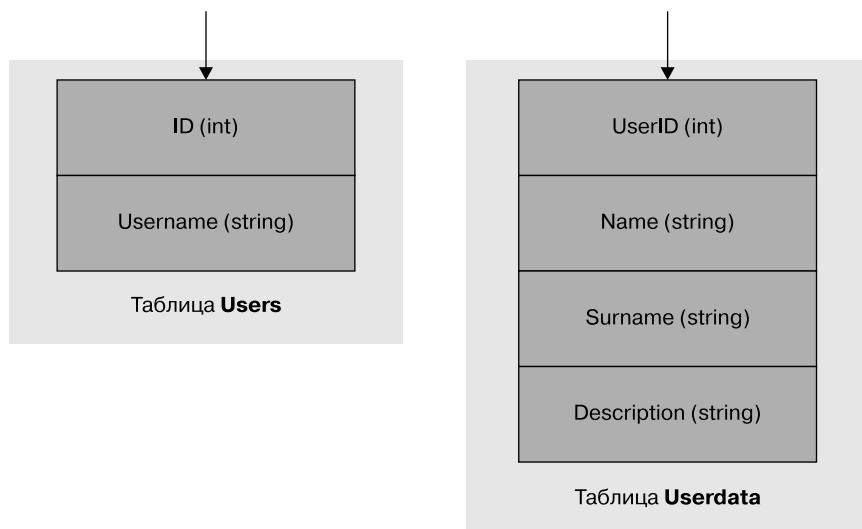


Рис. 5.3. Две таблицы базы данных, с которыми работает Go-пакет

Это простая схема, которая позволяет нам сохранять и обновлять пользовательские данные. Две таблицы объединяют *user ID*, который должен быть

уникальным. Кроме того, поле `Username` в таблице `Users` также должно быть уникальным, поскольку два или более пользователя не могут применять одно и то же имя.

Эта схема уже существует на сервере базы данных PostgreSQL; это значит, Go-код предполагает, что соответствующие таблицы находятся в нужном месте и хранятся в правильной базе данных PostgreSQL. Помимо таблицы `Users`, существует также таблица `Userdata`, которая содержит информацию о каждом пользователе. Запись, добавленную в таблицу `Users`, нельзя изменить. А вот что может измениться, так это данные, хранящиеся в таблице `Userdata`.

Если вы хотите создать базу данных `Users` и две таблицы в базе данных `go`, то можете выполнить следующие операторы, которые сохранены в файле `create_tables.sql` с помощью утилиты `psql`:

```
DROP DATABASE IF EXISTS go;
CREATE DATABASE go;

DROP TABLE IF EXISTS Users;
DROP TABLE IF EXISTS Userdata;

\c go;

CREATE TABLE Users (
    ID SERIAL,
    Username VARCHAR(100) PRIMARY KEY
);

CREATE TABLE Userdata (
    UserID Int NOT NULL,
    Name VARCHAR(100),
    Surname VARCHAR(100),
    Description VARCHAR(200)
);
```

Утилита командной строки для работы с Postgres называется `psql`. Команда утилиты для выполнения кода `create_tables.sql` выглядит следующим образом:

```
$ psql -h localhost -p 5432 -U mtsouk master < create_tables.sql
```

Теперь, когда у нас готова необходимая инфраструктура, приступим к обсуждению Go-пакета. Задачи, которые должен выполнять Go-пакет, чтобы облегчить нашу жизнь, заключаются в следующем:

- создавать нового пользователя;
- удалять существующего пользователя;

- обновлять существующего пользователя;
- выдавать список всех пользователей.

Каждая из этих задач должна включать одну или несколько функций или Go-методов для ее работы, что мы и собираемся реализовать в Go-пакете:

- функция для инициирования подключения Postgres — сведения о подключении должны быть предоставлены пользователем, а пакет должен иметь возможность их использовать. Однако вспомогательная функция для инициирования соединения может быть и закрытой;
- в некоторых деталях подключения могут содержаться значения по умолчанию;
- функция, которая проверяет, существует ли данное имя пользователя, — это вспомогательная функция, которая может быть закрытой;
- функция, которая добавляет нового пользователя в базу данных;
- функция, которая удаляет существующего пользователя из базы данных;
- функция для обновления существующего пользователя;
- функция для перечисления всех пользователей.

Теперь, обсудив общую структуру и функциональность Go-пакета, мы должны приступить к его реализации.

Реализация Go-пакета

В этом подразделе мы реализуем Go-пакет для работы с базой данных Postgres и заданной схемой базы данных. Мы представим каждую функцию отдельно — если вы объедините все эти функции, то на выходе получите функционал всего пакета.



Во время разработки пакета вы должны регулярно фиксировать свои изменения в репозитории GitHub или GitLab.

Первый элемент, который вам нужен в вашем Go-пакете, — это одна или несколько *структур*, которые могут хранить данные из таблиц базы данных. В большинстве случаев вам нужно столько структур, сколько существует таблиц, — мы начнем с этого и посмотрим, что получится. Поэтому мы определим следующие структуры:

```
type User struct {
    ID      int
    Username string
}
```

```
type Userdata struct {
    ID      int
    Name    string
    Surname string
    Description string
}
```

Если хорошо подумать, то легко увидеть, что нет смысла создавать две отдельные Go-структуры. Это связано с тем, что структура `User` не содержит реальных данных и нет смысла передавать несколько структур функциям, которые обрабатывают данные для таблиц PostgreSQL `Users` и `Userdata`. Таким образом, мы можем создать единую Go-строктуру для хранения всех определяемых данных:

```
type Userdata struct {
    ID      int
    Username string
    Name    string
    Surname string
    Description string
}
```

Для простоты я решил назвать структуру как таблицу базы данных, однако в текущем случае это не совсем правильно, поскольку структура `Userdata` содержит больше полей, чем таблица базы данных `Userdata`. Дело в том, что нам не нужно все подряд из таблицы базы данных `Userdata`.

Преамбула пакета выглядит следующим образом:

```
package post05

import (
    "database/sql"
    "errors"
    "fmt"
    "strings"

    _ "github.com/lib/pq"
)
```

Впервые в книге мы видим имя пакета, отличное от `main`, которым в данном случае является `post05`. Поскольку пакет взаимодействует с PostgreSQL, мы импортируем пакет `github.com/lib/pq` и используем `_` перед путем к пакету. Как обсуждалось ранее, это происходит потому, что импортированный пакет регистрирует себя в качестве обработчика базы данных для пакета `sql`, но не применяется непосредственно в коде. Он используется только через пакет `sql`.

Далее у вас должны быть переменные для хранения сведений о соединении. В случае с `post05` это может быть реализовано с помощью таких глобальных переменных:

```
// детали подключения
var (
    Hostname = ""
    Port = 2345
    Username = ""
    Password = ""
    Database = ""
)
```

Помимо переменной `Port`, которая имеет начальное значение, другие глобальные переменные имеют значение по умолчанию для своего типа данных, которым является `string`. Все эти переменные должны быть должным образом инициализированы Go-кодом, использующим пакет `post05`, и должны быть доступны извне пакета. Это означает, что их первая буква должна быть в верхнем регистре.

Функция `openConnection()`, которая является *закрытой* и доступна только в рамках пакета, определяется как:

```
func openConnection() (*sql.DB, error) {
    // строка подключения
    conn := fmt.Sprintf("host=%s port=%d user=%s password=%s dbname=%s
        sslmode=disable", Hostname, Port, Username, Password, Database)

    // открыть базу данных
    db, err := sql.Open("postgres", conn)
    if err != nil {
        return nil, err
    }
    return db, nil
}
```

Вы уже видели этот код в утилите `GetSchema.go`. Вы создаете строку подключения и передаете ее в `sql.Open()`.

Теперь рассмотрим функцию `exists()`, которая также является закрытой:

```
// Функция возвращает ID пользователя username
// -1, если пользователь не существует
func exists(username string) int {
    username = strings.ToLower(username)

    db, err := openConnection()
    if err != nil {
        fmt.Println(err)
        return -1
    }
    defer db.Close()
```

```
userID := -1
statement := fmt.Sprintf(`SELECT "id" FROM "users" where
    username = '%s'`, username)
rows, err := db.Query(statement)
```

Здесь мы определяем запрос, который показывает, существует ли указанное имя пользователя в базе данных. Поскольку все наши данные хранятся в БД, нам необходимо постоянно с ней взаимодействовать.

```
for rows.Next() {
    var id int
    err = rows.Scan(&id)
    if err != nil {
        fmt.Println("Scan", err)
        return -1
    }
}
```

Если `rows.Scan(&id)` выполняется без каких-либо ошибок, то мы знаем, что был возвращен результат, который является желаемым идентификатором пользователя.

```
    userID = id
}
defer rows.Close()
return userID
}
```

Последняя часть `exists()` устанавливает или закрывает запрос, чтобы освободить ресурсы, и возвращает значение идентификатора имени пользователя, которое задается в качестве параметра `exists()`.



Во время разработки я включаю в код пакета множество операторов `fmt.Println()` для целей отладки. Однако я удалил большинство из них в окончательной версии Go-пакета и заменил их значениями ошибок. Эти значения передаются программе, использующей пакет, и уже она отвечает за принятие решения о том, что делать с сообщениями об ошибках и ошибочных состояниями. Вы также можете использовать для этого журналирование — вывод можно перенаправить в стандартный или даже в `/dev/null`, когда он не нужен.

Реализация функции `AddUser()`:

```
// AddUser добавляет нового пользователя в базу данных
// Возвращаем новый ID пользователя
// -1, если произошла ошибка
func AddUser(d Userdata) int {
    d.Username = strings.ToLower(d.Username)
```

Все имена пользователей преобразуются в нижний регистр, чтобы избежать дублирования. Это проектное решение.

```
db, err := openConnection()
if err != nil {
    fmt.Println(err)
    return -1
}
defer db.Close()

userID := exists(d.Username)
if userID != -1 {
    fmt.Println("User already exists:", Username)
    return -1
}

insertStatement := `insert into "users" ("username") values ($1)`
```

Вот как мы создаем запрос, который принимает параметры. Для представленного запроса требуется одно значение \$1.

```
_ , err = db.Exec(insertStatement, d.Username)
```

Вот как вы передаете желаемое значение, `d.Username`, в переменную `insertStatement`.

```
if err != nil {
    fmt.Println(err)
    return -1
}

userID = exists(d.Username)
if userID == -1 {
    return userID
}

insertStatement = `insert into "userdata" ("userid", "name", "surname",
    "description") values ($1, $2, $3, $4)`
```

Представленному запросу нужны четыре значения: \$1, \$2, \$3 и \$4.

```
_ , err = db.Exec(insertStatement, userID, d.Name, d.Surname, d.Description)
if err != nil {
    fmt.Println("db.Exec()", err)
    return -1
}
```

Поскольку нам нужно передать четыре переменные в `insertStatement`, мы поместим четыре значения в вызов `db.Exec()`.

```
return userID
}
```

Это завершение функции, добавляющей нового пользователя в базу данных. Реализация функции `deleteUser()` заключается в следующем.

```
// deleteUser удаляет существующего пользователя
func DeleteUser(id int) error {
    db, err := openConnection()
    if err != nil {
        return err
    }
    defer db.Close()

    // Существует ли идентификатор?
    statement := fmt.Sprintf(`SELECT "username" FROM "users" where id = %d`, id)
    rows, err := db.Query(statement)
```

Здесь мы лишний раз проверяем, существует ли данный идентификатор пользователя в таблице `users`.

```
var username string
for rows.Next() {
    err = rows.Scan(&username)
    if err != nil {
        return err
    }
}
defer rows.Close()

if exists(username) != id {
    return fmt.Errorf("User with ID %d does not exist", id)
}
```

Если ранее возвращенное имя пользователя существует и имеет тот же идентификатор пользователя, что и параметр `deleteUser()`, то можно продолжить процесс удаления, который содержит два шага: во-первых, удаление соответствующих пользовательских данных из таблицы `userdata` и, во-вторых, удаление данных из таблицы `users`.

```
// удалить из Userdata
deleteStatement := `delete from "userdata" where userid=$1`
_, err = db.Exec(deleteStatement, id)
if err != nil {
    return err
}

// удалить из Users
deleteStatement = `delete from "users" where id=$1`
_, err = db.Exec(deleteStatement, id)
if err != nil {
    return err
}

return nil
}
```

Теперь рассмотрим реализацию функции `ListUsers()`.

```
func ListUsers() ([]Userdata, error) {
    Data := []Userdata{}
    db, err := openConnection()
    if err != nil {
        return Data, err
    }
    defer db.Close()
```

Напомню, нам нужно открыть соединение с базой данных перед выполнением запроса к базе данных.

```
rows, err := db.Query(`SELECT
    "id", "username", "name", "surname", "description"
    FROM "users", "userdata"
    WHERE users.id = userdata.userid`)
if err != nil {
    return Data, err
}

for rows.Next() {
    var id int
    var username string
    var name string
    var surname string
    var description string
    err = rows.Scan(&id, &username, &name, &surname, &description)
    temp := Userdata{ID: id, Username: username, Name: name,
        Surname: surname, Description: description}
```

На этом этапе мы сохраним данные, полученные из запроса `SELECT`, в структуре `Userdata`. Они добавляются к срезу, который будет возвращен из функции `ListUsers()`. Процесс продолжается до тех пор, пока чтение возможно.

```
    Data = append(Data, temp)
    if err != nil {
        return Data, err
    }
}
defer rows.Close()
return Data, nil
}
```

После обновления содержимого `Data` с помощью `append()` мы завершаем запрос, и функция возвращает список доступных пользователей, сохраненный в `Data`.

Наконец, рассмотрим функцию `UpdateUser()`:

```
// UpdateUser предназначена для обновления данных существующего пользователя
func UpdateUser(d Userdata) error {
    db, err := openConnection()
    if err != nil {
```

```

        return err
    }
    defer db.Close()

    userID := exists(d.Username)
    if userID == -1 {
        return errors.New("User does not exist")
    }
}

```

Для начала нам нужно убедиться, что данное имя пользователя существует в базе данных — процесс обновления основан на имени пользователя.

```

d.ID = userID
updateStatement := `update "userdata" set "name"=$1, "surname"=$2,
    "description"=$3 where "userid"=$4`
_, err = db.Exec(updateStatement, d.Name, d.Surname, d.Description, d.ID)
if err != nil {
    return err
}

return nil
}

```

Оператор обновления, хранящийся в `updateStatement`, которая выполняется с использованием желаемых параметров с помощью `db.Exec()`, обновляет пользовательские данные.

Теперь, когда мы в подробностях знаем, как реализовать каждую функцию в пакете `post05`, пришло время начать его использовать!

Тестирование Go-пакета

Чтобы протестировать пакет, мы должны разработать утилиту командной строки, называемую `postGo.go`.



Поскольку `postGo.go` использует внешний пакет (пусть даже и наш), не забывайте скачивать его последнюю версию с помощью `go get` или `go get -u`.

Поскольку `postGo.go` используется только в целях тестирования, мы жестко закодировали большую часть данных, кроме имени пользователя, которое вводим в базу данных. Все имена пользователей генерируются случайным образом.

Код `postGo.go` выглядит следующим образом:

```

package main

import (
    "fmt"

```

```

"math/rand"
"time"

"github.com/mactsouk/post05"
)

```

Поскольку `post05` работает с Postgres, здесь нет необходимости импортировать `lib/pq`:

```

var MIN = 0
var MAX = 26

func random(min, max int) int {
    return rand.Intn(max-min) + min
}

func getString(length int64) string {
    startChar := "A"
    temp := ""
    var i int64 = 1
    for {
        myRand := random(MIN, MAX)
        newChar := string(startChar[0] + byte(myRand))
        temp = temp + newChar
        if i == length {
            break
        }
        i++
    }
    return temp
}

```

Обе функции, `random()` и `getString()`, помогают генерировать случайные строки, заданные в качестве имен пользователей.

```

func main() {
    post05.Hostname = "localhost"
    post05.Port = 5432
    post05.Username = "mtsouk"
    post05.Password = "pass"
    post05.Database = "go"

```

Здесь определяются параметры подключения к серверу Postgres, а также база данных, в которой вы собираетесь работать (`go`). Поскольку все эти переменные находятся в пакете `post05`, доступ к ним осуществляется соответствующим образом.

```

data, err := post05.ListUsers()
if err != nil {
    fmt.Println(err)
    return
}

```

```
for _, v := range data {
    fmt.Println(v)
}
```

Мы начнем с перечисления существующих пользователей.

```
SEED := time.Now().Unix()
rand.Seed(SEED)
random_username := getString(5)
```

Затем мы генерируем случайную строку, которая используется в качестве имени пользователя. Все случайно сгенерированные имена пользователей имеют длину пять символов из-за вызова `getString(5)`. По желанию вы можете изменить это значение.

```
t := post05.Userdata{
    Username:    random_username,
    Name:        "Mihalis",
    Surname:     "Tsoukalos",
    Description: "This is me!"}

id := post05.AddUser(t)
if id == -1 {
    fmt.Println("There was an error adding user", t.Username)
}
```

Здесь мы добавляем нового пользователя в базу. Данные пользователя, включая имя, хранятся в структуре `post05.Userdata`. Эта структура передается функции `post05.AddUser()`, которая возвращает идентификатор нового пользователя.

```
err = post05.DeleteUser(id)
if err != nil {
    fmt.Println(err)
}
```

Здесь мы удаляем созданного пользователя, используя значение идентификатора пользователя, возвращаемое `post05.AddUser(t)`.

```
// Попытка повторного удаления!
err = post05.DeleteUser(id)
if err != nil {
    fmt.Println(err)
}
```

Если вы попытаетесь удалить того же пользователя еще раз, то процесс завершится неудачей, поскольку пользователя уже не существует.

```
id = post05.AddUser(t)
if id == -1 {
    fmt.Println("There was an error adding user", t.Username)
}
```

Здесь мы снова добавляем того же пользователя, однако, поскольку значения идентификатора пользователя генерируются Postgres, на этот раз у пользователя будет новое значение идентификатора.

```
t = post05.Userdata{  
    Username:     "random_username",  
    Name:         "Mihalis",  
    Surname:      "Tsoukalos",  
    Description:  "This might not be me!"}
```

Здесь мы обновляем поле `Description` структуры `post05.Userdata` перед передачей его в `post05.UpdateUser()`, чтобы обновить информацию, хранящуюся в базе данных.

```
err = post05.UpdateUser(t)  
if err != nil {  
    fmt.Println(err)  
}  
}
```

Работа с `postGo.go` дает следующий вывод:

```
$ go run postGo.go  
{4 mhmxz Mihalis Tsoukalos This might not be me!}  
{6 wsdlg Mihalis Tsoukalos This might not be me!}  
User with ID 7 does not exist
```

В этом выводе подтверждается, что `postGo.go` работает должным образом, поскольку может подключаться к базе данных, добавлять нового пользователя и удалять существующего. Это также означает, что пакет `post05` работает должным образом. Теперь, выяснив, как создавать Go-пакеты, кратко обсудим Go-модули.

Модули

Go-модуль подобен Go-пакету с версией, однако Go-модули могут состоять из нескольких пакетов. Go использует *семантическое управление версиями* для управления версиями модулей. Это означает, что версии начинаются с буквы `v`, за которой следуют номера версий в виде `major.minor.patch`. Таким образом, у вас могут получиться такие версии, как `v1.0.0`, `v1.0.5` и `v2.0.2`. Части `v1`, `v2` и `v3` обозначают основную версию Go-пакета, которая обычно не имеет обратной совместимости. Это означает, что если ваша программа Go работает с `v1`, то не обязательно будет работать с `v2` или `v3`, — это может сработать, но не стоит на это рассчитывать.

Второе число в версии касается функций. Обычно версия v1.1.0 обладает большим количеством функций, чем версия v1.0.2 или v1.0.0, при этом она совместима со всеми старыми версиями. Наконец, третье число — это просто исправления ошибок без каких-либо новых функций. Обратите внимание, что семантическое управление версиями также используется для версий Go.



Go-модули были введены в Go v1.11, но доработаны в Go v1.13.

Если вы хотите узнать больше о модулях, то прочтайте статью <https://blog.golang.org/using-go-modules>, состоящую из пяти частей, а также посетите <https://golang.org/doc/modules/developing>. Просто помните: *Go-модуль подобен, но не идентичен обычному Go-пакету с версией* и может состоять из нескольких пакетов.

Создание более качественных пакетов

В этом разделе приведены полезные советы, которые помогут разрабатывать более совершенные пакеты Go. Вот несколько хороших правил, которым стоит следовать при создании высококачественных Go-пакетов.

- Первое неофициальное правило успешного пакета заключается в том, что его элементы должны быть как-то связаны. Таким образом, вы можете создать пакет для работы с автомобилями, но было бы не очень хорошей идеей создавать единый пакет для работы с автомобилями, велосипедами и самолетами. Проще говоря, лучше разделить избыточную функциональность на несколько пакетов вместо того, чтобы добавлять большой объем функциональности в один Go-пакет.
- Второе практическое правило заключается в том, что сначала стоит использовать собственные пакеты в течение разумного периода времени, прежде чем предоставлять их широкой публике. Это поможет вам обнаружить глупые ошибки и убедиться, что ваши пакеты работают как часы. После этого передайте их коллегам-разработчикам для дополнительного тестирования, прежде чем делать их общедоступными. Кроме того, вы всегда должны писать тесты для любого пакета, который собираетесь дать использовать другим.
- Убедитесь, что ваш пакет имеет понятный и полезный API, чтобы любой пользователь мог быстро начать продуктивно с ним работать.
- Попробуйте ограничить общедоступный API ваших пакетов только тем, что абсолютно необходимо. Кроме того, дайте вашим функциям описательные, но не очень длинные имена.
- Интерфейсы, а также Go-дженерики могут повысить полезность ваших функций, поэтому, когда вы считаете это целесообразным, используйте ин-

терфейс вместо одного типа в качестве параметра функции или возвращающего типа.

- При обновлении одного из ваших пакетов старайтесь не нарушать работу и не создавать несовместимости со старыми версиями, если это не является абсолютно необходимым.
- При разработке нового пакета Go попробуйте использовать несколько файлов, чтобы сгруппировать похожие задачи или концепции.
- Не создавайте с нуля уже существующие пакеты. Внесите изменения в существующий пакет или создайте собственную его версию.
- Никому не нужен Go-пакет, который выводит информацию журнала на экран. Было бы более профессионально иметь флаг для включения ведения журнала, когда это необходимо. Go-код ваших пакетов должен гармонировать с Go-кодом ваших программ. Это означает, что если вы смотрите на программу, которая использует ваши пакеты, а имена ваших функций плохо выделяются в коде, то лучше изменить имена этих функций. Поскольку имя пакета используется почти везде, старайтесь использовать краткие и выразительные имена.
- Удобнее размещать новые определения Go-типов рядом с тем местом, где они используются впервые, поскольку никто, включая вас, не захочет искать определения новых типов данных в исходных файлах.
- Попробуйте создать тестовые файлы для своих пакетов, поскольку пакеты с тестовыми файлами считаются более профессиональными, чем пакеты без них; такие мелкие детали имеют решающее значение и позволяют другим людям быть уверенными в том, что вы серьезный разработчик! Обратите внимание, что писать тесты для ваших пакетов не обязательно и что следует избегать использования пакетов, которые не включают тесты. Больше информации о тестировании вы узнаете в главе 11.

Всегда помните: помимо того факта, что фактический Go-код в пакете не должен содержать ошибок, следующим по важности элементом успешного пакета является его документация, а также примеры кода, которые разъясняют его использование и демонстрируют особенности функций пакета. В следующем разделе мы обсудим создание документации в Go.

Создание документации

В этом разделе вы узнаете, как создать *документацию* для вашего Go-кода, использовав в качестве примера код пакета `post05`. Новый пакет переименован и теперь называется `document`.

В плане документации Go следует простому правилу: чтобы задокументировать функцию, метод, переменную или даже сам пакет, вы можете, как обычно, написать

комментарии, которые должны располагаться непосредственно перед элементом, который вы хотите задокументировать, без каких-либо пустых строк между ними. Вы можете использовать один или несколько однострочных комментариев, которые представляют собой строки, начинающиеся с `//`, или *блочные* комментарии, которые начинаются с `/*` и заканчиваются `*/`, — все, что находится между этими символами, считается комментарием.



Настоятельно рекомендуется, чтобы каждый создаваемый вами пакет Go имел блок комментария, предшествующий объявлению `package`, который познакомит разработчиков с пакетом, а также объяснит, что именно тот делает.

Вместо того чтобы представлять весь код пакета `post05`, мы оставим только важную часть; это значит, реализации функций будут содержать только операторы `return`. Новая версия `post05.go` называется `document.go` и поставляется со следующим кодом и комментариями:

```
/*
```

```
The package works on 2 tables on a PostgreSQL data base server.
```

```
The names of the tables are:
```

```
* Users  
* Userdata
```

```
The definitions of the tables in the PostgreSQL server are:
```

```
CREATE TABLE Users (  
    ID SERIAL,  
    Username VARCHAR(100) PRIMARY KEY  
) ;
```

```
CREATE TABLE Userdata (  
    UserID Int NOT NULL,  
    Name VARCHAR(100),  
    Surname VARCHAR(100),  
    Description VARCHAR(200)  
) ;
```

```
This is rendered as code
```

```
This is not rendered as code
```

```
 */  
package document
```

Это первый блок документации, который расположен прямо перед названием пакета. Это самое подходящее место для документирования функциональности

пакета, а также другой важной информации. В данном случае мы представляем команды создания SQL, полностью описывающие таблицы базы данных, с которыми мы собираемся работать. Еще один важный элемент — указание сервера базы данных, с которым взаимодействует пакет. Другая информация, которую вы можете поместить в начало пакета, — это автор, лицензия и версия пакета.

Если строка в комментарии к блоку начинается с табуляции, то в графическом выводе она отображается по-разному, что полезно для разграничения различных видов информации в документации:

```
// BUG(1): Function ListUsers() not working as expected
// BUG(2): Function AddUser() is too slow
```

Ключевое слово `BUG` выполняет особую функцию при написании документации. Go знает, что ошибки являются частью кода и поэтому также должны быть за-документированы. Вы можете написать любое сообщение после ключевого слова `BUG`, а также размещать их в любом месте (предпочтительно рядом с ошибками, которые они описывают).

```
import (
    "database/sql"
    "fmt"
    "strings"
)
```

В `github.com/lib/pq` пакет был удален из блока `import`, чтобы уменьшить размер файла.

```
/*
This block of global variables holds the connection details
to the Postgres server
Hostname: is the IP or the hostname of the server
Port: is the TCP port the DB server listens to
Username: is the username of the database user
Password: is the password of the database user
Database: is the name of the Database in PostgreSQL
*/
var (
    Hostname = ""
    Port = 2345
    Username = ""
    Password = ""
    Database = ""
)
```

Здесь мы видим способ документирования множества переменных одновре-менно — в данном случае глобальных. Преимущество этого способа заключа-ется в том, что вам не нужно ставить комментарий перед каждой глобальной

переменной и тем самым делать код менее читаемым. Единственным недостатком этого метода является то, что вы должны помнить об обновлении комментариев, если захотите внести какие-либо изменения в код. Однако одновременное документирование нескольких переменных может приводить к неправильному отображению веб-страниц godoc. По этой причине вы можете решить документировать каждое поле напрямую.

```
// Структура Userdata предназначена для хранения полных данных
// о пользователе из таблицы Userdata и имени пользователя
// из таблицы Users
type Userdata struct {
    ID int
    Username string
    Name string
    Surname string
    Description string
}
```

В вышеприведенном отрывке показано, как документировать Go-структурку, — это особенно полезно, когда в исходном файле много структур и вы хотите быстро просмотреть их.

```
// openConnection() предназначена для открытия соединения
// с Postgres, чтобы его могли использовать другие функции пакета
func openConnection() (*sql.DB, error) {
    var db *sql.DB
    return db, nil
}
```

При документировании функции полезно начинать первую строку комментариев с ее имени. Кроме того, вы можете написать в комментариях любую информацию, которую считаете важной.

```
// Функция возвращает ID пользователя username
// -1, если пользователя не существует
func exists(username string) int {
    fmt.Println("Searching user", username)
    return 0
}
```

В этом случае мы объясним возвращаемые значения функции `exists()`, поскольку они имеют особое значение.

```
// AddUser добавляет нового пользователя в базу данных
//
// Возвращает новый ID пользователя
// -1, если произошла ошибка
func AddUser(d Userdata) int {
```

```
d.Username = strings.ToLower(d.Username)
return -1
}

/*
DeleteUser удаляет существующего пользователя. Для этого требуется
идентификатор пользователя, которого необходимо удалить.
*/
func DeleteUser(id int) error {
    fmt.Println(id)
    return nil
}
```

Вы можете использовать блочные комментарии в любом желаемом месте, а не только в начале пакета.

```
// ListUsers выводит список всех пользователей в базе данных
// и возвращает срез Userdata.
func ListUsers() ([]Userdata, error) {
    // Data содержит записи, возвращаемые SQL-запросом
    Data := []Userdata{}
    return Data, nil
}
```

Когда вы запрашиваете документацию по структуре `Userdata`, Go автоматически представляет функции, которые используют `Userdata` в качестве входных или выходных данных.

```
// UpdateUser предназначена для обновления существующего пользователя,
// заданного структурой Userdata.
// Идентификатор обновляемого пользователя
// находится внутри функции.
func UpdateUser(d Userdata) error {
    fmt.Println(d)
    return nil
}
```

Мы еще не закончили, поскольку нам еще нужно как-то увидеть всю документацию пакета. Есть два способа ознакомиться с ней. Первый предполагает использование `go get`, что также означает создание репозитория пакета на GitHub, как мы это сделали с `post05`. Однако наша цель — тестирование, поэтому мы поступим проще: скопируем ее в `~/go/src` и получим доступ оттуда. Поскольку пакет называется `document`, мы создадим каталог с таким же именем внутри `~/go/src`. После этого скопируем `document.go` в `~/go/src/document`, и на этом все. Для более сложных пакетов процесс также будет более сложным. В таких случаях было бы лучше использовать `go get` и получать пакет из его репозитория.

В любом случае команда `go doc` будет отлично работать с пакетом `document`:

```
$ go doc document
package document // import "document"

The package works on 2 tables on a PostgreSQL data base server.

The names of the tables are:

* Users
* Userdata

The definitions of the tables in the PostgreSQL server are:

CREATE TABLE Users (
    ID SERIAL,
    Username VARCHAR(100) PRIMARY KEY
);

CREATE TABLE Userdata (
    UserID Int NOT NULL,
    Name VARCHAR(100),
    Surname VARCHAR(100),
    Description VARCHAR(200)
);

This is rendered as code

This is not rendered as code

var Hostname = "" ...
func AddUser(d Userdata) int
func DeleteUser(id int) error
func UpdateUser(d Userdata) error
type Userdata struct{ ... }
    func ListUsers() ([]Userdata, error)

BUG: Function ListUsers() not working as expected

BUG: Function AddUser() is too slow
```

Если вы хотите просмотреть информацию о конкретной функции, вам стоит использовать `go doc` следующим образом:

```
$ go doc document ListUsers
package document // import "document"

func ListUsers() ([]Userdata, error)
ListUsers lists all users in the database and returns a slice of
Userdata.
```

Кроме того, мы можем использовать веб-версию документации Go, доступ к которой можно получить после запуска утилиты `godoc` и перехода в раздел `Third Party`, — по умолчанию веб-сервер, инициированный `godoc`, прослушивает порт с номером 6060 и может быть доступен по адресу `http://localhost:6060`.

Часть страницы документации для пакета `document` показана на рис. 5.4.

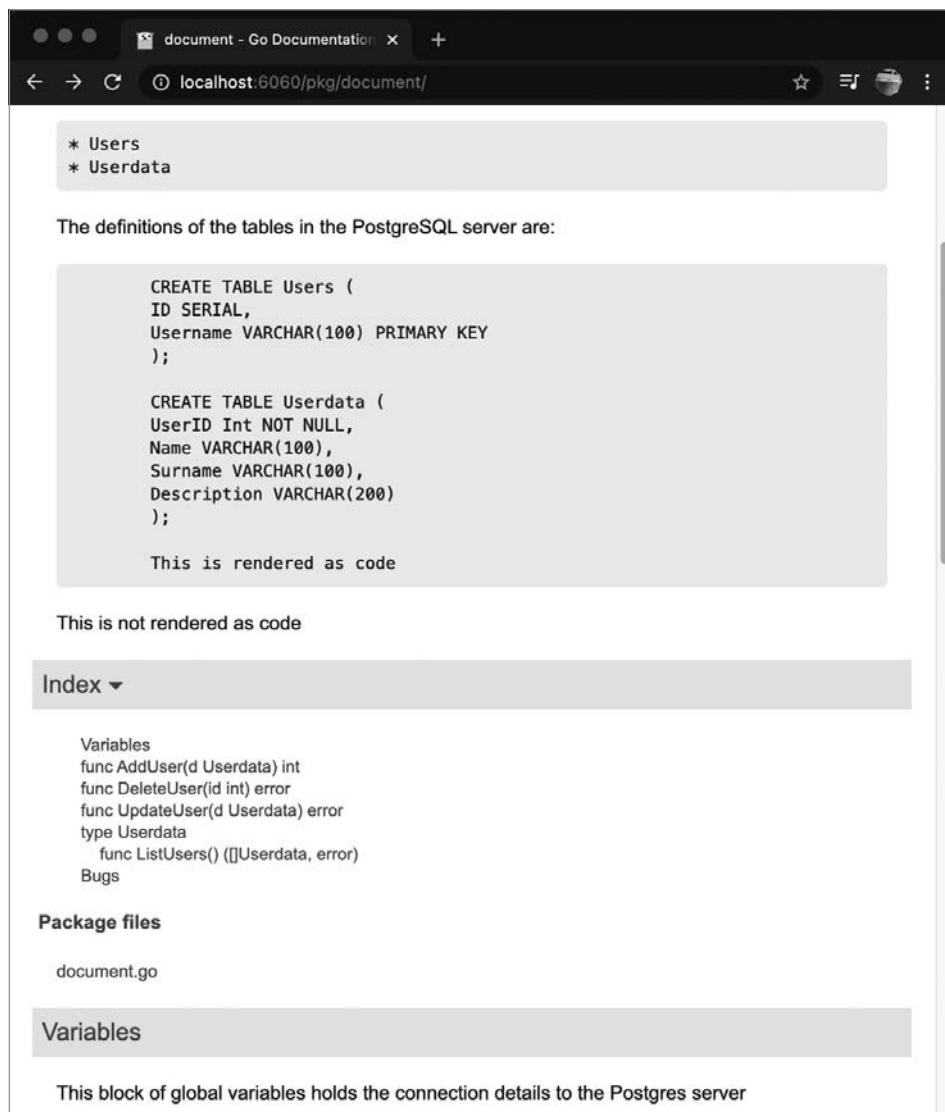


Рис. 5.4. Просмотр документации разработанного пользователем Go-пакета



Go автоматически помещает ошибки в конец текстового и графического вывода.

По моему личному мнению, рендеринг документации намного лучше при использовании графического интерфейса, что удобнее, когда вы не знаете, что ищете. В то же время использование `go doc` из командной строки является намного быстрым и позволяет обрабатывать выходные данные с помощью традиционных инструментов командной строки UNIX.

В следующем разделе кратко представлены системы CI/CD GitLab и GitHub, начиная с GitLab Runners, которые могут быть полезны для автоматизации разработки и развертывания пакетов.

GitLab Runners и Go

При разработке пакетов Go и документации вам нужна возможность протестировать результаты и найти ошибки как можно быстрее. Когда все будет работать ожидаемым образом, вы, вероятно, захотите показать свою работу миру автоматически, не тратя на это больше времени. Одним из лучших решений для этого является использование системы CI/CD для автоматизации задач. В данном разделе кратко показано, как использовать GitLab Runners для автоматизации проектов Go.



Для работы с данным разделом вам необходимо иметь учетную запись GitLab, создать выделенный репозиторий GitLab и хранить там соответствующие файлы.

Мы начнем с репозитория GitLab, который содержит следующие файлы:

- `hw.go` — это пример программы, которую мы используем, чтобы убедиться, что все работает;
- `.gitignore` — не обязательно иметь такой файл, но он очень удобен для игнорирования некоторых файлов и каталогов;
- `usePost05.go` — пример файла Go, который использует внешний пакет. Пожалуйста, обратитесь к репозиторию <https://gitlab.com/mactsouk/runners-go/>, чтобы просмотреть его содержимое;
- `README.md` — этот файл автоматически отображается на веб-странице репозитория и обычно используется для объяснения назначения репозитория.

Существует также каталог `.git`, который содержит информацию и метаданные о репозитории.

Начальная версия файла конфигурации

Первая версия конфигурационного файла предназначена для того, чтобы убедиться, что с нашей настройкой GitLab все в порядке. Имя файла конфигурации — `.gitlab-ci.yml`, и это файл YAML, который должен находиться в корневом каталоге репозитория GitLab. Эта начальная версия конфигурационного файла `.gitlab-ci.yml` компилирует `hw.go` и создает двоичный файл, который выполняется на этапе, отличном от того, на котором он был создан. Это означает, что мы должны создать артефакт для хранения и передачи этого двоичного файла:

```
$ cat .gitlab-ci.yml
image: golang:1.15.7

stages:
  - download
  - execute

compile:
  stage: download
  script:
    - echo "Getting System Info"
    - uname -a
    - mkdir bin
    - go version
    - go build -o ./bin/hw hw.go
  artifacts:
    paths:
      - bin/

execute:
  stage: execute
  script:
    - echo "Executing Hello World!"
    - ls -l bin
    - ./bin/hw
```

Важным моментом в вышеприведенном файле конфигурации является то, что мы используем образ, который поставляется с уже установленным Go. Это избавляет нас от необходимости устанавливать его с нуля и позволяет указать версию Go, которую мы хотим использовать.

Однако если вы когда-нибудь захотите установить дополнительное программное обеспечение, то сможете сделать это в зависимости от используемого

дистрибутива Linux. После сохранения файла вам нужно отправить изменения в GitLab, чтобы конвейер начал работать. Чтобы увидеть результаты, вы должны нажать на опцию CI/CD на левой панели пользовательского интерфейса GitLab.

На рис. 5.5 показана информация о конкретном рабочем процессе, основанном на вышеупомянутом файле YAML. Все, что выделено зеленым цветом, — это хорошо, в то время как красный используется в ситуациях ошибок. Если вы хотите узнать больше информации о конкретном этапе, то можете нажать его кнопку и увидеть более подробный вывод.

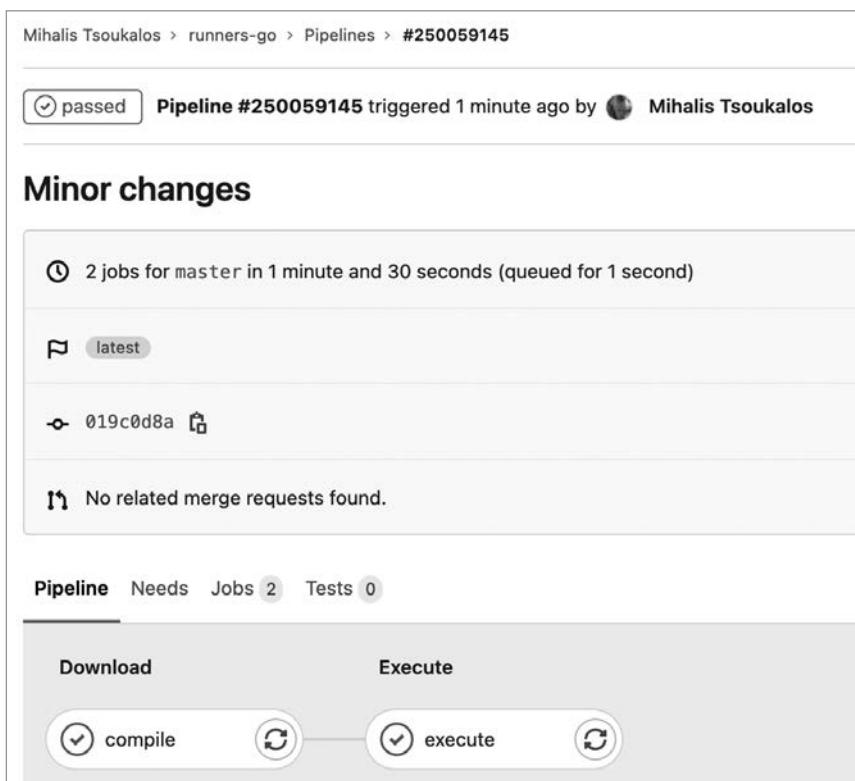


Рис. 5.5. Просмотр прогресса конвейера GitLab

Поскольку все выглядит нормально, мы готовы создать окончательную версию `.gitlab-ci.yml`. Обратите внимание, что если в рабочем процессе возникнет ошибка, то вы, скорее всего, получите электронное письмо на адрес электронной почты, который использовали при регистрации в GitLab. Если все в порядке, то электронное письмо не будет отправлено.

Окончательная версия конфигурационного файла

Окончательная версия конфигурационного файла CI/CD компилирует `usePost05.go`, который импортирует `post05`. Таким образом демонстрируется процесс скачивания внешних пакетов. Содержимое `.gitlab-ci.yml` выглядит следующим образом:

```
image: golang:1.15.7

stages:
  - download
  - execute

compile:
  stage: download
  script:
    - echo "Compiling usePost05.go"
    - mkdir bin
    - go get -v -d ./...
```

Использование команды `go get -v -d ./...` — простой способ загрузки всех зависимостей пакета проекта. После этого вы можете свободно создавать свой проект и генерировать исполняемый файл:

```
  - go build -o ./bin/usePost05 usePost05.go
artifacts:
  paths:
    - bin/
```

Каталог `bin` вместе с его содержимым будет доступен для состояния `execute`:

```
execute:
  stage: execute
  script:
    - echo "Executing usePost05"
    - ls -l bin
    - ./bin/usePost05
```

Отправка его в GitLab автоматически запускает его выполнение. Это показано на рис. 5.6, более подробно отображающем ход этапа `compile`.

Здесь мы видим, что все необходимые пакеты загружаются и `usePost05.go` компилируется без каких-либо проблем. Поскольку у нас нет доступного экземпляра PostgreSQL, мы не можем попробовать взаимодействовать с PostgreSQL, но можем выполнить `usePost05.go` и посмотреть значения глобальных переменных `Hostname` и `Port`. Это показано на рис. 5.7.

До сих пор мы изучали, как использовать GitLab Runners для автоматизации разработки и тестирования пакетов Go. Далее мы собираемся создать сценарий CI/CD в GitHub, используя GitHub Actions в качестве другого метода автоматизации публикации программного обеспечения.

```

11 Fetching changes with git depth set to 50...
12 Initialized empty Git repository in /builds/mactsouk/runners-go/.git/
13 Created fresh repository.
14 Checking out 8ba14df7 as master...
15 Skipping Git submodules setup
16 Executing "step_script" stage of the job script
17 $ echo "Compiling usePost05.go"
18 Compiling usePost05.go
19 $ mkdir bin
20 $ go get -v -d ./...
21 github.com/mactsouk/post05 (download)
22 github.com/lib/pq (download)
23 $ go build -o ./bin/usePost05 usePost05.go
24 Uploading artifacts for successful job
25 Uploading artifacts...
26 bin/: found 2 matching files and directories
27 Uploading artifacts as "archive" to coordinator... ok  id=1001258536 responseStatus=201 Created t
oken=_JMwiwyt
28 Cleaning up file based variables
29 Job succeeded

```

Рис. 5.6. Просмотр детального вида этапа

```

19 Downloading artifacts
20 Downloading artifacts for compile (1001263660)...
21 Downloading artifacts from coordinator... ok      id=1001263660 responseStatus=200 OK token=Skr
LxVpx
22 Executing "step_script" stage of the job script
23 $ echo "Executing usePost05"
24 Executing usePost05
25 $ ls -l bin
26 total 5752
27 -rwxr-xr-x. 1 root root 5883134 Feb  2 07:23 usePost05
28 $ ./bin/usePost05
29 2345
30 localhost
31 localhost
32 Cleaning up file based variables
33 Job succeeded

```

Рис. 5.7. Просмотр более подробной информации о этапе execute

GitHub Actions и Go

В этом разделе GitHub Actions будет использоваться для отправки образа Docker, содержащего исполняемый Go-файл, в Docker Hub.



Чтобы работать с этим разделом, вам нужно создать учетную запись GitHub, создать выделенный репозиторий GitHub и хранить там соответствующие файлы.

Мы начнем с репозитория GitHub, который содержит следующие файлы:

- `.gitignore` — это необязательный файл, который используется для игнорирования файлов и каталогов во время операций `git push`;
- `usePost05.go` — это тот же файл, что и раньше;
- `Dockerfile` — данный файл используется для создания образа Docker с исполняемым Go-файлом. Пожалуйста, обратитесь к <https://github.com/mactsouk/actions-with-go>, чтобы получить его содержимое;
- `README.md` — как и прежде, это файл Markdown, содержащий информацию о репозитории.

Чтобы настроить действия GitHub, нам нужно создать каталог `.github`, а затем создать в нем другой каталог `workflows`. Каталог `.github/workflows` содержит файлы YAML с конфигурацией конвейера.

На рис. 5.8 показан обзорный экран рабочих процессов выбранного репозитория GitHub.

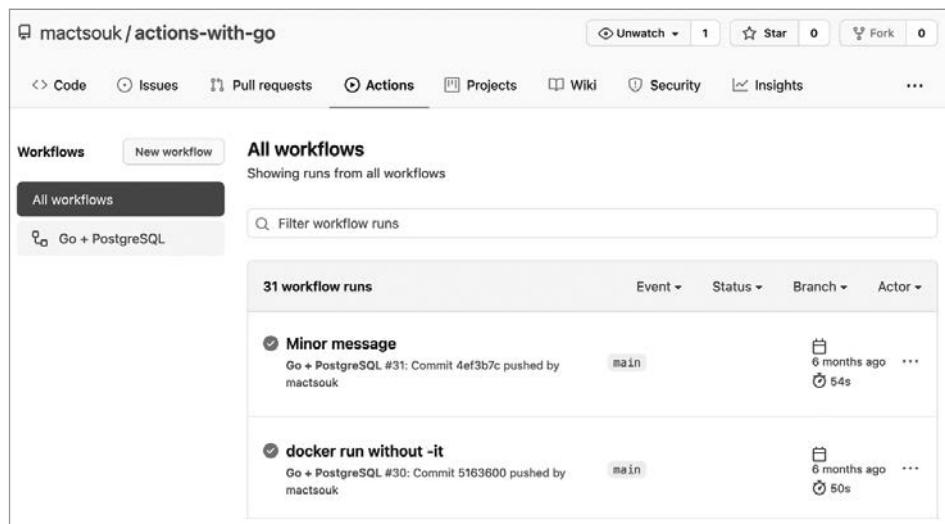


Рис. 5.8. Отображение рабочих процессов, связанных с данным репозиторием GitHub

Чтобы отправить изображение в Docker Hub, вам необходимо войти в систему. Этот процесс требует использования пароля, который является конфиденциальной информацией, и в следующем подразделе показано, как хранить секреты в GitHub.

Хранение секретов в GitHub

Учетные данные для подключения к Docker Hub хранятся в GitHub с использованием функционала секретов, который есть почти во всех системах CI/CD. Однако точная реализация может различаться.



Вы также можете использовать хранилище HashiCorp в качестве основного места хранения паролей и других конфиденциальных данных. К сожалению, рассказ о хранилище HashiCorp выходит за рамки этой книги.

В вашем репозитории GitHub перейдите на вкладку **Settings** и выберите **Secrets** в левой колонке. Вы увидите свои существующие секреты, если таковые имеются, и ссылку **Add new secret**, которую вам необходимо выбрать. Выполните этот процесс дважды, чтобы сохранить ваше имя пользователя и пароль Docker Hub.

На рис. 5.9 показаны секреты, связанные с репозиторием GitHub, используемым в этом разделе. Представленные секреты содержат имя пользователя и пароль, которые использовались для подключения к Docker Hub.

Repository secrets			
	PASSWORD	Updated 7 hours ago	<button>Update</button> <button>Remove</button>
	USERNAME	Updated 7 hours ago	<button>Update</button> <button>Remove</button>

Рис. 5.9. Секреты хранилища [mactsouk/actions-with-go](#)

Окончательная версия конфигурационного файла

Окончательная версия файла конфигурации компилирует Go-код, помещает его в образ Docker, как описано в `Dockerfile`, подключается к Docker Hub, используя указанные учетные данные, и отправляет образ в Docker Hub, используя предоставленные данные. Это очень распространенный способ автоматизации при создании образов Docker. Содержимое `go.yml` выглядит следующим образом:

```
name: Go + PostgreSQL
```

```
on: [push]
```

Эта строка в файле конфигурации указывает, что конвейер запускается только при push-операциях.

```
jobs:  
  build:  
    runs-on: ubuntu-18.04
```

Это образ Linux, который будет использоваться. Поскольку двоичный Go-файл встроен в образ Docker, нам не нужно устанавливать Go на виртуальную машину Linux.

```
steps:  
  # Checks-out your repository under $GITHUB_WORKSPACE,  
  # so your job can access it  
  - uses: actions/checkout@v2  
  - uses: actions/setup-go@v2  
  with:  
    stable: 'false'  
    go-version: '1.15.7'  
  - name: Publish Docker Image  
    env:  
      USERNAME: ${{ secrets.USERNAME }}  
      PASSWORD: ${{ secrets.PASSWORD }}
```

Вот как вы получаете доступ и храните секреты.

```
IMAGE_NAME: gopost  
run: |  
  docker images  
  docker build -t "${IMAGE_NAME}" .  
  docker images  
  echo "$PASSWORD" | docker login --username "$USERNAME"  
  --password-stdin  
  docker tag "${IMAGE_NAME}" "$USERNAME/${IMAGE_NAME}:latest"  
  docker push "$USERNAME/${IMAGE_NAME}:latest"  
  echo "* Running Docker Image"  
  docker run ${IMAGE_NAME}:latest
```

На этот раз большая часть работы выполняется командой `docker build`, поскольку исполняемый файл Go встроен в образ Docker. На рис. 5.10 показаны некоторые выходные данные конвейера, определенного `go.yml`.

Автоматизация экономит ваше время на разработку, поэтому старайтесь автоматизировать как можно больше вещей, особенно когда вы делаете свое программное обеспечение общедоступным.

```
146 WARNING! Your password will be stored unencrypted in /home/runner/.docker
147   /config.json.
148 Login Succeeded
149 Configure a credential helper to remove this warning. See
150   https://docs.docker.com/engine/reference/commandline/login/#credentials-store
151 The push refers to repository [docker.io/**/gopost]
152 327a1f07ee35: Preparing
153 5516549e02ba: Preparing
154 72e830a4dff5: Preparing
155 72e830a4dff5: Mounted from library/alpine
156 5516549e02ba: Pushed
157 327a1f07ee35: Pushed
158 latest: digest:
159 sha256:82beefee80b89ba4a28b2fc7aba11d17d3b3034d4752c257eece18862e9a94ce size:
160 945
161 * Running Docker Image
162 2345
163 localhost
```

Рис. 5.10. Конвейер для отправки образов Docker в Docker Hub

Утилиты управления версиями

Одна из самых сложных задач — как автоматически задать уникальные версии для утилит командной строки, особенно при использовании системы CI/CD. В этом разделе представлен метод, который использует значение GitHub для создания версии утилиты командной строки на вашем локальном компьютере.



Вы можете применить тот же метод к GitLab — просто найдите доступные переменные и значения GitLab и выберите ту, которая соответствует вашим потребностям.

Этот метод используется, среди прочего, как утилитами `docker`, так и `kubectl`:

```
$ docker version
Client: Docker Engine - Community
Cloud integration: 1.0.4
Version:           20.10.0
API version:       1.41
Go version:        go1.13.15
Git commit:        7287ab3
Built:             Tue Dec 8 18:55:43 2020
OS/Arch:           darwin/amd64
...
```

В этом выводе показано, что `docker` использует значение коммита Git для управления версиями — мы собираемся задействовать немного другое значение, которое длиннее применяемого `docker`.

Используемая утилита, которая сохранена как `GitVersion.go`, реализована следующим образом:

```
package main

import (
    "fmt"
    "os"
)

var VERSION string
```

`VERSION` — это переменная, которая будет установлена во время выполнения программы с помощью компоновщика Go.

```
func main() {
    if len(os.Args) == 2 {
        if os.Args[1] == "version" {
            fmt.Println("Version:", VERSION)
        }
    }
}
```

Здесь говорится, что если есть аргумент командной строки и его значение равно `version`, то выведите сообщение о версии с помощью переменной `VERSION`.

Что нам нужно сделать, так это сообщить компоновщику Go, что мы собираемся определить значение переменной `VERSION`. Это делается с помощью флага `-ldflags` (расшифровывается как флаги компоновщика). Он передает значения в пакет `cmd/link`, что позволяет нам изменять значения в импортированных пакетах во время сборки. Используемое значение `-X` требует пары «ключ — значение», где ключ — это имя переменной, а значение — это значение, которое мы хотим установить для данного ключа. В нашем случае ключ имеет форму `main.Variable`, поскольку мы изменяем значение переменной в пакете `main`. Имя переменной в `GitVersion.go` — `VERSION`, поэтому ключ — `main.VERSION`.

Но сначала нам нужно определиться со значением GitHub, которое мы собираемся использовать в качестве строки версии. Команда `git rev-list HEAD` возвращает полный список коммитов для текущего репозитория от самых последних до самых старых. Нам нужен только последний — самый свежий, — который мы можем получить, используя `git rev-list -1 HEAD` или `git rev-list HEAD | head -1`. Итак, нам нужно присвоить это значение переменной среды и передать ее компилятору Go. Поскольку это значение меняется каждый раз при выполнении коммита, а вы всегда хотите иметь последнее значение, следует пересматривать при каждом выполнении `go build`, что вскоре будет продемонстрировано.

Чтобы предоставить `GitVersion.go` значение желаемой переменной среды, мы должны выполнить следующее:

```
$ export VERSION=$(git rev-list -1 HEAD)
$ go build -ldflags "-X main.VERSION=$VERSION" gitVersion.go
```



Это работает как в оболочках `bash`, так и в оболочках `zsh`. Если вы используете другую оболочку, то вам следует убедиться, что вы правильно определяете переменную среды.

Выполнить две команды одновременно можно следующим образом:

```
$ export VERSION=$(git rev-list -1 HEAD) && go build -ldflags "-X main.
VERSION=$VERSION" gitVersion.go
```

При выполнении сгенерированного исполняемого файла, который называется `GitVersion`, мы получаем такой вывод:

```
$ ./gitVersion version
Version: 99745c8fbaff94790b5818edf55f15d309f5bfeb
```

Ваш результат будет различаться, поскольку ваш репозиторий GitHub будет другим. GitHub генерирует случайные и уникальные значения, поэтому у вас никогда не будет повторяющегося номера версии!

Упражнения

- Можете ли вы написать функцию, которая сортирует три значения `int?` Попробуйте написать две версии функции: одну с именованными возвращаемыми значениями и другую без именованных возвращаемых значений. Какой вариант, по-вашему, лучше?
- Перепишите утилиту `GetSchema.go` так, чтобы она работала с пакетом `jackc/pgx`.
- Перепишите утилиту `GetSchema.go` так, чтобы она работала с базами данных MySQL.
- Используйте GitLab CI/CD для отправки образов Docker в Docker Hub.

Резюме

В этой главе были представлены две основные темы: функции и пакеты. Функции — это полноправные обитатели Go, что делает их эффективным и удобным средством. Не забывайте, что все, что начинается с заглавной буквы, общедоступно. Единственным исключением из этого правила являются имена пакетов.

Частные переменные, функции, имена типов данных и поля структуры могут использоваться и вызываться строго внутри пакета, в то время как общедоступные открыты всем. К тому же вы больше узнали о ключевом слове `defer`. Кроме того, запомните, что Go-пакеты не похожи на классы Java: Go-пакет может быть настолько большим, насколько это необходимо. Что касается Go-модулей, то имейте в виду: Go-модуль — это несколько пакетов с версией.

Наконец, в этой главе мы обсуждали создание документации, GitHub Actions и GitLab Runner, а также то, как две системы CI/CD могут помочь вам автоматизировать скучные процессы и присвоить уникальные номера версий вашим утилитам.

В следующей главе мы более подробно рассмотрим системное программирование в целом, а также файловый ввод-вывод.

Дополнительные ресурсы

- Последние изменения модулей в Go 1.16: <https://blog.golang.org/go116-module-changes>.
- Как структурировать свои приложения Go? Выступление Кэт Зиен на GopherCon UK 2018: <https://www.youtube.com/watch?v=1rxDzs0zgcE>.
- PostgreSQL: <https://www.postgresql.org/>.
- PostgreSQL Go package: <https://github.com/lib/pq>.
- PostgreSQL Go package: <https://github.com/jackc/pgx>.
- HashiCorp Vault: <https://www.vaultproject.io/>.
- Документация по базе данных /sql: <https://golang.org/pkg/database/sql/>.
- Больше информации о переменных среды GitHub Actions: <https://docs.github.com/en/actions/reference/environment-variables>.
- Переменные GitLab CI/CD: <https://docs.gitlab.com/ee/ci/variables/>.
- Документация к пакету `cmd/link`: <https://golang.org/cmd/link/>.
- `golang.org` переезжает на `go.dev`: <https://go.dev/blog/tidy-web>.

6

Даем указания системе UNIX

Из этой главы вы узнаете о *системном программировании* в Go. Системное программирование включает в себя работу с файлами и каталогами, управление процессами, обработку сигналов, сетевое программирование, системные файлы, файлы конфигурации, а также файловый ввод и вывод (I/O). Как вы помните из главы 1, причина написания системных утилит с поддержкой Linux заключается в том, что часто программное обеспечение Go выполняется в среде Docker — образы Docker используют операционную систему Linux, а это значит, что вам, вероятно, потребуется *разрабатывать собственные утилиты с учетом этой ОС*. Но поскольку Go-код универсален, большинство системных утилит работают на компьютерах с Windows без каких-либо изменений или с незначительными модификациями. Среди прочего в этой главе реализованы две утилиты, одна из которых находит циклы в файловых системах UNIX, а другая преобразует данные JSON в данные XML и наоборот. Кроме того, в этой главе мы собираемся дополнить приложение телефонной книги с помощью пакета *cobra*.



Важное примечание. Начиная с Go 1.16, переменная среды `GO111MODULE` по умолчанию имеет значение `on` — это влияет на использование Go-пакетов, которые не принадлежат стандартной библиотеке Go. На практике это означает, что вы должны поместить свой код в `~/go/src`. Всегда можно вернуться к предыдущему поведению, установив для `GO111MODULE` значение `auto`, но лучше этого не делать, ведь будущее за модулями. Причина заключается в том, что `viper`, и `cobra` предпочтительнее рассматривать как Go-модули, а не как пакеты, что изменяет процесс разработки, но не код.

В этой главе:

- `stdin`, `stdout` и `stderr`;
- процессы UNIX;
- обработка сигналов UNIX;
- файловый ввод и вывод;
- чтение обычных текстовых файлов;
- запись в файл;
- работа с JSON;
- работа с XML;
- работа с YAML;
- пакет `viper`;
- пакет `cobra`;
- поиск циклов в файловой системе UNIX;
- новое в Go 1.16;
- обновление приложения телефонной книги.

stdin, stdout и stderr

Каждая операционная система UNIX содержит три файла, постоянно открытых для ее процессов. Помните, что UNIX рассматривает все (даже принтер или мышь) как файлы. UNIX использует *файловые дескрипторы*, которые представляют собой положительные целочисленные значения, в качестве внутреннего представления для доступа к открытым файлам, что гораздо изящнее использования длинных путей. Итак, по умолчанию все системы UNIX поддерживают три специальных и стандартных имени файлов: `/dev/stdin`, `/dev/stdout` и `/dev/stderr`, к которым также можно получить доступ с помощью файловых дескрипторов 0, 1 и 2 соответственно. Эти три файловых дескриптора также называются *стандартным вводом*, *стандартным выводом* и *стандартной ошибкой* соответственно. Кроме того, к файловому дескриптору 0 можно получить доступ как к `/dev/fd/0` на компьютере с macOS и как к `/dev/fd/0` и `/dev/pts/0` на компьютере с Debian Linux.

Go использует `OC.Stdin`, `os.Stdout` и `os.Stderr` для доступа к стандартному вводу, стандартному выводу и стандартной ошибке соответственно. Хотя вы можете применять `/dev/stdin`, `/dev/stdout` и `/dev/stderr` или соответствующие значения файловых дескрипторов для доступа к этим же устройствам, будет лучше, безопаснее и полезнее для совместимости придерживаться использования `OC.Stdin`, `os.Stdout` и `os.Stderr`.

Процессы UNIX

Поскольку Go-серверы, Go-утилиты и образы Docker в основном выполняются в Linux, полезно знать о процессах и потоках Linux.

Строго говоря, *процесс* — это среда выполнения, которая содержит инструкции, пользовательские данные и части системных данных, а также другие типы ресурсов, получаемые во время выполнения. В то время как *программа* — это двоичный файл, содержащий инструкции и данные, которые служат для инициализации частей процесса, связанных с инструкциями и пользовательскими данными. Каждый запущенный процесс UNIX уникально идентифицируется целым числом без знака, которое называется *идентификатором процесса*.

Существуют три категории процессов: *пользовательские процессы*, *демоны* и *процессы ядра*. Пользовательские процессы выполняются в пользовательском пространстве и обычно не имеют специальных прав доступа. Демоны — это программы, которые можно найти в пользовательском пространстве и запускать в фоновом режиме без помощи терминала. Процессы ядра выполняются только в пространстве ядра и имеют полный доступ ко всем структурам данных ядра.

Способ создания новых процессов на языке C включает в себя использование системного вызова `fork(2)`. Возвращаемое значение `fork(2)` позволяет программисту различать родительский и дочерний процессы. Вы можете создать новый процесс в Go с помощью пакета `exec`, но Go не позволит вам управлять потоками. Вместо этого язык предлагает *горутины*, создаваемые пользователем поверх потоков, которые создаются и обрабатываются средой выполнения Go.

Обработка сигналов UNIX

Сигналы UNIX предлагают очень удобный способ *асинхронного взаимодействия с вашими приложениями*. Однако обработка сигналов UNIX требует использования Go-каналов, которые применяются исключительно для этой задачи. Таким образом, было бы неплохо немного поговорить о модели параллелизма в Go, которая требует использования горутин и каналов для обработки сигналов.

Горутина — это наименьшая исполняемая сущность в Go. Чтобы создать новую горутину, вы должны использовать ключевое слово `go`, за которым следует предопределенная или анонимная функция — методы эквивалентны. *Канал* в Go — это механизм, который, помимо прочего, позволяет горутинам взаимодействовать и обмениваться данными. Если вы программист-любитель или впервые слышите о горутинах и каналах, то не паникуйте. О них мы гораздо более подробно поговорим в главе 7.



Чтобы горутина или функция завершила работу всего Go-приложения, она должна вызвать `os.Exit()` вместо `return`. Однако в большинстве случаев вам следует выходить из горутины или функции с помощью `return`, поскольку чаще всего необходимо выйти из конкретной подпрограммы или функции, а не останавливать все приложение.

Представленная программа по отдельности обрабатывает `SIGINT`, который в Go именуется `syscall.SIGINT`, и `SIGINFO` и использует `default` в блоке `switch` для обработки остальных случаев (других сигналов). Реализация этого блока позволит различать сигналы в соответствии с вашими потребностями.

Существует *выделенный канал*, который принимает все сигналы, что определено функцией `signal.Notify()`. Go-каналы могут обладать пропускной способностью. Пропускная способность этого конкретного канала равна 1, что дает возможность принимать и сохранять один сигнал за раз. Это имеет смысл, поскольку один сигнал может завершить программу и нет необходимости пытаться одновременно обработать другой. Обычно присутствует анонимная функция, которая выполняется как горутина и лишь обрабатывает сигнал. Основная задача этой горутины — прослушивать канал для получения данных. Как только сигнал получен, он отправляется в этот канал, считывается горутиной и сохраняется в переменной — в этот момент канал может принимать следующие сигналы. Эта переменная обрабатывается оператором `switch`.



Некоторые сигналы не могут быть перехвачены, и операционная система не может их игнорировать. Таким образом, сигналы `SIGKILL` и `SIGSTOP` не могут быть заблокированы, перехвачены или проигнорированы, и причина этого в том, что они позволяют привилегированным пользователям, а также ядру UNIX завершать любой процесс, который они пожелают.

Создайте текстовый файл и введите следующий код. Хорошим именем для этого файла будет `signals.go`:

```
package main

import (
    "fmt"
    "os"
    "os/signal"
    "syscall"
    "time"
)
func handleSignal(sig os.Signal) {
    fmt.Println("handleSignal() Caught:", sig)
}
```

Функция `handleSignal()` — отдельная и предназначена для обработки сигналов. Однако вы также можете обрабатывать сигналы *внутри* (*inline*), в ветках оператора `switch`.

```
func main() {
    fmt.Printf("Process ID: %d\n", os.Getpid())
    sigs := make(chan os.Signal, 1)
```

Мы создаем канал с данными типа `os.Signal`, поскольку все каналы должны иметь определенный тип.

```
    signal.Notify(sigs)
```

Вышеприведенный оператор означает следующее: обрабатывать все сигналы, которые могут быть обработаны.

```
    start := time.Now()
    go func() {
        for {
            sig := <-sigs
```

Ждем, пока будут прочитаны данные (`<-`) из канала `sigs`, и сохраняем их в переменной `sig`.

```
        switch sig {
```

В зависимости от считанного значения действуем соответствующим образом. Вот как мы различаем сигналы.

```
    case syscall.SIGINT:
        duration := time.Since(start)
        fmt.Println("Execution time:", duration)
```

Для обработки `syscall.SIGINT` мы вычисляем время, прошедшее с начала выполнения программы, и выводим его на экран.

```
    case syscall.SIGINFO:
        handleSignal(sig)
```

Код блока `syscall.SIGINFO` вызывает функцию `handleSignal()` — разработчик должен сам принять решение о деталях реализации. На компьютерах с Linux вам следует заменить `syscall.SIGINFO` другим сигналом, таким как `syscall.SIGUSR1` или `syscall.SIGUSR2`, поскольку `syscall.SIGINFO` недоступен в Linux (<https://github.com/golang/go/issues/1653>).

```
// Не используйте здесь return, так какgorутина завершает
// работу, но time.Sleep() продолжит работать!
os.Exit(0)
```

```
    default:  
        fmt.Println("Caught:", sig)  
    }
```

Если совпадения нет, то блок `default` обрабатывает остальные значения и просто выводит сообщение.

```
    }  
}  
  
for {  
    fmt.Print("+")  
    time.Sleep(10 * time.Second)  
}  
}
```

Бесконечный цикл `for` в конце функции `main()` предназначен для эмуляции работы реальной программы. Без него программа завершается почти сразу.

При выполнении `signals.go` и взаимодействии с ним мы получаем такой вывод:

```
$ go run signals.go  
Process ID: 74252  
+Execution time: 9.989863093s  
+Caught: user defined signal 1  
+signal: killed
```

Вторая строка вывода была сгенерирована путем нажатия `Ctrl+C` на клавиатуре, которая на машинах UNIX отправляет программе сигнал `syscall.SIGINT`. Третья строка вывода была вызвана путем выполнения `kill -USR1 74252` на другом терминале.

Последняя строка в выходных данных была сгенерирована командой `kill -9 74252`. Поскольку сигнал `KILL`, который также представлен значением 9, не может быть обработан, он завершает программу, и оболочка выдает сообщение `killed`.

Обработка двух сигналов. Если вы хотите обрабатывать ограниченное количество сигналов, а не все из них, то вам следует заменить оператор `signal.Notify(sigs)` на следующий:

```
signal.Notify(sigs, syscall.SIGINT, syscall.SIGINFO)
```

После этого вам необходимо внести соответствующие изменения в код подпрограммы, ответственной за обработку сигналов, чтобы идентифицировать и обработать `syscall.SIGINT` и `syscall.SIGINFO` — текущая версия (`signals.go`) уже обрабатывает их.

Теперь нам нужно научиться читать и записывать файлы в Go.

Файловый ввод-вывод

В этом разделе мы обсудим файловый ввод-вывод в Go, в который входит использование интерфейсов `io.Reader` и `io.Writer`, буферизованный и небуферизованный ввод-вывод, а также пакет `bufio`.



Пакет `io/ioutil` (<https://golang.org/pkg/io/ioutil/>) устарел в Go версии 1.16. Существующий Go-код, который использует функциональность `io/ioutil`, продолжит работать, но лучше прекратить использование этого пакета.

Интерфейсы `io.Reader` и `io.Writer`

В этом подразделе представлены определения популярных интерфейсов `io.Reader` и `io.Writer`, поскольку они являются основой файлового ввода-вывода в Go. Первый позволяет вам читать из файла, тогда как второй дает возможность записывать в файл. Определение интерфейса `io.Reader` выглядит следующим образом:

```
type Reader interface {
    Read(p []byte) (n int, err error)
}
```

Это определение, к которому нужно вернуться, если мы захотим, чтобы один из наших типов данных удовлетворял интерфейсу `io.Reader`. Оно сообщает нам следующее:

- интерфейс `Reader` требует реализации единственного метода;
- параметр `Read()` представляет собой байтовый срез;
- возвращаемые значения `Read()` представляют собой целое число и ошибку.

Метод `Read()` принимает в качестве входных данных байтовый срез, который будет заполнен данными *согласно его длине*, и возвращает количество прочитанных байтов, а также переменную `error`.

Определение интерфейса `io.Writer` выглядит следующим образом:

```
type Writer interface {
    Write(p []byte) (n int, err error)
}
```

К определению выше следует вернуться, если мы захотим, чтобы один из наших типов данных удовлетворял интерфейсу `io.Writer` в целях возможности записи в файл. Определение дает нам такую информацию:

- интерфейс требует реализации одного метода;
- параметр `Write()` представляет собой байтовый срез;
- возвращаемые значения `Write()` представляют собой целое число и значение `error`.

Метод `Write()` принимает в качестве входных данных байтовый срез из данных, которые нужно записать, и возвращает количество записанных байтов и переменную `error`.

Правильное и неправильное использование `io.Reader` и `io.Writer`

В приведенном ниже коде демонстрируется использование интерфейсов `io.Reader` и `io.Writer` для пользовательских типов данных, которые здесь представляют собой две Go-структурь: `S1` и `S2`.

Для структуры `S1` представленный код реализует оба интерфейса для считывания пользовательских данных из терминала и вывода их в терминал соответственно. Хотя это излишне, поскольку у нас уже есть `fmt.Scanln()` и `fmt.Printf()`, но это хорошее упражнение, которое показывает, насколько универсальны и гибки оба интерфейса. В другой ситуации вы могли бы использовать `io.Writer` для записи в сервис ведения журнала или сохранения резервной копии записанных данных или чего-нибудь еще, что соответствует вашим потребностям. Однако это также пример интерфейсов, позволяющих вам делать необычные или, если хотите, сумасшедшие вещи, — только от разработчика зависит желаемая функциональность и использование соответствующих концепций и функций Go!

Метод `Read()` использует `fmt.Scanln()` для получения пользовательского ввода с терминала, в то время как метод `Write()` с помощью `fmt.Printf()` выводит содержимое своего параметра буфера столько раз, сколько указано в поле `F1` структуры.

Для структуры `S2` представленный код реализует `io.Reader` только традиционным способом. Метод `Read()` считывает поле `text` структуры `S2`, представляющее собой байтовый срез. Когда считывать больше нечего, метод `Read()` ожидаемо возвращает `io.EOF`, которая на самом деле является не ошибкой, а ожидаемой ситуацией. Наряду с методом `Read()` существуют два вспомогательных метода: `eof()`, который объявляет, что считывать больше нечего, и `ReadByte()`, который байт за байтом считывает поле `text` структуры `S2`. После выполнения метода `Read()` поле `text` структуры `S2`, которое используется в качестве буфера, очищается.

С помощью этой реализации `io.Reader` для `S2` можно использовать для чтения традиционным способом, который в данном случае осуществляется с помощью `bufio.NewReader()` и нескольких вызовов `Read()` — количество вызовов зависит от размера используемого буфера, который в нашем случае представляет собой байтовый срез с двумя разделами для данных.

Ведите следующий код и сохраните его как `ioInterface.go`:

```
package main

import (
    "bufio"
    "fmt"
    "io"
)
```

Здесь показано, что для работы с файлами мы используем пакеты `io` и `bufio`.

```
type S1 struct {
    F1 int
    F2 string
}

type S2 struct {
    F1 S1
    text []byte
}
```

С этими двумя структурами мы и собираемся работать.

```
// используем указатель на S1 для сохранения изменений
// при завершении метода
func (s *S1) Read(p []byte) (n int, err error) {
    fmt.Print("Give me your name: ")
    fmt.Scanln(&p)
    s.F2 = string(p)
    return len(p), nil
}
```

Здесь мы реализуем интерфейс `io.Reader()` для `S1`.

```
func (s *S1) Write(p []byte) (n int, err error) {
    if s.F1 < 0 {
        return -1, nil
    }

    for i := 0; i < s.F1; i++ {
        fmt.Printf("%s ", p)
    }
}
```

```

    fmt.Println()
    return s.F1, nil
}

```

В этом методе мы реализуем `io.Writer` для `S1`.

```

func (s *S2) eof() bool {
    return len(s.text) == 0
}

func (s *S2) readByte() byte {
    // эта функция предполагает, что проверка eof() была выполнена ранее
    temp := s.text[0]
    s.text = s.text[1:]
    return temp
}

```

Данная функция представляет собой реализацию `bytes.Buffer.ReadByte` из стандартной библиотеки.

```

func (s *S2) Read(p []byte) (n int, err error) {
    if s.eof() {
        err = io.EOF
        return
    }

    l := len(p)
    if l > 0 {
        for n < l {

```

Вышеприведенные функции считывают данные из буфера до тех пор, пока он не опустеет. Здесь мы реализуем `io.Reader` для `S2`.

```

        p[n] = s.readByte()
        n++
        if s.eof() {
            s.text = s.text[0:0]
            break
        }
    }
    return
}

```

Когда все данные считаны, соответствующее поле структуры очищается. Вышеприведенный метод реализует `io.Reader` для `S2`. Однако операция `Read()` поддерживается `eof()` и `ReadByte()`, которые также определяются пользователем.

Напомним, что Go позволяет вам именовать возвращаемые значения функций, — в этом случае оператор `return` без каких-либо дополнительных аргументов

автоматически возвращает текущее значение каждой именованной возвращаемой переменной в том порядке, в котором они отображаются в сигнатуре функции. Метод `Read()` использует эту функцию.

```
func main() {
    s1var := S1{4, "Hello"}
    fmt.Println(s1var)
```

Мы инициализируем переменную `S1` с именем `s1var`.

```
buf := make([]byte, 2)
_, err := s1var.Read(buf)
```

Эта строка читает переменную `s1var` с использованием буфера с 2 байтами.

```
if err != nil {
    fmt.Println(err)
    return
}
fmt.Println("Read:", s1var.F2)
_, _ = s1var.Write([]byte("Hello There!"))
```

В этой строке мы вызываем метод `Write()` для `s1var`, чтобы записать содержимое байтового среза.

```
s2var := S2{F1: s1var, text: []byte("Hello world!!")}
```

Здесь мы инициализировали переменную `S2` с именем `s2var`.

```
// читать s2var.text
r := bufio.NewReader(&s2var)
```

Теперь мы создаем считыватель для `s2var`.

```
for {
    n, err := r.Read(buf)
    if err == io.EOF {
        break
    }
}
```

Мы продолжаем чтение из `s2var` до тех пор, пока не возникнет состояние `io.EOF`.

```
} else if err != nil {
    fmt.Println("*", err)
    break
}
fmt.Println(***, n, string(buf[:n]))
```

При выполнении `ioInterface.go` мы получаем такой вывод:

```
$ go run ioInterface.go
{4 Hello}
```

Первая строка выходных данных выводит содержимое переменной `s1var`.

```
Give me your name: Mike
```

Вызов метода `Read()` переменной `s1var`.

```
Read: Mike
Hello There! Hello There! Hello There! Hello There!
```

Предыдущая строка — это вывод `s1var.Write([]byte("Hello There!"))`.

```
** 2 He
** 2 ll
** 2 o
** 2 wo
** 2 rl
** 2 d!
** 1 !
```

В последней части выходных данных показан процесс чтения с помощью буфера, размер которого равняется 2. В следующем разделе мы рассмотрим буферизованные и небуферизованные операции.

Буферизованный и небуферизованный файловый ввод-вывод

Буферизованный ввод-вывод файлов — это использование буфера для временного хранения данных перед чтением или записью данных. Таким образом, вместо того чтобы читать файл побайтово, вы читаете сразу множество данных. Вы помещаете данные в буфер и ожидаете, пока кто-нибудь их не прочитает желаемым образом.

Небуферизованный ввод-вывод файлов происходит, когда буфер для временного хранения данных не используется перед их фактическим чтением или записью, что может повлиять на производительность ваших программ.

Следующий вопрос, который вы можете задать, заключается в том, как решить, когда использовать буферизованный, а когда небуферизованный файловый ввод-вывод. При работе с критически важными данными небуферизованный файловый ввод-вывод, как правило, является лучшим выбором, поскольку буферизованное чтение может привести к использованию устаревших данных, а небуферизованная запись — к потере данных при прерывании питания вашего компьютера. Однако в большинстве случаев однозначного ответа на этот вопрос нет. Это означает, что вы можете использовать все, что облегчает реализацию ваших задач. Однако имейте в виду, что *буферизованные считыватели также могут улучшить производительность* за счет уменьшения количества

системных вызовов, необходимых для чтения из файла или сокета. Реально повлиять на производительность может то, что программист решит использовать.

Есть также пакет `bufio`. Как следует из названия, `bufio` — это буферизованный ввод-вывод. Внутренне пакет `bufio` реализует интерфейсы `io.Reader` и `io.Writer`, для которых создается обертка, что в итоге приводит к появлению `bufio.Reader` и `bufio.Writer` соответственно. Пакет `bufio` очень популярен при работе с обычными текстовыми файлами, и вы увидите, как он функционирует, в следующем разделе.

Чтение текстовых файлов

В этом разделе вы узнаете, как читать обычные текстовые файлы, а также как использовать UNIX-устройство `/dev/random`, которое предлагает вам способ получения случайных чисел.

Чтение текстового файла построчно

Функция для чтения файла построчно находится в `byLine.go` и называется `lineByLine()`. Метод чтения текстового файла построчно также используется при чтении обычного текстового файла по словам, а также при чтении обычного текстового файла посимвольно, поскольку обычно вы обрабатываете текстовые файлы построчно. Представленная утилита выводит каждую прочитанную строку, что делает ее упрощенной версией утилиты `cat(1)`.

Сначала нужно создать новое средство чтения для требуемого файла, используя вызов `bufio.NewReader()`. Затем вы применяете это средство чтения с помощью `bufio.ReadString()`, чтобы прочитать входной файл построчно. Хитрость состоит в использовании параметра `bufio.ReadString()`, который представляет собой символ, дающий `bufio.ReadString()` указание продолжать чтение до тех пор, пока этот символ не будет найден. Постоянный вызов `bufio.ReadString()`, когда этим параметром является символ новой строки (`\n`), приводит к чтению входного файла построчно.

Реализация `lineByLine()` выглядит так:

```
func lineByLine(file string) error {
    f, err := os.Open(file)
    if err != nil {
        return err
    }
    defer f.Close()

    r := bufio.NewReader(f)
```

Убедившись, что можно открыть данный файл для чтения (`os.Open()`), мы создаем новое средство чтения с помощью `bufio.NewReader()`.

```
for {
    line, err := r.ReadString('\n')
```

`bufio.ReadString()` возвращает два значения: прочитанную строку и переменную `error`.

```
if err == io.EOF {
    break
} else if err != nil {
    fmt.Printf("error reading file %s", err)
    break
}
fmt.Println(line)
```

Использование `fmt.Println()` вместо `fmt.Println()` для вывода строки ввода показывает, что символ новой строки включен в каждую строку ввода.

```
}
```

```
return nil
}
```

При выполнении `byLine.go` мы получаем такой вывод:

```
$ go run byLine.go ~/csv.data
Dimitris,Tsoukalos,2101112223,1600665563
Mihalis,Tsoukalos,2109416471,1600665563
Jane,Doe,0800123456,1608559903
```

В этом выводе показано содержимое `~/csv.data`, представленное построчно с помощью `byLine.go`. В следующем подразделе мы поговорим о том, как читать обычный текстовый файл слово за словом.

Чтение текстового файла слово за словом

Чтение простого текстового файла слово за словом — единственная полезная функция для работы с файлом, поскольку обычно требуется обрабатывать файл по одному слову. Это показано в данном подразделе с помощью кода в `byWord.go`. Желаемая функциональность реализована в функции `wordByWord()`. Функция `wordByWord()` использует *регулярные выражения* для разделения слов, найденных в каждой строке входного файла. Регулярное выражение, определенное в операторе `regexp.MustCompile("[^\s]+")`, гласит, что для отделения одного слова от другого мы используем символ пробела.

Реализация функции `wordByWord()` выглядит следующим образом:

```
func wordByWord(file string) error {
    f, err := os.Open(file)
    if err != nil {
        return err
    }
    defer f.Close()

    r := bufio.NewReader(f)
    for {
        line, err := r.ReadString('\n')
        if err == io.EOF {
            break
        } else if err != nil {
            fmt.Printf("error reading file %s", err)
            return err
        }
    }

    r := regexp.MustCompile("[^\s]+")
```

Это то место, где мы определяем регулярное выражение, которое хотим использовать.

```
words := r.FindAllString(line, -1)
```

Здесь мы используем регулярное выражение для разделения переменной `line` на поля.

```
for i := 0; i < len(words); i++ {
    fmt.Println(words[i])
}
```

Этот цикл `for` просто выводит поля среза `words`. Если вы хотите узнать количество слов в строке ввода, то можете просто посмотреть значение вызова `len(words)`.

```
}
```

При выполнении `byWord.go` мы получаем такой вывод:

```
$ go run byWord.go ~/csv.data
Dimitris,Tsoukalos,2101112223,1600665563
Mihalis,Tsoukalos,2109416471,1600665563
Jane,Doe,0800123456,1608559903
```

Поскольку `~/csv.data` не содержит никаких пробелов, каждая строка считается одним словом!

Чтение текстового файла символ за символом

В этом подразделе вы узнаете, как читать текстовый файл посимвольно, что довольно редкое требование, если только вы не хотите разработать текстовый редактор. Вы берете каждую прочитанную строку и разделяете ее, используя цикл `for` с `range`, который возвращает два значения. Вы отбрасываете первое (местоположение текущего символа в переменной строки) и используете второе. Однако это значение является руной, следовательно, вы должны преобразовать его в символ с помощью `string()`.

Реализация `charByChar()`:

```
func charByChar(file string) error {
    f, err := os.Open(file)
    if err != nil {
        return err
    }
    defer f.Close()

    r := bufio.NewReader(f)
    for {
        line, err := r.ReadString('\n')
        if err == io.EOF {
            break
        } else if err != nil {
            fmt.Printf("error reading file %s", err)
            return err
        }

        for _, x := range line {
            fmt.Println(string(x))
        }
    }
    return nil
}
```

Обратите внимание: из-за использования оператора `fmt.Println(string(x))` каждый символ выводится в отдельной строке; это значит, что выходные данные программы будут объемными. Если вы хотите получить более сжатый вывод, то стоит использовать `fmt.Print()`.

При выполнении файла `byCharacter.go` и фильтрации его с помощью `head(1)` без каких-либо параметров мы получаем такой вывод:

```
$ go run byCharacter.go ~/csv.data | head
D
...
,
T
```

Использование утилиты `head(1)` без каких-либо параметров ограничивает вывод всего десятью строками.

Следующий подраздел посвящен чтению из `/dev/random`, который является системным файлом UNIX.

Чтение из `/dev/random`

В этом подразделе вы узнаете, как выполнять чтение с системного устройства `/dev/random`. Назначение системного устройства `/dev/random` — генерация случайных данных, которые можно использовать для тестирования своих программ или, в данном случае, в качестве исходных данных для генератора случайных чисел. Получение данных из `/dev/random` может показаться сложным, и это основная причина, по которой мы специально обсуждаем данный процесс.

Код `devRandom.go` следующий:

```
package main

import (
    "encoding/binary"
    "fmt"
    "os"
)

func main() {
    f, err := os.Open("/dev/random")
    defer f.Close()

    if err != nil {
        fmt.Println(err)
        return
    }

    var seed int64
    binary.Read(f, binary.LittleEndian, &seed)
    fmt.Println("Seed:", seed)
}
```

Есть два представления, *прямой порядок* и *обратный порядок*, которые имеют отношение к *порядку байтов* во внутреннем представлении. В нашем случае мы используем прямой порядок. *Порядок* здесь имеет отношение к тому, как различные вычислительные системы упорядочивают несколько байтов информации.



Реальный пример порядка — это то, как в разных языках по-разному читается текст: европейские языки, как правило, предполагают чтение слева направо, в то время как арабские тексты читаются справа налево.

В представлении с обратным порядком байтычитываются слева направо, в то время как прямой порядок подразумевает считывание справа налево. Для значения `0x01234567`, хранение которого требует четырех байтов, представление с обратным порядком равно `01 | 23 | 45 | 67`, тогда как прямой порядок выглядит как `67 | 45 | 23 | 01`.

При выполнении `devRandom.go` мы получаем такой вывод:

```
$ go run devRandom.go
Seed: 422907465220227415
```

Это означает, что использование устройства `/dev/random` — хороший способ получения случайных данных, включая начальное значение для вашего генератора случайных чисел.

Считывание определенного объема данных из файла

В этом подразделе рассказывается, как считывать определенный объем данных из файла. Описанная утилита может пригодиться, когда вы хотите просмотреть небольшую часть файла. Числовое значение, заданное в качестве аргумента командной строки, определяет размер буфера, который будет использоваться для чтения. Наиболее важной частью кода `readSize.go` является реализация функции `readSize()`:

```
func readSize(f *os.File, size int) []byte {
    buffer := make([]byte, size)
    n, err := f.Read(buffer)
```

Все волшебство происходит в определении переменной `buffer`, поскольку именно здесь мы определяем максимальный объем данных, которые хотим прочитать. Следовательно, каждый раз, когда вы вызываете `readSize()`, функция будет считывать из `f` не более `size` символов.

```
// io.EOF является особым случаем и рассматривается отдельно
if err == io.EOF {
    return nil
}
```

```

if err != nil {
    fmt.Println(err)
    return nil
}
return buffer[0:n]
}

```

Оставшийся код касается условий ошибки; `io.EOF` — это специальное и ожидаемое условие, которое должно обрабатываться отдельно и возвращать в вызывающую функцию байтовый срез прочитанных символов.

При выполнении `readSize.go` мы получаем такой вывод:

```
$ go run readSize.go 12 readSize.go
package main
```

В этом случае мы считываем 12 символов из самого `readSize.go` из-за параметра `12`.

Теперь, когда вы узнали, как читать из файлов, пришло время научиться в них записывать.

Запись в файл

До сих пор мы работали со способами чтения файлов. В этом разделе показано, как записывать данные в файлы четырьмя различными способами и как добавлять данные в уже существующий файл. Код `WriteFile.go` выглядит следующим образом:

```

package main

import (
    "bufio"
    "fmt"
    "io"
    "os"
)

func main() {
    buffer := []byte("Data to write\n")
    f1, err := os.Create("/tmp/f1.txt")
}
```

`os.Create()` возвращает `*os.File`, связанный с путем к файлу, который передается в качестве параметра. Обратите внимание, что если файл уже существует, то `os.Create()` обрезает его.

```

if err != nil {
    fmt.Println("Cannot create file", err)
    return
}
```

```
    }
    defer f1.Close()
    fmt.Fprintf(f1, string(buffer))
```

Функция `fmt.Fprintf()`, для которой требуется переменная `string`, помогает вам записывать данные в ваши собственные файлы, используя нужный формат. Единственное требование — наличие интерфейса `io.Writer`, в который требуется запись. В этом случае в дело вступает действительная переменная `*os.File`, которая удовлетворяет данному интерфейсу.

```
f2, err := os.Create("/tmp/f2.txt")
if err != nil {
    fmt.Println("Cannot create file", err)
    return
}
defer f2.Close()
n, err := f2.WriteString(string(buffer))

os.WriteString() записывает содержимое строки в действительную переменную
*os.File.

fmt.Printf("wrote %d bytes\n", n)

f3, err := os.Create("/tmp/f3.txt")
```

Здесь мы самостоятельно создаем временный файл. Позже в главе вы узнаете о том, как с помощью `os.CreateTemp()` создавать временные файлы.

```
if err != nil {
    fmt.Println(err)
    return
}
w := bufio.NewWriter(f3)
```

Эта функция возвращает `bufio.Writer`, который удовлетворяет интерфейсу `io.Writer`.

```
n, err = w.WriteString(string(buffer))
fmt.Printf("wrote %d bytes\n", n)
w.Flush()

f := "/tmp/f4.txt"
f4, err := os.Create(f)
if err != nil {
    fmt.Println(err)
    return
}
defer f4.Close()

for i := 0; i < 5; i++ {
    n, err = io.WriteString(f4, string(buffer))
    if err != nil {
```

```

        fmt.Println(err)
        return
    }
    fmt.Printf("wrote %d bytes\n", n)
}

// добавление в файл
f4, err = os.OpenFile(f, os.O_APPEND|os.O_CREATE|os.O_WRONLY, 0644)

```

`os.OpenFile()` предоставляет лучший способ создать или открыть файл для записи. `os.O_APPEND` сообщает, что если файл уже существует, то вы должны добавить к нему, а не обрезать его. `os.O_CREATE` сообщает, что если файл еще не существует, то его следует создать. Наконец, `os.O_WRONLY` сообщает, что программа должна открывать файл только для записи.

```

if err != nil {
    fmt.Println(err)
    return
}
defer f4.Close()

// для Write() требуется байтовый срез
n, err = f4.Write([]byte("Put some more data at the end.\n"))

```

Метод `Write()` получает свои входные данные из среза байтов, который является способом записи в Go. Все вышеописанные методы использовали строки; это не лучший способ, особенно при работе с двоичными данными. Однако использовать строки вместо байтовых срезов более практично, поскольку удобнее манипулировать значениями `string`, чем элементами байтового среза, особенно при работе с символами Юникода. С другой стороны, использование значений `string` увеличивает выделение памяти и может привести к большей нагрузке на сборщик мусора.

```

if err != nil {
    fmt.Println(err)
    return
}
fmt.Printf("wrote %d bytes\n", n)
}

```

При выполнении `WriteFile.go` мы получаем некоторую информацию о байтах, записанных на диск. Что действительно интересно, так это файлы, созданные в папке `/tmp`:

```
$ ls -l /tmp/f?.txt
-rw-r--r-- 1 mtsouk  wheel  14 Feb 27 19:44 /tmp/f1.txt
-rw-r--r-- 1 mtsouk  wheel  14 Feb 27 19:44 /tmp/f2.txt
-rw-r--r-- 1 mtsouk  wheel  14 Feb 27 19:44 /tmp/f3.txt
-rw-r--r-- 1 mtsouk  wheel 101 Feb 27 19:44 /tmp/f4.txt
```

В этом выводе показано, что в `f1.txt`, `f2.txt` и `f3.txt` был записан одинаковый объем информации; это значит, что представленные методы записи эквивалентны.

В следующем разделе показано, как работать с данными JSON в Go.

Работа с JSON

В стандартную библиотеку Go входит `encoding/json`, предназначенная для работы с данными JSON. Кроме того, Go позволяет добавить поддержку полей JSON в структурах Go с помощью *тегов*, что является темой подраздела «Структуры и JSON» (см. далее). Теги управляют кодированием и декодированием записей JSON в структуры Go и из них. Но сначала мы должны поговорить о маршалинге и демаршалинге записей JSON.

Использование `Marshal()` и `Unmarshal()`

И маршалинг, и демаршалинг JSON — важные процедуры для работы с данными JSON с помощью структур Go. *Маршалинг* — процесс преобразования структуры Go в запись JSON. Обычно он нужен для передачи данных в формате JSON по компьютерным сетям или для сохранения их на диске. *Демаршалинг* — процесс преобразования записи JSON, заданной в виде байтового среза, в структуру Go. Обычно он требуется при получении данных JSON через компьютерные сети или при загрузке данных JSON из файлов на диске.



Главная ошибка при преобразовании записей JSON в структуры Go и наоборот заключается в том, что требуемые поля ваших структур Go не экспортятся. Если у вас возникли проблемы с маршалингом и демаршалингом, начните процесс отладки отсюда.

В коде `encodeDecode.go` показан как маршалинг, так и демаршалинг записей JSON с помощью жестко закодированных данных (для простоты):

```
package main

import (
    "encoding/json"
    "fmt"
)

type UseAll struct {
    Name    string `json:"username"`
    Surname string `json:"surname"`
    Year    int     `json:"created"`
}
```

Эти метаданные говорят нам о том, что поле `Name` структуры `UseAll` преобразуется в `username` в записи JSON и *наоборот*, поле `Surname` преобразуется в `surname` и *наоборот*, а поле структуры `Year` преобразуется в `created` в записи JSON и *наоборот*. Эта информация имеет отношение к маршалингу и демаршалингу данных JSON. В остальном вы рассматриваете и используете `UseAll` как обычную Go-структурку.

```
func main() {
    useall := UseAll{Name: "Mike", Surname: "Tsoukalos", Year: 2021}

    // обычная структура
    // кодирование данных JSON -> преобразовать структуру Go
    // в запись JSON с полями
    t, err := json.Marshal(&useall)
```

`Json.Marshal()` требует указателя на структурную переменную (ее реальным типом данных является пустая интерфейсная переменная) и возвращает байтовый срез с закодированной информацией и переменной `error`.

```
if err != nil {
    fmt.Println(err)
} else {
    fmt.Printf("Value %s\n", t)

    // декодирование данных JSON, заданных в виде строки
    str := `{"username": "M.", "surname": "Ts", "created":2020}`
```

Данные JSON обычно представляются в виде строки.

```
// преобразовать строку в байтовый срез
jsonRecord := []byte(str)
```

Однако поскольку `json.Unmarshal()` требует байтовый срез, вам нужно преобразовать эту строку в срез, прежде чем передавать ее в `json.Unmarshal()`.

```
// создаем структурную переменную для хранения результата
temp := UseAll{}
err = json.Unmarshal(jsonRecord, &temp)
```

`json.Unmarshal()` требует байтовый срез с записью JSON и указатель на переменную Go-структурки, которая будет хранить запись JSON и возвращает переменную `error`.

```
if err != nil {
    fmt.Println(err)
} else {
    fmt.Printf("Data type: %T with value %v\n", temp, temp)
}
```

При выполнении `encodeDecode.go` мы получаем такой вывод:

```
$ go run encodeDecode.go
Value {"username":"Mike","surname":"Tsoukalos","created":2021}
Data type: main.UseAll with value {M. Ts 2020}
```

В следующем подразделе более подробно показано, как определить теги JSON в Go-структуре.

Структуры и JSON

Представьте, что у вас есть Go-структура, которую вы хотите преобразовать в запись JSON без включения каких-либо пустых полей. В следующем коде показано, как выполнить эту задачу с помощью `omitempty`:

```
// игнорирование пустых полей в JSON
type NoEmpty struct {
    Name     string `json:"username"`
    Surname  string `json:"surname"`
    Year     int    `json:"creationyear,omitempty"`
}
```

Наконец, представьте, что у вас есть некие конфиденциальные данные в некоторых полях Go-структуры, которые вы не хотите включать в записи JSON. Вы можете сделать это, включив специальное значение `"-"` в нужные теги структуры `json:`. Это показано в следующем фрагменте кода:

```
// удаление закрытых полей и игнорирование пустых
type Password struct {
    Name     string `json:"username"`
    Surname  string `json:"surname,omitempty"`
    Year     int    `json:"creationyear,omitempty"`
    Pass     string `json:"-"`
}
```

Таким образом, поле `Pass` будет игнорироваться при преобразовании структуры `Password` в запись JSON с помощью `json.Marshal()`.

Эти два метода показаны в `tagsJSON.go`, который можно найти в каталоге `ch06` репозитория GitHub данной книги. При выполнении `tagsJSON.go` мы получаем такой вывод:

```
$ go run tagsJSON.go
noEmptyVar decoded with value {"username":"Mihalis","surname":""}
password decoded with value {"username":"Mihalis"}
```

Для первой строки вывода мы имеем следующее: значение `noEmpty`, которое преобразуется в структурную переменную `NoEmpty` с именем `noEmptyVar`, равно

NoEmpty{Name: "Mihalis"}. Структура `noEmpty` содержит значения по умолчанию для полей `Surname` и `Year`. Однако поскольку они конкретно не определены, `json.Marshal()` игнорирует поле `Year`, так как оно имеет тег `omitempty`, но не игнорирует поле `Surname`, которое имеет пустое строковое значение.

Для второй строки вывода: значение переменной `password` равно `Password{Name: "Mihalis", Pass: "myPassword"}`. Когда переменная `password` преобразуется в запись JSON, поле `Pass` не включается в выходные данные. Оставшиеся два поля структуры `Password`, называемые `Surname` и `Year`, опущены из-за тега `omitempty`. В итоге остается только поле `username`.

До сих пор мы работали с отдельными записями JSON. Но что делать, когда нужно обработать несколько записей? В следующем подразделе можно найти ответ на этот и многие другие вопросы!

Чтение и запись данных JSON в виде потоков

Представьте, что у вас есть срез Go-структур, представляющий записи JSON, которые требуется обработать. Должны ли вы обрабатывать записи одну за другой? Это возможно, но будет ли это эффективным решением? Конечно, нет! Хорошая новость в том, что Go поддерживает обработку нескольких записей JSON в виде потоков, что быстрее и эффективнее. В этом подразделе рассказывается, как выполнить это с помощью утилиты `JSONstreams.go`, которая содержит следующие две функции:

```
// Deserialize декодирует сериализованный фрагмент с записями JSON
func Deserialize(e *json.Decoder, slice interface{}) error {
    return e.Decode(slice)
}
```

Функция `Deserialize()` используется для считывания входных данных в виде записей JSON, их декодирования и помещения в срез. Функция записывает срез, который имеет тип данных `interface{}` и задается в качестве параметра, а также получает его входные данные из буфера параметра `*json.Decoder`. Параметр `*json.Decoder` вместе со своим буфером определяется в функции `main()`, что позволяет избежать постоянного выделения памяти и, следовательно, потерии производительности и эффективности использования данного типа. То же самое касается применения `*json.Encoder`:

```
// Serialize сериализует фрагмент с записями JSON
func Serialize(e *json.Encoder, slice interface{}) error {
    return e.Encode(slice)
}
```

Функция `Serialize()` принимает два параметра, `*json.Encoder` и срез любого типа данных (отсюда и использование `interface{}`). Функция обрабатывает

срез и записывает выходные данные в буфер `json.Encoder`, который передается в качестве параметра кодировщику во время его создания.



Обе функции, `Serialize()` и `DeSerialize()`, могут работать с любым типом записи JSON благодаря использованию `interface{}`.

Утилита `JSONstreams.go` генерирует случайные данные. При выполнении `JSONstreams.go` мы получаем такой вывод:

```
$ go run JSONstreams.go
After Serialize:[{"key": "XVLBZ", "value": 16}, {"key": "BAICM", "value": 89}]
After DeSerialize:
0 {XVLBZ 16}
1 {BAICM 89}
```

Входной срез структур, который генерируется в `main()`, сериализуется, что видно из первой строки выходных данных. После этого он десериализуется в исходный срез структур.

Структурный вывод записи JSON

В этом подразделе показано, как *структурно выводить* записи JSON. Термин означает вывод записей JSON в удобном для чтения формате без знания формата Go-структуры, в которой хранятся записи JSON. Поскольку читать записи JSON можно двумя способами (по отдельности и в виде потока), есть два способа структурного вывода данных JSON: в виде отдельных записей JSON и в виде потока. Поэтому мы реализуем две отдельные функции `prettyPrint()` и `JSONStream()` соответственно.

Реализация функции `prettyPrint()` выглядит так:

```
func PrettyPrint(v interface{}) (err error) {
    b, err := json.MarshalIndent(v, "", "\t")
    if err == nil {
        fmt.Println(string(b))
    }
    return err
}
```

Вся работа выполняется с помощью `json.MarshalIndent()`, которая *применяет отступ* для форматирования выходных данных.

Хотя и `json.MarshalIndent()`, и `json.Marshal()` выдают текстовый результат в формате JSON (*байтовый срез*), только `json.MarshalIndent()` позволяет применять настраиваемый отступ, тогда как `json.Marshal()` генерирует более компактный вывод.

Для структурного вывода потоков данных JSON вам следует использовать функцию `JSONStream()`:

```
func JSONstream(data interface{}) (string, error) {
    buffer := new(bytes.Buffer)
    encoder := json.NewEncoder(buffer)
    encoder.SetIndent("", "\t")
```

Функция `json.NewEncoder()` возвращает новый `Encoder`, который записывает в устройство записи, передающееся в качестве параметра в `json.NewEncoder()`. `Encoder` записывает значения JSON в выходной поток. Аналогично `json.MarshalIndent()`, метод `SetIndent()` позволяет применить настраиваемый отступ к потоку.

```
err := encoder.Encode(data)
if err != nil {
    return "", err
}
return buffer.String(), nil
}
```

Закончив настройку кодировщика, мы можем свободно обрабатывать поток JSON с помощью `Encode()`.

Эти две функции показаны в `prettyPrint.go`, который генерирует записи JSON с помощью случайных данных. При выполнении `prettyPrint.go` мы получаем такой вывод:

```
Last record: {BAICM 89}
{
    "key": "BAICM",
    "value": 89
}
[
    {
        "key": "XVLBZ",
        "value": 16
    },
    {
        "key": "BAICM",
        "value": 89
    }
]
```

Здесь показан улучшенный вывод одной записи JSON, за которым следует улучшенный вывод среза с двумя записями JSON. Все записи JSON представлены в виде Go-структур.

Следующий раздел посвящен работе с XML в Go.

Работа с XML

В этом разделе кратко описывается, как работать с XML-данными в Go с помощью записей. Идея, лежащая в основе взаимодействия XML и Go, та же, что и в случае с JSON и Go. Чтобы указать теги XML, вы помещаете теги в Go-структуры и по-прежнему можете сериализовать и десериализовать записи XML с помощью `xml.Unmarshal()` и `xml.Marshal()` из пакета `encoding/xml`. Однако есть и кое-какие различия, которые показаны в файле `xml.go`:

```
package main

import (
    "encoding/xml"
    "fmt"
)

type Employee struct {
    XMLName  xml.Name `xml:"employee"`
    ID       int      `xml:"id,attr"`
    FirstName string   `xml:"name>first"`
    LastName  string   `xml:"name>last"`
    Height    float32  `xml:"height,omitempty"`
    Address
    Comment string `xml:",comment"`
}
```

Здесь задается структура XML-данных. Однако имеется и дополнительная информация, касающаяся имени и типа каждого XML-элемента. Поле `XMLName` содержит имя XML-записи, которым в данном случае будет `employee`.

Поле с тегом `"comment"` служит комментарием и как таковое форматируется в выходных данных. Поле с тегом `attr` отображается в качестве атрибута к предоставленному имени поля (в данном случае `id`) в выходных данных. Нотация `name>first` указывает Go на встраивание тега `first` внутри тега `name`.

Наконец, поле с параметром `omitempty` исключается из выходных данных, если оно пустое. *Пустым значением* является любое из значений `0`, `false`, указателя или интерфейса `nil`, а также любой массив, срез, карта или строка с нулевой длиной.

```
type Address struct {
    City, Country string
}

func main() {
    r := Employee{ID: 7, FirstName: "Mihalis", LastName: "Tsoukalos"}
    r.Comment = "Technical Writer + DevOps"
    r.Address = Address{"SomeWhere 12", "12312, Greece"}

    output, err := xml.MarshalIndent(&r, " ", " ")
```

Как и в случае с JSON, `xml.MarshalIndent()` предназначен для приведения выходных данных в читаемый вид.

```
if err != nil {
    fmt.Println("Error:", err)
}
output = []byte(xml.Header + string(output))
fmt.Printf("%s\n", output)
}
```

Выходные данные программы `xml.go` следующие:

```
<?xml version="1.0" encoding="UTF-8"?>
<employee id="7">
  <name>
    <first>Mihalis</first>
    <last>Tsoukalos</last>
  </name>
  <City>SomeWhere 12</City>
  <Country>12312, Greece</Country>
  <!--Technical Writer + DevOps-->
</employee>
```

В этом выводе показана XML-версия Go-структуры, предоставленная в качестве входных данных для программы.

В следующем подразделе мы разработаем утилиту, которая преобразует записи JSON в XML и наоборот.

Преобразование JSON в XML и обратно

Как и было обещано, мы напишем утилиту, которая преобразует записи между форматами JSON и XML. Входные данные задаются в качестве аргумента командной строки. Утилита пытается угадать формат входных данных, начиная с XML. С одной стороны, если `xml.Unmarshal()` завершается неудачей, то утилита пытается использовать `json.Unmarshal()`. Если и это не срабатывает, то пользователь информируется о состоянии ошибки. С другой стороны, если `xml.Unmarshal()` выполняется успешно, то данные сохраняются в переменной `XMLrec`, а затем преобразуются в переменную `JSONrec`. То же самое происходит и с `json.Unmarshal()` в случае, когда вызов `xml.Unmarshal()` не сработал.

Логику работы утилиты можно проследить в Go-структуратах:

```
type XMLrec struct {
    Name      string `xml:"username"`
    Surname   string `xml:"surname,omitempty"`
    Year      int    `xml:"creationyear,omitempty"`
}
```

```
type JSONrec struct {
    Name     string `json:"username"`
    Surname string `json:"surname,omitempty"`
    Year     int    `json:"creationyear,omitempty"`
}
```

Обе структуры хранят одни и те же данные. Однако первая (`XMLrec`) предназначена для хранения XML-данных, тогда как вторая (`JSONrec`) — для хранения данных JSON.

При выполнении `JSON2XML.go` мы получаем такой вывод:

```
$ go run JSON2XML.go '<XMLrec><username>Mihalis</username></XMLrec>
<XMLrec><username>Mihalis</username></XMLrec>
{"username": "Mihalis"}'
```

Итак, мы предоставляем XML-запись в качестве входных данных, которая преобразуется в запись JSON. В следующем выводе показан обратный процесс:

```
$ go run JSON2XML.go '{"username": "Mihalis"}'
{"username": "Mihalis"}
<XMLrec><username>Mihalis</username></XMLrec>
```

В этом выводе входные данные представляли собой запись JSON, а выходные — запись XML.

В следующем разделе мы обсудим работу Go с файлами YAML.

Работа с YAML

В этом разделе мы кратко обсудим, как в Go обрабатываются файлы YAML. В стандартную библиотеку Go не входит поддержка YAML; это значит, для поддержки YAML вам придется обратиться к внешним пакетам. Существуют три основных пакета, которые позволяют работать с YAML в Go:

- <https://github.com/kylelemmons/go-gypsy>;
- <https://github.com/go-yaml/yaml>;
- <https://github.com/goccy/go-yaml>;

Выбор одного из них — вопрос личных предпочтений. В этом разделе мы будем работать с пакетом `go-yaml`, используя код в файле `yaml.go`. Благодаря использованию Go-модулей `yaml.go` разработан в `~/go/src/github.com/mactsouk/yaml` — вы также можете найти его в репозитории GitHub данной книги. Самая важная его часть выглядит так:

```
var yamlfile = `
image: Golang
```

```
matrix:
  docker: python
  version: [2.7, 3.9]
```

Переменная `yamlfile` содержит данные YAML. Обычно мы считываем данные из файла, но здесь просто используем переменную, чтобы сэкономить немного места.

```
type Mat struct {
  DockerImage string `yaml:"docker"`
  Version     []float32 `yaml:",flow"`
}
```

Структура `Mat` определяет два поля и их связи с файлом YAML. Поле `Version` представляет собой срез значений `float32`. Поскольку для поля `Version` нет имени, его именем будет `version`. Ключевое слово `flow` говорит о том, что маршалинг использует потоковый стиль, который хорошо работает для структур, последовательностей и карт.

```
type YAML struct {
  Image string
  Matrix Mat
}
```

Структура YAML включает структуру `Mat` и содержит поле `Image`, которое связано с `image` в файле YAML. Функция `main()` содержит ожидаемые вызовы `yaml.Unmarshal()` и `yaml.Marshal()`.

Как только у вас в нужном месте появится исходный файл, запустите следующие команды (если же вам нужно выполнить какие-либо дополнительные команды, то двоичный файл `go` поможет вам в этом):

```
$ go mod init
$ go mod tidy
```

Команда `go mod init` инициализирует и записывает новый файл `go.mod` в текущий каталог, в то время как команда `go mod tidy` синхронизирует этот файл с исходным кодом.



Если вы захотите перестраховаться и используете пакеты, которые не принадлежат к стандартной библиотеке, то разработка внутри `~/go/src`, фиксация в репозитории GitHub и использование Go-модулей для всех зависимостей будут наилучшим вариантом. Однако это не означает, что вы должны разрабатывать собственные пакеты в виде Go-модулей.

При выполнении `yaml.go` мы получаем такой вывод:

```
$ go run yaml.go
After Unmarshal (Structure):
{Golang {python [2.7 3.9]}}
After Marshal (YAML code):
image: Golang
matrix:
  docker: python
  version: [2.7, 3.9]
```

В этом выводе показано, как текст `{Golang {python [2.7 3.9]}}` преобразуется в файл YAML и наоборот. Теперь, когда мы обсудили работу в Go с данными JSON, XML и YAML, пришло время узнать о пакете `viper`.

Пакет `viper`

Флаги — это специально отформатированные строки, которые передаются в программу в целях управления ее поведением. Самостоятельная работа с флагами может быть проблемной, если вам требуется поддерживать несколько флагов и опций. Для работы с опциями командной строки, параметрами и флагами Go предлагает пакет `flag`. Хотя `flag` может многое, он далеко не так эффективен, как другие внешние Go-пакеты. Таким образом, если вы разрабатываете простые утилиты командной строки UNIX, то пакет `flag` может показаться вам очень интересным и полезным. Но ведь вы читаете эту книгу не для того, чтобы создавать простые утилиты командной строки! Поэтому я пропущу `flag` и познакомлю вас с внешним пакетом `viper`, который представляет собой эффективный Go-пакет, поддерживающий множество опций. Он использует пакет `pflag` вместо `flag`, что также показано в коде, который мы рассмотрим в следующих разделах.

Все проекты `viper` следуют определенной схеме. Сначала вы инициализируете `viper`, а затем определяете интересующие вас элементы. После этого получаете эти элементы, считываете их значения и используете. Желаемые значения можно получить либо напрямую (как это происходит, когда вы используете пакет `flag` из стандартной библиотеки Go), либо косвенно, с помощью файлов конфигурации. При использовании отформатированных конфигурационных файлов в формате JSON, YAML, TOML, HCL или свойств Java пакет `viper` выполняет весь синтаксический анализ за вас, что избавляет от необходимости писать и отлаживать большое количество Go-кода.

Кроме того, пакет позволяет извлекать и сохранять значения в Go-структурках. Однако для этого требуется, чтобы поля Go-структуры соответствовали ключам файла конфигурации.

Домашняя страница пакета `viper` находится на GitHub (<https://github.com/spf13/viper>). Обратите внимание, что вы не обязаны использовать все возможности `viper` в своих инструментах. Используйте только те функции, которые вам нужны. Общее правило заключается в использовании функций Viper, которые упрощают ваш код. Проще говоря, если вашей утилите командной строки требуется слишком много параметров и флагов командной строки, то вместо всего этого было бы лучше использовать файл конфигурации.

Использование флагов командной строки

В первом примере показано, как написать простую утилиту, которая принимает два значения в качестве параметров командной строки и выводит их на экран для проверки. Это означает, что для этих параметров нам понадобятся два флага командной строки.

Начиная с *версии Go 1.16* применение модулей является поведением по умолчанию, которое должен использовать пакет `viper`. Следовательно, чтобы все заработало, вам нужно поместить `useViper.go` (это имя исходного файла) внутрь `~/go`. Поскольку мое имя пользователя на GitHub — `mactsouk`, мне придется выполнить следующие команды:

```
$ mkdir ~/go/src/github.com/mactsouk/useViper
$ cd ~/go/src/github.com/mactsouk/useViper
$ vi useViper.go
$ go mod init
$ go mod tidy
```

Вы можете либо отредактировать `useViper.go` самостоятельно, либо скопировать его из репозитория книги на GitHub. Имейте в виду, что последние две команды должны быть выполнены, когда `useViper.go` будет готов к использованию и содержать все необходимые внешние пакеты.

Реализация `useViper.go` выглядит следующим образом:

```
package main

import (
    "fmt"

    "github.com/spf13/pflag"
    "github.com/spf13/viper"
)
```

Нам нужно импортировать пакеты `pflag` и `viper`, поскольку мы используем функционал обоих.

```
func aliasNormalizeFunc(f *pflag.FlagSet, n string) pflag.  
NormalizedName {  
    switch n {  
        case "pass":  
            n = "password"  
            break  
        case "ps":  
            n = "password"  
            break  
    }  
    return pflag.NormalizedName(n)  
}
```

С помощью функции `aliasNormalizeFunc()` создаются дополнительные псевдонимы для флага — в данном случае для `--password`. Согласно существующему коду, к флагу `--password` можно получить доступ либо как к `--pass`, либо как к `--ps`.

```
func main() {  
    pflag.StringP("name", "n", "Mike", "Name parameter")
```

Здесь мы создаем новый флаг `name`, к которому также можно получить доступ как к `-n`. Его значение по умолчанию — `Mike`, а его описание, которое появляется при использовании утилиты, — `Name parameter`.

```
pflag.StringP("password", "p", "hardToGuess", "Password")  
pflag.CommandLine.SetNormalizeFunc(aliasNormalizeFunc)
```

Мы создаем другой флаг `password`, к которому также можно получить доступ как к `-p` и который имеет значение по умолчанию `hardToGuess`, а также описание. Кроме того, мы регистрируем *функцию нормализации* для генерации псевдонимов для флага `password`.

```
pflag.Parse()  
viper.BindPFlags(pflag.CommandLine)
```

Вызов `pflag.Parse()` следует использовать после определения всех флагов командной строки. Его цель в том, чтобы преобразовать флаги командной строки в определенные флаги.

Кроме того, вызов `viper.BindPFlags()` делает все флаги доступными для пакета `viper` — строго говоря, мы утверждаем, что `viper.BindPFlags()` привязывает существующий набор флагов `pflag` (`pflag.FlagSet`) к `viper`.

```
name := viper.GetString("name")  
password := viper.GetString("password")
```

Эти команды показывают, как считывать значения двух флагов типа `string` из командной строки.

```
fmt.Println(name, password)

// чтение переменной среды
viper.BindEnv("GOMAXPROCS")
val := viper.Get("GOMAXPROCS")
if val != nil {
    fmt.Println("GOMAXPROCS:", val)
}
```

Пакет `viper` может работать с переменными среды. Сначала нужно вызвать `viper.BindEnv()`, чтобы передать `viper` интересующую нас переменную среды, а затем мы можем прочитать ее значение, вызвав `viper.Get()`. Если `GOMAXPROCS` еще не установлена, команда `fmt.Println()` выполнена не будет.

```
// установка переменной среды
viper.Set("GOMAXPROCS", 16)
val = viper.Get("GOMAXPROCS")
fmt.Println("GOMAXPROCS:", val)
}
```

Аналогично, мы можем изменить текущее значение переменной среды с помощью `viper.Set()`.

Хорошо то, что `viper` автоматически дает информацию об использовании:

```
$ go run useViper.go --help
Usage of useViper:
  -n, --name string      Name parameter (default "Mike")
  -p, --password string  Password (default "hardToGuess")
pflag: help requested
exit status 2
```

Использование `useViper.go` без каких-либо аргументов командной строки приводит к следующему выводу. Помните, что мы находимся внутри `~/go/src/github.com/mactsouk/useViper`:

```
$ go run useViper.go
Mike hardToGuess
GOMAXPROCS: 16
```

Однако если мы предоставим значения для флагов командной строки, то результат будет немного различаться:

```
$ go run useViper.go -n mtsouk -p hardToGuess
mtsouk hardToGuess
GOMAXPROCS: 16
```

Во втором случае мы использовали ярлыки для флагов командной строки, поскольку так быстрее.

В следующем подразделе мы обсудим использование файлов JSON для хранения информации о конфигурации.

Чтение конфигурационных файлов JSON

Пакет `viper` умеет считывать файлы JSON для получения своей конфигурации, и в этом подразделе показано, как это делается. Использование текстовых файлов для хранения сведений о конфигурации может пригодиться при написании сложных приложений, требующих значительного объема данных и настроек. Это показано в `jsonViper.go`.

Еще раз: нам нужно поместить `jsonViper.go` внутрь `~/go/src/github.com/mactsouk/jsonViper` — пожалуйста, настройте эту команду так, чтобы она соответствовала вашему собственному имени пользователя GitHub. Тем не менее, если вы не хотите создавать репозиторий GitHub, вы можете использовать `mactsouk`. Код `jsonViper.go` выглядит следующим образом:

```
package main

import (
    "encoding/json"
    "fmt"
    "os"

    "github.com/spf13/viper"
)

type ConfigStructure struct {
    MacPass     string `mapstructure:"macos"`
    LinuxPass   string `mapstructure:"linux"`
    WindowsPass string `mapstructure:"windows"`
    PostHost    string `mapstructure:"postgres"`
    MySQLHost   string `mapstructure:"mysql"`
    MongoHost   string `mapstructure:"mongodb"`
}
```

Отмету *важный момент*: мы используем файл JSON для хранения конфигурации, но Go-структура использует `mapstructure` вместо `json` для полей файла конфигурации JSON.

```
var CONFIG = ".config.json"

func main() {
    if len(os.Args) == 1 {
        fmt.Println("Using default file", CONFIG)
```

```

} else {
    CONFIG = os.Args[1]
}

viper.SetConfigType("json")
viper.SetConfigFile(CONFIG)
fmt.Printf("Using config: %s\n", viper.ConfigFileUsed())
viper.ReadInConfig()

```

С помощью этих четырех операторов мы объявляем, что используем файл JSON, сообщаем `viper` путь к файлу конфигурации, выводим используемый файл конфигурации, а также читаем его и анализируем.

Имейте в виду, что `viper` не проверяет, действительно ли файл конфигурации существует и доступен для чтения. Если файл не может быть найден или прочитан, то `viper.ReadInConfig()` будет работать так, будто файл конфигурации пуст.

```

if viper.IsSet("macos") {
    fmt.Println("macos:", viper.Get("macos"))
} else {
    fmt.Println("macos not set!")
}

```

Вызов `viper.IsSet()` проверяет, есть ли в конфигурации ключ `macos`. Если он установлен, то его значение считывается с помощью `viper.Get("macos")` и выводится на экран.

```

if viper.IsSet("active") {
    value := viper.GetBool("active")
    if value {
        postgres := viper.Get("postgres")
        mysql := viper.Get("mysql")
        mongo := viper.Get("mongodb")
        fmt.Println("P:", postgres, "My:", mysql, "Mo:", mongo)
    }
} else {
    fmt.Println("active is not set!")
}

```

В вышеупомянутом коде мы проверяем, существует ли ключ `active`, прежде чем считывать его значение. Если оно `true`, то мы считываем значения еще из трех ключей: `postgres`, `mysql` и `mongodb`.

Поскольку `active` должен содержать логическое значение, для чтения мы используем `viper.GetBool()`.

```

if !viper.IsSet("DoesNotExist") {
    fmt.Println("DoesNotExist is not set!")
}

```

Как и ожидалось, попытка прочитать несуществующий ключ завершается неудачей.

```
var t ConfigStructure
err := viper.Unmarshal(&t)
if err != nil {
    fmt.Println(err)
    return
}
```

Вызов `viper.Unmarshal()` позволяет поместить информацию из файла конфигурации JSON в должным образом определенную Go-строктуру (это не обязательно, но удобно).

```
PrettyPrint(t)
}
```

Реализация функции `PrettyPrint()` была представлена в `prettyPrint.go` ранее в этой главе.

Теперь нам нужно скачать зависимости `jsonViper.go`:

```
$ go mod init
$ go mod tidy # Эта команда требуется не всегда
```

Содержимое текущего каталога выглядит следующим образом:

```
$ ls -l
total 44
-rw-r--r-- 1 mtsouk users     85 Feb 22 18:46 go.mod
-rw-r--r-- 1 mtsouk users 29678 Feb 22 18:46 go.sum
-rw-r--r-- 1 mtsouk users   1418 Feb 22 18:45 jsonViper.go
-rw-r--r-- 1 mtsouk users    189 Feb 22 18:46 myConfig.json
```

Содержимое файла `MyConfig.json`, используемого для тестирования, выглядит следующим образом:

```
{
    "macos": "pass_macos",
    "linux": "pass_linux",
    "windows": "pass_windows",

    "active": true,
    "postgres": "machine1",
    "mysql": "machine2",
    "mongodb": "machine3"
}
```

При выполнении `jsonViper.go` с этим файлом JSON мы получаем такой вывод:

```
$ go run jsonViper.go myConfig.json
Using config: myConfig.json
macos: pass_macos
P: machine1 My: machine2 Mo: machine3
DoesNotExist is not set!
{
    "MacPass": "pass_macos",
    "LinuxPass": "pass_linux",
    "WindowsPass": "pass_windows",
    "PostHost": "machine1",
    "MySQLHost": "machine2",
    "MongoHost": "machine3"
}
```

Этот вывод генерируется `jsonViper.go` при разборе `MyConfig.json` и попытке найти нужную информацию.

В следующем разделе обсуждается Go-пакет для создания эффективных и профессиональных утилит командной строки, таких как `docker` и `kubectl`.

Пакет `cobra`

Go-пакет `cobra` очень удобный и популярный, он позволяет разрабатывать утилиты командной строки с командами, подкомандами и псевдонимами. Если вы когда-либо использовали `hugo`, `docker` или `kubectl`, то сразу поймете, что именно делает пакет `cobra`, поскольку все эти инструменты разработаны с помощью `cobra`. Команды могут иметь один или несколько псевдонимов, что очень удобно, когда вы хотите угодить как любителям, так и опытным пользователям. Пакет `cobra` также поддерживает *постоянные и локальные флаги*, которые служат флагами, доступными для всех команд, и флагами, доступными только для заданных команд, соответственно. Кроме того, по умолчанию `cobra` использует `viper` для анализа собственных аргументов командной строки.

Все проекты на `cobra` следуют одной и той же схеме разработки. Вы используете утилиту `cobra`, затем создаете команды, после чего вносите желаемые изменения в сгенерированные файлы исходного Go-кода, чтобы реализовать желаемую функциональность. В зависимости от сложности вашей утилиты вам может потребоваться внести в созданные файлы множество изменений. Хотя `cobra` экономит много времени, нам все равно придется писать код, реализующий желаемую функциональность для каждой команды.

Чтобы правильно скачать двоичный файл `cobra`, придется предпринять некоторые дополнительные шаги:

```
$ GO111MODULE=on go get -u -v github.com/spf13/cobra/cobra
```

Эта команда скачивает двоичный файл `cobra` и необходимые зависимости с помощью Go-модулей, даже если вы используете версию Go старше 1.16.



Не обязательно знать обо всех поддерживаемых переменных среды, таких как `GO111MODULE`, но иногда они могут помочь решить сложные проблемы с установкой Go. Итак, если вы хотите узнать о вашей текущей среде Go, то можете использовать команду `env`.

Для целей данного раздела нам понадобится репозиторий GitHub — он необязателен, но для читателей это единственный вариант получить доступ к представленному коду.

Путь к репозиторию GitHub следующий: <https://github.com/mactsovuk/go-cobra>. Первое, что требуется сделать, — это разместить файлы репозитория GitHub в нужном месте. Все будет намного проще, если вы поместите все внутрь `~/go`; точное место зависит от репозитория GitHub, поскольку компилятору Go не придется искать Go-файлы.

В нашем случае мы собираемся поместить все в `~/go/src/github.com/mactsovuk`, поскольку `mactsovuk` и есть мое имя пользователя на GitHub. Для этого необходимо выполнить следующие команды:

```
$ cd ~/go/src/github.com
$ mkdir mactsovuk # only required if the directory is not there
$ cd mactsovuk
$ git clone git@github.com:mactsovuk/go-cobra.git
$ cd go-cobra
$ ~/go/bin/cobra init --pkg-name github.com/mactsovuk/go-cobra
Using config file: /Users/mtsouk/.cobra.yaml
Your Cobra application is ready at
/Users/mtsouk/go/src/github.com/mactsovuk/go-cobra
$ go mod init
go: creating new go.mod: module github.com/mactsovuk/go-cobra
```

Поскольку пакет `cobra` лучше работает с модулями, мы определяем зависимости проекта с помощью Go-модулей. Чтобы указать, что проект Go использует Go-модули, необходимо выполнить `go mod init`. Команда создает два файла: `go.sum` и `go.mod`.

```
$ go run main.go
go: finding module for package github.com/spf13/cobra
go: finding module for package github.com/mitchellh/go-homedir
go: finding module for package github.com/spf13/viper
go: found github.com/mitchellh/go-homedir in github.com/mitchellh/
gohomedir
v1.1.0
go: found github.com/spf13/cobra in github.com/spf13/cobra v1.1.3
go: found github.com/spf13/viper in github.com/spf13/viper v1.7.1
```

A longer description that spans multiple lines and likely contains examples and usage of using your application. For example:

Cobra is a CLI library for Go that empowers applications.
This application is a tool to generate the needed files
to quickly create a Cobra application.

Все строки, начинающиеся с `go:`, имеют отношение к Go-модулям и встречаются только раз. Последние строки — сообщение по умолчанию проекта `cobra`. Позже мы изменим это сообщение. Теперь вы готовы приступить к работе с инструментом `cobra`.

Утилита с тремя командами

В этом подразделе показано использование команды `cobra add`, которая служит для добавления новых команд в проект `cobra`. Названия команд — `one`, `two` и `three`:

```
$ ~/go/bin/cobra add one
Using config file: /Users/mtsouk/.cobra.yaml
one created at /Users/mtsouk/go/src/github.com/mactsouk/go-cobra
$ ~/go/bin/cobra add two
$ ~/go/bin/cobra add three
```

Эти команды создают в папке `cmd` три новых файла `one.go`, `two.go` и `three.go`, которые являются начальными реализациями трех команд.

Первое, что обычно нужно сделать, — удалить ненужный код из `root.go` и изменить сообщения утилиты и каждой команды, как это описано в полях `Short` и `Long`. Однако при желании вы можете оставить исходные файлы без изменений.

В следующем подразделе мы дополним утилиту, добавляя флаги командной строки к командам.

Добавление флагов командной строки

Мы собираемся создать два глобальных флага командной строки и один флаг командной строки, который привязан к единственной команде (`two`) и не поддерживается двумя другими. Глобальные флаги командной строки определены в файле `./cmd/root.go`. Мы определим два глобальных флага: `directory`, который является строкой, и `depth`, который является целым числом без знака.

Оба глобальных флага определены в функции `init()` файла `./cmd/root.go`.

```
rootCmd.PersistentFlags().StringP("directory", "d", "/tmp", "Path to use.")
rootCmd.PersistentFlags().Uint("depth", 2, "Depth of search.")
viper.BindPFlag("directory", rootCmd.PersistentFlags().
Lookup("directory"))
viper.BindPFlag("depth", rootCmd.PersistentFlags().Lookup("depth"))
```

Мы используем `rootCmd.PersistentFlags()` для определения глобальных флагов, за которыми следует их тип данных. Имя первого флага — `directory`, а его ярлык — `d`, тогда как имя второго флага — `depth`, и ярлыка у него нет. Если вы хотите его добавить, то вам следует вместо этого использовать метод `UintP()`. После определения двух флагов мы передаем управление `viper`, вызывая `viper.BindPFlag()`. Первый флаг — это `string`, тогда как второй — значение `uint`. Поскольку оба они доступны в проекте `cobra`, мы вызываем `viper.GetString("directory")`, чтобы получить значение флага `directory`, и `viper.GetUint("depth")`, чтобы получить значение флага `depth`.

Наконец, мы добавляем флаг командной строки, который доступен только для команды `two`, используя следующую строку в файле `./cmd/two.go`:

```
twoCmd.Flags().StringP("username", "u", "Mike", "Username value")
```

Имя флага — `username`, а его ярлык — `u`. Поскольку это локальный флаг, доступный лишь для команды `two`, мы можем получить его значение путем вызова `cmd.Flags().GetString("username")` внутри файла `./cmd/two.go`.

В следующем подразделе мы создадим псевдонимы для существующих команд.

Создание псевдонимов команд

В этом подразделе мы продолжим дополнять код из предыдущего подраздела, создав псевдонимы для существующих команд. Это означает, что команды `one`, `two` и `three` также будут доступны как `cmd1`, `cmd2` и `cmd3` соответственно.

Чтобы проделать это, вам необходимо добавить дополнительное поле `Aliases` в структуру `cobra`. Структура `Command` каждой команды — тип данных поля `Aliases` — это срез `string`. Итак, для команды `one` начало структуры `cobra.Command` в `./cmd/one.go` будет выглядеть следующим образом:

```
var oneCmd = &cobra.Command{  
    Use:      "one",  
    Aliases: []string{"cmd1"},  
    Short:    "Command one",
```

Вы должны внести аналогичные изменения в файлы `./cmd/two.go` и `./cmd/three.go`. Пожалуйста, имейте в виду, что *внешнее имя* команды `one` — `oneCmd`. Другие команды имеют аналогичные внутренние имена.



Если вы случайно добавите псевдоним `cmd1` (или любой другой) к нескольким командам, то компилятор Go никак не отреагирует. Однако выполниться будет только его первое вхождение.

В следующем подразделе мы дополним утилиту, добавляя подкоманды для команд `one` и `two`.

Создание подкоманд

В этом подразделе показано, как создать две подкоманды для команды `three`. Именами двух подкоманд будут `list` и `delete`. Способ их создания с помощью утилиты `cobra` заключается в следующем:

```
$ ~./go/bin/cobra add list -p 'threeCmd'  
Using config file: /Users/mtsouk/.cobra.yaml  
list created at /Users/mtsouk/go/src/github.com/mactsouk/go-cobra  
$ ~./go/bin/cobra add delete -p 'threeCmd'  
Using config file: /Users/mtsouk/.cobra.yaml  
delete created at /Users/mtsouk/go/src/github.com/mactsouk/go-cobra
```

Эти команды создают внутри `./cmd` два новых файла: `delete.go` и `list.go`. За флагом `-p` следует *внешнее имя* команды, с которой вы хотите связать подкоманды. Внутренним именем команды `three` является `threeCmd`. Вы можете проверить, что эти две команды связаны с командой `three`, следующим образом:

```
$ go run main.go three delete  
delete called  
$ go run main.go three list  
list called
```

Если вы запустите `go run main.go two list`, то Go посчитает `list` аргументом командной строки команды `two` и не выполнит код в файле `./cmd/list.go`. Окончательная версия проекта `go-cobra` имеет следующую структуру и содержит следующие файлы, сгенерированные утилитой `tree(1)`:

```
$ tree  
. .  
|__ LICENSE  
|__ README.md  
|__ cmd  
|   |__ delete.go  
|   |__ list.go  
|   |__ one.go  
|   |__ root.go  
|   |__ three.go  
|   \__ two.go  
|__ go.mod  
|__ go.sum  
|__ main.go  
  
1 directory, 11 files
```



На данном этапе вы можете задаться вопросом о том, что происходит, когда требуется создать две подкоманды с одинаковыми именами для двух разных команд. В этом случае вы создаете первую подкоманду и переименовываете ее файл перед созданием второй.

Поскольку особого смысла в больших фрагментах кода здесь нет, вы можете найти код проекта `go-cobra` в <https://github.com/mactsovuk/go-cobra>. Пакет `cobra` также показан в заключительном разделе, где мы радикально обновляем приложение телефонной книги.

Поиск циклов в файловой системе UNIX

В этом разделе реализована практическая утилита командной строки UNIX, которая может находить циклы (петли) в файловых системах UNIX. Идея, лежащая в основе утилиты, заключается в том, что с помощью *символьных ссылок* у UNIX есть возможность создавать в нашей файловой системе циклы. Это может не только затруднить работу программного обеспечения для резервного копирования, такого как `tar(1)`, или утилит наподобие `find(1)`, но и создать проблемы, связанные с безопасностью. Представленная утилита `FScycles.go` пытается информировать нас о подобных ситуациях.

Идея, лежащая в ее основе, заключается в сохранении пути к каждому посещенному каталогу в карте. Если такой путь появляется во второй раз, то у нас цикл. Кarta называется `visited` и определяется как `map[string]int`.



Если вам интересно, почему мы используем строку, а не байтовый срез или какой-либо другой вид среза в качестве ключа для `visited`, то это лишь потому, что карты не могут иметь срезы в качестве ключей, так как срезы нельзя сравнивать.

Выходные данные утилиты зависят от корневого пути, используемого для инициализации процесса поиска, — этот путь передается утилите в качестве аргумента командной строки.

По замыслу, функция `filepath.Walk()` не переходит по символическим ссылкам, что позволяет избегать циклов. Однако в нашем случае мы хотим использовать именно символические ссылки на каталоги, чтобы обнаруживать циклы. Мы решим эту проблему через некоторое время.

Утилита использует функцию `IsDir()`, помогающую идентифицировать каталоги. Нас интересуют только каталоги, поскольку лишь они вкупе с символьическими ссылками на них могут создавать циклы в файловых системах. Наконец, утилита использует `os.Lstat()` из-за возможности обрабатывать символьические ссылки. Кроме того, `os.Lstat()` возвращает информацию о символьической ссылке, не переходя по ней, чего нельзя сказать о `os.Stat()` — в нашем случае нам не требуется автоматически переходить по символьическим ссылкам.

Важная часть кода `FScycles.go` располагается в реализации `walkFunction()`:

```
func walkFunction(path string, info os.FileInfo, err error) error {
    fileInfo, err := os.Stat(path)
    if err != nil {
        return nil
    }

    fileInfo, _ = os.Lstat(path)
    mode := fileInfo.Mode()
```

Сначала мы убеждаемся, что путь существует, а затем вызываем `os.Lstat()`.

```
// сначала ищем обычные каталоги
if mode.IsDir() {
    abs, _ := filepath.Abs(path)
    _, ok := visited[abs]
    if ok {
        fmt.Println("Found cycle:", abs)
        return nil
    }
    visited[abs]++
    return nil
}
```

Если обычный каталог уже посещался, то обнаружен цикл. Карта `visited` отслеживает все посещенные каталоги.

```
// поиск символьических ссылок на каталоги
if fileInfo.Mode()&os.ModeSymlink != 0 {
    temp, err := os.Readlink(path)
    if err != nil {
        fmt.Println("os.Readlink():", err)
        return err
    }

    newPath, err := filepath.EvalSymlinks(temp)
    if err != nil {
        return nil
    }
```

Функция `filepath.EvalSymlinks()` используется для определения того, куда ведут символические ссылки. Если местом назначения является другой каталог, то приведенный ниже код гарантирует, что он также будет посещен. Для этого используется дополнительный вызов `filepath.Walk()`.

```
linkFileInfo, err := os.Stat(newPath)
if err != nil {
    return err
}

linkMode := linkFileInfo.Mode()
if linkMode.IsDir() {
    fmt.Println("Following...", path, "-->", newPath)
```

Оператор `linkMode.IsDir()` гарантирует, что отслеживаются только каталоги.

```
abs, _ := filepath.Abs(newPath)
```

Вызов `filepath.Abs()` возвращает абсолютный путь от пути, который задан в качестве параметра. Ключи среза `visited` — это значения, возвращаемые `filepath.Abs()`.

```
_ , ok := visited[abs]
if ok {
    fmt.Println("Found cycle!", abs)
    return nil
}
visited[abs]++
err = filepath.Walk(newPath, walkFunction)
if err != nil {
    return err
}
return nil
}
}
return nil
}
```

При выполнении `FScycles.go` мы получаем такой вывод:

```
$ go run FScycles.go ~
Following... /home/mtsouk/.local/share/epiphany/databases/indexeddb/v0
--> /home/mtsouk/.local/share/epiphany/databases/indexeddb
Found cycle! /home/mtsouk/.local/share/epiphany/databases/indexeddb
```

Этот вывод сообщает нам, что в домашнем каталоге текущего пользователя обнаружен цикл — как только он определен, его необходимо удалить самостоятельно.

В остальных разделах этой главы мы обсудим ряд новых функций, которые появились в Go версии 1.16.

Новое в Go 1.16

Go 1.16 поставляется с некоторыми новыми функциями, включая встраивание файлов в двоичные файлы Go, а также функцию `os.ReadDir()`, тип `os.DirEntry` и пакет `io/fs`.

Поскольку эти функции связаны с системным программированием, они добавлены в книгу и исследуются в текущей главе. Мы начнем со знакомства со встраиванием файлов в двоичные исполняемые файлы Go.

Встраивание файлов

В этом подразделе представлен функционал, впервые появившийся в Go 1.16. Он позволяет *встраивать статические ресурсы* в двоичные файлы Go. Разрешенные типы данных для хранения встроенного файла – `string`, `[]byte` и `embed.FS`. Это означает, что двоичный файл Go может содержать файл, который вам не придется скачивать вручную при выполнении двоичного файла Go! Представленная утилита встраивает два разных файла, которые она может извлекать на основе заданного аргумента командной строки.

В приведенном ниже коде, сохраненном в файле `embedFiles.go`, показана эта новая функция Go:

```
package main

import (
    _ "embed"
    "fmt"
    "os"
)
```

Чтобы встраивать любые файлы в ваши двоичные файлы Go, вам понадобится пакет `embed`. Поскольку пакет `embedded` напрямую не используется, вам нужно поставить перед ним `_`, чтобы компилятор Go не ругался.

```
//go:embed static/image.png
var f1 []byte
```

Вам нужно начать строку с `//go:embed`, которая отмечает Go-комментарий, но обрабатывается особым образом; за ним следует путь к файлу, который вы хотите внедрить. В этом случае мы внедряем `static/image.png`, представляющий собой двоичный файл. Следующая строка должна задавать переменную, которая будет содержать данные встроенного файла, который в данном случае представляет собой байтовый срез `f1`. Для двоичных файлов рекомендуется исполь-

зователь байтовый срез, поскольку мы собираемся напрямую использовать его для сохранения двоичного файла.

```
//go:embed static/textfile
var f2 string
```

В данном случае мы сохраняем содержимое обычного текстового файла, `static/textfile`, в переменной `string`, названной `f2`.

```
func writeToFile(s []byte, path string) error {
    fd, err := os.OpenFile(path, os.O_CREATE|os.O_WRONLY, 0644)
    if err != nil {
        return err
    }
    defer fd.Close()

    n, err := fd.Write(s)
    if err != nil {
        return err
    }
    fmt.Printf("wrote %d bytes\n", n)
    return nil
}
```

`writeToFile()` служит для сохранения байтового среза в файл и является вспомогательной функцией, которую можно использовать и в других случаях.

```
func main() {
    arguments := os.Args
    if len(arguments) == 1 {
        fmt.Println("Print select 1|2")
        return
    }

    fmt.Println("f1:", len(f1), "f2:", len(f2))
```

Данный оператор выводит длины переменных `f1` и `f2`, чтобы убедиться, что они содержат размер встроенных файлов.

```
switch arguments[1] {
case "1":
    filename := "/tmp/temporary.png"
    err := writeToFile(f1, filename)
    if err != nil {
        fmt.Println(err)
        return
    }
case "2":
```

```

        fmt.Println(f2)
    default:
        fmt.Println("Not a valid option!")
    }
}
}

```

Блок `switch` отвечает за возврат нужного файла пользователю — в случае `static/textfile` содержимое файла выводится на экран. Что же касается двоичного файла, мы сохраним его как `/tmp/temporary.png`.

На этот раз в целях реализма мы скомпилируем `embedFiles.go`, поскольку это исполняемый двоичный файл, который содержит встроенные файлы. Двоичный файл мы создаем с помощью `go build embedfiles.go`. При выполнении `embedFiles` мы получаем такой вывод:

```

$ ./embedFiles 2
f1: 75072 f2: 14
Data to write
$ ./embedFiles 1
f1: 75072 f2: 14
wrote 75072 bytes

```

Следующий вывод проверяет, что `temporary.png` находится по правильному пути (`/tmp/temporary.png`):

```

$ ls -l /tmp/temporary.png
-rw-r--r-- 1 mt souk wheel 75072 Feb 25 15:20 /tmp/temporary.png

```

С помощью функционала встраивания мы можем создать утилиту, которая встраивает собственный исходный код и выводит его на экран при выполнении! Это интересный способ использования встраивания. Исходный код файла `printSource.go` выглядит следующим образом:

```

package main
import (
    _ "embed"
    "fmt"
)

//go:embed printSource.go
var src string

func main() {
    fmt.Print(src)
}

```

Как и прежде, встраиваемый файл задается в строке `//go:embed`. При выполнении `printSource.go` мы получаем вышеупомянутый код.

ReadDir и DirEntry

В этом подразделе мы обсудим `os.ReadDir()` и `os.DirEntry`. Однако начнем с обсуждения устаревания пакета `io/ioutil`. Его функциональность была перенесена в другие пакеты. Итак, мы имеем следующее:

- новая функция `os.ReadDir()` возвращает `[]DirEntry`. Это означает, что она не может напрямую заменить `ioutil.ReadDir()`, которая возвращает `[]FileInfo`. Хотя ни `os.ReadDir()`, ни `os.DirEntry` не предлагают ничего нового, они делают все быстрее и проще, что тоже немаловажно;
- функция `os.ReadFile()` непосредственно заменяет `ioutil.ReadFile()`;
- функция `os.WriteFile()` может непосредственно заменить `ioutil.WriteFile()`;
- аналогично, `os.MkdirTemp()` может заменить `ioutil.TempDir()` без каких-либо изменений. Однако имя `os.TempDir()` уже занято, поэтому новое имя функции различается;
- функция `os.CreateTemp()` такая же, как `ioutil.TempFile()`. Хотя имя `os.TempFile()` не было заимствовано, разработчики Go решили дать функции название `os.CreateTemp()` для соответствия с `os.MkdirTemp()`.



И `os.ReadDir()`, и `os.DirEntry` можно найти под именами `fs.ReadDir()` и `fs.DirEntry` в пакете `io/fs` для работы с интерфейсом файловой системы в `io/fs`.

Утилита `ReadDirEntry.go` демонстрирует использование `os.ReadDir()`. Кроме того, в следующем разделе мы увидим в действии `fs.DirEntry` в сочетании с `fs.WalkDir()`. Пакет `io/fs` поддерживает только `WalkDir()`, которая по умолчанию использует `DirEntry`. И `fs.WalkDir()`, и `filepath.WalkDir()` используют `DirEntry` вместо `FileInfo`. Это означает следующее: чтобы увидеть какие-либо улучшения производительности при обходе деревьев каталогов, вам придется заменить вызовы `filepath.Walk()` вызовами `filepath.WalkDir()`.

Представленная утилита использует `os.ReadDir()` для вычисления размера дерева каталогов с помощью следующей функции:

```
func GetSize(path string) (int64, error) {
    contents, err := os.ReadDir(path)
    if err != nil {
        return -1, err
    }

    var total int64
    for _, entry := range contents {
        // посетить элементы каталога
        if entry.IsDir() {
```

Если мы обрабатываем каталог, то нам придется идти глубже.

```
temp, err := GetSize(filepath.Join(path, entry.Name()))
if err != nil {
    return -1, err
}
total += temp
// получить размер каждого элемента, не относящегося к каталогу
} else {
```

Если это файл, то нам просто нужно получить его размер. Для этого осуществим вызов `Info()` для получения общей информации о файле, а затем `Size()` для получения размера файла:

```
info, err := entry.Info()
if err != nil {
    return -1, err
}
// возвращает значение int64
total += info.Size()
}
}
return total, nil
}
```

При выполнении `ReadDirEntry.go` мы получаем следующий вывод, который указывает, что утилита работает должным образом:

```
$ go run ReadDirEntry.go /usr/bin
Total Size: 1170983337
```

Наконец, имейте в виду, что и `ReadDir`, и `DirEntry` скопированы из языка программирования Python.

В следующем подразделе мы познакомимся с пакетом `io/fs`.

Пакет `io/fs`

В этом подразделе показан функционал пакета `io/fs`, который впервые появился в Go 1.16. Поскольку `io/fs` содержит уникальные функции, мы начнем этот подраздел с объяснения того, что может делать этот пакет. Проще говоря, `io/fs` предлагает интерфейс файловой системы `FS`, доступный *только для чтения*. Обратите внимание, что `embed.FS` реализует интерфейс `fs.FS`; это значит, `embed.FS` имеет возможность пользоваться некоторыми функциональными возможностями пакета `io/fs`. Следовательно, ваши приложения могут создавать собственные внутренние файловые системы и работать с их файлами.

Приведенный ниже пример кода, сохраненный как `ioFS.go`, создает с помощью `embed` файловую систему, после чего помещает туда все файлы из папки `./static`.

Файл `ioFS.go` поддерживает следующие функциональные возможности: перечисление всех файлов, поиск по имени файла и извлечение файла с помощью `list()`, `search()` и `extract()` соответственно. Мы начнем со знакомства с `list()`:

```
func list(f embed.FS) error {
    return fs.WalkDir(f, ".", walkFunction)
}
```

Все волшебство творится в функции `walkFunction()`, которая реализована следующим образом:

```
func walkFunction(path string, d fs.DirEntry, err error) error {
    if err != nil {
        return err
    }
    fmt.Printf("Path=%q, isDir=%v\n", path, d.IsDir())
    return nil
}
```

Функция `walkFunction()` довольно компактна, поскольку вся функциональность реализована самим Go.

Далее мы переходим к реализации функции `extract()`:

```
func extract(f embed.FS, filepath string) ([]byte, error) {
    s, err := fs.ReadFile(f, filepath)
    if err != nil {
        return nil, err
    }
    return s, nil
}
```

Функция `ReadFile()` используется для извлечения файла, который идентифицируется по пути к файлу, из файловой системы `embed.FS` в виде байтового среза, возвращаемого из функции `extract()`.

Наконец, у нас есть реализация функции `search()`, которая основана на `walkSearch()`:

```
func walkSearch(path string, d fs.DirEntry, err error) error {
    if err != nil {
        return err
    }
    if d.Name() == searchString {
```

`searchString` — это глобальная переменная, которая содержит строку поиска. Когда совпадение найдено, соответствующий путь выводится на экран.

```
    fileInfo, err := fs.Stat(f, path)
    if err != nil {
        return err
    }
```

```
    fmt.Println("Found", path, "with size", fileInfo.Size())
    return nil
}
```

Прежде чем выводить совпадение, мы вызываем `fs.Stat()` с целью получить о нем более подробную информацию.

```
return nil
}
```

Функция `main()` последовательно вызывает эти три функции. При выполнении `ioFS.go` мы получаем такой вывод:

```
$ go run ioFS.go
Path=.", isDir=true
Path="static", isDir=true
Path="static/file.txt", isDir=false
Path="static/image.png", isDir=false
Path="static/textfile", isDir=false
Found static/file.txt with size 14
wrote 14 bytes
```

Изначально утилита выводит список всех файлов в файловой системе (строки, начинающиеся с `Path`). Затем проверяет, что `static/file.txt` присутствует в файловой системе. И наконец, проверяет, что запись 14 байт в новый файл прошла успешно, поскольку все они были записаны.

В итоге мы видим, что в Go версии 1.16 были внесены важные дополнения и улучшения производительности.

Обновление приложения телефонной книги

В этом разделе мы изменим формат, который телефонное приложение использует для хранения своих данных. На этот раз оно будет использовать JSON где только можно. Кроме того, мы используем пакет `cobra` для реализации поддерживаемых команд. В результате весь соответствующий код находится в собственном репозитории GitHub, а не в каталоге `ch06` репозитория GitHub этой книги. Путь к репозиторию GitHub — <https://github.com/mactsouk/phonebook>. Вы можете клонировать каталог с помощью `git clone`, но попробуйте найти время и создать собственную версию.



При разработке реальных приложений не забывайте время от времени выполнять `git commit` и `git push`, чтобы все этапы разработки сохранялись в GitHub или GitLab. Помимо всего прочего, это хороший способ хранить резервные копии!

Использование **cobra**

Сначала нам нужно создать пустой репозиторий GitHub и клонировать его:

```
$ cd ~/go/src/github.com/mactsouk  
$ git clone git@github.com:mactsouk/phonebook.git  
$ cd phonebook
```

Вывод команды `git clone` нам не важен, поэтому он опущен.

Первой задачей после клонирования репозитория GitHub, который на данный момент практически пуст, является запуск команды `cobra init` с соответствующими параметрами.

```
$ ~/go/bin/cobra init --pkg-name github.com/mactsouk/phonebook  
Using config file: /Users/mtsouk/.cobra.yaml  
Your Cobra application is ready at  
/Users/mtsouk/go/src/github.com/mactsouk/phonebook
```

Затем необходимо создать структуру приложения, используя двоичный файл `cobra`. Как только структура готова, гораздо легче понять, что вам нужно реализовать. Структура приложения основана на поддерживаемых командах.

```
$ ~/go/bin/cobra add list  
Using config file: /Users/mtsouk/.cobra.yaml  
list created at /Users/mtsouk/go/src/github.com/mactsouk/phonebook  
$ ~/go/bin/cobra add delete  
$ ~/go/bin/cobra add insert  
$ ~/go/bin/cobra add search
```

На данный момент структура проекта должна быть следующей:

```
$ tree  
. .  
└── LICENSE  
└── README.md  
└── cmd  
    ├── delete.go  
    ├── insert.go  
    ├── list.go  
    ├── root.go  
    └── search.go  
└── go.mod  
└── go.sum  
└── main.go  
  
1 directory, 10 files
```

После этого необходимо объявить, что мы хотим использовать Go-модули, выполнив следующую команду:

```
$ go mod init
go: creating new go.mod: module github.com/mactsouk/phonebook
```

При необходимости вы можете запустить `go mod tidy` после `go mod init`. На данном этапе выполнение `go run main.go` должно загрузить все необходимые зависимости пакета и сгенерировать вывод `cobra` по умолчанию.

В следующем подразделе мы обсудим хранение данных JSON на диске.

Хранение и загрузка данных в формате JSON

Эта функциональность вспомогательной функции `saveJSONFile()` реализована в `./cmd/root.go` с помощью следующей функции:

```
func saveJSONFile(filepath string) error {
    f, err := os.Create(filepath)
    if err != nil {
        return err
    }
    defer f.Close()

    err = Serialize(&data, f)
    if err != nil {
        return err
    }
    return nil
}
```

Итак, в принципе все, что нам нужно сделать, — это сериализовать срез структур с помощью `Serialize()` и сохранить результат в файл. Далее нужно загрузить данные JSON из файла.

Функционал загрузки реализован в `./cmd/root.go` с помощью вспомогательной функции `readJSONFile()`. Нам необходимо прочитать файл данных с данными JSON и поместить их в срез структур, десериализовав его.

Реализация команды `delete`

Команда `delete` удаляет существующие записи из приложения телефонной книги — она реализована в файле `./cmd/delete.go`:

```
var deleteCmd = &cobra.Command{
    Use:   "delete",
    Short: "delete an entry",
```

```
Long: `delete an entry from the phone book application.`,
Run: func(cmd *cobra.Command, args []string) {
    // получить ключ
    key, _ := cmd.Flags().GetString("key")
    if key == "" {
        fmt.Println("Not a valid key:", key)
        return
    }
}
```

Сначала мы считываем соответствующий флаг командной строки (`key`), чтобы получить возможность идентифицировать удаляемую запись.

```
// удаление данных
err := deleteEntry(key)
if err != nil {
    fmt.Println(err)
    return
}
},
```

Затем мы вызываем вспомогательную функцию `deleteEntry()`, чтобы уже фактически удалить ключ. После успешного удаления функция вызывает `saveJSONFile()`, чтобы изменения вступили в силу.

В следующем подразделе мы обсудим команду `insert`.

Реализация команды `insert`

Команда `insert` требует ввода пользователем, а для этого должна поддерживать локальные флаги командной строки. Поскольку каждая запись содержит три поля, то для выполнения команды требуются три флага командной строки. Затем она вызывает вспомогательную функцию `insert()` для записи данных на диск. Пожалуйста, обратитесь к исходному файлу `./cmd/insert.go`, чтобы получить подробную информацию о реализации команды `insert`.

Реализация команды `list`

Команда `list` выводит список содержимого в приложении телефонной книги. Она не требует аргументов командной строки и в основном реализуется с помощью функции `list()`:

```
func list() {
    sort.Sort(PhoneBook(data))
    text, err := PrettyPrintJSONStream(data)
    if err != nil {
        fmt.Println(err)
    }
}
```

```

        return
    }
    fmt.Println(text)
    fmt.Printf("%d records in total.\n", len(data))
}

```

Функция сортирует данные перед вызовом `PrettyPrintJSONStream()`, чтобы привести сгенерированный вывод в порядок.

Реализация команды search

Команда `search` используется для поиска заданного телефонного номера в приложении телефонной книги. Она реализована в файле `./cmd/search.go` с помощью функции `search()`, которая перебирает карту индексов для данного ключа. Если ключ найден, то возвращается соответствующая запись.



За исключением операций, связанных с JSON, и изменений, связанных с использованием JSON и `cobra`, весь остальной Go-код почти такой же, как версия приложения телефонной книги из главы 4.

Работа с утилитой телефонной книги приводит к следующему результату:

```
$ go run main.go list
[
    {
        "name": "Mastering",
        "surname": "Go",
        "tel": "333123",
        "lastaccess": "1613503772"
    }
]

1 records in total.
```

Это результат выполнения команды `list`. Для добавления записи нужно всего лишь выполнить команду:

```
$ go run main.go insert -n Mike -s Tsoukalos -t 9416471
```

С помощью выполнения команды `list` проверяется успешность выполнения команды `insert`:

```
$ go run main.go list
[
    {
        "name": "Mastering",
```

```
        "surname": "Go",
        "tel": "333123",
        "lastaccess": "1613503772"
    },
    {
        "name": "Mike",
        "surname": "Tsoukalos",
        "tel": "9416471",
        "lastaccess": "1614602404"
    }
]
2 records in total.
```

Затем можно удалить эту запись, выполнив `go run main.go delete --key 9416471`. Как указывалось ранее, ключами приложения служат номера телефонов, так что мы удаляем записи на основе этих номеров. Однако ничто не мешает вам осуществлять удаление на основе других свойств.

Если команда не найдена, то вы получите следующий вывод:

```
$ go run main.go doesNotExist
Error: unknown command "doesNotExist" for "phonebook"
Run 'phonebook --help' for usage.
Error: unknown command "doesNotExist" for "phonebook"
exit status 1
```

Поскольку команда `DoesNotExist` не поддерживается приложением командной строки, `cobra` выводит описательное сообщение об ошибке (`unknown command`).

Упражнения

- Используйте функциональность `byCharacter.go`, `byLine.go` и `byWord.go`, чтобы создать упрощенную версию утилиты `wc(1)` UNIX.
- Создайте полную версию утилиты `wc(1)` UNIX, используя пакет `viper` для обработки параметров командной строки.
- Создайте полную версию утилиты `wc(1)` UNIX, используя команды вместо параметров командной строки, с помощью пакета `cobra`.
- Измените `JSONstreams.go`, чтобы принимать пользовательские данные или данные из файла.
- Измените `embedFiles.go`, чтобы сохранять двоичный файл в выбиремом пользователем месте.
- Измените `ioFS.go`, чтобы получить нужную команду, а также строку поиска в качестве аргумента командной строки.

- Сделайте `iofs.go` проектом `cobra`.
- Утилита `byLine.go` использует `ReadString('\'\n')` для чтения входного файла. Измените код, чтобы использовать для чтения `Scanner` (<https://golang.org/pkg/bufio/#Scanner>).
- Аналогично `byWord.go` использует `ReadString('\'\n')` для чтения входного файла. Измените код, чтобы вместо этого использовался `Scanner`.
- Измените код `yaml.go`, чтобы читать данные YAML из внешнего файла.

Резюме

Эта глава была посвящена системному программированию и файловому вводу-выводу в Go. В ней описаны такие темы, как обработка сигналов, работа с аргументами командной строки, чтение и запись обычных текстовых файлов, работа с данными JSON и создание эффективных утилит командной строки с использованием `cobra`.

Это одна из самых важных глав в книге, поскольку невозможно создать какую-либо реальную утилиту, не взаимодействуя с операционной системой, а также с файловой системой.

Следующая глава посвящена параллелизму в Go, и основными ее темами будутgorутины, каналы и *совместное* использование данных.

Дополнительные ресурсы

- Пакет `viper`: <https://github.com/spf13/viper>.
- Пакет `cobra`: <https://github.com/spf13/cobra>.
- Документация по пакету `encoding/json`: <https://golang.org/pkg/encoding/json>.
- Документация пакета `io/fs`: <https://golang.org/pkg/io/fs/>.
- Порядок байтов: <https://en.wikipedia.org/wiki/Endianness>.
- Примечания к выпуску Go 1.16: <https://golang.org/doc/go1.16>.

7

Параллельное выполнение в Go

Ключевым компонентом модели распараллеливания в Go служит *горутина*, которая является минимальной исполняемой сущностью в этом языке. В Go вообще все выполняется в виде горутин либо явно, либо по задумке. Каждая выполняемая программа Go имеет по крайней мере одну горутину, которая используется для запуска функции `main()` пакета `main`. Каждая горутина выполняется в единственном потоке ОС в соответствии с инструкциями *планировщика Go*, который и отвечает за выполнение горутин. Планировщик ОС не определяет, сколько потоков нужно создавать среде выполнения Go. Среда создаст достаточное количество потоков с целью гарантировать, что для запуска Go-кода доступно `GOMAXPROCS` потоков.

Однако горутини не могут напрямую взаимодействовать друг с другом. Обмен данными в Go осуществляется с использованием либо *каналов*, либо *общей памяти*. Каналы действуют как клей, соединяющий несколько горутин. Помните: горутини могут обрабатывать данные и выполнять команды, но не могут напрямую взаимодействовать друг с другом. Тем не менее им доступны другие способы, включая каналы, локальные сокеты и общую память. В то же время каналы не могут обрабатывать данные или выполнять код, но могут отправлять данные в горутини, получать данные или иметь какое-то иное специальное назначение.

Объединяя несколько каналов и горутин, вы получаете возможность создавать потоки данных, которые в терминологии Go также называются конвейерами.

Итак, у вас может бытьgorутина, которая считывает данные из базы и отправляет их в канал, и вторая горутина, которая считывает их из него, обрабатывает и отправляет их в другой канал, откуда их можно прочитать с помощью другой горутины до внесения изменений и сохранения в другой базе данных.

В этой главе:

- процессы, потоки и горутины;
- планировщик Go;
- горутины;
- каналы;
- состояния гонки;
- ключевое слово `select`;
- установка тайм-аута горутины;
- каналы в Go;
- общая память и общие переменные;
- закрытые переменные и оператор `go`;
- пакет `context`;
- пакет `semaphore`.

Процессы, потоки и горутины

Процесс – это представление операционной системой запущенной программы, в то время как *программа* представляет собой двоичный файл на диске, содержащий всю информацию, необходимую для создания процесса ОС. Двоичный файл записан в определенном формате (ELF в Linux) и содержит все инструкции, которые будет выполнять процессор, а также множество других полезных разделов. Программа загружается в память, и инструкции выполняются, создавая запущенный процесс. Таким образом, *процесс* содержит дополнительные ресурсы, такие как память, описания открытых файлов и пользовательские данные, а также другие типы ресурсов, которые добавляются во время выполнения.

Поток – это объект меньшего размера и значимости. Процессы состоят из одного или нескольких потоков, у которых имеется свой поток команд управления и стек. Быстрый, но слегка упрощенный способ отличить поток от процесса – рассматривать процесс как запущенный двоичный файл, а поток – как подмножество процесса.

Горутина — это минимальная сущность в Go, которая может выполняться параллельно. Использование слова «минимальная» здесь очень важно, поскольку горутины не являются автономными объектами, подобными процессам UNIX. Горутины работают в потоках ОС, которые, в свою очередь, работают в процессах ОС. Хорошо то, что горутины легче потоков, которые, в свою очередь, легче процессов, так что запуск тысяч или сотен тысяч горутин на одной машине не станет проблемой. Среди причин того, что горутины легче потоков, можно выделить следующие: меньший стек, который может расти, а также более быстрое время запуска. Кроме того, они могут взаимодействовать друг с другом по каналам с низкой задержкой.

На практике это означает, что процесс может иметь несколько потоков, а также множество горутин, тогда как для существования горутины требуется среда процесса. Итак, чтобы создать горутину, вам нужно иметь процесс хотя бы с одним потоком. Операционная система заботится о планировании процессов и потоков, в то время как Go создает необходимые потоки, а разработчик создает желаемое количество горутин.

Теперь, когда вы познакомились с основами процессов, программ, потоков и горутин, немного поговорим о *планировщике Go*.

Планировщик Go

Планировщик ядра ОС отвечает за выполнение потоков программы. Аналогично среда выполнения Go имеет собственный планировщик, который отвечает за выполнение горутин. При этом используется метод, известный как *планирование t:n*, где *t* горутин выполняется с помощью *n* потоков ОС с использованием мультиплексирования. Планировщик Go — это компонент, отвечающий за способ и порядок выполнения горутин Go-программы. Это делает планировщик Go поистине важной частью языка программирования Go. Планировщик Go и сам выполняется как горутина.



Имейте в виду, что, поскольку планировщик Go имеет дело только с горутинами одной программы, он работает проще и быстрее, чем планировщик ядра, и обходится дешевле.

Go использует модель распараллеливания, известную как *разветвление-соединение*. Часть «разветвление», которую не следует путать с системным вызовом `fork(2)`, утверждает, что дочерняя ветвь может быть создана в любой точке программы. Аналогично часть модели «соединение» — это окончание дочерней

ветви и соединение ее с родительской. Имейте в виду: и операторы `sync.Wait()`, и каналы, которые собирают результаты горутин, — это точки соединения, тогда как каждая новая горутина создает дочернюю ветвь.

Стратегия справедливого планирования, которая довольно проста и имеет простую реализацию, распределяет всю нагрузку равномерно между доступными процессорами. На первый взгляд это может показаться идеальной стратегией, поскольку не нужно принимать во внимание множество вещей, сохраняя при этом одинаковую загруженность всех процессоров. Но на поверку это не совсем так. Большинство распределенных задач обычно зависят от других задач. Следовательно, некоторые процессоры используются недостаточно или (эквивалентно) больше других. Горутина — это задача, тогда как все, что происходит после ее вызывающего оператора, является *следованием*. В диспетчериизации на основе *перехвата работы*, используемой планировщиком Go, (логический) процессор, который используется недостаточно, забирает дополнительную работу у других процессоров.

Согласно названию, находя такие задания, он перехватывает их у другого процессора или процессоров. Кроме того, алгоритм перехвата работы в Go ставит в очередь и перехватывает и следования. Стоящее соединение, как следует из его названия, — это точка, в которой поток выполнения останавливается при соединении и пытается найти для выполнения другую работу.

Хотя и перехват задачи, и перехват следования приводят к остановке соединений, следования происходят чаще, чем задачи; поэтому алгоритм планирования Go работает со следованиями, а не с задачами.

Основным недостатком перехвата следования является то, что это требует дополнительной работы от компилятора языка программирования. К счастью, Go располагает такими возможностями и поэтому использует перехват следования в своем алгоритме перехвата работы. Одним из преимуществ перехвата следования является то, что вы получаете те же результаты при использовании вызовов функций вместо горутин или одного потока с несколькими горутинами. Это отличное решение, поскольку в любой момент в обоих случаях выполняется только что-то одно.

Планировщик Go работает с применением трех основных типов объектов: потоков операционной системы (M), которые связаны с используемой операционной системой; горутин (G); и логических процессоров (P). Количество процессоров, которые могут использоваться Go-программой, определяется переменной среды `GOMAXPROCS` — в любой момент времени существует не более `GOMAXPROCS` процессоров. Теперь вернемся к алгоритму планирования $m:n$, используемому в Go. Строго говоря, в любой момент у нас имеется m горутин, которые выполняются и, следовательно, запланированы к запуску в n потоках ОС, использующих не более `GOMAXPROCS` логических процессоров. Вскоре мы подробнее поговорим о `GOMAXPROCS`.

На рис. 7.1 показано, что существуют два различных типа очередей выполнения: глобальная и локальная, подключенная к каждому логическому процессору. Горутины из глобальной очереди назначаются очереди логического процессора, чтобы в какой-то момент выполниться.

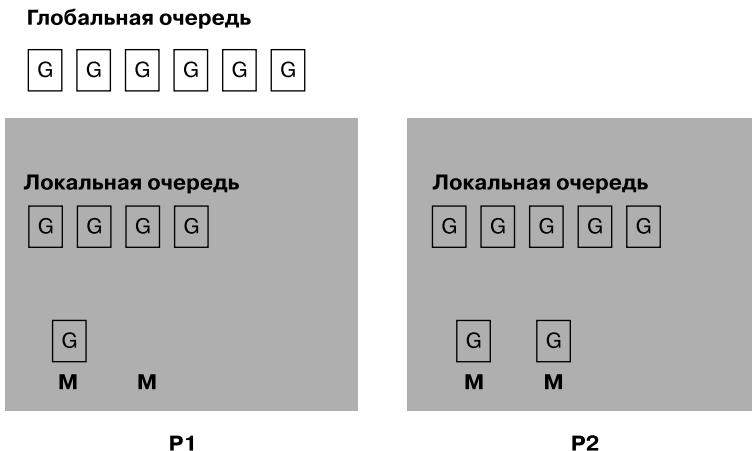


Рис. 7.1. Работа планировщика Go

Каждый логический процессор может иметь несколько потоков, и перехват происходит между локальными очередями доступных логических процессоров. Наконец, имейте в виду, что планировщику Go разрешено создавать больше потоков операционной системы, если это необходимо. Потоки ОС довольно дороги с точки зрения ресурсов. Это означает, что чрезмерная работа с потоками ОС может замедлить работу ваших приложений Go.

Далее мы обсудим значение и использование `GOMAXPROCS`.

Переменная среды `GOMAXPROCS`

Переменная среды `GOMAXPROCS` позволяет вам задать количество потоков операционной системы (*процессоров*), которые могут одновременно выполнять Go-код пользователяского уровня. Начиная с версии Go 1.5, значением `GOMAXPROCS` по умолчанию должно быть количество логических ядер, доступных на вашем компьютере. Существует также функция `runtime.GOMAXPROCS()`, которая позволяет устанавливать и получать значение `GOMAXPROCS` программно.

Если вы решите присвоить `GOMAXPROCS` значение, меньшее, чем количество ядер на вашем компьютере, то это может повлиять на производительность вашей программы. Однако использование значения `GOMAXPROCS`, превышающего количество доступных ядер, не обязательно ускорит работу ваших Go-программ.

Как упоминалось ранее, вы можете программно установить и получить значение переменной среды `GOMAXPROCS` — это показано в файле `maxprocs.go`, где также будут показаны дополнительные возможности пакета `runtime`. Функция `main()` реализована следующим образом:

```
func main() {
    fmt.Println("You are using ", runtime.Compiler, " ")
    fmt.Println("on a", runtime.GOARCH, "machine")
    fmt.Println("Using Go version", runtime.Version())
```

Переменная `runtime.Compiler` содержит набор инструментов компилятора, используемый для сборки запущенного двоичного файла. Двумя наиболее известными значениями являются `gc` и `gccgo`. Переменная `runtime.GOARCH` содержит текущую архитектуру, а `runtime.Version()` возвращает текущую версию компилятора Go. Эта информация не является необходимой для использования `runtime.GOMAXPROCS()`, но всегда хорошо лучше знать, как работает система.

```
    fmt.Printf("GOMAXPROCS: %d\n", runtime.GOMAXPROCS(0))
}
```

Что происходит с вызовом `runtime.GOMAXPROCS(0)`? Функция `runtime.GOMAXPROCS()` всегда возвращает предыдущее значение *максимального количества процессоров, которые могут выполняться одновременно*. Когда параметр `runtime.GOMAXPROCS()` равен или больше 1, `runtime.GOMAXPROCS()` также изменяет текущую настройку. Поскольку мы используем 0, наш вызов не меняет текущую настройку.

При выполнении `maxprocs.go` мы получаем такой вывод:

```
You are using gc on a amd64 machine
Using Go version go1.16.2
GOMAXPROCS: 8
```

Вы можете изменять значение `GOMAXPROCS` динамически, используя следующий метод:

```
$ export GOMAXPROCS=100; go run maxprocs.go
You are using gc on a amd64 machine
Using Go version go1.16.2
GOMAXPROCS: 100
```

Эта команда временно изменяет значение `GOMAXPROCS` на 100 и выполняет `maxprocs.go`.

Помимо случаев тестирования производительности вашего кода с использованием меньшего количества ядер, вам, скорее всего, не потребуется менять `GOMAXPROCS`. В следующем подразделе мы обсудим сходства и различия между распараллеливанием и параллелизмом.

Параллелизм и распараллеливание

Распространенное заблуждение состоит в том, что *параллелизм* — это то же самое, что и *распараллеливание*. Это совершенно не так! Параллелизм — это одновременное выполнение нескольких объектов какого-либо вида, тогда как распараллеливание — это способ структурирования компонентов таким образом, чтобы они могли выполняться независимо, когда это возможно.

Только создавая программные компоненты параллельными, вы можете безопасно выполнять их параллельно, когда и если ваша операционная система и ваше оборудование это позволяют. Язык программирования Erlang сделал это давным-давно, задолго до того, как процессоры получили несколько ядер, а компьютеры — большой объем оперативной памяти.

При хорошем проектировании добавление параллельных сущностей ускоряет работу всей системы, поскольку параллельно может выполняться больше задач. Таким образом, желаемый параллелизм достигается за счет лучшего параллельного выражения и реализации задачи. Разработчик несет ответственность за распараллеливание на этапе проектирования системы и получает прирост производительности от потенциального параллельного выполнения компонентов системы. Таким образом, разработчик должен думать не о параллелизме, а о разбиении проекта на независимые компоненты, которые при объединении решают исходную проблему.

Даже если на своей машине вы не можете выполнять функции параллельно, правильный дизайн с распараллеливанием задач все равно улучшает ваш проект и удобство обслуживания ваших программ.

Другими словами, распараллеливание лучше, чем параллелизм! Теперь, прежде чем перейти к каналам, которые являются основными компонентами модели распараллеливания в Go, поговорим о горутинах.

Горутины

Вы можете определить, создать и выполнить новую горутину, используя ключевое слово `go`, за которым следует имя функции или *анонимная функция*. Ключевое слово `go` немедленно возвращает выполнение из вызова функции, в то время как функция начинает работать в фоновом режиме в виде горутины, тогда как остальная часть программы тоже продолжает свое выполнение. Вы не можете контролировать или делать какие-либо предположения о *порядке*, в котором будут выполняться ваши программы, поскольку это зависит от планировщика ОС, планировщика Go и загрузки ОС.

Создание горутины

В этом подразделе вы узнаете, как создавать горутины. Программа, в которой показана данная техника, называется `create.go`. Реализация функции `main()` заключается в следующем:

```
func main() {
    go func(x int) {
        fmt.Printf("%d ", x)
    }(10)
```

Так запускается анонимная функция в качестве горутины. `(10)` в конце — это способ, с помощью которого передается параметр анонимной функции. Анонимная функция выше просто выводит значение на экран.

```
    go printme(15)
```

Так в виде горутины выполняется функция. Как правило, функции, которые выполняются как горутины, *не возвращают никаких значений напрямую*. Обмен данными с горутинаами происходит с помощью общей памяти, каналов или какого-либо другого механизма.

```
    time.Sleep(time.Second)
    fmt.Println("Exiting...")
}
```

Поскольку Go-программа перед выходом не ожидает завершения своих горутин, нам нужно отложить выход вручную, что мы и делаем с помощью `runtime.Sleep()`. Мы исправим это в ближайшее время и перед выходом дождемся завершения всех горутин.

При выполнении `create.go` мы получаем такой вывод:

```
$ go run create.go
10 * 15
Exiting...
```

Часть `10` в выходных данных взята из анонимной функции, тогда как часть `* 15` — из оператора `go printme(15)`. Однако, запустив `create.go` несколько раз, вы можете получить другие выходные данные, поскольку две горутины не всегда выполняются в одном и том же порядке:

```
$ go run create.go
* 15
10 Exiting...
```

В следующем подразделе показано, как запускать переменное количество горутин.

Создание нескольких горутин

В этом подразделе вы узнаете, как создать переменное количество горутин. Программа, в которой показана данная техника, называется `multiple.go`. Количество горутин передается программе в качестве аргумента командной строки. Важным кодом из реализации функции `main()` является следующий:

```
fmt.Printf("Going to create %d goroutines.\n", count)
for i := 0; i < count; i++ {
```

Ничто не мешает вам использовать цикл `for` для создания нескольких горутин, особенно если вам их нужно достаточно много.

```
    go func(x int) {
        fmt.Printf("%d ", x)
    }(i)
}
time.Sleep(time.Second)
fmt.Println("\nExiting...")
```

И снова `time.Sleep()` приостанавливает завершение функции `main()`.

При выполнении `multiple.go` мы получаем такой вывод:

```
$ go run multiple.go 15
Going to create 15 goroutines.
3 0 8 4 5 6 7 11 9 12 14 13 1 2 10
Exiting...
```

Если вы запустите `multiple.go` многократно, то получите разные выходные данные. Таким образом, у нас все еще есть что улучшать. В следующем подразделе показано, как избавиться от вызова `time.Sleep()` и заставить программу ждать завершения работы горутин.

Ожидание завершения горутин

Недостаточно создать несколько горутин, нужно еще дождаться их завершения до завершения функции `main()`. Так что в этом подразделе вы узнаете о технике, улучшающей код `multiple.go`. Улучшенная версия называется `varGoroutines.go`. Но сначала придется объяснить, как все это работает.

Процесс синхронизации начинается с определения переменной `sync.WaitGroup` и использования методов `Add()`, `Done()` и `Wait()`. Если вы посмотрите на исходный код Go-пакета `sync`, а точнее на файл `waitgroup.go`, то увидите, что тип `sync.WaitGroup` — это не что иное, как структура с двумя полями:

```
type WaitGroup struct {
    noCopy noCopy
```

```
    state1 [3]uint32
}
```

Каждый вызов `sync.Add()` увеличивает счетчик в поле `state1`, которое представляет собой массив с тремя элементами `uint32`. Обратите внимание, что действительно важно вызвать `sync.Add()` перед операцией `go`, чтобы предотвратить какие-либо состояния гонки. О них вы узнаете в разделе «Состояния гонки» данной главы. Когдаgorутина завершает свою работу, должна быть выполнена функция `sync.Done()`, уменьшающая тот же счетчик на единицу. «За кулисами» `sync.Done()` просто совершает вызов `Add(-1)`. Метод `Wait()` ожидает, пока этот счетчик не станет равным 0, после чего возвращается. Возврат `Wait()` внутри функции `main()` означает, что `main()` собирается вернуться и программа завершится.



Вы можете вызвать `Add()` с целым положительным значением, отличным от 1, если нужно избежать многократного вызова `Add(1)`. Это удобно, когда вы заранее знаете количество gorутин, которые собираетесь создать. `Done()` не поддерживает эту функциональность.

Важной частью `varGoroutines.go` является следующий код:

```
var waitGroup sync.WaitGroup
fmt.Printf("%#v\n", waitGroup)
```

Здесь создается переменная `sync.WaitGroup`, которую мы и будем использовать. Вызов `fmt.Printf()` выводит содержимое структуры `sync.WaitGroup` — обычно это не нужно, но бывает полезно для получения дополнительной информации о структуре `sync.WaitGroup`.

```
for i := 0; i < count; i++ {
    waitGroup.Add(1)
```

Мы вызываем `Add(1)` непосредственно перед созданием gorутины, чтобы избежать состояний гонки.

```
go func(x int) {
    defer waitGroup.Done()
```

Вызов `Done()` будет выполнен непосредственно перед возвратом анонимной функции из-за ключевого слова `defer`.

```
    fmt.Printf("%d ", x)
}(i)
}

fmt.Printf("%#v\n", waitGroup)
waitGroup.Wait()
```

Функция `Wait()` будет ждать, пока счетчик в переменной `WaitGroup` не станет равным `0`, после чего произойдет возврат, чего мы, собственно, и добиваемся.

```
fmt.Println("\nExiting...")
```

Когда функция `Wait()` возвращается, выполняется оператор `fmt.Println()`. Таким образом, нам больше не нужно вызывать `time.Sleep()`!

При выполнении `varGoroutines.go` мы получаем такой вывод:

```
$ go run varGoroutines.go 15
Going to create 15 goroutines.
sync.WaitGroup{noCopy:sync.noCopy{}, state1:[3]uint32{0x0, 0x0, 0x0}}
sync.WaitGroup{noCopy:sync.noCopy{}, state1:[3]uint32{0x0, 0x0, 0xf}}
14 8 9 10 11 5 0 4 1 2 3 6 13 12 7
Exiting...
```

Значение в третьем месте среза `state1` равно `0xf`, что составляет `15` в десятичной системе, поскольку мы вызывали `Add(1)` 15 раз.



Помните, что использование большего количества горутин в программе не является панацеей для повышения производительности. Большое количество горутин, вдобавок к различным вызовам `sync.Add()`, `sync.Wait()` и `sync.Done()`, может замедлить работу вашей программы из-за дополнительной работы, которую придется выполнять планировщику Go.

Что делать, если количество вызовов `Add()` и `Done()` разное

Когда количество вызовов `sync.Add()` и `sync.Done()` равно, в вашей программе все будет в порядке. Однако в этом подразделе мы выясним, что происходит, если эти два числа не согласуются друг с другом.

В зависимости от того, имеются ли параметры командной строки, представленная программа действует по-разному. Без параметров количество вызовов `Add()` меньше, чем количество вызовов `Done()`. При наличии хотя бы одного параметра командной строки количество вызовов `Done()` меньше, чем количество вызовов `Add()`. Вы можете посмотреть Go-код `addDone.go` самостоятельно. Что важно, так это результат, который он дает. При выполнении `addDone.go` без аргументов командной строки мы получаем следующее сообщение об ошибке:

```
$ go run addDone.go
Going to create 20 goroutines.
sync.WaitGroup{noCopy:sync.noCopy{}, state1:[3]uint32{0x0, 0x0, 0x0}}
```

```

sync.WaitGroup{noCopy:sync.noCopy{}, state1:[3]uint32{0x0, 0x0, 0x13}}
19 14 15 16 17 18 10 8 9 11 12 1 0 2 5 6 3 4 7 13
Exiting...
panic: sync: negative WaitGroup counter

goroutine 19 [running]:
sync.(*WaitGroup).Add(0xc000014094, 0xffffffffffffffff)
    /usr/local/Cellar/go/1.16/libexec/src/sync/waitgroup.go:74
+0x147
sync.(*WaitGroup).Done(0xc000014094)
    /usr/local/Cellar/go/1.16/libexec/src/sync/waitgroup.go:99
+0x34
main.main.func1(0xc000014094, 0xd)
    /Users/mtsouk/ch07/addDone.go:26 +0xdb
created by main.main
    /Users/mtsouk/ch07/addDone.go:23 +0x1c6
exit status 2

```

Причину сообщения об ошибке можно найти в выводе. Сообщение `panic: sync: negative WaitGroup counter` вызвано количеством вызовов `Done()`, превышающим количество вызовов `Add()`. Обратите внимание, что иногда `addDone.go` не выдает никаких сообщений об ошибках и завершается нормально. Это проблема с параллельными программами в целом — они не всегда сбоят или ведут себя неправильно, поскольку порядок выполнения может измениться, вследствие чего меняется поведение программы. Это еще больше затрудняет отладку.

При выполнении `addDone.go` с одним аргументом командной строки мы получаем следующее сообщение об ошибке:

```

$ go run addDone.go 1
Going to create 20 goroutines.
sync.WaitGroup{noCopy:sync.noCopy{}, state1:[3]uint32{0x0, 0x0, 0x0}}
sync.WaitGroup{noCopy:sync.noCopy{}, state1:[3]uint32{0x0, 0x0, 0x15}}
19 5 6 7 8 9 0 1 3 14 11 12 13 4 17 10 2 15 16 18 fatal error: all
goroutines are asleep - deadlock!

goroutine 1 [semacquire]:
sync.runtime_Semacquire(0xc000014094)
    /usr/local/Cellar/go/1.16/libexec/src/runtime/sema.go:56 +0x45
sync.(*WaitGroup).Wait(0xc000014094)
    /usr/local/Cellar/go/1.16/libexec/src/sync/waitgroup.go:130
+0x65
main.main()
    /Users/mtsouk/ch07/addDone.go:38 +0x2b6
exit status 2

```

И снова причина сбоя выводится на экран: `fatal error: all goroutines are asleep - deadlock!`. Это означает, что программа обречена *бесконечно ждать* завершения горутины, то есть вызова `Done()`, которого никогда не будет.

Создание нескольких файлов с помощью горутин

В качестве практического примера использования горутин в данном подразделе представлена утилита командной строки, которая создает несколько файлов, заполненных случайно сгенерированными данными. Такие файлы могут использоваться для тестирования файловых систем или генерации данных для тестирования. Ключевой код `randomFiles.go` включает следующее:

```
var waitGroup sync.WaitGroup
for i := start; i <= end; i++ {
    waitGroup.Add(1)
    filepath := fmt.Sprintf("%s/%s%d", path, filename, i)
    go func(f string) {
        defer waitGroup.Done()
        createFile(f)
    }(filepath)
}
waitGroup.Wait()
```

Сначала мы создаем переменную `sync.WaitGroup`, чтобы должным образом дождаться завершения всех программ. Каждая горутина создает один файл. Что здесь важно, так это то, что каждый файл имеет уникальное имя, — это реализовано с помощью переменной `filepath`, которая содержит значение счетчика цикла `for`. По мере создания горутинами файлов происходит несколько выполнений функций `CreateFile()`. Это простой, но очень эффективный способ создания нескольких файлов.

При выполнении `randomFiles.go` мы получаем такой вывод:

```
$ go run randomFiles.go
Usage: randomFiles firstInt lastInt filename directory
```

Итак, утилите требуются четыре параметра: первое и последнее значение цикла `for`, имя файла и каталог, в который файлы будут записаны. Так что запустим утилиту с правильным количеством параметров:

```
$ go run randomFiles.go 2 5 masterGo /tmp
/tmp/masterGo5 created!
/tmp/masterGo3 created!
/tmp/masterGo2 created!
/tmp/masterGo4 created!
```

Все выглядит как надо, и четыре файла были созданы в соответствии с нашими инструкциями! Теперь, когда вы узнали о горутинах, переключимся на каналы.

Каналы

Канал — это механизм связи, который, помимо прочего, позволяет горутинам обмениваться данными. Во-первых, каждый канал допускает обмен определенным типом данных, который также называется типом элемента канала, а во-вторых, для правильной работы канала вам понадобится кто-то, кто будет получать то, что отправляется по нему. Новый канал можно объявить, используя `make()` и ключевое слово `chan` (`make(chan int)`), а закрыть канал — с помощью функции `close()`. Вы можете объявить и размер канала, написав что-то вроде `make(chan int, 1)`.

Конвойер — это виртуальный метод соединения горутин и каналов, так что выходные данные одной горутины становятся входными данными другой, а для передачи ваших данных используются каналы. Одним из преимуществ использования конвойеров является то, что в вашей программе будет постоянный поток данных, так как ни одна горутина или канал не обязаны ждать завершения всего, чтобы начать выполнение. Кроме того, вы используете меньше переменных и, следовательно, меньше места в памяти, так как нет необходимости хранить все в переменной. Наконец, использование конвойеров упрощает разработку программы и повышает доступность ее обслуживания.

Запись в канал и чтение из него

Записать значение `val` в канал `ch` очень легко. Достаточно записать `ch <- val`. Стрелка показывает направление значения, и у вас не возникнет проблем с этим оператором, если `var` и `ch` имеют один и тот же тип данных.

Прочитать значение из канала `c` можно, выполнив команду `<-c`. В данном случае направление — от канала во внешний мир. Вы можете сохранить это значение в переменной, использовав `aVar := <-c`.

Как чтение, так и запись каналов показаны в `channels.go` с помощью кода, представленного ниже:

```
package main

import (
    "fmt"
    "sync"
)

func writeToChannel(c chan int, x int) {
    c <- x
    close(c)
}
```

Эта функция просто записывает значение в канал и немедленно его закрывает.

```
func printer(ch chan bool) {
    ch <- true
}
```

Эта функция отправляет значение `true` в канал с типом `bool`.

```
func main() {
    c := make(chan int, 1)
```

Этот канал имеет *буфер* с размером 1. Это означает, что, как только мы заполним буфер, мы сможем закрыть канал, аgorутина продолжит свое выполнение и вернется. Канал, который *не буферизован*, ведет себя по-другому: когда вы отправляете значение в этот канал, он блокируется до момента, пока кто-то получит это значение. В нашем случае определенно нужен буферизованный канал, чтобы избежать каких-либо блокировок.

```
var waitGroup sync.WaitGroup
waitGroup.Add(1)
go func(c chan int) {
    defer waitGroup.Done()
    writeToChannel(c, 10)
    fmt.Println("Exit.")
}(c)

fmt.Println("Read:", <-c)
```

Здесь мы считываем из канала и выводим значение, не сохраняя его в отдельной переменной.

```
_ , ok := <-c
if ok {
    fmt.Println("Channel is open!")
} else {
    fmt.Println("Channel is closed!")
}
```

Код выше показывает метод *определения того, закрыт канал или нет*. В этом случае мы игнорируем читаемое значение — если бы канал был открыт, то значение было бы *отброшено*.

```
waitGroup.Wait()

var ch chan bool = make(chan bool)
for i := 0; i < 5; i++ {
    go printer(ch)
}
```

Здесь мы создадим небуферизованный канал и пять горутин *без какой-либо синхронизации*, поскольку не будем использовать вызовы `Add()`.

```
// диапазон по каналам
// ВАЖНО: поскольку канал с не закрыт,
// цикл по диапазону не завершается сам по себе
n := 0
for i := range ch {
```

Ключевое слово `range` работает с каналами! Однако цикл по `range` канала завершается только тогда, когда канал закрыт или использовано ключевое слово `break`.

```
fmt.Println(i)
if i == true {
    n++
}
if n > 2 {
    fmt.Println("n:", n)
    close(ch)
    break
}
```

Мы закрываем канал `ch` при выполнении условия и выходим из цикла `for` с помощью `break`.

```
for i := 0; i < 5; i++ {
    fmt.Println(<-ch)
}
}
```

При попытке чтения из закрытого канала мы получаем нулевое значение его типа данных, так что этот цикл `for` работает превосходно без каких-либо проблем.

При выполнении `channels.go` мы получаем такой вывод:

```
Exit.
Read: 10
```

После записи значения `10` в канал с помощью `writeToChannel(c, 10)` мы считываем это значение.

```
Channel is closed!
true
true
true
```

Цикл `for` с `range` завершается после трех итераций — каждая итерация выводит на экран значение `true`.

```
n: 3
false
false
false
false
false
```

Эти пять `false` выводятся последним циклом `for`.

Хотя с `channels.go` все вроде бы нормально, в нем присутствует логическая проблема, которую мы объясним и разрешим в разделе «Состояния гонки» данной главы. Кроме того, многократный запуск `channels.go` может привести к сбою. Однако в большинстве случаев сбоя не будет, что делает отладку еще более сложной.

Прием из закрытого канала

Чтение из закрытого канала возвращает нулевое значение его типа данных. Но если вы попытаетесь выполнить запись в закрытый канал, ваша программа завершится ошибкой (*panic*). Эти две ситуации рассматриваются в `readCloseCh.go`, а конкретно в реализации функции `main()`:

```
func main() {
    willClose := make(chan complex64, 10)
```

Если вы проделаете это каналом без буферизации, то произойдет сбой.

```
// записываем в канал какие-то данные
willClose <- -1
willClose <- 1i
```

Мы записываем два значения в канал `willClose`.

```
// считывание данных из пустой канал
<-willClose
<-willClose
close(willClose)
```

Затем мы читаем и отбрасываем два полученных значения, после чего закрываем канал.

```
// читаем еще раз — это закрытый канал
read := <-willClose
fmt.Println(read)
}
```

Последнее значение, которое мы считываем из канала, — это нулевое значение типа `complex64`. При выполнении `readCloseCh.go` мы получаем такой вывод:

```
(0+0i)
```

Итак, мы получили обратно нулевое значение типа `complex64`. Продолжим тему и обсудим, как работать с функциями, которые принимают каналы в качестве параметров.

Каналы как параметры функций

При использовании канала в качестве *параметра функции* вы можете указать его направление, то есть будет ли он использоваться для отправки или получения данных. На мой взгляд, если вы заранее знаете назначение канала, то вам следует использовать эту возможность, поскольку это делает ваши программы более надежными. Вы не сможете случайно отправить данные в канал, из которого должны только получать данные, или получить данные из канала, предназначеннего только для отправки. В результате если вы объявили, что параметр функции (канал) будет использоваться только для чтения, после чего попытаетесь в него что-то записать, то получите сообщение об ошибке, которое, скорее всего, избавит вас от неприятных ошибок в будущем.

Все это показано в программе `channelFunc.go`. Реализация функций, которые принимают параметры канала, выглядит так:

```
func printer(ch chan<- bool) {
    ch <- true
}
```

Функция принимает параметр канала, который доступен только для записи.

```
func writeToChannel(c chan<- int, x int) {
    fmt.Println("1", x)
    c <- x
    fmt.Println("2", x)
}
```

Параметр канала этой функции доступен только для чтения.

```
func f2(out <-chan int, in chan<- int) {
    x := <-out
    fmt.Println("Read (f2):", x)
    in <- x
    return
}
```

Последняя функция принимает два параметра канала. Однако `out` доступен только для чтения, в то время как `in` — для записи. Если вы попытаетесь вы-

полнить операцию с недопустимым параметром канала, компилятор Go выдаст ошибку. Это происходит, даже если функция не используется.

Тема следующего раздела — состояние гонки. Внимательно прочитайте его, чтобы избежать неопределенного поведения и неприятных ситуаций при работе с несколькими горутинами.

Состояния гонки

Состояние гонки данных — это ситуация, когда два или более запущенных элемента, таких как потоки и горутины, пытаются взять под контроль или изменить общий ресурс или общую переменную программы. Строго говоря, гонка данных возникает, когда две или более команд обращаются к одному и тому же адресу памяти, куда по крайней мере одна из них выполняет операцию записи (изменения). Если все операции являются операциями чтения, то состояние гонки отсутствует. На практике это означает, что вы можете получить разные выходные данные, если запустите свою программу несколько раз, и это очень плохо.

Использование флага `-race` при запуске или сборке исходного кода Go запускает Go-детектор гонки, который заставляет компилятор создавать модифицированную версию типичного исполняемого файла. Эта модифицированная версия может записывать все обращения к общим переменным, а также все происходящие события синхронизации, включая вызовы `sync.Mutex` и `sync.WaitGroup`, которые представлены далее в текущей главе. После анализа соответствующих событий детектор гонки выводит отчет, который поможет вам выявить и исправить потенциальные проблемы.

Go-детектор гонки. Вы можете запустить детектор гонки с помощью `go run -race`. Если мы проверим `channels.go` с помощью `go run -race`, то получим следующий результат:

```
$ go run -race channels.go
Exit.
Read: 10
Channel is closed!
true
true
true
n: 3
=====
WARNING: DATA RACE
Write at 0x00c00006e010 by main goroutine:
    runtime.closechan()
        /usr/local/Cellar/go/1.16.2/libexec/src/runtime/chan.go:355 +0x0
```

```

main.main()
/Users/mtsouk/ch07/channels.go:54 +0x46c

Previous read at 0x00c00006e010 by goroutine 12:
runtime.chansend()
/usr/local/Cellar/go/1.16.2/libexec/src/runtime/chan.go:158 +0x0
main.printer()
/Users/mtsouk/ch07/channels.go:14 +0x47

Goroutine 12 (running) created at:
main.main()
/Users/mtsouk/ch07/channels.go:40 +0x2b4
=====
false
false
false
false
false
Found 1 data race(s)
exit status 66

```

Таким образом, хотя `channels.go` и выглядит нормально, в нем имеется потенциальное состояние гонки. Проанализируем вывод, показанный выше, и подумаем, в чем заключается проблема с `channels.go`.

В строке 54 в `channels.go` происходит закрытие канала, а в строке 14 мы видим запись в тот же канал, которая, похоже, и является причиной проблемы с состоянием гонки. Стока 54 содержит `close(ch)`, тогда как строка 14 — `ch <- true`. Проблема в том, что мы не можем быть уверены, что именно произойдет и в *каком порядке* — это состояние гонки. Если вы запустите `channels.go` без детектора гонки, то все может пройти хорошо, но если проделать это несколько раз, то вы можете получить сообщение об ошибке `panic: send on closed channel`. В основном ошибка связана с порядком, в котором планировщик Go будет запускать горутины программы. Итак, если сначала произойдет закрытие канала, то запись в этот канал завершится неудачей — *состояние гонки!*

Исправление `channels.go` требует изменения кода и, более конкретно, реализации функции `printer()`. Исправленная версия `channels.go` называется `chRace.go`, и код в ней выглядит так:

```

func printer(ch chan<- bool, times int) {
    for i := 0; i < times; i++ {
        ch <- true
    }
    close(ch)
}

```

Первое, на что здесь следует обратить внимание, — вместо того чтобы использовать несколькоgorутин для записи в нужный канал, мы применяем одну. Одна горутина, записывающая данные в канал с последующим его закрытием, не может породить состояния гонки, поскольку все происходит последовательно.

```
func main() {
    // это не буферизованный канал
    var ch chan bool = make(chan bool)

    // записываем пять значений в канал с помощью одной горутины
    go printer(ch, 5)

    // ВАЖНО: Поскольку канал с закрыт,
    // цикл по диапазону завершится сам по себе
    for val := range ch {
        fmt.Println(val, " ")
    }
    fmt.Println()

    for i := 0; i < 15; i++ {
        fmt.Println(<-ch, " ")
    }
    fmt.Println()
}
```

При выполнении `go run -race chRace.go` мы получаем следующий результат, означающий, что состояния гонки больше нет:

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

Следующий раздел посвящен важному и эффективному ключевому слову `select`.

Ключевое слово `select`

Ключевое слово `select` поистине незаменимо, так как позволяет прослушивать несколько каналов одновременно. Блок `select` может иметь несколько разделов и необязательный раздел `default`, что в целом имитирует оператор `switch`. На всякий случай для блоков `select` всегда полезно иметь опцию тайм-аута. Наконец, `select` без каких-либо разделов (`select{}`) запустит вечное ожидание.

На практике это означает, что `select` позволяет горутине *ожидать выполнения нескольких* операций связи. Следовательно, `select` дает возможность

прослушивать несколько каналов с помощью одного блока `select`. Как следствие, вы можете выполнять неблокирующие операции с каналами при условии, что соответствующим образом реализовали свои блоки `select`.

Оператор `select` не вычисляется последовательно, так как все его каналы проверяются одновременно. Если ни один из каналов в операторе не готов, то он блокируется (*ожидает*), пока один из каналов не будет готов. *Если несколько каналов оператора select готовы, то среда выполнения Go делает случайный выбор из набора этих готовых каналов.*

В коде `select.go` показано простое использование `select` в горутине, имеющей три раздела. Но сначала посмотрим, как работает горутина, содержащая `select`:

```
wg.Add(1)
go func() {
    gen(0, 2*n, createNumber, end)
    wg.Done()
}()
```

Этот код говорит нам, что для выполнения `wg.Done()` функция `gen()` должна возвращаться первой. Взглянем на реализацию `gen()`:

```
func gen(min, max int, createNumber chan int, end chan bool) {
    time.Sleep(time.Second)
    for {
        select {
        case createNumber <- rand.Intn(max-min) + min:
        case <-end:
            fmt.Println("Ended!")
            // оператор return
    }
}
```

Правильная вещь, которую следует здесь сделать, — это добавить оператор `return`, чтобы `gen()` завершилась. Теперь представим, что мы забыли его добавить. Это означает, что функция не завершится после выполнения ветви `select`, связанной с параметром канала `end`, следовательно, `createNumber` не завершит функцию из-за отсутствия оператора `return`. Таким образом, блок `select` продолжает ждать продолжения. Решение можно увидеть в коде ниже:

```
case <-time.After(4 * time.Second):
    fmt.Println("time.After()!")
    return
}
}
```

Итак, что же на самом деле происходит в коде всего блока `select`? Этот конкретный оператор `select` имеет три раздела. Как указывалось ранее, для `select`

не обязательно указывать ветку `default`. Но вы можете считать третью ветвь оператора `select` вполне разумной веткой `default`. Причина в том, что `time.After()` ожидает истечения указанного времени (`4 * time.Second`), после чего выводит сообщение и правильно завершает `gen()` с помощью `return`. Это разблокирует оператор `select` в случае, если все остальные каналы по какой-либо причине заблокированы. Пропуск `return` из второй ветви будет ошибкой, но данное действие показывает, что наличие стратегии выхода — это всегда хорошо.

При выполнении `select.go` мы получаем такой вывод:

```
$ go run select.go 10
Going to create 10 random numbers.
13 0 2 8 12 4 13 15 14 19 Ended!
time.After()!
Exiting...
```

Мы будем наблюдать, как работает `select`, на протяжении оставшейся части главы, начиная со следующего раздела, в котором обсудим, как установить тайм-аут для горутины. Необходимо четко запомнить, что `select` позволяет прослушивать несколько каналов из одной точки.

Установка тайм-аута горутины

Бывают случаи, когда для завершения работы горутины требуется больше времени, чем ожидалось, — в таких ситуациях мы хотим установить тайм-аут для ее работы, чтобы иметь возможность разблокировать программу. В данном разделе представлены два метода, позволяющие сделать это.

Ограничение времени выполнения горутины — внутри `main()`

В этом подразделе представлена простая методика ограничения времени выполнения горутины. Соответствующий код можно найти в функции `main()` программы `timeOut1.go`:

```
func main() {
    c1 := make(chan string)
    go func() {
        time.Sleep(3 * time.Second)
        c1 <- "c1 OK"
    }()
}
```

Вызов `time.Sleep()` используется для эмуляции времени, которое обычно требуется для завершения работы функции. В данном случае анонимная функция, которая выполняется какgorутина, работает около трех секунд, прежде чем записать сообщение в канал `c1`.

```
select {
    case res := <-c1:
        fmt.Println(res)
    case <-time.After(time.Second):
        fmt.Println("timeout c1")
}
```

Цель вызова `time.After()` заключается в том, чтобы дождаться желаемого времени перед выполнением. Если же выполняется другая ветка, то время ожидания сбрасывается. В нашем случае нас не интересует фактическое значение, возвращаемое `time.After()`, но тот факт, что ветка `time.After()` была выполнена, означает, что время ожидания прошло. В этом случае, поскольку значение, передаваемое в функцию `time.After()`, меньше значения, используемого в вызове `time.Sleep()`, который был выполнен ранее, вы, скорее всего, получите сообщение о тайм-ауте. Причина, по которой я говорю «скорее всего», заключается в том, что Linux – это не ОС реального времени, и иногда планировщик ОС ведет себя загадочно, особенно когда имеет дело с высокой нагрузкой и планирует множество задач.

```
c2 := make(chan string)
go func() {
    time.Sleep(3 * time.Second)
    c2 <- "c2 OK"
}()

select {
    case res := <-c2:
        fmt.Println(res)
    case <-time.After(4 * time.Second):
        fmt.Println("timeout c2")
}
```

Оба фрагмента кода выполняют горутину, работа которой из-за вызова `time.Sleep()` занимает три секунды, и определяют период ожидания в четыре секунды в `select` с помощью `time.After(4 * time.Second)`. Если вызов `time.After(4 * time.Second)` возвращается после того, как вы получите значение из канала `c2` в первом разделе блока `select`, то время ожидания превышено не будет; в противном случае тайм-аут сработает.

Однако в данном случае значение `time.After()` дает достаточно времени для возврата вызова `time.Sleep()`, поэтому вы, скорее всего, не получите здесь сообщение о превышении времени ожидания.

Проверим все это. При выполнении `timeOut1.go` мы получаем такой вывод:

```
$ go run timeOut1.go
timeout c1
c2 OK
```

Как и ожидалось, время ожидания первой горутины истекло, а второй — нет. В следующем подразделе представлен другой метод установки времени ожидания.

Ограничение времени выполнения горутины — вне `main()`

В этом подразделе показан еще один метод ограничения времени выполнения горутин. Оператор `select` может располагаться в отдельной функции. Кроме того, период ожидания задается в качестве аргумента командной строки.

Интересная часть программы `timeOut2.go` заключается в реализации функции `timeout()`:

```
func timeout(t time.Duration) {
    temp := make(chan int)
    go func() {
        time.Sleep(5 * time.Second)
        defer close(temp)
    }()

    select {
    case <-temp:
        result <- false
    case <-time.After(t):
        result <- true
    }
}
```

В `timeout()` продолжительность времени, используемая в `time.After()`, передается в качестве параметра функции, а это означает, что она может меняться. И снова логика ограничения времени ожидания содержится в блоке `select`. Любой период ожидания, превышающий пять секунд, скорее всего, даст программе достаточно времени для завершения. Если `timeout()` запишет `false` в канал `result`, то тайм-аута не будет, тогда как при записи `true` — будет. При выполнении `timeOut2.go` мы получаем такой вывод:

```
$ go run timeOut2.go 100
Timeout period is 100ms
Time out!
```

Период ожидания составляет 100 миллисекунд, что не дает горутине достаточно времени для завершения, отсюда и сообщение о тайм-ауте.

```
$ go run timeOut2.go 5500
Timeout period is 5.5s
OK
```

На этот раз время ожидания составляет 5500 миллисекунд, и это означает, что у горутин было достаточно времени для завершения.

В следующем разделе мы рассмотрим и представим расширенные концепции, связанные с каналами.

Еще раз о каналах в Go

До сих пор вы изучали основные способы использования каналов. В этом же разделе представлены определение и использование `nil`-каналов, а также сигнальных и буферизованных каналов.

Полезно помнить, что нулевое значение типа канала равно `nil` и если вы отправляете сообщение на закрытый канал, то возникает состояние паники. Однако если вы попытаетесь выполнить чтение из закрытого канала, то получите нулевое значение типа этого канала. Итак, *после закрытия канала вы больше не можете записывать в него, но все еще можете читать из него*. Чтобы иметь возможность закрыть канал, его следует сделать доступным не только для чтения.

Кроме того, `nil`-канал всегда блокируется; это значит, что как чтение, так и запись из `nil`-каналов блокируются. Данное свойство каналов бывает очень полезным, когда необходимо отключить ветку инструкции `select`, присвоив переменной канала значение `nil`. Наконец, если вы попытаетесь закрыть `nil`-канал, то ваша программа вызовет состояние паники. Это наглядно показано в программе `closeNil.go`:

```
package main

func main() {
    var c chan string
```

В этом операторе мы определяем `nil`-канал с типа `string`.

```
    close(c)
}
```

При выполнении `closeNil.go` мы получаем такой вывод:

```
panic: close of nil channel

goroutine 1 [running]:
main.main()
    /Users/mtsouk/ch07/closeNil.go:5 +0x2a
exit status 2
```

В этом выводе показано сообщение, которое вы получите, если попытаетесь закрыть `nil`-канал.

Теперь перейдем к буферизации каналов.

Буферизованные каналы

Тема этого подраздела — буферизованные каналы. Они позволяют вам быстро помещать задания в очередь, чтобы вы могли работать с большим количеством запросов и обрабатывать их позже. Кроме того, буферизованные каналы можно использовать в качестве семафоров, ограничивая пропускную способность вашего приложения.

Представленная методика работает следующим образом: все входящие запросы перенаправляются на канал, который обрабатывает их по очереди. Завершая обработку запроса, канал отправляет исходному, вызвавшему сообщение о готовности обработать новый запрос. Таким образом, емкость буфера канала ограничивает количество одновременных запросов, которые он может хранить.

Файл, реализующий этот метод, называется `bufChannel.go` и содержит следующий код:

```
package main

import (
    "fmt"
)

func main() {
    numbers := make(chan int, 5)
```

Канал `numbers` не может хранить более пяти целых чисел — это буферный канал с емкостью 5.

```
    counter := 10

    for i := 0; i < counter; i++ {
```

```

select {
    // здесь происходит обработка
    case numbers <- i * i:
        fmt.Println("About to process", i)
    default:
        fmt.Print("No space for ", i, " ")
    }
}

```

Мы начинаем помещать данные в `numbers`, однако когда канал заполнен, он перестанет сохранять данные и будет выполняться ветка `default`.

```

}
fmt.Println()

for {
    select {
    case num := <-numbers:
        fmt.Print("*", num, " ")
    default:
        fmt.Println("Nothing left to read!")
        return
    }
}
}

```

Аналогично, мы пытаемся считывать данные из `numbers`, используя цикл `for`. Когда все данные из канала считаны, выполнится ветка `default` и программа завершится с помощью оператора `return`.

При выполнении `bufChannel.go` мы получаем такой вывод:

```
$ go run bufChannel.go
About to process 0
. .
About to process 4
No space for 5 No space for 6 No space for 7 No space for 8 No space
for 9
*0 *1 *4 *9 *16 Nothing left to read!
```

Теперь обсудим `nil`-каналы.

Nil-каналы

`Nil`-каналы *всегда блокируются!* Следовательно, вы должны использовать их, когда вам требуется именно такое поведение! В приведенном ниже коде показаны `nil`-каналы:

```

package main

import (

```

```

    "fmt"
    "math/rand"
    "sync"
    "time"
}

var wg sync.WaitGroup

```

Мы делаем `wg` глобальной переменной, чтобы она была доступна из любой точки кода и не передавалась каждой функции, которая в ней нуждается.

```

func add(c chan int) {
    sum := 0
    t := time.NewTimer(time.Second)

    for {
        select {
        case input := <-c:
            sum = sum + input
        case <-t.C:
            c = nil
            fmt.Println(sum)
            wg.Done()
        }
    }
}

```

Функция `send()` отправляет случайные числа в канал `c`. Не путайте этот канал, который является параметром функции канала, с каналом `t.C`, который является частью таймера `t`, — вы можете менять имя переменной `c`, но не имя поля `C`. Когда время таймера `t` истекает, он отправляет значение в канал `t.C`.

Это запускает выполнение соответствующей ветки оператора `select`, которая присваивает значение `nil` каналу `c`, выводит значение переменной `sum` и выполняет команду `wg.Done()`, которая разблокирует `wg.Wait()` в функции `main()`.

Кроме того, когда `c` становится равным `nil`, он останавливает/блокирует отправку в него любых данных с помощью `send()`.

```

func send(c chan int) {
    for {
        c <- rand.Intn(10)
    }
}

func main() {
    c := make(chan int)
    rand.Seed(time.Now().Unix())
    wg.Add(1)
    go add(c)
}

```

```
    go send(c)
    wg.Wait()
}
```

При выполнении `nilChannel.go` мы получаем такой вывод:

```
$ go run nilChannel.go
11168960
```

Поскольку количество выполнений первой ветки оператора `select` в `add()` не является фиксированным, каждый раз при выполнении `nilChannel.go` вы будете получать разные результаты.

В следующем подразделе рассматриваются пулы рабочих процессов.

Пулы рабочих процессов

Пул рабочих процессов — это набор потоков, которые обрабатывают назначенные им задания. Веб-сервер Apache и Go-пакет `net/http` работают примерно так: основной процесс принимает все входящие запросы, после чего они пересылаются рабочим процессам для обслуживания. Как только рабочий процесс завершает работу, он готов обслуживать нового клиента. Поскольку Go не имеет потоков, представленная реализация вместо потоков использует горутины. Кроме того, потоки обычно не завершаются после обслуживания запроса из-за высокой стоимости завершения потока и создания нового, в то время как горутины после завершения работы завершаются. Пулы рабочих процессов в Go реализуются с помощью буферизованных каналов, поскольку те позволяют ограничить количество одновременно запущенных горутин.

Представленная утилита реализует простую задачу: она обрабатывает целые числа и выводит их квадраты, используя единственную горутину для обслуживания каждого запроса. Код `wPools.go` выглядит так:

```
package main

import (
    "fmt"
    "os"
    "runtime"
    "strconv"
    "sync"
    "time"
)

type Client struct {
    id      int
    integer int
}
```

Структура `Client` используется для отслеживания запросов, которые программе нужно обработать.

```
type Result struct {
    job    Client
    square int
}
```

Структура `Result` используется для хранения данных каждого `Client`, а также сгенерированных клиентом результатов. Проще говоря, структура `Client` содержит входные данные каждого запроса, в то время как `Result` — результаты запроса. Если вы хотите обрабатывать сложные данные, то вам придется изменить эти структуры.

```
var size = runtime.GOMAXPROCS(0)
var clients = make(chan Client, size)
var data = make(chan Result, size)
```

Канал `clients` и каналы с буферизацией `data` используются для получения новых клиентских запросов и записи результатов соответственно. Если вам требуется ускорить работу программы, то можно увеличить значение `size`.

```
func worker(wg *sync.WaitGroup) {
    for c := range clients {
        square := c.integer * c.integer
        output := Result{c, square}
        data <- output
        time.Sleep(time.Second)
    }
    wg.Done()
}
```

Функция `worker()` обрабатывает запросы, читая канал `clients`. Как только обработка завершена, результат записывается в канал `data`. Задержка, которая вводится с помощью `time.Sleep()`, не является обязательной, но дает лучшее представление при выводе сгенерированных выходных данных.

```
func create(n int) {
    for i := 0; i < n; i++ {
        c := Client{i, i}
        clients <- c
    }
    close(clients)
}
```

Цель функции `create()` — правильно создать все запросы, а затем отправить их в буферизованный канал `clients` для обработки. Обратите внимание, что канал `clients` читается в `worker()`.

```
func main() {
    if len(os.Args) != 3 {
```

```

        fmt.Println("Need #jobs and #workers!")
        return
    }

nJobs, err := strconv.Atoi(os.Args[1])
if err != nil {
    fmt.Println(err)
    return
}

nWorkers, err := strconv.Atoi(os.Args[2])
if err != nil {
    fmt.Println(err)
    return
}

```

В этом коде мы прочитали параметры командной строки, определяющие количество заданий и рабочих процессов. Если количество рабочих процессов превышает размер буферизованного канала `clients`, то количество создаваемыхgorутин равно размеру канала `clients`. Аналогично, если количество заданий больше, чем количество рабочих процессов, то задания обслуживаются меньшими порциями.

```
go create(nJobs)
```

Вызов `create()` имитирует клиентские запросы, которые мы должны обрабатывать.

```
finished := make(chan interface{})
```

Канал `finished` используется для блокировки программы и, следовательно, не нуждается в определенном типе данных.

```

go func() {
    for d := range data {
        fmt.Printf("Client ID: %d\tint: ", d.job.id)
        fmt.Printf("%d\tsquare: %d\n", d.job.integer, d.square)
    }
    finished <- true
}
```

Оператор `finished <- true` используется для разблокировки программы, как только цикл `for range` завершается. Цикл `for range` заканчивается, когда канал `data` закрывается, что происходит после `wg.Wait()`, то есть после окончания работы всех рабочих процессов.

```

}()

var wg sync.WaitGroup
for i := 0; i < nWorkers; i++ {
```

```
        wg.Add(1)
        go worker(&wg)
    }
wg.Wait()
close(data)
```

Цель этого цикла `for` — сгенерировать необходимое количество горутин `worker()` для обработки всех запросов.

```
    fmt.Printf("Finished: %v\n", <-finished)
}
```

Оператор `<-finished` в `fmt.Printf()` блокируется до тех пор, пока не будет закрыт канал `finished`.

При выполнении `wPools.go` мы получаем такой вывод:

```
$ go run wPools.go 10 4
Client ID: 1    int: 1 square: 1
Client ID: 0    int: 0 square: 0
Client ID: 2    int: 2 square: 4
Client ID: 3    int: 3 square: 9
Client ID: 4    int: 4 square: 16
Client ID: 5    int: 5 square: 25
Client ID: 6    int: 6 square: 36
Client ID: 7    int: 7 square: 49
Client ID: 8    int: 8 square: 64
Client ID: 9    int: 9 square: 81
Finished: true
```

В этом выводе показано, что все запросы были обработаны. Такой метод позволяет обслуживать заданное количество запросов, что избавляет нас от перегрузки сервера. Цена, которую мы за это платим, состоит в необходимости писать больше кода.

В следующем подразделе представлены сигнальные каналы и показана методика их использования для определения порядка выполнения в случае небольшого числа горутин.

Сигнальные каналы

Сигнальный канал — это канал, который используется только для передачи сигналов. Проще говоря, вы можете использовать сигнальный канал, когда хотите сообщить о чем-то другой горутине.

Сигнальные каналы не должны использоваться для передачи данных. Сейчас вы увидите работу сигнальных каналов и то, как мы зададим порядок выполнения горутин.

Указание порядка выполнения для вашихgorутин

Здесь демонстрируется метод указания порядка выполнения горутин с помощью *сигнальных каналов*. Однако имейте в виду, что данный метод лучше всего работает, когда мы имеем дело с небольшим количеством горутин. Представленный пример кода содержит четыре горутины, которые мы хотим выполнить в следующем порядке: сначала горутина для функции A(), затем функция B(), затем C() и, наконец, D().

Код `defineOrder.go` без оператора `package` и блока `import` выглядит так:

```
var wg sync.WaitGroup

func A(a, b chan struct{}) {
    <-a
    fmt.Println("A()!")
    time.Sleep(time.Second)
    close(b)
}
```

Функция A() будет заблокирована до тех пор, пока канал a, который передается в качестве параметра, не закроется. Непосредственно перед его завершением он закроет канал b, который передается в качестве параметра. Это разблокирует следующую горутину, а именно функцию B().

```
func B(a, b chan struct{}) {
    <-a
    fmt.Println("B()!")
    time.Sleep(3 * time.Second)
    close(b)
}
```

Аналогично функция B() будет заблокирована до тех пор, пока канал a, который передается в качестве параметра, не будет закрыт. Непосредственно перед завершением B() закроется канал b, который передается в качестве параметра. Как и ранее, это разблокирует следующую функцию.

```
func C(a, b chan struct{}) {
    <-a
    fmt.Println("C()!")
    close(b)
}
```

Как и в случаях с функциями A() и B(), выполнение функции C() блокируется каналом a. Непосредственно перед ее окончанием канал b будет закрыт.

```
func D(a chan struct{}) {
    <-a
    fmt.Println("D()!")
    wg.Done()
}
```

Это последняя функция, которая будет выполнена. Поэтому она, хотя и заблокирована, не закрывает никаких каналов перед выходом. Кроме того, тот факт, что функция является последней, означает, что она может быть выполнена несколько раз, что не так для функций A(), B() и C(), поскольку *канал может быть закрыт только один раз*.

```
func main() {
    x := make(chan struct{})
    y := make(chan struct{})
    z := make(chan struct{})
    w := make(chan struct{})
```

Нам необходимо иметь столько каналов, сколько функций мы хотим выполнить в виде горутин.

```
wg.Add(1)
go func() {
    D(w)
}()
```

Это доказывает, что порядок выполнения, диктуемый Go-кодом, не имеет значения, поскольку D() будет выполняться последней.

```
wg.Add(1)
go func() {
    D(w)
}()
go A(x, y)

wg.Add(1)
go func() {
    D(w)
}()

go C(z, w)
go B(y, z)
```

Мы запускаем C() перед B(), однако C() завершится после завершения B().

```
wg.Add(1)
go func() {
    D(w)
}()

// это запускает процесс
close(x)
```

Закрытие первого канала — это то, что запускает выполнение горутин, поскольку разблокирует A().

```
wg.Wait()
}
```

При выполнении `defineOrder.go` мы получаем такой вывод:

```
$ go run defineOrder.go
A()!
B()!
C()!
D()! D()! D()!
```

Таким образом, четыре функции, которые выполняются как горутины, запускаются в желаемом порядке, а в случае последней — еще и желаемое количество раз. В следующем разделе рассказывается об общей памяти и общих переменных, что является очень удобным способом заставить горутины взаимодействовать друг с другом.

Общая память и общие переменные

Разделяемая память и совместно используемые переменные — это важные темы в параллельном программировании и наиболее распространенные способы взаимодействия потоков UNIX друг с другом. Те же принципы применимы к Go и горутинам, о чем и пойдет речь в данном разделе. Переменная *mutex*, название которой является аббревиатурой к *mutual exclusion variable* (*переменной взаимного исключения*), в основном служит для синхронизации потоков и защиты совместно используемых данных, когда одновременно может выполняться несколько операций записи. *Мьютекс работает как буферизованный канал с пропускной способностью в единицу*, что позволяет не более чем одной горутине получать доступ к общей переменной в один момент времени. Это означает, что у двух или более горутин нет возможности обновить эту переменную одновременно. Go содержит типы данных `sync.Mutex` и `sync.RWMutex`.

Критический раздел параллельной программы — это код, который не может быть выполнен одновременно всеми процессами, потоками или, в данном случае, горутинами. Это код, который должен быть защищен мьютексами. Таким образом, определение критических разделов вашего кода делает весь процесс программирования настолько более простым, что вам следует уделить этой задаче особое внимание. Критический раздел не может быть встроен в другой критический раздел, если оба они используют одну и ту же переменную `sync.Mutex` или `sync.RWMutex`.

Проще говоря, любой ценой избегайте распространения мьютексов по функциям, так как это действительно затрудняет понимание того, внедряете вы их или нет.

Тип sync.Mutex

Тип `sync.Mutex` — это реализация *мьюотекса* в Go. Его определение, расположенное в файле `mutex.go` каталога `sync`, выглядит следующим образом (вам не нужно знать определение `sync.Mutex` для его использования):

```
type Mutex struct {
    state int32
    sema  uint32
}
```

В определении `sync.Mutex` нет ничего особенного. Все самое интересное происходит в `sync.Lock()` и `sync.Unlock()`, которые могут соответственно блокировать и разблокировать переменную `sync.Mutex`. Блокировка мьюотекса означает, что никто другой не может заблокировать его до тех пор, пока он не будет освобожден с помощью функции `sync.Unlock()`. Все это показано в файле `mutex.go`:

```
package main

import (
    "fmt"
    "os"
    "strconv"
    "sync"
    "time"
)

var m sync.Mutex
var v1 int

func change(i int) {
    m.Lock()

    time.Sleep(time.Second)
    v1 = v1 + 1
    if v1 == 10 {
        v1 = 0
        fmt.Println("* ")
    }
    m.Unlock()
}
```

Эта функция меняет значение `v1`. Здесь начинается критический раздел.

```
    time.Sleep(time.Second)
    v1 = v1 + 1
    if v1 == 10 {
        v1 = 0
        fmt.Println("* ")
    }
    m.Unlock()
```

Конец критического раздела. Теперь другаяgorутина получает возможность заблокировать мьюотекс.

```
}
```

```
func read() int {
    m.Lock()
```

```
a := v1
m.Unlock()
return a
}
```

Эта функция используется для считывания значения `v1`, поэтому здесь необходимо использовать мьютекс, чтобы сделать процесс безопасным в плане параллелизма. В частности, мы хотим убедиться, что никто не попытается изменить значение `v1`, пока мы его читаем. Остальная часть программы содержит реализацию функции `main()` — не поленитесь ознакомиться с полным кодом `mutex.go` в репозитории книги на GitHub.

При выполнении `mutex.go` мы получаем такой вывод:

```
$ go run -race mutex.go 10
0 -> 1-> 2-> 3-> 4-> 5-> 6-> 7-> 8-> 9* -> 0-> 0
```

В этом выводе видно, что из-за использования мьютекса горутины не могут получить доступ к общим данным и поэтому не возникает скрытых состояний гонки.

В следующем пункте показано, что может произойти, если мы забудем разблокировать мьютекс.

Что будет, если забыть разблокировать мьютекс

Если забыть разблокировать мьютекс `sync.Mutex`, то даже в самой простой программе возникнет ситуация паники. То же самое относится и к мьютексу `sync.RWMutex`, который представлен далее.

Чтобы лучше понять эту неприятную ситуацию, рассмотрим пример кода, часть программы `forgetMutex.go`.

```
var m sync.Mutex
var w sync.WaitGroup

func function() {
    m.Lock()
    fmt.Println("Locked!")
}
```

Здесь мы блокируем мьютекс, не освобождая его впоследствии. Это означает, что если мы запустим `function()` как горутину более одного раза, то все экземпляры после первого будут заблокированы в ожидании `Lock()` общего мьютекса. В нашем случае мы запустим две горутины — не поленитесь ознакомиться с полным кодом `forgetMutex.go`, чтобы получить более полное представление. При выполнении `forgetMutex.go` мы получаем такой вывод:

```
Locked!
fatal error: all goroutines are asleep - deadlock!

goroutine 1 [semacquire]:
sync.runtime_Semacquire(0x118d3e8)
    /usr/local/Cellar/go/1.16.2/libexec/src/runtime/sema.go:56
+0x45
sync.(*WaitGroup).Wait(0x118d3e0)
    /usr/local/Cellar/go/1.16.2/libexec/src/sync/waitgroup.go:130
+0x65
main.main()
    /Users/mtsouk/ch07/forgetMutex.go:29 +0x95

goroutine 18 [semacquire]:
sync.runtime_SemacquireMutex(0x118d234, 0x0, 0x1)
    /usr/local/Cellar/go/1.16.2/libexec/src/runtime/sema.go:71
+0x47
sync.(*Mutex).lockSlow(0x118d230)
    /usr/local/Cellar/go/1.16.2/libexec/src/sync/mutex.go:138
+0x105
sync.(*Mutex).Lock(...)
    /usr/local/Cellar/go/1.16.2/libexec/src/sync/mutex.go:81
main.function()
    /Users/mtsouk/ch07/forgetMutex.go:12 +0xac
main.main.func1()
    /Users/mtsouk/ch07/forgetMutex.go:20 +0x4c
created by main.main
    /Users/mtsouk/ch07/forgetMutex.go:18 +0x52
exit status 2
```

Как и ожидалось, из-за тупиковой ситуации в программе возникает сбой. Чтобы избежать подобных ситуаций, не забывайте о разблокировке любых мьютексов, созданных в вашей программе.

Теперь обсудим `sync.RWMutex`, представляющий собой улучшенную версию `sync.Mutex`.

Тип `sync.RWMutex`

Тип данных `sync.RWMutex` является улучшенной версией `sync.Mutex` и определяется в файле `rwmutex.go` каталога `sync` стандартной библиотеки Go следующим образом:

```
type RWMutex struct {
    w           Mutex
    writerSem   uint32
    readerSem   uint32
    readerCount int32
    readerWait  int32
}
```

Другими словами, `sync.RWMutex` основан на `sync.Mutex` и имеет необходимые дополнения и улучшения. Вы можете спросить, а как `sync.RWMutex` улучшает `sync.Mutex`? Хотя одной функции разрешено выполнять операции записи с мьютексом `sync.RWMutex`, у вас может быть *несколько считывателей*, владеющих мьютексом `sync.RWMutex`. Это означает, что операции чтения с помощью `sync.RWMutex` обычно выполняются быстрее. Тем не менее есть одна важная деталь, о которой следует знать: пока *все считыватели* мьютекса `sync.RWMutex` не разблокируют его, вы не сможете заблокировать его для записи. Такова небольшая цена, которую нам придется платить за повышение производительности, получаемое за счет возможности иметь несколько считывателей.

Функциями, которые могут помочь в работе с `sync.RWMutex`, являются `RLock()` и `RUnlock()`. Они используются для блокировки и разблокировки мьютекса для чтения. Функции `Lock()` и `Unlock()`, используемые в `sync.Mutex`, точно так же следует применять, когда нужно заблокировать и разблокировать мьютекс `sync.RWMutex` для записи. Наконец, должно быть очевидно, что нельзя вносить изменения в какие-либо общие переменные внутри блоков кода `RLock()` и `RUnlock()`.

Все это показано в файле `RWMutex.go`. Наиболее важным кодом здесь является следующий:

```
var Password *secret
var wg sync.WaitGroup

type secret struct {
    RWM      sync.RWMutex
    password string
}
```

Это общая переменная программы — вы можете предоставить общий доступ к любому желаемому типу переменной.

```
func Change(pass string) {
    fmt.Println("Change() function")
    Password.RWM.Lock()
```

Это начало критического раздела.

```
    fmt.Println("Change() Locked")
    time.Sleep(4 * time.Second)
    Password.password = pass
    Password.RWM.Unlock()
```

А это конец критического раздела.

```
    fmt.Println("Change() UnLocked")
}
```

Функция `Change()` вносит изменения в общую переменную `Password` и, следовательно, должна использовать функцию `Lock()`, которая может выполняться только одним записывателем.

```
func show() {
    defer wg.Done()
    Password.RWM.RLock()
    fmt.Println("Show function locked!")
    time.Sleep(2 * time.Second)
    fmt.Println("Pass value:", Password.password)
    defer Password.RWM.RUnlock()
}
```

Функция `show()` считывает общую переменную `Password` и, следовательно, может использовать функцию `RLock()`, которая может храниться несколькими считывателями. Внутри `main()` три функции `show()` выполняются как горутины перед вызовом функции `Change()`, которая также выполняется как горутина. Ключевым моментом здесь является тот факт, что никаких состояний гонки не ожидается. При выполнении `RWMutex.go` мы получаем такой вывод:

```
$ go run rwMutex.go
Change() function
```

Функция `Change()` выполняется, но не может получить мьютекс, поскольку он уже занят одной или несколькими горутинаами `show()`.

```
Show function locked!
Show function locked!
```

Вывод выше подтверждает, что две программы `show()` успешно забрали мьютекс для чтения.

```
Change() function
```

Здесь мы видим, что вторая функция `Change()` запущена и ожидает получения мьютекса.

```
Pass value: myPass
Pass value: myPass
```

Это вывод двух горутин `show()`.

```
Change() Locked
Change() UnLocked
```

Мы видим, что однаgorутина `Change()` завершает свою работу.

```
Show function locked!
Pass value: 54321
```

После этого завершается другая горутина `show()`.

```
Change() Locked
Change() UnLocked
Current password value: 123456
```

Наконец, вторая горутина `Change()` тоже завершается. Последняя строка вывода предназначена для проверки того, что значение пароля изменилось. Пожалуйста, обратитесь к полному коду `RWMutex.go`, чтобы получить более подробную информацию.

В следующем подразделе мы обсудим, как с помощью пакета `atomic` избегать состояний гонки.

Пакет `atomic`

Атомарная операция — это операция, которая выполняется за один шаг относительно других потоков или, в данном случае, других горутин. Это означает, что *атомарную операцию нельзя прервать в середине ее работы*. Стандартная библиотека Go содержит пакет `atomic`, который в некоторых простых случаях может помочь избежать использования мьютекса. С помощью этого пакета вы получаете доступ к атомарным счетчикам из нескольких горутин, не имея проблем с синхронизацией и не беспокоясь о состояниях гонки. Однако мьютесксы более универсальны, чем атомарные операции.

Как показано в следующем коде, при использовании атомарной переменной во избежание состояний гонки *все операции чтения и записи атомарной переменной должны выполняться с помощью функций, предоставляемых пакетом `atomic`*.

Код в файле `atomic.go` выглядит следующим образом (он уменьшен за счет жесткого кодирования некоторых значений):

```
package main

import (
    "fmt"
    "sync"
    "sync/atomic"
)

type atomCounter struct {
    val int64
}
```

Это структура для хранения требуемой атомарной переменной `int64`.

```
func (c *atomCounter) Value() int64 {
    return atomic.LoadInt64(&c.val)
}
```

Это вспомогательная функция, которая возвращает текущее значение атомарной переменной `int64`, используя `atomic.LoadInt64()`.

```
func main() {
    X := 100
    Y := 4
    var waitGroup sync.WaitGroup
    counter := atomCounter{}
    for i := 0; i < X; i++ {
```

Мы создаем множество горутин, которые изменяют общую переменную. Как указывалось ранее, благодаря использованию пакета `atomic` для работы с общей переменной мы получаем простой способ избежать состояний гонки при изменении ее значения.

```
    waitGroup.Add(1)
    go func(no int) {
        defer waitGroup.Done()
        for i := 0; i < Y; i++ {
            atomic.AddInt64(&counter.val, 1)
        }
    }
```

Функция `atomic.AddInt64()` безопасно изменяет значение поля `val` структуры `counter`.

```
    }(i)
}

waitGroup.Wait()
fmt.Println(counter.Value())
}
```

При выполнении `atomic.go` в ходе проверки состояний гонки мы получаем такой вывод:

```
$ go run -race atomic.go
400
```

Таким образом, атомарная переменная изменяется несколькими горутинами без каких-либо проблем.

В следующем подразделе показано, как с помощью горутин совместно использовать память.

Совместное использование памяти с помощью горутин

В данном подразделе показано, как обмениваться данными с помощью *выделенной горутины*. Хотя общая память является традиционным способом взаимодействия потоков друг с другом, Go поставляется со встроенными функциями синхронизации, которые позволяют единственной горутине владеть общим фрагментом данных. Это означает, что другие горутины должны отправлять сообщения в ту, которой принадлежат общие данные, благодаря чему и предотвращается повреждение данных. Такая горутина называется *контролирующей*. В терминологии Go это *обмен информацией путем взаимодействия, а не взаимодействие путем совместного использования*.



Я предпочитаю использовать контролирующую горутину вместо традиционных методов с общей памятью, поскольку работа с ней безопаснее, ближе к философии Go и более проста для понимания.

Логику работы программы можно увидеть в реализации функции `monitor()`. А если более конкретно, то работой всей программы управляет оператор `select`. Когда возникает запрос на чтение, функция `read()` пытается выполнить чтение из канала `readValue`, который управляется функцией `monitor()`.

Это возвращает текущее значение переменной `value`. В то же время при желании изменить сохраненное значение вы вызываете `set()`. При этом выполняется запись в канал `writeValue`, который обрабатывается тем же оператором `select`. В результате никто не может задействовать общую переменную, не используя функцию `monitor()`, которая за это отвечает.

Код `monitor.go` выглядит следующим образом:

```
package main

import (
    "fmt"
    "math/rand"
    "os"
    "strconv"
    "sync"
    "time"
)

var readValue = make(chan int)
var writeValue = make(chan int)
```

```
func set(newValue int) {
    writeValue <- newValue
}
```

Функция отправляет данные в канал `writeValue`.

```
func read() int {
    return <-readValue
}
```

Когда вызывается функция `read()`, данныечитываются из канала `readValue`, и это считывание происходит внутри функции `monitor()`.

```
func monitor() {
    var value int
    for {
        select {
        case newValue := <-writeValue:
            value = newValue
            fmt.Printf("%d ", value)
        case readValue <- value:
        }
    }
}
```

Функция `monitor()` содержит логику программы с бесконечным циклом `for` и оператором `select`. Первый раздел получает данные из канала `writeValue`, соответствующим образом устанавливает переменную `value` и выводит это новое значение. Второй раздел отправляет значение переменной `value` в канал `readValue`. Поскольку весь трафик проходит через функцию `monitor()` и ее блок `select`, нет никакой возможности получить состояние гонки, так как `monitor()` запущена в единственном экземпляре.

```
func main() {
    if len(os.Args) != 2 {
        fmt.Println("Please give an integer!")
        return
    }
    n, err := strconv.Atoi(os.Args[1])
    if err != nil {
        fmt.Println(err)
        return
    }

    fmt.Printf("Going to create %d random numbers.\n", n)
    rand.Seed(time.Now().Unix())
    go monitor()
```

Важно, чтобы функция `monitor()` выполнялась первой, поскольку это горутина, которая управляет потоком программы.

```
var wg sync.WaitGroup

for r := 0; r < n; r++ {
    wg.Add(1)
    go func() {
        defer wg.Done()
        set(rand.Intn(10 * n))
    }()
}
}
```

Когда цикл `for` заканчивается, это означает, что мы создали желаемое количество случайных чисел.

```
wg.Wait()
fmt.Printf("\nLast value: %d\n", read())
}
```

Наконец, мы ожидаем завершения всех процедур `set()`, после чего выводим последнее случайное число.

При выполнении `monitor.go` мы получаем такой вывод:

```
$ go run monitor.go 10
Going to create 10 random numbers.
98 22 5 84 20 26 45 36 0 16
Last value: 16
```

Итак, десять случайных чисел генерируются десятью горутинами, и все они отправляют свои выходные данные в функцию `monitor()`, которая также выполняется как горутина. Помимо получения результатов, функция `monitor()` выводит их на экран, так что весь этот вывод генерируется `monitor()`.

В следующем разделе более подробно рассматривается инструкция `go`.

Закрытые переменные и оператор `go`

В этом разделе мы поговорим о *закрытых переменных*, которые являются переменными внутри замыканий, и операторе `go`. Обратите внимание, что закрытые переменные в горутинах вычисляются при фактическом запуске и при выполнении оператора `go` в целях создания новой горутины. Это означает, что закрытые переменные будут заменены их значениями, когда планировщик Go решит выполнить соответствующий код. Это показано в функции `main()` программы `goClosure.go`:

```
func main() {
    for i := 0; i <= 20; i++ {
        go func() {
            fmt.Println(i, " ")
        }()
    }
    time.Sleep(time.Second)
    fmt.Println()
}
```

При выполнении `goClosure.go` мы получаем такой вывод:

```
$ go run goClosure.go
3 7 21 21 21 21 21 21 21 21 21 21 21 21 21 21 21 21 21 21 21 21 21
```

В основном программа выводит число 21, которое является последним значением переменной цикла `for`, а не другие числа. Поскольку `i` является *закрытой переменной*, она *вычисляется во время выполнения*. Горутины запускаются, но при этом ждут, пока планировщик Go разрешит их выполнение, поэтому цикл `for` заканчивается, вследствие чего используемое значение `i` равно 21. Нужно отметить, что та же проблема касается и каналов Go, поэтому будьте осторожны.

При выполнении `goClosure.go` с Go-детектором состояний гонки выявляется проблема:

```
$ go run -race goClosure.go
=====
WARNING: DATA RACE
Read at 0x00c00013a008 by goroutine 7:
  main.main.func1()
    /Users/mtsouk/ch07/goClosure.go:11 +0x3c

Previous write at 0x00c00013a008 by main goroutine:
  main.main()
    /Users/mtsouk/ch07/goClosure.go:9 +0xa4

Goroutine 7 (running) created at:
  main.main()
    /Users/mtsouk/ch07/goClosure.go:10 +0x7e
=====
2 3 5 5 7 8 9 10 9 11 12 13 14 17 18 18 18 19 20 21
Found 1 data race(s)
exit status 66
```

Исправим файл `goClosure.go` и дадим ему новое имя, а именно `goClosureCorrect.go`. Его функция `main()` выглядит следующим образом:

```
func main() {
    for i := 0; i <= 20; i++ {
```

```
i := i
go func() {
    fmt.Println(i, " ")
}()
}
```

Это лишь один из способов устранения проблемы. Допустимый, но странный оператор `i := i` создает новый экземпляр переменной для горутины, которая содержит правильное значение.

```
time.Sleep(time.Second)
fmt.Println()

for i := 0; i <= 20; i++ {
    go func(x int) {
        fmt.Println(x, " ")
    }(i)
}
```

Вот совершенно другой способ избежания состояний гонки: передать текущее значение `i` анонимной функции в качестве параметра.

```
time.Sleep(time.Second)
fmt.Println()
}
```

Тестирование `goClosureCorrect.go` с детектором состояния гонки генерирует ожидаемый результат:

```
$ go run -race goClosureCorrect.go
0 1 2 4 3 5 6 9 8 7 10 11 13 12 14 16 15 17 18 20 19
0 1 2 3 4 5 6 7 8 10 9 12 13 11 14 15 16 17 18 19 20
```

В следующем разделе представлена функциональность пакета `context`.

Пакет `context`

Основная задача пакета `context` — определение типа `Context` и поддержка *отмены*. Да, вы правильно расслышали; бывают моменты, когда по какой-то причине требуется прервать то, что вы делаете. Однако было бы весьма полезно получить возможность включить некоторую дополнительную информацию о ваших решениях об отмене. Именно для этого и предназначен пакет `context`.

Если вы взглянете на исходный код пакета `context`, то увидите, что его реализация довольно проста. Несмотря на его исключительную важность, простой является даже реализация типа `Context`.

Тип `Context` представляет собой интерфейс с четырьмя методами: `Deadline()`, `Done()`, `Err()` и `Value()`. Хорошой новостью является то, что вам не требуется реализовывать все эти функции интерфейса `Context`, а нужно просто изменить переменную `Context`, используя такие методы, как `context.WithCancel()`, `context.WithDeadline()` и `context.WithTimeout()`.



Все три функции возвращают производный (дочерний) `Context` и функцию `CancelFunc()`. Вызов `CancelFunc()` удаляет родительскую ссылку на дочернюю и останавливает все связанные с ней таймеры. В качестве побочного эффекта это означает, что сборщик мусора Go сможет свободно собрать дочерние горутины, с которыми больше не связаны родительские. Чтобы сборка мусора сработала правильно, родительская горутина должна сохранять ссылку на каждую дочернюю горутину. Если та завершается без ведома родителя, происходит утечка памяти, длившаяся до тех пор, пока родительская программа также не будет отменена.

В приведенном ниже примере показано использование пакета `context`. Программа содержит четыре функции, включая `main()`. Функции `f1()`, `f2()` и `f3()` требуют только одного параметра, который является временной задержкой, а все остальное определено внутри их тела. В этом примере мы используем `context.Background()` для инициализации пустого `Context`. Другая функция, которая может создавать пустой `Context`, — это `context.TODO()`, представленная далее в этой главе.

```
package main

import (
    "context"
    "fmt"
    "os"
    "strconv"
    "time"
)

func f1(t int) {
    c1 := context.Background()
    c1, cancel := context.WithCancel(c1)
    defer cancel()

    // ...
```

Метод `WithCancel()` возвращает копию родительского контекста с новым каналом `Done`. Обратите внимание, что переменная `cancel` (на самом деле это функция) является одним из возвращаемых значений `context.CancelFunc()`. Функция `context.WithCancel()` использует существующий `Context` и создает дочерний элемент с отменой. Функция `context.WithCancel()` также возвращает

канал `Done`, который может быть закрыт либо при вызове функции `cancel()` (что показано в коде выше), либо при закрытии канала `Done` родительского контекста.

```
go func() {
    time.Sleep(4 * time.Second)
    cancel()
}()

select {
case <-c1.Done():
    fmt.Println("f1() Done:", c1.Err())
    return
case r := <-time.After(time.Duration(t) * time.Second):
    fmt.Println("f1():", r)
}
return
}
```

Функция `f1()` создает и выполняетgorутину. Вызов `time.Sleep()` имитирует время, которое потребовалось бы реальной горутине для выполнения своей работы. В данном случае это четыре секунды, но вы можете указать любой желаемый интервал времени. Если контекст `c1` вызывает функцию `Done()` менее чем за четыре секунды, то горутине не будет хватать времени для завершения.

```
func f2(t int) {
    c2 := context.Background()
    c2, cancel := context.WithTimeout(c2, time.Duration(t)*time.Second)
    defer cancel()
```

Переменная `cancel` в `f2()` берется из `context.WithTimeout()`, которому требуются два параметра: `Context` и `time.Duration`. По истечении времени ожидания функция `cancel()` вызывается автоматически.

```
go func() {
    time.Sleep(4 * time.Second)
    cancel()
}()

select {
case <-c2.Done():
    fmt.Println("f2() Done:", c2.Err())
    return
case r := <-time.After(time.Duration(t) * time.Second):
    fmt.Println("f2():", r)
}
return
}

func f3(t int) {
    c3 := context.Background()
```

```
deadline := time.Now().Add(time.Duration(2*t) * time.Second)
c3, cancel := context.WithDeadline(c3, deadline)
defer cancel()
```

Переменная `cancel` в `f3()` берется из `context.WithDeadline()`, которому требуются два параметра: переменная `Context` и время в будущем, задающее крайний срок выполнения операции. По истечении крайнего срока автоматически вызывается функция `cancel()`.

```
go func() {
    time.Sleep(4 * time.Second)
    cancel()
}()

select {
case <-c3.Done():
    fmt.Println("f3() Done:", c3.Err())
    return
case r := <-time.After(time.Duration(t) * time.Second):
    fmt.Println("f3():", r)
}
return
}
```

Логика в `f3()` такая же, как в `f1()` и `f2()`, — процессом управляет блок `select`.

```
func main() {
    if len(os.Args) != 2 {
        fmt.Println("Need a delay!")
        return
    }

    delay, err := strconv.Atoi(os.Args[1])
    if err != nil {
        fmt.Println(err)
        return
    }
    fmt.Println("Delay:", delay)

    f1(delay)
    f2(delay)
    f3(delay)
}
```

Эти три функции последовательно выполняются функцией `main()`.

При выполнении `useContext.go` мы получаем такой вывод:

```
$ go run useContext.go 3
Delay: 3
f1(): 2021-03-18 13:10:24.739381 +0200 EET m=+3.001331808
f2(): 2021-03-18 13:10:27.742732 +0200 EET m=+6.004804424
f3(): 2021-03-18 13:10:30.742793 +0200 EET m=+9.004988055
```

Длинные строки в выходных данных — это возвращаемые значения `time.After()`, которые показывают время, в течение которого `After()` отправлял текущее время по возвращаемому каналу. Все они обозначают нормальную работу программы.

Если вы определите большую задержку, то результат будет выглядеть примерно так:

```
$ go run useContext.go 13
Delay: 13
f1() Done: context canceled
f2() Done: context canceled
f3() Done: context canceled
```

Дело здесь в том, что работа программы отменяется при возникновении задержек в выполнении.

В следующем подразделе показан еще один способ использования пакета `context`.

Использование контекста в качестве хранилища ключей и значений. В этом подразделе мы передаем в `Context` значения и используем его как хранилище ключей и значений. В данном случае мы не передаем значения в контексты, чтобы предоставить дополнительную информацию о том, почему они были отменены. В программе `keyVal.go` показано использование функции `context.TODO()`, а также `contextWithValue()`.

Все это и многое другое можно найти в программе `keyVal.go`, которая выглядит следующим образом.

```
package main

import (
    "context"
    "fmt"
)

type aKey string

func searchKey(ctx context.Context, k aKey) {
    v := ctx.Value(k)
    if v != nil {
        fmt.Println("found value:", v)
        return
    } else {
        fmt.Println("key not found:", k)
    }
}
```

Функция `SearchKey()` извлекает значение из переменной `Context` с помощью `Value()` и проверяет, существует ли это значение.

```
func main() {
    myKey := aKey("mySecretValue")
    ctx := context.WithValue(context.Background(), myKey, "mySecret")
```

Функция `contextWithValue()`, используемая в `main()`, дает способ связать значение с `Context`. Следующие два оператора выполняют поиск значений двух ключей в существующем контексте (`ctx`).

```
searchKey(ctx, myKey)
searchKey(ctx, aKey("notThere"))
emptyCtx := context.TODO()
```

На этот раз мы создаем контекст, используя `context.TODO()` вместо `context.Background()`. Хотя обе функции возвращают ненулевой пустой `Context`, их цели различаются. Вы никогда не должны передавать контекст `nil`. Для создания подходящего контекста используйте функцию `context.TODO()`. Кроме того, применяйте функцию `context.TODO()`, если не уверены в том, какой `Context` хотите использовать. Функция `context.TODO()` означает, что мы намерены использовать контекст операций, но не уверены в этом до конца.

```
searchKey(emptyCtx, aKey("notThere"))
}
```

При выполнении `keyVal.go` мы получаем такой вывод:

```
$ go run keyVal.go
found value: mySecret
key not found: notThere
key not found: notThere
```

Первый вызов `SearchKey()` выполняется успешно, в то время как следующие два не могут найти в контексте нужный ключ. Таким образом, контексты позволяют хранить пары ключей и значений и выполнять поиск ключей.

Мы еще не закончили с `context`, поскольку в следующей главе будем использовать его для времени ожидания HTTP-взаимодействий на клиентской стороне соединения. В последнем разделе этой главы мы обсудим пакет `semaphore`, который не является частью стандартной библиотеки.

Пакет `semaphore`

В последнем разделе этой главы описывается пакет `semaphore`, который предоставляется командой Go. *Семафор* – это конструкция, которая может ограничивать или контролировать доступ к общему ресурсу. Поскольку мы говорим о Go, семафор может ограничить доступ горутин к общему ресурсу, но первоначально семафоры использовались для ограничения доступа к потокам.

Семафоры могут иметь *веса*, которые задают максимальное количество потоков или горутин, получающих доступ к ресурсу.

Процесс поддерживается с помощью методов `Acquire()` и `Release()`, определенных следующим образом:

```
func (s *Weighted) Acquire(ctx context.Context, n int64) error
func (s *Weighted) Release(n int64)
```

Второй параметр `Acquire()` определяет вес семафора.

Поскольку мы собираемся использовать внешний пакет, нам нужно поместить код внутрь `~/go/src`, чтобы использовать Go-модули: `~/go/src/github.com/mactsouk/semaphore`. Теперь рассмотрим код `semaphore.go`, в котором показана реализация *пула рабочих процессов* с применением семафоров:

```
package main

import (
    "context"
    "fmt"
    "os"
    "strconv"
    "time"

    "golang.org/x/sync/semaphore"
)

var Workers = 4
```

Эта переменная определяет максимальное количество горутин, которые могут быть выполнены данной программой.

```
var sem = semaphore.NewWeighted(int64(Workers))
```

Здесь мы определяем семафор с весом, идентичным максимальному количеству горутин, которые могут выполняться одновременно. Это означает, что получать семафор одновременно могут не более чем `Workers` горутин.

```
func worker(n int) int {
    square := n * n
    time.Sleep(time.Second)
    return square
}
```

Функция `worker()` выполняется как часть горутины. Однако поскольку мы используем семафор, нет необходимости возвращать результаты в канал.

```
func main() {
    if len(os.Args) != 2 {
        fmt.Println("Need #jobs!")
```

```

        return
    }

nJobs, err := strconv.Atoi(os.Args[1])
if err != nil {
    fmt.Println(err)
    return
}

```

Здесь мы считываем количество заданий, которые хотим запустить.

```

// где хранить результаты
var results = make([]int, nJobs)
// требуется для Acquire()
ctx := context.TODO()

for i := range results {
    err = sem.Acquire(ctx, 1)
    if err != nil {
        fmt.Println("Cannot acquire semaphore:", err)
        break
    }
}

```

В этой части мы пытаемся получить семафор столько раз, сколько заданий определено `nJobs`. Если `nJobs` больше, чем `Workers`, то вызов `Acquire()` будет заблокирован и дождется вызовов `Release()` для разблокировки.

```

go func(i int) {
    defer sem.Release(1)
    temp := worker(i)
    results[i] = temp
}(i)
}

```

Здесь мы запускаем горутины, которые выполняют эту задачу, и записываем результаты в срез `results`. Поскольку каждая горутина записывает данные в свой элемент среза, никаких состояний гонки нет.

```

err = sem.Acquire(ctx, int64(Workers))
if err != nil {
    fmt.Println(err)
}

```

Это весьма хитрый трюк: мы получаем все токены таким образом, чтобы вызов `sem.Acquire()` блокировался до тех пор, пока все рабочие процессы/горутины не завершат работу. Функционально это похоже на вызов `Wait()`.

```

for k, v := range results {
    fmt.Println(k, "->", v)
}

```

Последняя часть программы посвящена выводу результатов. После написания кода нам нужно выполнить следующие команды, чтобы получить необходимые Go-модули:

```
$ go mod init  
$ go mod tidy  
$ mod download golang.org/x/sync
```

Кроме первой, эти команды были указаны в выходных данных `go mod init`, так что вам не придется ничего запоминать.

В итоге запуск программы `semaphore.go` выдает следующий результат:

```
$ go run semaphore.go 6  
0 -> 0  
1 -> 1  
2 -> 4  
3 -> 9  
4 -> 16  
5 -> 25
```

Каждая строка в выводе показывает входное и выходное значения, разделенные символом `->`. Использование семафора поддерживает порядок происходящего.

Упражнения

- Попробуйте реализовать параллельную версию `wc(1)`, использующую буферизованный канал.
- Попробуйте реализовать параллельную версию `wc(1)`, которая использует общую память.
- Попробуйте реализовать параллельную версию `wc(1)`, использующую семафоры.
- Попробуйте реализовать параллельную версию `wc(1)`, которая сохраняет свои выходные данные в файл.
- Измените `wPools.go` так, чтобы каждый рабочий процесс реализовывал функциональность `wc(1)`.

Резюме

Эта важная глава была посвящена параллелизму Go,gorутинам,каналам,ключевому слову `select`,общей памяти и мьютексам,а также времени выполнения gorутин и использованию пакета `context`. Все эти знания позволят вам писать эффективные параллельные Go-приложения. Не стесняйтесь экспериментиро-

ваться с концепциями и примерами из этой главы, чтобы лучше разобраться с горутинами, каналами и общей памятью.

Следующая глава посвящена веб-сервисам и работе с протоколом HTTP в Go. Среди прочего мы собираемся преобразовать приложение телефонной книги в веб-сервис.

Дополнительные ресурсы

- Страница документации по синхронизации: <https://golang.org/pkg/sync/>.
- Информация о пакете `semaphore`: <https://pkg.go.dev/golang.org/x/sync/semaphore>.
- Узнайте больше о планировщике Go, прочитав серию статей, начав с <https://www.ardanlabs.com/blog/2018/08/scheduling-in-go-part1.html>.
- Реализация планировщика Go: <https://golang.org/src/runtime/proc.go>.

8

Создание веб-сервисов

Основной темой этой главы является работа с HTTP с использованием пакета `net/http`. Помните, что для работы любого веб-сервиса требуется веб-сервер. Кроме того, в этой главе мы собираемся преобразовать приложение телефонной книги в веб-приложение, которое принимает HTTP-соединения, а также создать клиент командной строки для удобной работы с ним. Наконец, мы выясним, как создавать серверы *FTP* (*протокол передачи файлов*), экспортить метрики из приложений Go в Prometheus и работать с пакетом `runtime/metrics`, чтобы получать заданные реализацией метрики, которые экспортятся средой выполнения Go.

В частности, в этой главе рассматриваются:

- пакет `net/http`;
- создание веб-сервера;
- обновление приложения телефонной книги;
- предоставление метрик для Prometheus;
- разработка веб-клиентов;
- создание клиента для сервиса телефонной книги;
- создание файловых серверов;
- время ожидания HTTP-соединений.

Пакет net/http

Пакет `net/http` содержит функции, которые позволяют разрабатывать веб-серверы и клиенты. Например, `http.Get()` и `http.NewRequest()` используются клиентами для выполнения HTTP-запросов, тогда как `http.ListenAndServe()` — для запуска веб-серверов через указание IP-адреса и TCP-порта, которые прослушивает сервер. Кроме того, `http.HandleFunc()` определяет поддерживаемые URL, а также функции, которые будут их обрабатывать.

В следующих трех подразделах описываются три важные структуры данных пакета `net/http` — вы можете использовать эти описания в качестве справки при чтении этой главы.

Тип http.Response

Структура `http.Response` реализует ответ на HTTP-запрос — как `http.Client`, так и `http.Transport` возвращают значения `http.Response` после получения заголовков ответов. Его определение можно найти по адресу <https://golang.org/src/net/http/response.go>:

```
type Response struct {
    Status      string // например, "200 OK"
    StatusCode int    // например, 200
    Proto       string // например, "HTTP/1.0"
    ProtoMajor  int    // например, 1
    ProtoMinor  int    // например, 0
    Header      Header
    Body        io.ReadCloser
    ContentLength int64
    TransferEncoding []string
    Close        bool
    Uncompressed bool
    Trailer      Header
    Request     *Request
    TLS          *tls.ConnectionState
}
```

Вам не обязательно задействовать все поля структуры, но всегда полезно знать, что они существуют. Однако некоторые из них, такие как `Status`, `StatusCode` и `Body`, особенно важны. Исходный файл Go, как и выходные данные `go doc http.Response`, содержит дополнительную информацию о назначении каждого поля, что также относится к большинству структурных типов данных стандартной библиотеки Go.

Тип `http.Request`

Тип `http.Request` представляет собой HTTP-запрос, созданный клиентом для отправки или получения HTTP-сервером. Общедоступные поля `http.Request` выглядят так:

```
type Request struct {
    Method string
    URL *url.URL
    Proto string
    ProtoMajor int
    ProtoMinor int
    Header Header
    Body io.ReadCloser
    GetBody func() (io.ReadCloser, error)
    ContentLength int64
    TransferEncoding []string
    Close bool
    Host string
    Form url.Values
    PostForm url.Values
    MultipartForm *multipart.Form
    Trailer Header
    RemoteAddr string
    RequestURI string
    TLS *tls.ConnectionState
    Cancel <-chan struct{}
    Response *Response
}
```

Поле `Body` содержит текст запроса. После прочтения тела запроса вам разрешается вызов `getBody()`, который вернет новую копию тела (не обязательно).

Теперь обратимся к структуре `http.Transport`.

Тип `http.Transport`

Определение `http.Transport`, дающего больший контроль над HTTP-соединениями, довольно длинное и сложное:

```
type Transport struct {
    Proxy func(*Request) (*url.URL, error)
    DialContext func(ctx context.Context, network, addr string)
    (net.Conn, error)
    Dial func(network, addr string) (net.Conn, error)
    DialTLSContext func(ctx context.Context, network, addr string)
    (net.Conn, error)
    DialTLS func(network, addr string) (net.Conn, error)
    TLSClientConfig *tls.Config
}
```

```
TLSHandshakeTimeout time.Duration  
DisableKeepAlives bool  
DisableCompression bool  
MaxIdleConns int  
MaxIdleConnsPerHost int  
MaxConnsPerHost int  
IdleConnTimeout time.Duration  
ResponseHeaderTimeout time.Duration  
ExpectContinueTimeout time.Duration  
TLSNextProto map[string]func(authority string, c *tls.Conn)  
RoundTripper  
    ProxyConnectHeader Header  
    GetProxyConnectHeader func(ctx context.Context, proxyURL *url.URL,  
        target string) (Header, error)  
    MaxResponseHeaderBytes int64  
    WriteBufferSize int  
    ReadBufferSize int  
    ForceAttemptHTTP2 bool  
}
```

Обратите внимание, что `http.Transport` довольно низкоуровневый, в то время как `http.Client` (который также используется в этой главе) реализует высокоДанный HTTP-клиент — каждый `http.Client` содержит поле `Transport`. Если его значение соответствует `nil`, то используется `DefaultTransport`. Вам не нужно задействовать `http.Transport` во всех ваших программах, и вы не обязаны работать со всеми его полями при каждом использовании. Если вы хотите узнать больше о `DefaultTransport`, то наберите `go doc http.DefaultTransport`. Теперь выясним, как разработать веб-сервер.

Создание веб-сервера

В этом разделе представлен простой веб-сервер, разработанный на Go, позволяющий лучше понять принципы, которые лежат в основе подобных приложений.



Веб-сервер на Go может выполнять многие задачи эффективно и безопасно, но если вам действительно нужен мощный веб-сервер, поддерживающий модульность, несколько сайтов и виртуальные хосты, то лучше использовать веб-сервер Apache, Nginx или Caddy, написанный на Go.

Вы можете спросить, почему представленный веб-сервер использует HTTP вместо *безопасного HTTP (HTTPS)*. Ответ прост: большинство веб-серверов Go развернуты как образы Docker и скрыты за веб-серверами, такими как Caddy и Nginx, которые и обеспечивают безопасную работу HTTP, используя

соответствующие учетные данные безопасности. Нет никакого смысла использовать защищенный протокол HTTPS вместе с необходимыми учетными данными безопасности, не зная при этом, как и под каким доменным именем будет развернуто приложение. Это обычная практика в микросервисах, а также в обычных веб-приложениях, которые развертываются в образах Docker.

Пакет `net/http` предлагает функции и типы данных, которые позволяют разрабатывать мощные веб-серверы и клиенты. Методы `http.Set()` и `http.Get()` могут использоваться для выполнения запросов HTTP и HTTPS, тогда как `http.ListenAndServe()` — для создания веб-серверов с учетом указанной пользователем функции обработчика или функций, которые обрабатывают входящие запросы. Поскольку большинство веб-сервисов требуют поддержки нескольких конечных точек, вам рано или поздно потребуется несколько отдельных функций для обработки входящих запросов, что также улучшает структуру ваших сервисов.

Самый простой способ определить поддерживаемые конечные точки, а также функцию обработчика, которая отвечает на каждый запрос клиента, — это использовать `http.HandleFunc()`, которая может быть вызвана многократно.

После такого краткого и в некоторой степени теоретического введения пришло время перейти к более практическим темам, начав с реализации простого веб-сервера, что показано на примере `wwwServer.go`:

```
package main

import (
    "fmt"
    "net/http"
    "os"
    "time"
)

func myHandler(w http.ResponseWriter, r *http.Request) {
    fmt.Fprintf(w, "Serving: %s\n", r.URL.Path)
    fmt.Printf("Served: %s\n", r.Host)
}


```

Это функция-обработчик, которая возвращает сообщение клиенту, используя `w` `http.ResponseWriter`, который также является интерфейсом, реализующим `io.Writer`, и применяется для отправки ответа сервера.

```
func timeHandler(w http.ResponseWriter, r *http.Request) {
    t := time.Now().Format(time.RFC1123)
    Body := "The current time is:"
    fmt.Fprintf(w, "<h1 align=\"center\">%s</h1>", Body)
    fmt.Fprintf(w, "<h2 align=\"center\">%s</h2>\n", t)
```

```
    fmt.Fprintf(w, "Serving: %s\n", r.URL.Path)
    fmt.Printf("Served time for: %s\n", r.Host)
}
```

Это еще одна функция-обработчик, называемая `timeHandler`, которая возвращает текущее время в формате HTML. Все вызовы `fmt.Fprintf()` отправляют данные обратно HTTP-клиенту, тогда как выходные данные `fmt.Printf()` выводятся на терминале, на котором работает веб-сервер. Первым аргументом `fmt.Fprintf()` служит `w` `http.ResponseWriter`, который реализует `io.Writer` и, следовательно, может принимать данные.

```
func main() {
    PORT := ":8001"
```

Здесь мы определяем номер порта, который прослушивает ваш веб-сервер.

```
arguments := os.Args
if len(arguments) != 1 {
    PORT = ":" + arguments[1]
}
fmt.Println("Using port number: ", PORT)
```

Если вы не хотите использовать предопределенный номер порта (`8001`), то вам следует вызывать `wwwServer.go` с вашим собственным номером порта в качестве аргумента командной строки.

```
http.HandleFunc("/time", timeHandler)
http.HandleFunc("/", myHandler)
```

Таким образом, веб-сервер поддерживает URL `/time`, а также `/`. Путь `/` соответствует любому URL, не совпадающему с другими обработчиками. Тот факт, что мы связываем `MyHandler()` с `/`, делает `MyHandler()` функцией обработчика по умолчанию.

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

Вызов `Http.ListenAndServe()` запускает HTTP-сервер, используя предопределенный номер порта. Поскольку в строке `PORT` не указано имя хоста, веб-сервер будет прослушивать все доступные сетевые интерфейсы. Номер порта и имя хоста должны быть разделены двоеточием `(:)`, которое должно присутствовать, даже если имени хоста нет. В этом случае сервер прослушивает все доступные сетевые интерфейсы и, следовательно, все поддерживаемые имена хостов. Это причина того, что значение `PORT` равно `:8001` вместо `8001`.

Частью пакета `net/http` выступает тип `ServeMux` (`go doc http.ServeMux`), который является *мультиплексором HTTP-запросов*, предоставляющим несколько иной способ определения функций обработчика и конечных точек, отличающийся от функции по умолчанию, которая используется в `wwwServer.go`. Итак, если мы не создадим и не настроим нашу собственную переменную `ServeMux`, то `http.HandleFunc()` задействует `DefaultServeMux`, который служит `ServeMux` по умолчанию. Итак, в данном случае мы собираемся реализовать веб-сервис, используя *маршрутизатор Go по умолчанию*. Именно по этой причине второй параметр `http.ListenAndServe()` равен `nil`.

При выполнении файла `wwwServer.go` и взаимодействии с ним с помощью `curl(1)` мы получаем такой вывод:

```
$ go run wwwServer.go
Using port number: :8001
Served: localhost:8001
Served time for: localhost:8001
Served: localhost:8001
```

Обратите внимание: поскольку `wwwServer.go` не завершается автоматически, вам необходимо остановить его самостоятельно.

На стороне `curl(1)` взаимодействие выглядит следующим образом:

```
$ curl localhost:8001
Serving: /
```

В этом первом случае мы посещаем путь веб-сервера `/`, и нас обслуживает `MyHandler()`.

```
$ curl localhost:8001/time
<h1 align="center">The current time is:</h1><h2 align="center">Mon, 29
Mar 2021 08:26:27 EEST</h2>
Serving: /time
```

В этом случае мы посещаем `/time` и получаем HTML-вывод из `timeHandler()`.

```
$ curl localhost:8001/doesNotExist
Serving: /doesNotExist
```

В этом последнем случае мы посещаем `/DoesNotExist`, которого не существует. Поскольку этот путь не может быть сопоставлен ни с каким другим, он обслуживается обработчиком по умолчанию, то есть функцией `MyHandler()`.

Следующий раздел мы посвятим превращению приложения телефонной книги в веб-приложение!

Обновление приложения телефонной книги

На этот раз приложение телефонной книги будет работать как веб-сервис. Вот две основные задачи, которые необходимо выполнить: определение API вместе с конечными точками и реализация API. Третья задача касается *обмена данными* между сервером приложений и его клиентами. Существуют четыре основных подхода к обмену данными:

- использование обычного текста;
- использование HTML;
- использование JSON;
- гибридный подход, который сочетает в себе обычный текст и данные JSON.

Поскольку JSON рассматривается в главе 10, а HTML – не лучший вариант для сервиса (так как придется отделять данные от HTML-тегов и анализировать их), мы используем первый подход. Таким образом, сервис будет работать с *обычными текстовыми данными*. Мы начнем с определения API, поддерживающего работу приложения телефонной книги.

Определение API

API поддерживает следующие URL:

- `/list` – перечисление всех доступных записей;
- `/insert/name/surname/telephone/` – вставка новой записи. Позже мы узнаем, как извлечь нужную информацию из URL, содержащего пользовательские данные;
- `/delete/telephone/` – удаление записи на основе значения `telephone`;
- `/search/telephone/` – поиск записи на основе значения `telephone`;
- `/status` – это дополнительный URL, который возвращает количество записей в телефонной книге.



Список конечных точек не соответствует стандартным соглашениям REST – они будут представлены в главе 10.

На этот раз мы не будем использовать маршрутизатор Go по умолчанию, а следовательно, определим и настроим нашу собственную переменную

`http.NewServeMux()`. Это меняет способ предоставления функций обработчика: функция обработчика с сигнатурой `func(http.ResponseWriter, *http.Request)` должна быть преобразована в тип `http.HandlerFunc` и использоваться *типовом ServeMux* и его собственным методом `Handle()`. Поэтому в случае применения `ServeMux`, который отличается от используемого по умолчанию, мы должны выполнить это преобразование явно, вызвав `http.HandlerFunc()`. Это заставит *тип* `http.HandlerFunc` действовать в качестве *адаптера*, который позволяет задействовать обычные функции в качестве HTTP-обработчиков при условии, что они обладают требуемой сигнатурой. Это не проблема при использовании маршрутизатора Go по умолчанию (`DefaultServeMux`), поскольку функция `http.HandleFunc()` выполняет это преобразование автоматически.



Чтобы было понятнее, тип `http.HandlerFunc` поддерживает метод `HandlerFunc()` — и тип, и метод определены в пакете `http`. Аналогично, названная функция `http.HandleFunc()` (без `r`) используется с маршрутизатором Go по умолчанию.

Например, для конечной точки `/time` и функции-обработчика `timeHandler()` вам нужно вызвать `mux.Handle()` следующим образом: `mux.Handle("/time", http.HandlerFunc(timeHandler))`. Если бы вы использовали `http.HandleFunc()` и, как следствие, `DefaultServeMux`, то вызов выглядел бы как `http.HandleFunc("/time", timeHandler)`.

Предметом обсуждения в следующем подразделе является реализация конечных точек HTTP.

Реализация обработчиков

Новая версия телефонной книги будет создана в специальном репозитории GitHub: <https://github.com/mactsouk/www-phone>. После создания репозитория нам необходимо выполнить следующее:

```
$ cd ~/go/src/github.com/mactsouk # Replace with your own path
$ git clone git@github.com:mactsouk/www-phone.git
$ cd www-phone
$ touch handlers.go
$ touch www-phone.go
```

Файл `www-phone.go` содержит код, который и определяет работу веб-сервера. Обычно обработчики помещаются в отдельный пакет, но по соображениям простоты мы решили поместить обработчики в отдельный файл в том же пакете `handlers.go`. Содержимое файла `handlers.go`, который вмещает в себя все

функциональные возможности, связанные с обслуживанием клиентов, выглядит следующим образом:

```
package main

import (
    "fmt"
    "log"
    "net/http"
    "strings"
)
```

Все необходимые пакеты для `handlers.go` импортируются, даже если некоторые из них уже импортированы в `www-phone.go`. Обратите внимание, что название пакета — `main`, что также относится к `www-phone.go`.

```
const PORT = ":1234"
```

Это номер порта по умолчанию, который прослушивает HTTP-сервер.

```
func defaultHandler(w http.ResponseWriter, r *http.Request) {
    log.Println("Serving:", r.URL.Path, "from", r.Host)
    w.WriteHeader(http.StatusOK)
    Body := "Thanks for visiting!\n"
    fmt.Fprintf(w, "%s", Body)
}
```

Это обработчик по умолчанию, который обслуживает все запросы, не совпадающие ни с одним из других обработчиков.

```
func deleteHandler(w http.ResponseWriter, r *http.Request) {
    // получить телефон
    paramStr := strings.Split(r.URL.Path, "/")
```

А это функция-обработчик для пути `/delete`, которая начинается с разделения URL с целью получения нужной информации.

```
fmt.Println("Path:", paramStr)
if len(paramStr) < 3 {
    w.WriteHeader(http.StatusNotFound)
    fmt.Println(w, "Not found: "+r.URL.Path)
    return
}
```

Если нам не хватает параметров, то следует направить клиенту сообщение об ошибке вместе с нужным HTTP-кодом, которым в данном случае выступает `http.StatusNotFound`. Вы можете использовать любой HTTP-код, подходящий по смыслу. Метод `WriteHeader()` возвращает заголовок с предоставленным кодом состояния перед записью тела ответа.

```
log.Println("Serving:", r.URL.Path, "from", r.Host)
```

Именно здесь HTTP-сервер отправляет данные в файлы журналов — в основном это делается в целях отладки.

```
telephone := paramStr[2]
```

Поскольку процесс удаления основан на номере телефона, то все, что нам требуется, — это корректный номер телефона. Именно здесь данный параметр и считывается после разделения предоставленного URL.

```
err := deleteEntry(telephone)
if err != nil {
    fmt.Println(err)
    Body := err.Error() + "\n"
    w.WriteHeader(http.StatusNotFound)
    fmt.Fprintf(w, "%s", Body)
    return
}
```

Как только у нас есть номер телефона, мы вызываем `deleteEntry()` для удаления. Возвращаемое `deleteEntry()` значение определяет результат операции и, следовательно, ответ клиенту.

```
Body := telephone + " deleted!\n"
w.WriteHeader(http.StatusOK)
fmt.Fprintf(w, "%s", Body)
}
```

На данный момент мы знаем, что операция удаления прошла успешно, поэтому отправляем клиенту соответствующее сообщение вместе с кодом статуса `http.StatusOK`. Наберите `go doc http.StatusOK` для получения списка кодов.

```
func listHandler(w http.ResponseWriter, r *http.Request) {
    log.Println("Serving:", r.URL.Path, "from", r.Host)
    w.WriteHeader(http.StatusOK)
    Body := list()
    fmt.Fprintf(w, "%s", Body)
}
```

Вспомогательная функция `list()`, которая используется в пути `/list`, не может завершиться сбоем. Следовательно, `http.StatusOK` всегда возвращается при обработке `/list`. Однако иногда возвращаемое значение `list()` может оказаться пустым.

```
func statusHandler(w http.ResponseWriter, r *http.Request) {
    log.Println("Serving:", r.URL.Path, "from", r.Host)
    w.WriteHeader(http.StatusOK)
    Body := fmt.Sprintf("Total entries: %d\n", len(data))
    fmt.Fprintf(w, "%s", Body)
}
```

Здесь мы определяем функцию обработчика для URL `/status`. Он просто возвращает информацию об общем количестве записей в нашей телефонной книге. Его можно использовать для проверки того, что веб-сервис работает нормально.

```
func insertHandler(w http.ResponseWriter, r *http.Request) {
    // разделяем URL
    paramStr := strings.Split(r.URL.Path, "/")
    fmt.Println("Path:", paramStr)
```

Как и в случае с `delete`, нам необходимо разделить данный URL, чтобы получить нужную информацию. В данном случае нам требуются три элемента, поскольку мы пытаемся вставить в приложение телефонной книги новую запись.

```
if len(paramStr) < 5 {
    w.WriteHeader(http.StatusNotFound)
    fmt.Fprintln(w, "Not enough arguments: "+r.URL.Path)
    return
}
```

Необходимость извлечения трех элементов из URL означает, что `ParamStr` должен содержать по крайней мере четыре элемента, отсюда и условие `len(ParamStr) < 5`.

```
name := paramStr[2]
surname := paramStr[3]
tel := paramStr[4]

t := strings.ReplaceAll(tel, "-", "")
if !matchTel(t) {
    fmt.Println("Not a valid telephone number:", tel)
    return
}
```

В этой части мы получаем нужные данные и убеждаемся, что телефонный номер содержит только цифры, — это делается с помощью вспомогательной функции `matchTel()`.

```
temp := &Entry{Name: name, Surname: surname, Tel: t}
err := insert(temp)
```

Поскольку вспомогательной функции `insert()` требуется значение `*Entry`, мы создаем его перед вызовом.

```
if err != nil {
    w.WriteHeader(http.StatusNotModified)
    Body := "Failed to add record\n"
    fmt.Fprintf(w, "%s", Body)
} else {
```

```

    log.Println("Serving:", r.URL.Path, "from", r.Host)
    Body := "New record added successfully\n"
    w.WriteHeader(http.StatusOK)
    fmt.Fprintf(w, "%s", Body)
}

log.Println("Serving:", r.URL.Path, "from", r.Host)
}

```

Это конец обработчика `/insert`. Последняя часть реализации `insertHandler()` имеет дело с возвращаемым значением `insert()`. Если ошибки не было, то клиенту возвращается `http.StatusOK`. В обратном случае возвращается `http.StatusNotModified`, что означает отсутствие изменений в телефонной книге. Задача клиента — проверить код состояния взаимодействия, а задача сервера — вернуть клиенту соответствующий код состояния.

```

func searchHandler(w http.ResponseWriter, r *http.Request) {
    // получить значение Search из URL
    paramStr := strings.Split(r.URL.Path, "/")
    fmt.Println("Path:", paramStr)

    if len(paramStr) < 3 {
        w.WriteHeader(http.StatusNotFound)
        fmt.Fprintln(w, "Not found: "+r.URL.Path)
        return
    }

    var Body string
    telephone := paramStr[2]
}

```

На данном этапе мы извлекаем номер телефона из URL, как делали это в случае `/delete`.

```

t := search(telephone)
if t == nil {
    w.WriteHeader(http.StatusNotFound)
    Body = "Could not be found: " + telephone + "\n"
} else {
    w.WriteHeader(http.StatusOK)
    Body = t.Name + " " + t.Surname + " " + t.Tel + "\n"
}

fmt.Println("Serving:", r.URL.Path, "from", r.Host)
fmt.Fprintf(w, "%s", Body)
}

```

На этом месте заканчивается последняя функция `handlers.go`, и она касается конечной точки `/search`. Вспомогательная функция `search()` проверяет, существует ли данная запись в телефонной книге, и действует соответствующим

образом. Кроме того, реализация функции `main()`, которую можно найти в файле `www-phone.go`, состоит в следующем:

```
func main() {
    err := readCSVFile(CSVFILE)
    if err != nil {
        fmt.Println(err)
        return
    }

    err = createIndex()
    if err != nil {
        fmt.Println("Cannot create index.")
        return
    }
}
```

Эта первая часть `main()` имеет отношение к инициализации приложения телефонной книги.

```
mux := http.NewServeMux()
s := &http.Server{
    Addr: PORT,
    Handler: mux,
    IdleTimeout: 10 * time.Second,
    ReadTimeout: time.Second,
    WriteTimeout: time.Second,
}
```

Здесь мы сохраняем параметры HTTP-сервера в структуре `http.Server` и применяем собственный `http.NewServeMux()` вместо используемого по умолчанию.

```
mux.HandleFunc("/list", http.HandlerFunc(listHandler))
mux.HandleFunc("/insert/", http.HandlerFunc(insertHandler))
mux.HandleFunc("/insert", http.HandlerFunc(insertHandler))
mux.HandleFunc("/search", http.HandlerFunc(searchHandler))
mux.HandleFunc("/search/", http.HandlerFunc(searchHandler))
mux.HandleFunc("/delete/", http.HandlerFunc(deleteHandler))
mux.HandleFunc("/status", http.HandlerFunc(statusHandler))
mux.HandleFunc("/", http.HandlerFunc(defaultHandler))
```

Это список поддерживаемых URL. Обратите внимание, что `/search` и `/search/` обрабатываются одной и той же функцией-обработчиком, даже если `/search` завершается неудачей (поскольку не содержит требуемого аргумента). В то время как `/delete/` обрабатывается особым образом — это будет показано при тестировании приложения. Поскольку мы используем `http.NewServeMux()`, а не маршрутизатор Go по умолчанию, при определении функций обработчика нам требуется использовать `http.HandlerFunc()`.

```
fmt.Println("Ready to serve at", PORT)
err = s.ListenAndServe()
```

```
if err != nil {  
    fmt.Println(err)  
    return  
}  
}
```

Метод `ListenAndServe()` запускает HTTP-сервер, используя параметры, определенные ранее в структуре `http.Server`. Оставшаяся часть `www-phone.go` содержит вспомогательные функции, связанные с работой телефонной книги. Обратите внимание, что важно сохранять и обновлять содержимое приложения телефонной книги как можно чаще, поскольку приложение работает в режиме реального времени и вы можете потерять данные в случае сбоя.

Следующая команда позволяет запустить приложение — вам нужно предоставить оба файла `go run`:

```
$ go run www-phone.go handlers.go  
Ready to serve at :1234  
2021/03/29 17:13:49 Serving: /list from localhost:1234  
2021/03/29 17:13:53 Serving: /status from localhost:1234  
Path: [ search 2109416471]  
Serving: /search/2109416471 from localhost:1234  
Path: [ search ]  
2021/03/29 17:28:34 Serving: /list from localhost:1234  
Path: [ search 2101112223]  
Serving: /search/2101112223 from localhost:1234  
Path: [ delete 2109416471]  
2021/03/29 17:29:24 Serving: /delete/2109416471 from localhost:1234  
Path: [ insert Mike Tsoukalos 2109416471]  
2021/03/29 17:29:56 Serving: /insert/Mike/Tsoukalos/2109416471 from  
localhost:1234  
2021/03/29 17:29:56 Serving: /insert/Mike/Tsoukalos/2109416471 from  
localhost:1234  
Path: [ insert Mike Tsoukalos 2109416471]  
2021/03/29 17:30:18 Serving: /insert/Mike/Tsoukalos/2109416471 from  
localhost:1234
```

На стороне клиента, которым является `curl(1)`, мы получаем следующий вывод:

```
$ curl localhost:1234/list  
Dimitris Tsoukalos 2101112223  
Jane Doe 0800123456  
Mike Tsoukalos 2109416471
```

Здесь мы получаем все записи из приложения телефонной книги, посетив `/list`.

```
$ curl localhost:1234/status  
Total entries: 3
```

Далее мы посещаем `/status` и получаем ожидаемый результат.

```
$ curl localhost:1234/search/2109416471
Mike Tsoukalos 2109416471
```

Эта команда выполняет поиск существующего телефонного номера — сервер отвечает его полной записью.

```
$ curl localhost:1234/delete/2109416471
2109416471 deleted!
```

В этом выводе показано, что мы удалили запись с номером телефона **2109416471**. В REST для этого потребуется метод `DELETE`, но по соображениям простоты подробную информацию об этом вы узнаете в главе 10.

Теперь попробуем зайти в `/delete` вместо `/delete/`:

```
$ curl localhost:1234/delete
<a href="/delete/">Moved Permanently</a>.
```

Маршрутизатор Go сообщает нам, что мы должны попробовать вместо этого `/delete/`, поскольку `/delete` был перемещен навсегда. Это тип сообщения, которое мы получаем, не определяя в маршрутах конкретно `/delete` и `/delete/`.

Теперь вставим новую запись:

```
$ curl localhost:1234/insert/Mike/Tsoukalos/2109416471
New record added successfully
```

В REST для этого потребуется метод `POST`, но вы узнаете об этом в главе 10.

Если мы попытаемся вставить ту же запись снова, то ответ будет следующим:

```
$ curl localhost:1234/insert/Mike/Tsoukalos/2109416471
Failed to add record
```

Похоже, все работает как положено. Теперь мы можем подключить телефонное приложение к сети и взаимодействовать с ним с помощью нескольких HTTP-запросов, поскольку пакет `http` использует несколько подпрограмм для взаимодействия с клиентами. На практике это означает, что приложение телефонной книги *работает в многопоточном режиме!*

Позже в этой главе мы создадим клиент командной строки для сервера телефонной книги. Кроме того, в главе 11 будет показано, как протестировать ваш код.

В следующем разделе вы узнаете, как предоставлять метрики в Prometheus и создавать образы Docker для серверных приложений.

Предоставление метрик для Prometheus

Представьте, что у вас есть приложение, которое записывает файлы на диск, а вы хотите получить метрики для этого приложения, чтобы лучше понять, как запись нескольких файлов влияет на общую производительность (вам нужно собрать данные о производительности, чтобы понять поведение вашего приложения). Представленное приложение использует тип метрики *gauge* только потому, что он подходит для информации, которая отправляется в Prometheus, однако Prometheus принимает множество типов данных. Список поддерживаемых типов данных для метрик представлен ниже.

- *Счетчик* — это накопительное значение, которое используется для представления увеличивающихся счетчиков. Значение счетчика может оставаться неизменным, увеличиваться или сбрасываться до нуля, но не может уменьшаться. Счетчики обычно служат для представления совокупных значений, таких как количество запросов, обработанных на данный момент, общее количество ошибок и т. д.
- *Шкала* — это единственное числовое значение, которое разрешено увеличивать или уменьшать. Шкалы обычно используются для представления значений, которые могут увеличиваться или уменьшаться, таких как количество запросов, продолжительность времени и т. д.
- *Гистограмма* — используется для выборки наблюдений, подсчета и сегментации. Гистограммы обычно служат для подсчета длительности запроса, времени отклика и т. д.
- *Сводка* — похожа на гистограмму, но также может вычислять квантили по скользящим окнам, работающие со временем.

Как гистограммы, так и сводки полезны и удобны для выполнения статистических вычислений и определения свойств. Обычно счетчик или шкала — это все, что нам нужно для хранения системных метрик.

В этом разделе показано, как собрать систему и предоставить ее в Prometheus. По соображениям простоты представленное приложение будет генерировать случайные значения. Мы начнем с использования пакета `runtime/metrics`, который предоставляет метрики, связанные со средой выполнения Go.

Пакет runtime/metrics

Пакет `runtime/metrics` делает метрики, экспортруемые средой Go, доступными разработчику. Каждое имя метрики задается путем. В качестве примера доступ к количеству работающихgoroutines осуществляется в виде `/sched/goroutines:goroutines`. Однако если вы хотите собрать все доступные метрики, то вам следует использовать `metrics.All()` – это избавит вас от необходимости писать много кода для ручного сбора всех показателей.

Показатели сохраняются с помощью типа данных `metrics.Sample`. Определение структуры данных `metrics.Sample` выглядит так:

```
type Sample struct {
    Name string
    Value Value
}
```

Значение `Name` должно соответствовать имени одного из описаний метрики, возвращаемых `metrics.All()`. Если вы уже знаете описание метрики, то необходимости использовать `metrics.All()` нет.

Использование пакета `runtime/metrics` показано в файле `metrics.go`. Представленный код получает значение `/sched/goroutines:goroutines` и выводит его на экран:

```
package main

import (
    "fmt"
    "runtime/metrics"
    "sync"
    "time"
)

func main() {
    const nGo = "/sched/goroutines:goroutines"
```

Переменная `nGo` содержит путь к метрике, которую мы хотим получить.

```
// срез для получения образцов метрик getMetric
getMetric := make([]metrics.Sample, 1)
getMetric[0].Name = nGo
```

После этого, чтобы сохранить значение метрики, мы создаем срез типа `metrics.Sample`. Начальный размер среза равен 1, поскольку мы получаем значения только для одной метрики. Мы устанавливаем значение `Name` в `/sched/goroutines:goroutines`, как сохранено в `nGo`.

```
var wg sync.WaitGroup
for i := 0; i < 3; i++ {
```

```
wg.Add(1)
go func() {
    defer wg.Done()
    time.Sleep(4 * time.Second)
}()
```

Здесь мы вручную создаем триgorутины, чтобы иметь соответствующие данные для получения.

```
// получить фактические данные
metrics.Read(getMetric)
if getMetric[0].Value.Kind() == metrics.KindBad {
    fmt.Printf("metric %q no longer supported\n", nGo)
}
```

Функция `metrics.Read()` собирает нужные показатели на основе данных в срезе `getMetric`.

```
mVal := getMetric[0].Value.Uint64()
fmt.Printf("Number of goroutines: %d\n", mVal)
}
```

После считывания желаемой метрики мы преобразуем ее в числовое значение (в данном случае неподписанный `int64`), чтобы позже использовать в нашей программе.

```
wg.Wait()

metrics.Read(getMetric)
mVal := getMetric[0].Value.Uint64()
fmt.Printf("Before exiting: %d\n", mVal)
}
```

Последние строки кода позволяют убедиться, что после завершения всех подпрограмм значение метрики будет равно 1, что является горутиной, используемой для запуска функции `main()`.

При выполнении `metrics.go` мы получаем такой вывод:

```
$ go run metrics.go
Number of goroutines: 2
Number of goroutines: 3
Number of goroutines: 4
Before exiting: 1
```

Мы создали три горутины, и у нас уже есть горутина для запуска функции `main()`. Следовательно, максимальное количество горутин действительно равно 4.

В следующих подразделах показано, как сделать любую собранную вами метрику доступной для Prometheus.

Предоставление метрик

Сбор метрик — совершенно иная задача, нежели предоставление их Prometheus. В этом подразделе показано, как сделать метрики доступными для сбора.

Код `samplePro.go` выглядит следующим образом:

```
package main

import (
    "fmt"
    "net/http"

    "math/rand"
    "time"

    "github.com/prometheus/client_golang/prometheus"
    "github.com/prometheus/client_golang/promhttp"
)
```

Для связи с Prometheus нам придется использовать два внешних пакета.

```
var PORT = ":1234"

var counter = prometheus.NewCounter(
    prometheus.CounterOpts{
        Namespace: "mtsouk",
        Name:      "my_counter",
        Help:      "This is my counter",
    })
}
```

Так мы определяем новую переменную `counter` и указываем желаемые параметры. Поле `Namespace` очень важно, поскольку позволяет группировать метрики в наборы.

```
var gauge = prometheus.NewGauge(
    prometheus.GaugeOpts{
        Namespace: "mtsouk",
        Name:      "my_gauge",
        Help:      "This is my gauge",
    })
}
```

Так мы определяем новую переменную `gauge` и указываем желаемые параметры.

```
var histogram = prometheus.NewHistogram(
    prometheus.HistogramOpts{
        Namespace: "mtsouk",
        Name:      "my_histogram",
        Help:      "This is my histogram",
    })
}
```

Так мы определяем новую переменную `histogram` и указываем желаемые параметры.

```
var summary = prometheus.NewSummary(
    prometheus.SummaryOpts{
        Namespace: "mtsouk",
        Name:      "my_summary",
        Help:      "This is my summary",
    })
}
```

Так мы определяем новую переменную `summary` и указываем желаемые параметры. Однако, как вы сейчас увидите, одного определения метрической переменной недостаточно. Вам также необходимо ее зарегистрировать.

```
func main() {
    rand.Seed(time.Now().Unix())

    prometheus.MustRegister(counter)
    prometheus.MustRegister(gauge)
    prometheus.MustRegister(histogram)
    prometheus.MustRegister(summary)
}
```

В этих четырех операторах мы регистрируем четыре переменные метрик. Теперь Prometheus знает о них.

```
go func() {
    for {
        counter.Add(rand.Float64() * 5)
        gauge.Add(rand.Float64()*15 - 5)
        histogram.Observe(rand.Float64() * 10)
        summary.Observe(rand.Float64() * 10)
        time.Sleep(2 * time.Second)
    }
}()
```

Эта горутина выполняется до тех пор, пока с помощью бесконечного цикла `for` работает веб-сервер. В этой горутине из-за использования оператора `time.Sleep(2 * time.Second)` показатели обновляются каждые две секунды (в данной ситуации — с помощью случайных значений).

```
http.Handle("/metrics", promhttp.Handler())
fmt.Println("Listening to port", PORT)
fmt.Println(http.ListenAndServe(PORT, nil))
}
```

Как вы уже знаете, каждый URL обрабатывается функцией-обработчиком, которую мы обычно реализуем самостоятельно. Однако в данном случае мы используем обработчик `promhttp.Handler()`, который поставляется с пакетом github.com/prometheus/client_golang/prometheus/promhttp, и это избавляет нас от необходимости писать собственный код. Тем не менее, прежде чем мы

запустим веб-сервер, нам все еще нужно зарегистрировать `promhttp.Handler()`, используя для этого `http.HandleFunc()`. Обратите внимание, что метрики находятся по пути `/metrics`, — Prometheus знает об этом.

Когда `samplePro.go` запущена, получить список метрик, относящихся к пространству имен `mtsouk`, можно просто путем выполнения следующей команды `curl(1)`:

```
$ curl localhost:1234/metrics --silent | grep mtsouk
# HELP mtsouk_my_counter This is my counter
# TYPE mtsouk_my_counter counter
mtsouk_my_counter 19.948239343027772
```

Это выходные данные переменной `counter`. Если часть `| grep mtsouk` опущена, то вы получите список всех доступных метрик.

```
# HELP mtsouk_my_gauge This is my gauge
# TYPE mtsouk_my_gauge gauge
mtsouk_my_gauge 29.335329668135287
```

Это выходные данные переменной `gauge`.

```
# HELP mtsouk_my_histogram This is my histogram
# TYPE mtsouk_my_histogram histogram
mtsouk_my_histogram_bucket{le="0.005"} 0
mtsouk_my_histogram_bucket{le="0.01"} 0
mtsouk_my_histogram_bucket{le="0.025"} 0
...
mtsouk_my_histogram_bucket{le="5"} 4
mtsouk_my_histogram_bucket{le="10"} 9
mtsouk_my_histogram_bucket{le="+Inf"} 9
mtsouk_my_histogram_sum 44.52262035556937
mtsouk_my_histogram_count 9
```

Это выходные данные переменной `histogram`. Гистограммы содержат *сегменты*, отсюда и большое количество выходных строк.

```
# HELP mtsouk_my_summary This is my summary
# TYPE mtsouk_my_summary summary
mtsouk_my_summary_sum 19.407554729772105
mtsouk_my_summary_count 9
```

Последние строки выходных данных относятся к типу `summary`.

Итак, метрики есть и готовы к использованию Prometheus — на практике это означает, что каждое приложение Go может экспортить метрики, с помощью которых можно измерять его производительность и выявлять узкие места. Однако мы еще не закончили, так как нам нужно научиться создавать образы Docker для Go-приложений.

Создание образа Docker для Go-сервера

В этом подразделе показано, как создать образ Docker для Go-приложения. Основное преимущество здесь заключается в том, что вы можете развернуть его в среде Docker, не беспокоясь о компиляции и наличии необходимых ресурсов — все уже включено в образ Docker.

Тем не менее вы можете спросить: «*Почему бы не использовать обычный двоичный Go-файл вместо образа Docker?*» Ответ прост: образы Docker можно поместить в файлы `docker-compose.yml` и развернуть с помощью Kubernetes. К двоичным Go-файлам это уже не относится.

Обычно вы начинаете с базового образа Docker, который уже включает Go, и создаете в нем желаемый двоичный файл. Ключевым моментом здесь является то, что `samplePro.go` использует внешний пакет, который следует загрузить в образ Docker перед сборкой исполняемого двоичного файла.

Процесс должен начинаться с `go mod init` и `go mod tidy`. Содержимое файла `Dockerfile`, который можно найти по имени `dFilev2` в репозитории книги на GitHub, выглядит следующим образом:

```
# WITH Go Modules

FROM golang:alpine AS builder
RUN apk update && apk add --no-cache git
```

Поскольку `golang: alpine` использует последнюю версию Go, которая не поставляется с `git`, мы установим ее вручную.

```
RUN mkdir $GOPATH/src/server
ADD ./samplePro.go $GOPATH/src/server
```

Если вы хотите использовать Go-модули, то вам следует поместить свой код в `$GOPATH/src`.

```
WORKDIR $GOPATH/src/server
RUN go mod init
RUN go mod tidy
RUN go mod download
RUN mkdir /pro
RUN go build -o /pro/server samplePro.go
```

Мы скачиваем зависимости, используя различные команды `go mod`. Построение двоичного файла такое же, как и раньше.

```
FROM alpine:latest

RUN mkdir /pro
COPY --from=builder /pro/server /pro/server
EXPOSE 1234
WORKDIR /pro
CMD ["/pro/server"]
```

На этом втором этапе мы помещаем двоичный файл в нужное местоположение (`/pro`) и открываем нужный порт, который в данном случае соответствует 1234. Номер порта зависит от кода в `samplePro.go`.

Создать образ Docker из `dFilev2` достаточно просто с помощью следующей команды:

```
$ docker build -f dFilev2 -t go-app116 .
```

Когда образ Docker создан, нет никакой разницы в том, как использовать его в файле `docker-compose.yml`, — соответствующая запись в файле `docker-compose.yml` будет выглядеть следующим образом:

```
goapp:  
  image: goapp  
  container_name: goapp-int  
  restart: always  
  ports:  
    - 1234:1234  
  networks:  
    - monitoring
```

Имя образа Docker — `goapp`, тогда как внутреннее имя контейнера — `goapp-int`. Итак, если другой контейнер из сети `monitoring` захочет получить доступ к этому контейнеру, он должен использовать имя хоста `goapp-int`. Наконец, единственным открытым портом является порт с номером 1234.

В следующем подразделе показано, как предоставить метрики Prometheus.

Предоставление желаемых метрик

Здесь показано, как предоставить метрики из пакета `runtime/metrics` в Prometheus. В нашем случае мы используем `/sched/goroutines:goroutines` и `/memory/classes/total:bytes`. О первой вы уже знаете, она содержит общее количество горутин. Вторая метрика — это объем памяти, выделенный средой выполнения Go для текущего процесса в режиме чтения-записи.



Поскольку представленный код использует внешний пакет, он должен быть помещен внутрь `~/go /src`, а Go-модули должны быть включены с помощью `go mod init`.

Go-код `prometheus.go` содержит следующее:

```
package main  
  
import (  
    "log"  
    "math/rand"
```

```

    "net/http"
    "runtime"
    "runtime/metrics"
    "time"

    "github.com/prometheus/client_golang/prometheus"
    "github.com/prometheus/client_golang/promhttp"
)

```

Первый внешний пакет — это клиентская Go-библиотека для Prometheus, а второй предназначен для использования функции обработчика по умолчанию (`promhttp.Handler()`).

```

var PORT = ":1234"

var n_goroutines = prometheus.NewGauge(
    prometheus.GaugeOpts{
        Namespace: "packt",
        Name:      "n_goroutines",
        Help:       "Number of goroutines"})

var n_memory = prometheus.NewGauge(
    prometheus.GaugeOpts{
        Namespace: "packt",
        Name:      "n_memory",
        Help:       "Memory usage"})

```

Здесь мы определим две метрики Prometheus.

```

func main() {
    rand.Seed(time.Now().Unix())
    prometheus.MustRegister(n_goroutines)
    prometheus.MustRegister(n_memory)

    const nGo = "/sched/goroutines:goroutines"
    const nMem = "/memory/classes/heap/free:bytes"
}

```

А здесь определим метрики, которые хотим считать из пакета `runtime/metrics`.

```

getMetric := make([]metrics.Sample, 2)
getMetric[0].Name = nGo
getMetric[1].Name = nMem

http.Handle("/metrics", promhttp.Handler())

```

Здесь зарегистрируем функцию обработчика для пути `/metrics`. Мы используем `promhttp.Handler()`.

```

go func() {
    for {
        for i := 1; i < 4; i++ {

```

```
go func() {
    _ = make([]int, 1000000)
    time.Sleep(time.Duration(rand.Intn(10)) * time.Second)
}()
}
```

Обратите внимание, что такая программа обязательно должна иметь *по крайней мере две горутины*: одну для запуска HTTP-сервера и другую для сбора метрик. Обычно HTTP-сервер работает в горутине, которая запускает функцию `main()`, а сбор метрик происходит в пользовательской горутине.

Внешний цикл `for` гарантирует, чтоgorутина выполняется вечно, в то время как внутренний цикл `for` создает дополнительные горутины, так что значение метрики `/sched/goroutines:goroutines` постоянно меняется.

```
    runtime.GC()
    metrics.Read(getMetric)
    goVal := getMetric[0].Value.Uint64()
    memVal := getMetric[1].Value.Uint64()
    time.Sleep(time.Duration(rand.Intn(15)) * time.Second)

    n_goroutines.Set(float64(goVal))
    n_memory.Set(float64(memVal))
}
}()
```

Функция `runtime.GC()` сообщает сборщику мусора Go о запуске и вызывается для изменения метрики `/memory/classes/heap/free:bytes`. Два вызова `Set()` обновляют значения метрик.



Больше информации о работе сборщика мусора Go вы можете прочитать в приложении.

```
    log.Println("Listening to port", PORT)
    log.Println(http.ListenAndServe(PORT, nil))
}
```

Последнее выражение запускает веб-сервер, используя маршрутизатор Go по умолчанию. Запуск `prometheus.go` из каталога внутри `~/go/src/github.com/mactsouk` требует выполнения следующих команд:

```
$ cd ~/go/src/github.com/mactsouk/Prometheus # use any path inside ~/go/src  
$ go mod init  
$ go mod tidy  
$ go mod download  
$ go run prometheus.go  
2021/04/01 12:18:11 Listening to port :1234
```

Хотя `prometheus.go` не генерирует никаких выходных данных, кроме вышеприведенной строки, в следующем подразделе показано, как считывать из него нужные метрики с помощью `curl(1)`.

Чтение метрик

Чтобы убедиться, что приложение работает должным образом, вы можете получить список метрик из `prometheus.go` с помощью `curl(1)`. Я всегда тестирую работу такого приложения с помощью `curl(1)` или другой подобной утилиты (например, `wget(1)`) прежде, чем пытаюсь получить метрики с помощью Prometheus.

```
$ curl localhost:1234/metrics --silent | grep packt
# HELP packt_n_goroutines Number of goroutines
# TYPE packt_n_goroutines gauge
packt_n_goroutines 5
# HELP packt_n_memory Memory usage
# TYPE packt_n_memory gauge
packt_n_memory 794624
```

Эта команда предполагает, что `curl(1)` выполняется на том же компьютере, что и сервер, и что сервер прослушивает TCP-порт с номером 1234. Далее мы должны позволить Prometheus извлечь метрики. Самый простой способ дать Docker-образу Prometheus увидеть Go-приложение с его метриками — выполнить оба в виде образов Docker. Здесь есть важный момент: пакет `runtime/metrics` был впервые представлен в Go начиная с версии 1.16. Это означает, что для создания исходного файла Go, использующего `runtime/metrics`, нам нужно задействовать Go версии 1.16 или новее и, следовательно, применять модули для создания образа Docker. Таким образом, мы будем использовать следующий файл `Dockerfile`:

```
FROM golang:alpine AS builder
```

Это имя базового образа Docker, который используется для создания двоичного файла. `golang:alpine` всегда содержит последнюю версию Go (при регулярном обновлении).

```
RUN apk update && apk add --no-cache git
```

Поскольку `golang:alpine` не поставляется с `git`, нам нужно установить его вручную.

```
RUN mkdir $GOPATH/src/server
ADD ./prometheus.go $GOPATH/src/server
WORKDIR $GOPATH/src/server
```

```
RUN go mod init  
RUN go mod tidy  
RUN go mod download
```

Эти команды скачивали необходимые зависимости перед попыткой создания двоичного файла.

```
RUN mkdir /pro  
RUN go build -o /pro/server prometheus.go  
  
FROM alpine:latest  
  
RUN mkdir /pro  
COPY --from=builder /pro/server /pro/server  
EXPOSE 1234  
WORKDIR /pro  
CMD ["/pro/server"]
```

Чтобы создать желаемый образ Docker, который будет называться `goapp`, можно просто выполнить следующую команду:

```
$ docker build -f Dockerfile -t goapp .
```

Как обычно, вывод образов Docker подтверждает успешное создание Docker-образа `goapp`. В моем случае соответствующая запись выглядит следующим образом:

goapp	latest	a1f0cd4bd8f5	5 seconds ago	16.9MB
-------	--------	--------------	---------------	--------

Теперь обсудим, как настроить Prometheus для получения желаемых метрик.

Ввод метрик в Prometheus

Чтобы извлекать метрики, Prometheus нужен правильный файл конфигурации. Мы используем вот такой файл:

```
# prometheus.yml  
scrape_configs:  
  - job_name: GoServer  
    scrape_interval: 5s  
    static_configs:  
      - targets: ['goapp:1234']
```

Мы просим Prometheus подключиться к хосту `goapp`, использовав порт с номером 1234. Prometheus извлекает данные каждые пять секунд в соответствии со значением поля `scrape_interval`. Вам следует разместить `prometheus.yml` в каталоге `prometheus`, который должен находиться в том же каталоге, что и файл `docker-compose.yml`, который представлен далее.

Prometheus, а также Grafana и Go-приложение будут запускаться как контейнеры Docker с помощью файла `docker-compose.yml`:

```
version: "3"

services:
  goapp:
    image: goapp
    container_name: goapp
    restart: always
    ports:
      - 1234:1234
    networks:
      - monitoring
```

Это та часть, которая касается Go-приложения, собирающего метрики. Имя образа Docker, а также внутреннее имя хоста контейнера Docker — `goapp`. Вы должны задать номер порта, который будет открыт для подключений. В нашем случае номера как внутреннего, так и внешнего порта равны `1234`. Внутренний сопоставляется с внешним. Кроме того, вы должны поместить все образы Docker в одну и ту же сеть, которая в данном случае называется `monitoring` (скоро мы определим ее).

```
prometheus:
  image: prom/prometheus:latest
  container_name: prometheus
  restart: always
  user: "0"
  volumes:
    - ./prometheus/:/etc/prometheus/
```

Так мы передаем свою копию `prometheus.yml` в образ Docker, который будет использоваться Prometheus. Итак, `./prometheus/prometheus.yml` с локального компьютера можно получить как `/etc/prometheus/prometheus.yml` из образа Docker.

```
- ./prometheus_data/:/prometheus/
command:
  - '--config.file=/etc/prometheus/prometheus.yml'
```

Здесь мы указываем Prometheus, какой файл конфигурации использовать.

```
- '--storage.tsdb.path=/prometheus'
- '--web.console.libraries=/etc/prometheus/console_libraries'
- '--web.console.templates=/etc/prometheus/consoles'
- '--storage.tsdb.retention.time=200h'
- '--web.enable-lifecycle'
ports:
  - 9090:9090
networks:
  - monitoring
```

На этом заканчивается определение части сценария, касающейся Prometheus. Используемый образ Docker называется `prom/prometheus:latest`, а его внутреннее имя — `prometheus`. Prometheus прослушивает порт с номером `9090`.

```
grafana:
  image: grafana/grafana
  container_name: grafana
  depends_on:
    - prometheus
  restart: always
  user: "0"
  ports:
    - 3000:3000
  environment:
    - GF_SECURITY_ADMIN_PASSWORD=helloThere
```

Это текущий пароль пользователя `admin` — он нужен для подключения к Grafana.

```
- GF_USERS_ALLOW_SIGN_UP=false
- GF_PANELS_DISABLE_SANITIZE_HTML=true
- GF_SECURITY_ALLOW_EMBEDDING=true
networks:
  - monitoring
volumes:
  - ./grafana_data:/var/lib/grafana/
```

Наконец, мы представляем часть, касающуюся Grafana. Grafana прослушивает порт номер `3000`.

```
volumes:
  grafana_data: {}
  prometheus_data: {}
```

Эти две строки в сочетании с двумя полями `volumes` позволяют как Grafana, так и Prometheus сохранять свои данные локально, чтобы они не терялись при каждом перезапуске образов Docker.

```
networks:
  monitoring:
    driver: bridge
```

Внутренне все три контейнера известны по значению их поля `container_name`. Однако внешне вы можете подключиться к открытым портам со своего локального компьютера как `http://localhost:port` или с другой машины, используя `http://hostname:port`. Второй способ не очень безопасен и должен быть заблокирован брандмауэром. Наконец, вам нужно запустить `docker-compose up`, и все заработает! Go-приложение начинает предоставлять данные, а Prometheus — собирает их.

На рис. 8.1 показано, как пользовательский интерфейс Prometheus (<http://hostname:9090>) отображает простой график `packt_n_goroutines`.

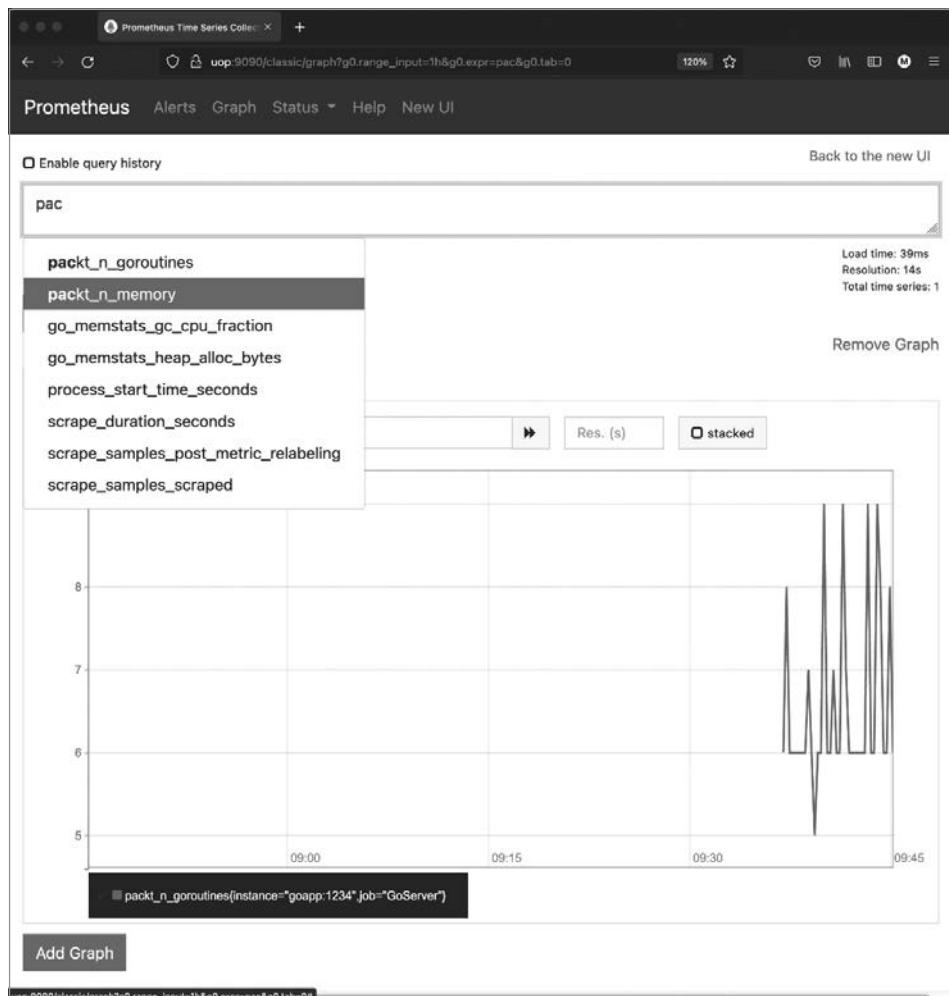


Рис. 8.1. Пользовательский интерфейс Prometheus

Этот вывод, который показывает значения метрик графическим способом, очень удобен для отладки, но далек от того, чтобы быть по-настоящему профессиональным методом, поскольку Prometheus не является инструментом визуализации. В следующем подразделе показано, как соединить Prometheus с Grafana для создания впечатляющих графиков.

Визуализация метрик Prometheus в Grafana

Нет смысла собирать показатели, если ничего с ними не делать. Я имею в виду визуализацию. Prometheus и Grafana очень хорошо работают вместе, поэтому мы используем Grafana в части визуализации. Единственная наиболее важная задача, которую необходимо выполнить в Grafana, — это подключить ее к экземпляру Prometheus. В терминологии Grafana вы должны создать *источник данных* Grafana, который позволяет Grafana получать данные из Prometheus.

Шаги по созданию источника данных с помощью нашей установки Prometheus представлены ниже.

1. Сначала перейдите на <http://localhost:3000>, чтобы подключиться к Grafana.
2. Имя пользователя администратора — `admin`, тогда как пароль определен в значении параметра `GF_SECURITY_ADMIN_PASSWORD` файла `docker-compose.yml`.
3. Затем выберите `Add your first data source`. Из списка источников данных выберите `Prometheus`, который обычно находится в верхней части списка.
4. Введите `http://prometheus:9090` в поле URL, а затем нажмите кнопку `Save & Test`. Благодаря внутренней сети, которая существует между образами Docker, контейнер Grafana знает контейнер Prometheus по имени хоста `prometheus` — это значение поля `container_name`. Как вы уже знаете, к Prometheus можно также подключиться и со своего локального компьютера через `http://localhost:9090`. Мы закончили! Имя источника данных — `Prometheus`.

Выполнив эти шаги, создайте новую панель инструментов на начальном экране Grafana и поместите новую панель на нее. Выберите `Prometheus` в качестве источника данных панели, если он еще не выбран. Затем перейдите в раскрывающееся меню `Metrics` и выберите нужные метрики. Нажмите `Save`. Все готово. Создавайте столько панелей, сколько захотите.

На рис. 8.2 показано, как Grafana визуализирует две метрики из Prometheus, предоставленные файлом `prometheus.go`.

Grafana предоставляет гораздо больше возможностей, чем те, которые описаны здесь. Если вы работаете с системными показателями и хотите проверить производительность своих Go-приложений, то Prometheus и Grafana — хороший выбор.

Ознакомившись с HTTP-серверами, в следующем разделе мы выясним, как разрабатывать HTTP-клиенты.

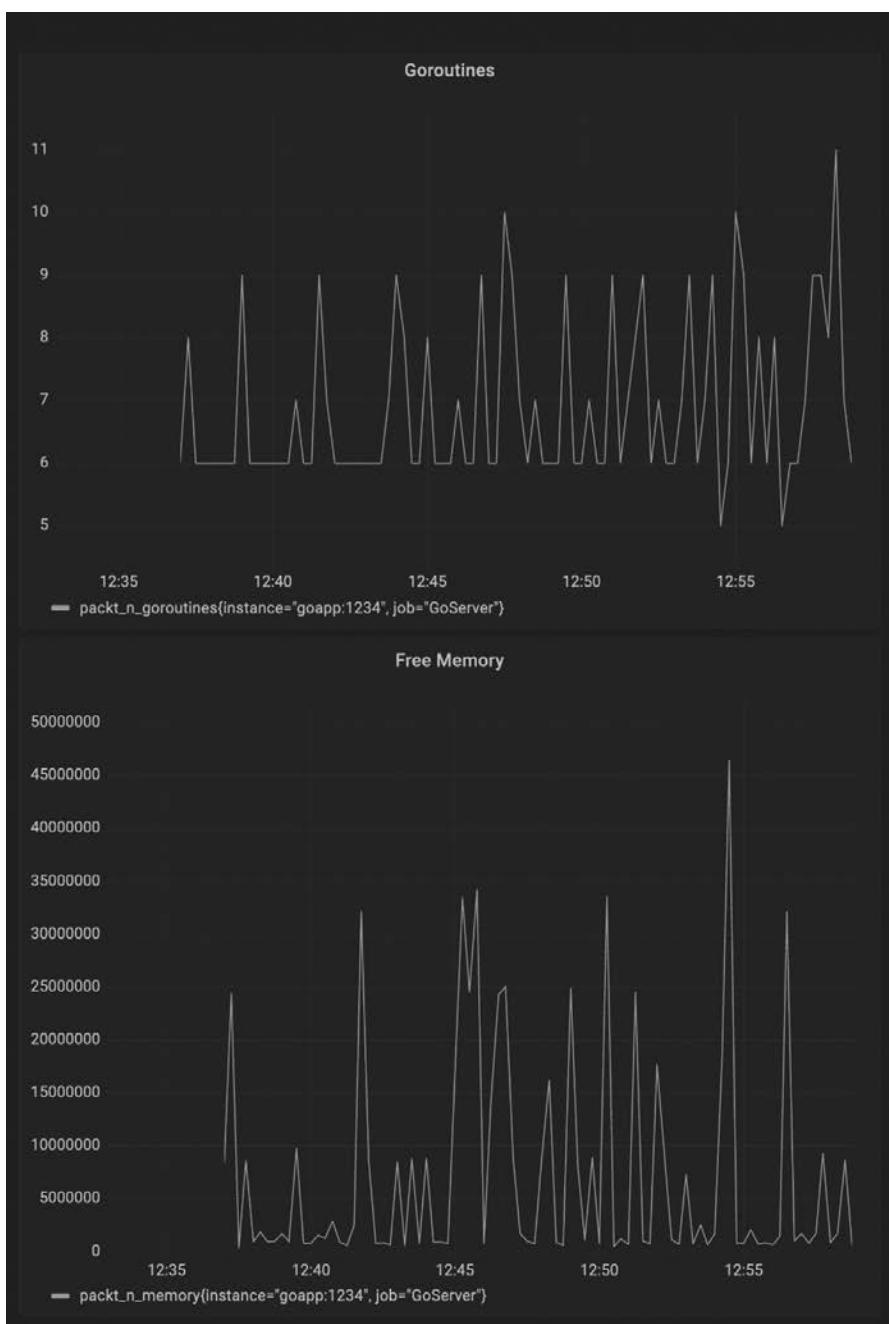


Рис. 8.2. Визуализация метрик в Grafana

Разработка веб-клиентов

В этом разделе показано, как разрабатывать HTTP-клиенты, начиная с упрощенной версии и заканчивая более расширенной. В упрощенной версии всю работу выполняет вызов `http.Get()`, что довольно удобно, когда вы не хотите иметь дело со множеством опций и параметров. Однако этот тип вызова не дает процессу быть гибким. Обратите внимание, что `http.Get()` возвращает значение `http.Response`. Все это показано в файле `simpleClient.go`:

```
package main

import (
    "fmt"
    "io"
    "net/http"
    "os"
    "path/filepath"
)

func main() {
    if len(os.Args) != 2 {
        fmt.Printf("Usage: %s URL\n", filepath.Base(os.Args[0]))
        return
    }
}
```

Функция `filepath.Base()` возвращает последний элемент пути. Когда в качестве параметра передается `os.Args[0]`, будет возвращено имя исполняемого двоичного файла.

```
URL := os.Args[1]
data, err := http.Get(URL)
```

В этих двух операторах мы получаем URL и его данные с помощью `http.Get()`, которая возвращает `*http.Response` и переменную `error`. Значение `*http.Response` уже содержит всю информацию, поэтому вам не нужно выполнять никаких дополнительных вызовов `http.Get()`.

```
if err != nil {
    fmt.Println(err)
    return
}

_, err = io.Copy(os.Stdout, data.Body)
```

Функция `io.Copy()` считывает данные из `data.Body`, где содержится тело ответа сервера, и записывает их в `os.Stdout`. Поскольку `os.Stdout` всегда открыт, вам не нужно открывать его для записи. Таким образом, все данные записываются в стандартный вывод, которым обычно служит окно терминала.

```
    if err != nil {
        fmt.Println(err)
        return
    }
    data.Body.Close()
}
```

Наконец, мы закрываем `data.Body`, облегчив работу сборщику мусора.

Работа с файлом `simpleClient.go` приводит к следующему выводу, который в нашем случае дается в сокращенном виде:

```
$ go run simpleClient.go https://www.golang.org
<!DOCTYPE html>
<html lang="en">
<meta charset="utf-8">
<meta name="description" content="Go is an open source programming
language that makes it easy to build simple, reliable, and efficient
software.">
...
</script>
```

Хотя `simpleClient.go` и выполняет задачу по проверке того, что данный URL существует и доступен, он не дает никакого контроля над процессом. В следующем подразделе мы разработаем расширенный HTTP-клиент, который обрабатывает ответ сервера.

Использование `http.NewRequest()` для улучшения работы клиента

Поскольку веб-клиент, описанный в предыдущем разделе, относительно прост и не предоставляет никакой гибкости, то в этом подразделе мы разберемся, как прочитать URL, не используя функцию `http.Get()` и имея большое количество опций. Однако дополнительная гибкость потребует определенных затрат, поскольку вам необходимо написать больше кода.

Код `wwwClient.go`, без блока `import`, выглядит следующим образом:

```
package main

// чтобы увидеть блок import, перейдите в репозиторий на GitHub

func main() {
    if len(os.Args) != 2 {
        fmt.Printf("Usage: %s URL\n", filepath.Base(os.Args[0]))
        return
    }
```

Хотя использование `filepath.Base()` и не обязательно, оно сделает ваш вывод более профессиональным.

```
URL, err := url.Parse(os.Args[1])
if err != nil {
    fmt.Println("Error in parsing:", err)
    return
}
```

Функция `url.Parse()` преобразует строку в структуру URL. Это означает, что если данный аргумент не является допустимым URL, то `url.Parse()` это заметит. Как обычно, проверяем переменную `error`.

```
c := &http.Client{
    Timeout: 15 * time.Second,
}

request, err := http.NewRequest(http.MethodGet, URL.String(), nil)
if err != nil {
    fmt.Println("Get:", err)
    return
}
```

Функция `http.NewRequest()` возвращает объект `http.Request` на основе переданного метода, URL и необязательного тела запроса. Параметр `http.MethodGet` определяет, что мы хотим получить данные, используя метод HTTP GET, тогда как `URL.String()` возвращает строковое значение переменной `http.URL`.

```
httpData, err := c.Do(request)
if err != nil {
    fmt.Println("Error in Do():", err)
    return
}
```

Функция `http.Do()` отправляет HTTP-запрос (`http.Request`) с помощью `http.Client` и получает `http.Response`. Таким образом, `http.Do()` выполняет работу `http.Get()` более детально.

```
fmt.Println("Status code:", httpData.Status)
```

`httpData.Status` содержит код состояния HTTP ответа — это действительно важно, поскольку позволяет понять, что на самом деле произошло с запросом.

```
header, _ := httputil.DumpResponse(httpData, false)
fmt.Print(string(header))
```

Функция `httputil.DumpResponse()` используется для получения ответа от сервера и в основном в целях отладки. Второй аргумент `httputil.DumpResponse()` — это логическое значение, которое указывает, будет ли функция включать в вывод

тело ответа. В нашем случае ему присвоено значение `false`, что исключает тело ответа из выходных данных и выводит лишь заголовок. Если вы хотите сделать то же самое на стороне сервера, то вам следует использовать `httputil.DumpRequest()`.

```
contentType := httpData.Header.Get("Content-Type")
characterSet := strings.SplitAfter(contentType, "charset=")
if len(characterSet) > 1 {
    fmt.Println("Character Set:", characterSet[1])
}
```

Здесь мы узнаем о наборе символов ответа, выполнив поиск по значению `Content-Type`.

```
if httpData.ContentLength == -1 {
    fmt.Println("ContentLength is unknown!")
} else {
    fmt.Println("ContentLength:", httpData.ContentLength)
}
```

Здесь мы пытаемся получить длину содержимого из ответа, прочитав `httpData.ContentLength`. Однако если значение не задано, то мы выводим соответствующее сообщение.

```
length := 0
var buffer [1024]byte
r := httpData.Body
for {
    n, err := r.Read(buffer[0:])
    if err != nil {
        fmt.Println(err)
        break
    }
    length = length + n
}
fmt.Println("Calculated response data length:", length)
```

В последней части программы мы используем метод самостоятельного определения размера HTTP-ответа сервера. Если бы нам требовалось отобразить вывод HTML на экране, то мы могли бы вывести содержимое переменной буфера `r`.

Работа с `wwwClient.go` и посещение <https://www.golang.org> приводят к следующему выводу, который является результатом `fmt.Println("Status code:", httpData.Status)`:

```
$ go run wwwClient.go https://www.golang.org
Status code: 200 OK
```

Далее мы видим вывод оператора `fmt.Println(string(header))` с данными заголовка ответа HTTP-сервера:

```
HTTP/2.0 200 OK
Alt-Svc: h3-29=:443"; ma=2592000,h3-T051=:443";
ma=2592000,h3-Q050=:443"; ma=2592000,h3-Q046=:443";
ma=2592000,h3-Q043=:443"; ma=2592000,quic=:443"; ma=2592000;
v="46,43"
Content-Type: text/html; charset=utf-8
Date: Sat, 27 Mar 2021 19:19:25 GMT
Strict-Transport-Security: max-age=31536000; includeSubDomains; preload
Vary: Accept-Encoding
Via: 1.1 google
```

Последняя часть выходных данных относится к набору символов (`utf-8`) и длине содержимого ответа (9216), рассчитанной с помощью кода:

```
Character Set: utf-8
ContentLength is unknown!
EOF
Calculated response data length: 9216
```

В следующем подразделе показано, как создать клиент для веб-сервиса телефонной книги, которую мы разработали ранее.

Создание клиента для сервиса телефонной книги

В этом подразделе мы создаем утилиту командной строки, которая взаимодействует с веб-сервисом телефонной книги, разработанным ранее в данной главе. Эта версия клиента телефонной книги будет создана с помощью пакета `cobra`, а значит, ей потребуется выделенный репозиторий GitHub или GitLab. В нашем случае репозиторий можно найти по адресу <https://github.com/mactsouk/phone-cli>. Первое, что нужно сделать после запуска `git clone`, — это связать репозиторий с каталогом, который будет использоваться для разработки:

```
$ cd ~/go/src/github.com/mactsouk
$ git clone git@github.com:mactsouk/phone-cli.git
$ cd phone-cli
$ ~/go/bin/cobra init --pkg-name github.com/mactsouk/phone-cli
$ go mod init
$ go mod tidy
$ go mod download
```

Далее мы должны создать команды для утилиты. Структура утилиты реализована с помощью этих команд `cobra`:

```
$ ~/go/bin/cobra add search
$ ~/go/bin/cobra add insert
```

```
$ ~/go/bin/cobra add delete
$ ~/go/bin/cobra add status
$ ~/go/bin/cobra add list
```

Итак, у нас есть утилита командной строки с пятью командами: `search`, `insert`, `delete`, `status` и `list`. Затем команды нужно реализовать и определить их локальные параметры, чтобы можно было взаимодействовать с сервером телефонной книги.

Теперь посмотрим на реализации команд, начиная с реализации функции `init()` файла `root.go`, поскольку именно здесь определяются глобальные параметры командной строки:

```
func init() {
    rootCmd.PersistentFlags().StringP("server", "S", "localhost", "Server")
    rootCmd.PersistentFlags().StringP("port", "P", "1234", "Port number")

    viper.BindPFlag("server", rootCmd.PersistentFlags().Lookup("server"))
    viper.BindPFlag("port", rootCmd.PersistentFlags().Lookup("port"))
}
```

Итак, мы определяем два глобальных параметра `server` и `port`, которые являются именем хоста и номером порта соответственно. Оба параметра имеют псевдонимы и обрабатываются `viper`.

Теперь рассмотрим реализацию команды `status` в `status.go`:

```
SERVER := viper.GetString("server")
PORT := viper.GetString("port")
```

Все команды считывают значения параметров командной строки `server` и `port`, получая информацию о сервере, и команда `status` здесь не является исключением.

```
// создаем запрос
URL := "http://" + SERVER + ":" + PORT + "/status"
```

После этого мы создаем полный URL запроса.

```
data, err := http.Get(URL)
if err != nil {
    fmt.Println(err)
    return
}
```

Затем мы отправляем запрос GET на сервер, используя `http.Get()`.

```
// проверяем код состояния HTTP
if data.StatusCode != http.StatusOK {
    fmt.Println("Status code:", data.StatusCode)
    return
}
```

После этого мы проверяем код состояния HTTP-запроса, чтобы убедиться, что все в порядке.

```
// считываем данные
responseData, err := io.ReadAll(data.Body)
if err != nil {
    fmt.Println(err)
    return
}
fmt.Print(string(responseData))
```

Если все в порядке, то мы считываем весь текст ответа сервера, который представляет собой байтовый срез, и выводим его на экран в виде строки. Реализация `list` почти идентична реализации `status`. Различия заключаются лишь в том, что реализация расположена в `list.go`, а полный URL сконструирован следующим образом:

```
URL := "http://" + SERVER + ":" + PORT + "/list"
```

Далее посмотрим, как команда `delete` реализована в `delete.go`:

```
SERVER := viper.GetString("server")
PORT := viper.GetString("port")
number, _ := cmd.Flags().GetString("tel")
if number == "" {
    fmt.Println("Number is empty!")
    return
}
```

Помимо считывания значений глобальных параметров `server` и `port`, мы получаем значение параметра `tel`. Если `tel` не содержит значения, то команда возвращается.

```
// создаем запрос
URL := "http://" + SERVER + ":" + PORT + "/delete/" + number
```

Повторюсь, мы создаем полный URL запроса перед подключением к серверу.

```
// отправить запрос на сервер
data, err := http.Get(URL)
if err != nil {
    fmt.Println(err)
    return
}
```

Затем мы отправляем запрос на сервер.

```
// проверяем код состояния HTTP
if data.StatusCode != http.StatusOK {
    fmt.Println("Status code:", data.StatusCode)
    return
}
```

Если в ответе сервера ошибка, то команда `delete` завершается.

```
// считываем данные
responseData, err := io.ReadAll(data.Body)
if err != nil {
    fmt.Println(err)
    return
}
fmt.Print(string(responseData))
```

Если все прошло гладко, то текст ответа сервера выводится на экран.

Функция `init()` в программе `delete.go` содержит определение локального параметра командной строки `tel`:

```
func init() {
    rootCmd.AddCommand(deleteCmd)
    deleteCmd.Flags().StringP("tel", "t", "", "Telephone number to
    delete")
}
```

Это локальный флаг, доступный только для команды `delete`. Далее обратимся к команде `search` и узнаем о том, как она реализована в программе `search.go`. Реализация такая же, как и в случае `delete`, за исключением полного URL запроса:

```
URL := "http://" + SERVER + ":" + PORT + "/search/" + number
```

Команда `search` также поддерживает параметр командной строки `tel`, позволяющий получить телефонный номер для поиска, — это определено в функции `init()` в `search.go`.

Последней представленной командой является команда `insert`, которая поддерживает три локальных параметра командной строки, определенных в функции `init()` в программе `insert.go`:

```
func init() {
    rootCmd.AddCommand(insertCmd)
    insertCmd.Flags().StringP("name", "n", "", "Name value")
    insertCmd.Flags().StringP("surname", "s", "", "Surname value")
    insertCmd.Flags().StringP("tel", "t", "", "Telephone value")
}
```

Эти три параметра необходимы для получения требуемого пользовательского ввода. Обратите внимание, что псевдоним для `surname` — это *строчная* буква `s`, тогда как псевдоним для `server`, определенной в `root.go`, — *прописная* `S`. Команды и их псевдонимы определяются пользователем, так что при их выборе руководствуйтесь здравым смыслом.

Команда реализована с использованием следующего кода:

```
SERVER := viper.GetString("server")
PORT := viper.GetString("port")
```

Сначала мы считываем глобальные параметры `server` и `port`.

```
number, _ := cmd.Flags().GetString("tel")
if number == "" {
    fmt.Println("Number is empty!")
    return
}
name, _ := cmd.Flags().GetString("name")
if name == "" {
    fmt.Println("Name is empty!")
    return
}
surname, _ := cmd.Flags().GetString("surname")
if surname == "" {
    fmt.Println("Surname is empty!")
    return
}
```

Затем получаем значения трех локальных параметров командной строки. Если какой-либо из них содержит пустое значение, то команда возвращается, не отправив запрос на сервер.

```
URL := "http://" + SERVER + ":" + PORT + "/insert/"
URL = URL + "/" + name + "/" + surname + "/" + num
```

Здесь мы в два этапа (для удобства чтения) создаем запрос сервера.

```
data, err := http.Get(URL)
if err != nil {
    fmt.Println("**", err)
    return
}
```

Затем мы отправляем запрос на сервер.

```
if data.StatusCode != http.StatusOK {
    fmt.Println("Status code:", data.StatusCode)
    return
}
```

Проверка кода состояния HTTP — это всегда хорошо. Так что если с ответом сервера все в порядке, то мы продолжаем чтение данных. В противном случае выводим код состояния и завершаем работу.

```
responseData, err := io.ReadAll(data.Body)
if err != nil {
    fmt.Println("**", err)
    return
}
fmt.Print(string(responseData))
```

Прочитав тело ответа сервера, который хранится в байтовом срезе, выводим его на экран в виде строки, используя для этого `string(responseData)`.

Клиентское приложение генерирует следующий вид выходных данных:

```
$ go run main.go list
Dimitris Tsoukalos 2101112223
Jane Doe 0800123456
Mike Tsoukalos 2109416471
```

Это вывод команды `list`.

```
$ go run main.go status
Total entries: 3
```

Вывод команды `status` информирует нас о количестве записей в телефонной книге.

```
$ go run main.go search --tel 0800123456
Jane Doe 0800123456
```

Здесь показано использование команды `search` при успешном нахождении номера.

```
$ go run main.go search --tel 0800
Status code: 404
```

Здесь показано использование команды `search`, когда номер не найден.

```
$ go run main.go delete --tel 2101112223
2101112223 deleted!
```

Это результат выполнения команды `delete`.

```
$ go run main.go insert -n Michalis -s Tsoukalos -t 2101112223
New record added successfully
```

Это работа команды `insert`. Если вы попытаетесь вставить один и тот же номер более одного раза, то на выходе сервера будет `Status code: 304`.

В следующем разделе показано, как создать FTP-сервер с помощью `net/http`.

Создание файловых серверов

Файловый сервер сам по себе не является веб-сервером, но тесно связан с веб-сервисами, поскольку реализуется с использованием аналогичных Go-пакетов. Кроме того, файловые серверы часто используются для поддержки функциональности веб-серверов и веб-сервисов.

Go предлагает для этого обработчик `http.FileServer()`, а также `http.ServeFile()`. Самая большая разница между ними заключается в том, что `http.FileServer()` —

это `http.Handler`, тогда как в случае `http.ServeFile()` это не так. Кроме того, `http.ServeFile()` лучше обслуживает отдельные файлы, а `http.FileServer()` — целые деревья каталогов.

Простой пример кода `http.FileServer()` представлен в файле `FileServer.go`:

```
package main

import (
    "fmt"
    "log"
    "net/http"
)

var PORT = ":8765"

func defaultHandler(w http.ResponseWriter, r *http.Request) {
    log.Println("Serving:", r.URL.Path, "from", r.Host)
    w.WriteHeader(http.StatusOK)
    Body := "Thanks for visiting!\n"
    fmt.Fprintf(w, "%s", Body)
}
```

Это ожидаемый обработчик HTTP-сервера по умолчанию.

```
func main() {
    mux := http.NewServeMux()
    mux.HandleFunc("/", defaultHandler)

    fileServer := http.FileServer(http.Dir("/tmp/"))
    mux.Handle("/static/", http.StripPrefix("/static", fileServer))
```

`mux.Handle()` регистрирует файловый сервер в качестве обработчика для всех URL, которые начинаются с `/static/`. Но когда найдено совпадение, мы удаляем префикс `/static/` до того, как файловый сервер попытается обработать такой запрос, поскольку `/static/` не является частью местоположения, в котором файлы находятся фактически. Что касается Go, то `http.FileServer()` — это просто еще один обработчик.

```
fmt.Println("Starting server on:", PORT)
err := http.ListenAndServe(PORT, mux)
fmt.Println(err)
```

Наконец, мы запускаем HTTP-сервер с помощью `http.ListenAndServe()`.

Использование `curl(1)` для посещения `/static/` приводит к следующему виду вывода в формате HTML:

```
$ curl http://localhost:8765/static/
<pre>
```

```
<a href="AlTest1.out">AlTest1.out</a>
<a href="adobegc.log">adobegc.log</a>
<a href="com.google.Keystone/">com.google.Keystone/</a>
<a href="data.csv">data.csv</a>
<a href="fsevents-d-uuid">fsevents-d-uuid</a>
<a href="powerlog/">powerlog/</a>
</pre>
```

Вы также можете посетить `http://localhost:8765/static/` в вашем браузере или FTP-клиенте, чтобы просмотреть файлы и каталоги FTP-сервера.

В следующем подразделе показано, как использовать `http.ServeFile()` для обслуживания отдельных файлов.

Загрузка содержимого приложения телефонной книги

В этом подразделе мы создадим и реализуем конечную точку, которая позволит нам скачивать содержимое одного файла. Код создает *временный файл с другим именем файла для каждого запроса* с содержимым приложения телефонной книги. По соображениям простоты представленный код поддерживает две конечные точки HTTP: одну — для маршрутизатора по умолчанию, а другую — для обслуживания файла. Мы обслуживаем один файл, поэтому используем `http.ServeFile()`, который отвечает на запрос содержимым указанного файла или каталога.

Каждый временный файл хранится в файловой системе в течение *30 секунд*, прежде чем будет удален. Чтобы имитировать реальную ситуацию, представленная утилитой считывает `data.csv`, помещает его в срез и создает файл на основе содержимого `data.csv`. Название утилиты — `getEntries.go`, а ее наиболее важным кодом служит реализация функции `getFileHandler()`:

```
func getFileHandler(w http.ResponseWriter, r *http.Request) {
    var tempFileName string

    // создаем временное имя файла
    f, err := os.CreateTemp("", "data*.txt")
    tempFileName = f.Name()
```

Временный путь создается с помощью `os.CreateTemp()` на основе заданного шаблона и путем добавления случайной строки в конец. Если шаблон содержит `*`, то случайно сгенерированная строка заменяет последнюю `*`. Точное место, где создается файл, зависит от используемой операционной системы.

```
// удалить файл
defer os.Remove(tempFileName)
```

Мы не хотим, чтобы в конечном итоге осталось множество временных файлов, поэтому удаляем файл, когда функция обработчика возвращается.

```
// сохраняем в него данные
err = saveCSVFile(tempFileName)
if err != nil {
    fmt.Println(err)
    w.WriteHeader(http.StatusNotFound)
    fmt.Fprintln(w, "Cannot create: "+tempFileName)
    return
}

fmt.Println("Serving ", tempFileName)

http.ServeFile(w, r, tempFileName)
```

Именно здесь временный файл отправляется клиенту.

```
// 30 секунд для получения файла
time.Sleep(30 * time.Second)
}
```

Вызов `time.Sleep()` задерживает удаление временного файла на 30 секунд — вы можете задать любой период задержки, который вам нравится.

Что касается функции `main()`, то `getFileHandler()` — это обычная функция-обработчик, используемая в операторе `mux.HandleFunc("/getContents/", getFileHandler)`. Следовательно, каждый раз, когда клиент запрашивает `/getContents/`, содержимое файла возвращается HTTP-клиенту.

При выполнении `getEntries.go` и посещении `/getContents/` мы получаем такой вывод:

```
$ curl http://localhost:8765/getContents/
Dimitris,Tsoukalos,2101112223,1617028128
Jane,Doe,0800123456,1608559903
Mike,Tsoukalos,2109416471,1617028196
```

Поскольку мы возвращаем данные в виде обычного текста, выходные данные отображаются на экране.



В главе 10 представлен другой способ создания файлового сервера, который поддерживает как загрузку, так и выгрузку файлов с помощью пакета `gorilla/mux`.

В следующем разделе объясняется, как установить время ожидания HTTP-соединений.

Время ожидания HTTP-соединений

В этом разделе представлены методы установки времени ожидания HTTP-соединений, которые завершаются слишком долго и работают либо на стороне сервера, либо на стороне клиента.

Использование функции `SetDeadline()`

Функция `SetDeadline()` используется `net` для установки крайних сроков чтения и записи сетевых подключений. Из-за того, как она работает, вам придется вызывать ее перед любой операцией чтения или записи. Имейте в виду, что Go использует крайние сроки для реализации периодов ожидания, поэтому вам не нужно сбрасывать их каждый раз, когда ваше приложение получает или отправляет данные. Использование `SetDeadline()` показано в файле `WithDeadline.go` и более конкретно — в реализации функции `Timeout()`:

```
var timeout = time.Duration(time.Second)

func Timeout(network, host string) (net.Conn, error) {
    conn, err := net.DialTimeout(network, host, timeout)
    if err != nil {
        return nil, err
    }

    conn.SetDeadline(time.Now().Add(timeout))
    return conn, nil
}
```

Глобальная переменная `timeout` определяет период ожидания, используемый в вызове `SetDeadline()`.

Эта функция используется в следующем коде внутри `main()`:

```
t := http.Transport{
    Dial: Timeout,
}
client := http.Client{
    Transport: &t,
}
```

Итак, `http.Transport` использует `Timeout()` в поле `Dial` и `http.Client` использует `http.Transport`. Когда вы вызываете метод `client.Get()` с желаемым URL, который здесь не показан, из-за определения `http.Transport` автоматически используется `Timeout`. Если функция `Timeout` возвращается до получения ответа сервера, то у нас сработал тайм-аут.

Использование программы `WithDeadline.go` приводит к следующему выводу:

```
$ go run WithDeadline.go http://www.golang.org
Timeout value: 1s
<!DOCTYPE html>
...
```

Вызов прошел успешно, и его завершение заняло менее секунды, поэтому тайм-аута не было.

```
$ go run WithDeadline.go http://localhost:80
Timeout value: 1s
Get "http://localhost:80": read tcp 127.0.0.1:52492->127.0.0.1:80: i/o
timeout
```

На этот раз у нас есть тайм-аут, так как серверу потребовалось слишком много времени для ответа. Далее мы покажем, как установить время ожидания соединения с помощью пакета `context`.

Установка периода ожидания на стороне клиента

В этом подразделе представлен метод установки времени ожидания сетевых подключений, завершение которых на *стороне клиента* занимает слишком много времени. Итак, если клиент не получает ответа от сервера за нужное время, то закрывает соединение. Этот метод показан в исходном коде файла `timeoutClient.go` без блока `import`.

```
package main

// для блока импорта перейдите в репозиторий кода книги

var myUrl string
var delay int = 5
var wg sync.WaitGroup

type myData struct {
    r    *http.Response
    err error
}
```

В коде выше мы определили глобальные переменные и структуру, которые будут использоваться в остальной части программы.

```
func connect(c context.Context) error {
    defer wg.Done()
```

```
data := make(chan myData, 1)
tr := &http.Transport{}
httpClient := &http.Client{Transport: tr}
req, _ := http.NewRequest("GET", myUrl, nil)
```

Здесь мы инициализируем переменные HTTP-соединения. Канал `data` используется в операторе `select`, который приводится ниже. Кроме того, параметр `c context.Context` поставляется со встроенным каналом, который также используется в операторе `select`.

```
go func() {
    response, err := httpClient.Do(req)
    if err != nil {
        fmt.Println(err)
        data <- myData{nil, err}
        return
    } else {
        pack := myData{response, err}
        data <- pack
    }
}()
```

Вышеприведенная горутина используется для взаимодействия с HTTP-сервером. Это обычное взаимодействие HTTP-клиента с HTTP-сервером, так что здесь нет ничего особенного.

```
select {
case <-c.Done():
    tr.CancelRequest(req)
    <-data
    fmt.Println("The request was canceled!")
    return c.Err()
```

Код, который выполняет этот блок `select`, основывается на том, истекает ли время ожидания контекста. Если сначала истекает время ожидания контекста, то клиентское соединение отменяется с помощью `tr.CancelRequest(req)`.

```
case ok := <-data:
    err := ok.err
    resp := ok.r
    if err != nil {
        fmt.Println("Error select:", err)
        return err
    }
    defer resp.Body.Close()

    realHTTPData, err := io.ReadAll(resp.Body)
    if err != nil {
        fmt.Println("Error select:", err)
```

```

        return err
    }
    fmt.Printf("Server Response: %s\n", realHTTPData)
}
return nil
}

```

Вторая ветвь `select` имеет дело с данными, полученными с HTTP-сервера, которые обрабатываются обычным способом.

```

func main() {
    if len(os.Args) == 1 {
        fmt.Println("Need a URL and a delay!")
        return
    }

    myUrl = os.Args[1]
    if len(os.Args) == 3 {
        t, err := strconv.Atoi(os.Args[2])
        if err != nil {
            fmt.Println(err)
            return
        }
        delay = t
    }
}

```

URL считывается напрямую, поскольку он уже является строковым значением, тогда как период задержки преобразуется в числовое значение с помощью `strconv.Atoi()`.

```

fmt.Println("Delay:", delay)
c := context.Background()
c, cancel := context.WithTimeout(c, time.Duration(delay)*time.Second)
defer cancel()

```

Период ожидания определяется методом `context.WithTimeout()`. Считается хорошей практикой использовать `context.Background()` в функции `main()` или `init()` пакета или же в тестах.

```

fmt.Printf("Connecting to %s \n", myUrl)
wg.Add(1)
go connect(c)
wg.Wait()
fmt.Println("Exiting...")
}

```

Функция `connect()`, которая также выполняется какgorутина, либо завершается нормально, либо при выполнении функции `cancel()`. Функция `cancel()` — это то, что вызывает метод `Done()` в `c`.

Работа с файлом `timeoutClient.go` и ситуация тайм-аута генерируют следующий вывод:

```
$ go run timeoutClient.go http://localhost:80
Delay: 5
Connecting to http://localhost:80
Get "http://localhost:80": net/http: request canceled
The request was canceled!
Exiting...
```

В следующем подразделе показано, как установить время ожидания HTTP-запроса на стороне сервера.

Установка времени ожидания на стороне сервера

В этом подразделе представлен метод установки времени ожидания сетевых подключений, завершение которых на *стороне сервера* занимает слишком много времени. Это гораздо важнее, чем клиентская сторона, поскольку сервер со слишком большим количеством открытых подключений может быть не в состоянии обрабатывать больше запросов, если некоторые из уже открытых подключений не будут закрыты. Обычно это происходит по двум причинам. Первая — это программные ошибки, а вторая — когда сервер подвергается атаке типа «Отказ в обслуживании» (Denial of Service, DoS)!

Функция `main()` в файле `timeoutServer.go` демонстрирует эту технику:

```
func main() {
    PORT := ":8001"
    arguments := os.Args
    if len(arguments) != 1 {
        PORT = ":" + arguments[1]
    }
    fmt.Println("Using port number: ", PORT)

    m := http.NewServeMux()
    srv := &http.Server{
        Addr:         PORT,
        Handler:      m,
        ReadTimeout:  3 * time.Second,
        WriteTimeout: 3 * time.Second,
    }
```

Здесь определяются периоды времени ожидания. Обратите внимание, что вы можете определить периоды ожидания для процессов как чтения, так и записи. Значение поля `ReadTimeout` указывает максимальное время, разрешенное для чтения всего клиентского запроса, включая тело, в то время как значение поля

`WriteTimeout` указывает максимальную продолжительность времени до отправки ответа клиента.

```
m.HandleFunc("/time", timeHandler)
m.HandleFunc("/", myHandler)

err := srv.ListenAndServe()
if err != nil {
    fmt.Println(err)
    return
}
}
```

Помимо параметров в определении `http.Server`, остальная часть кода совершенно обычна. Она содержит функции обработчика и вызывает `ListenAndServe()` для запуска HTTP-сервера.

Работа с файлом `timeoutServer.go` не генерирует выходных данных. Однако, если клиент подключается к нему без отправки каких-либо запросов, клиентское соединение завершится через три секунды. То же самое произойдет, если клиенту потребуется более трех секунд, чтобы получить ответ сервера.

Упражнения

- Поместите все обработчики из `www-phone.go` в другой Go-пакет и соответствующим образом измените `www-phone.go`. Вам понадобится другой репозиторий для хранения нового пакета.
- Измените `wwwClient.go` так, чтобы сохранять вывод HTML во внешний файл.
- Включите функциональность утилиты `getEntries.go` в приложение телефонной книги.
- Реализуйте простую версию `ab(1)`, используя горутины и каналы. `ab(1)` — это инструмент сравнительного анализа HTTP-сервера Apache.

Резюме

В этой главе мы узнали, как работать с HTTP, создавать образы Docker из кода Go, предоставлять метрики Prometheus, а также разрабатывать HTTP-клиенты и серверы. Мы также обновили приложение телефонной книги до веб-приложения и запрограммировали для него клиент командной строки. Кроме того, мы узнали, как устанавливать время ожидания HTTP-соединений и разрабатывать файловые серверы.

Теперь мы готовы приступить к разработке эффективных и многопоточных HTTP-приложений. Однако мы еще не закончили с HTTP. В главе 10 мы расставим все точки над *i* и разберемся с тем, как разрабатывать мощные серверы и клиенты RESTful.

Но сначала нам нужно выяснить, как работать с TCP/IP, TCP, UDP и WebSocket, что и является предметом обсуждения в следующей главе.

Дополнительные ресурсы

- Сервер Caddy: <https://caddyserver.com/>.
- Сервер Nginx: <https://nginx.org/en/>.
- Гистограммы в Prometheus: <https://prometheus.io/docs/practices/histograms/>.
- Пакет `net/http`: <https://golang.org/pkg/net/http/>.
- Официальные Docker-образы Go: https://hub.docker.com/_/golang/.

9

Работа с TCP/IP и WebSocket

В этой главе вы узнаете, как с помощью пакета `net` работать с протоколами более низкого уровня TCP/IP, которыми являются TCP и UDP. Это даст возможность разрабатывать серверы и клиенты TCP/IP. Кроме того, в данной главе показано, как разрабатывать серверы и клиенты для протокола *WebSocket*, который основан на HTTP, а также доменные сокеты UNIX для сервисов программирования, которые работают только на локальном компьютере.

В частности, в этой главе рассматриваются:

- TCP/IP;
- пакет `net`;
- разработка TCP-клиента;
- разработка TCP-сервера;
- разработка UDP-клиента;
- разработка UDP-сервера;
- разработка параллельных TCP-серверов;
- работа с доменными сокетами UNIX;
- создание сервера WebSocket;
- создание клиента WebSocket.

TCP/IP

TCP/IP – это семейство протоколов, которые помогают Интернету работать. Название происходит от двух наиболее известных протоколов: TCP и IP. TCP расшифровывается как *Transmission Control Protocol* (протокол управления передачей). Программное обеспечение TCP передает данные между машинами, используя сегменты, которые также называются *TCP-пакетами*. Основной характеристикой TCP выступает надежность протокола, а это значит, что он гарантирует доставку пакета, не требуя от программиста никакого дополнительного кода. Если подтверждения доставки пакета не происходит, то TCP отправляет этот конкретный пакет повторно. Помимо прочего, TCP-пакеты могут использоваться для установления соединений, передачи данных, отправки подтверждений и закрытия соединений.

Когда две машины соединяются по TCP, между ними создается полнодуплексный виртуальный канал, аналогичный телефонному звонку. Две машины постоянно обмениваются данными с целью гарантировать, что данные отправляются и принимаются правильно. Если по какой-либо причине соединение прерывается, то две машины пытаются локализовать проблему и сообщить об этом соответствующему приложению. Заголовок TCP каждого пакета включает поля *порта источника* и *порта назначения*. Эти два поля, а также IP-адреса источника и назначения объединяются в целях уникальной идентификации каждого отдельного TCP-соединения. Все эти детали обрабатываются TCP/IP до тех пор, пока вы предоставите необходимые данные без каких-либо дополнительных усилий.



При создании серверных процессов TCP/IP помните: номера портов 0-1024 имеют ограниченный доступ и могут использоваться только пользователем root; это значит, вам потребуются права администратора, чтобы использовать порты в этом диапазоне. Запуск процесса с правами root представляет собой угрозу безопасности, и его следует избегать.

IP расшифровывается как *Internet Protocol* (межсетевой протокол). Основной характеристикой IP служит то, что по своей природе он не является надежным протоколом. IP инкапсулирует данные, которые передаются по сети TCP/IP, поскольку отвечает за доставку пакетов от исходного хоста к конечному в соответствии с их IP-адресами. Чтобы эффективно отправить пакет адресату, IP должен иметь способ адресации. Существуют выделенные устройства, называемые маршрутизаторами, которые выполняют IP-маршрутизацию; тем не менее каждое устройство TCP/IP должно уметь выполнять некую

базовую маршрутизацию. Первая версия IP-протокола теперь называется *IPv4*, чтобы отличать ее от последней версии IP-протокола, которая называется *IPv6*.

Основная проблема с IPv4 заключается в том, что у него вскоре закончатся доступные IP-адреса, что и является основной причиной создания протокола IPv6. Это произошло потому, что адрес IPv4 представлен только 32 битами, что позволяет использовать в общей сложности 232 (4 294 967 296) различных IP-адреса. IPv6 использует уже 128 бит для определения каждого из своих адресов. Формат адреса IPv4 — 10.20.32.245 (четыре части, разделенные точками), в то время как формат адреса IPv6 — 3fce:1706:4523:3:150:f8f f:fe21:56cf (восемь частей, разделенных двоеточиями).

UDP (User Datagram Protocol; протокол пользовательских датаграмм) основан на IP; это значит, он также ненадежен. UDP проще, чем TCP, главным образом потому, что ненадежен по своей конструкции. В результате UDP-сообщения могут теряться, дублироваться или приходить не по порядку. Кроме того, пакеты могут приходить быстрее, чем получатель успевает их обрабатывать. Таким образом, UDP используется, когда скорость важнее надежности.

В этой главе мы реализуем как TCP, так и UDP — службы TCP и UDP являются основой Интернета, и полезно знать, как разрабатывать серверы и клиенты TCP/IP в Go. Но сначала поговорим об утилите `nc(1)`.

Утилита командной строки nc(1). Утилита `nc(1)`, которая также называется `netcat(1)`, очень удобна, когда нужно протестировать серверы и клиенты TCP /IP. На самом деле `nc(1)` — это утилита для всего, что связано с TCP и UDP, а также IPv4 и IPv6, включая открытие TCP-соединений, отправку и получение UDP-сообщений и выполнение функций TCP-сервера.

Вы можете использовать `nc(1)` в качестве клиента для службы TCP, которая выполняется на компьютере с IP-адресом 10.10.1.123 и прослушивает номер порта 1234 следующим образом:

```
$ nc 10.10.1.123 1234
```

Параметр -1 дает `netcat(1)` указание действовать в качестве сервера; это значит, `netcat(1)` начинает прослушивать входящие соединения с заданным номером порта. По умолчанию `nc(1)` использует протокол TCP. Однако если вы выполняете `nc(1)` с флагом -u, то `nc(1)` использует протокол UDP либо в качестве клиента, либо в качестве сервера. Наконец, параметры -v и -vv дают `netcat(1)` указание генерировать подробный вывод, который может пригодиться, когда вы хотите устранить неполадки в сетевых подключениях.

Пакет `net`

Пакет `net` стандартной библиотеки Go полностью посвящен TCP/IP, UDP, разрешению доменных имен и доменным сокетам UNIX. Функция `net.Dial()` используется для подключения к сети в качестве клиента, тогда как `net.Listen()` служит для того, чтобы давать Go-программе указание принимать входящие сетевые подключения и, таким образом, действовать в качестве сервера. Возвращаемое значение как `net.Dial()`, так и `net.Listen()` имеет тип данных `net.Conn`, который реализует `io.Reader` и `io.Writer`. Это означает, что вы можете как читать, так и записывать в подключение `net.Conn`, используя код, связанный с файловым вводом-выводом. Первый параметр `net.Dial()` и `net.Listen()` — тип сети, однако на этом их сходство заканчивается.

Функция `net.Dial()` используется для подключения к удаленному серверу. Первый параметр функции `net.Dial()` определяет сетевой протокол, который будет использоваться, в то время как второй определяет адрес сервера, который также должен включать и номер порта. Допустимые значения для первого параметра — `tcp`, `tcp4` (только для IPv4), `tcp6` (только для IPv6), `udp`, `udp4` (только для IPv4), `udp6` (только для IPv6), `ip`, `ip4` (только для IPv4), `ip6` (только для IPv6), `unix` (сокеты UNIX), `unixgram` и `unixpacket`. Допустимые значения для `net.Listen()` — это `tcp`, `tcp4`, `tcp6`, `unix` и `unixpacket`.



Выполните команды `go doc net.Listen` и `go doc net.Dial`, чтобы получить более подробную информацию об этих двух функциях.

Разработка TCP-клиента

В этом разделе представлены два эквивалентных способа разработки TCP-клиентов.

Разработка TCP-клиента с помощью `net.Dial()`

В первую очередь мы рассмотрим наиболее широко используемый способ, который реализован в файле `tcpC.go`:

```
package main

import (
```

```
"bufio"
"fmt"
"net"
"os"
"strings"
)
```

Блок `import` включает такие пакеты, как `bufio` и `fmt`, которые также работают с файловыми операциями ввода-вывода.

```
func main() {
    arguments := os.Args
    if len(arguments) == 1 {
        fmt.Println("Please provide host:port.")
        return
    }
}
```

Сначала прочитаем сведения о TCP-сервере, к которому хотим подключиться.

```
connect := arguments[1]
c, err := net.Dial("tcp", connect)
if err != nil {
    fmt.Println(err)
    return
}
```

Зная детали подключения, мы вызываем `net.Dial()` — его первым параметром служит протокол, который мы хотим использовать (в нашем случае `tcp`), а вторым — сведения о соединении. Успешный вызов `net.Dial()` возвращает открытое соединение (интерфейс `net.Conn`), которое представляет собой общее потоковое сетевое соединение.

```
for {
    reader := bufio.NewReader(os.Stdin)
    fmt.Print("> ")
    text, _ := reader.ReadString('\n')
    fmt.Fprintf(c, text+"\n")

    message, _ := bufio.NewReader(c).ReadString('\n')
    fmt.Print("->: " + message)
    if strings.TrimSpace(string(text)) == "STOP" {
        fmt.Println("TCP client exiting...")
        return
    }
}
```

Заключительная часть TCP-клиента продолжает считывать вводимые пользователем данные до тех пор, пока в качестве входных данных не будет получено

слово `STOP`. В этом случае клиент *ожидает ответа сервера, прежде чем завершить* работу после `STOP`, поскольку именно так построен цикл `for`. В основном это происходит потому, что у сервера может быть полезный для нас ответ и мы не хотим его пропустить. Весь заданный пользовательский ввод отправляется (записывается) в открытое TCP-соединение с помощью `fmt.Fprintf()`, тогда как `bufio.NewReader()` используется для чтения данных из TCP-соединения, точно так же, как вы бы поступили с обычным файлом.

Использование `tcpC.go` для подключения к TCP-серверу, который в данном случае реализован с помощью `nc(1)`, выдает следующие выходные данные:

```
$ go run tcpC.go localhost:1234
>> Hello!
->: Hi from nc -l 1234
>> STOP
->: Bye!
TCP client exiting...
```

Строки, начинающиеся с `>>`, обозначают пользовательский ввод, тогда как строки, начинающиеся с `->`, — это сообщения сервера. После отправки `STOP` мы ждем ответа сервера, а затем клиент завершает TCP-соединение. В вышеупомянутом коде показан подход, позволяющий создать правильный TCP-клиент в Go с некоторой дополнительной логикой (ключевое слово `STOP`).

В следующем подразделе показан еще один способ создания TCP-клиента.

Разработка TCP-клиента, использующего `net.DialTCP()`

В этом подразделе представлен альтернативный способ разработки TCP-клиента. Разница заключается не в функциональности клиента, а в функциях Go, которые используются для установления TCP-соединения: `net.DialTCP()` и `net.ResolveTCPAddr()`.

Код `otherTCPclient.go` выглядит следующим образом:

```
package main

import (
    "bufio"
    "fmt"
    "net"
    "os"
    "strings"
)
```

Хотя мы работаем с подключениями TCP/IP, нам нужны такие пакеты, как `bufio`, поскольку UNIX рассматривает сетевые подключения как файлы, поэтому мы в основном работаем с операциями ввода-вывода поверх сетей.

```
func main() {
    arguments := os.Args
    if len(arguments) == 1 {
        fmt.Println("Please provide a server:port string!")
        return
    }
}
```

Нам нужно прочитать подробную информацию о TCP-сервере, к которому мы хотим подключиться, включая желаемый номер порта. При работе с TCP/IP утилита не может работать с параметрами по умолчанию, если только мы не разрабатываем крайне специализированный TCP-клиент.

```
connect := arguments[1]
tcpAddr, err := net.ResolveTCPAddr("tcp4", connect)
if err != nil {
    fmt.Println("ResolveTCPAddr:", err)
    return
}
```

Функция `net.ResolveTCPAddr()` характерна для TCP-соединений (отсюда и ее название). Она преобразует указанный адрес в значение `*net.TCPAddr`, представляющее собой структуру, содержащую адрес конечной точки TCP, — в данном случае конечной точкой является TCP-сервер, к которому мы хотим подключиться.

```
conn, err := net.DialTCP("tcp4", nil, tcpAddr)
if err != nil {
    fmt.Println("DialTCP:", err)
    return
}
```

Имея под рукой конечную точку TCP, мы вызываем `net.DialTCP()` для подключения к серверу. Помимо использования `net.ResolveTCPAddr()` и `net.DialTCP()`, остальная часть кода, имеющая отношение к взаимодействию TCP-клиента и TCP-сервера, точно такая же.

```
for {
    reader := bufio.NewReader(os.Stdin)
    fmt.Print(">> ")
    text, _ := reader.ReadString('\n')
    fmt.Fprintf(conn, text+"\n")

    message, _ := bufio.NewReader(conn).ReadString('\n')
    fmt.Print("->: " + message)
    if strings.TrimSpace(string(text)) == "STOP" {
```

```
        fmt.Println("TCP client exiting...")
        conn.Close()
        return
    }
}
}
```

Наконец, для взаимодействия с TCP-сервером используется бесконечный цикл `for`. TCP-клиент считывает пользовательские данные, которые отправляются на сервер. После этого он считывает данные с TCP-сервера. Повторюсь, ключевое слово `STOP` завершает TCP-соединение на стороне клиента с помощью метода `Close()`.

Работа с программой `otherTCPclient.go` и взаимодействие с процессом TCP-сервера приводят к следующему выводу:

```
$ go run otherTCPclient.go localhost:1234
>> Hello!
->: Hi from nc -l 1234
>> STOP
->: Thanks for connection!
TCP client exiting...
```

Взаимодействие такое же, как и в случае с `tcpC.go` — мы только что изучили другой способ разработки TCP-клиентов. Если вы хотите знать мое мнение, то я предпочитаю реализацию `tcpC.go`, поскольку она использует более общие функции. Однако это всего лишь дело личных предпочтений.

В следующем разделе показано, как программировать TCP-серверы.

Разработка TCP-сервера

В этом разделе представлены два способа разработки TCP-сервера, который умеет взаимодействовать с TCP-клиентами точно так же, как мы это делали с TCP-клиентом.

Разработка TCP-сервера с помощью `net.Listen()`

TCP-сервер, представленный в этом разделе и использующий `net.Listen()`, возвращает клиенту текущую дату и время в одном сетевом пакете. На практике это означает, что, приняв клиентское подключение, сервер получает время и дату из операционной системы и отправляет эти данные клиенту. Функция `net.Listen()` прослушивает соединения, в то время как метод `net.Accept()`

ожидает следующего подключения и возвращает общую переменную `Conn` с информацией о клиенте.

Код `tcpS.go` выглядит следующим образом:

```
package main

import (
    "bufio"
    "fmt"
    "net"
    "os"
    "strings"
    "time"
)

func main() {
    arguments := os.Args
    if len(arguments) == 1 {
        fmt.Println("Please provide port number")
        return
    }

    PORT := ":" + arguments[1]
    l, err := net.Listen("tcp", PORT)
    if err != nil {
        fmt.Println(err)
        return
    }
    defer l.Close()
```

TCP-сервер должен знать номер порта, который он собирается использовать, — он задается в качестве аргумента командной строки.

```
    c, err := l.Accept()
    if err != nil {
        fmt.Println(err)
        return
    }
```

Функция `net.Listen()` прослушивает соединения и выступает тем, что делает эту конкретную программу серверным процессом. Если второй параметр функции содержит номер порта без IP-адреса или имени хоста, то она прослушивает все доступные IP-адреса локальной системы, как в данном случае.

```
c, err := l.Accept()
if err != nil {
    fmt.Println(err)
    return
}
```

Мы просто вызываем `Accept()` и ждем подключения клиента — `Accept()` блокируется до тех пор, пока не придет соединение. В этом конкретном TCP-сервере есть кое-что необычное: он может обслуживать только первого TCP-клиента, который собирается подключиться к нему, поскольку вызов `Accept()` находится

вне цикла `for` и поэтому вызывается лишь единожды. Каждый отдельный клиент должен быть указан с помощью другого вызова `Accept()`.

Исправление этой ситуации оставлено в качестве упражнения для читателя.

```

for {
    netData, err := bufio.NewReader(c).ReadString('\n')
    if err != nil {
        fmt.Println(err)
        return
    }
    if strings.TrimSpace(string(netData)) == "STOP" {
        fmt.Println("Exiting TCP server!")
        return
    }

    fmt.Print("-> ", string(netData))
    t := time.Now()
    myTime := t.Format(time.RFC3339) + "\n"
    c.Write([]byte(myTime))
}
}

```

Данный бесконечный цикл `for` продолжает взаимодействовать с одним и тем же TCP-клиентом до тех пор, пока от него не поступит слово `STOP`. Как это было и в случае с TCP-клиентами, `bufio.NewReader()` используется для чтения данных из сетевого подключения, тогда как `Write()` служит для отправки данных TCP-клиенту.

При выполнении `tcpS.go` и взаимодействии с TCP-клиентом мы получаем такой вывод:

```
$ go run tcpS.go 1234
-> Hello!
-> Have to leave now!
EOF
```

Соединение с сервером автоматически завершилось подключением к клиенту, поскольку цикл `for` завершился, когда `bufio.NewReader(c).ReadString('\n')` больше нечего было читать. Клиентом выступала `nc(1)`, которая произвела следующий вывод:

```
$ nc localhost 1234
Hello!
2021-04-12T08:53:32+03:00
Have to leave now!
2021-04-12T08:53:51+03:00
```

Чтобы выйти из `nc(1)`, нужно нажать сочетание клавиш `Ctrl+D`, что в UNIX означает `EOF` (Конец файла).

Итак, теперь вы знаете, как разработать TCP-сервер в Go. Как и в случае с TCP-клиентом, существует альтернативный способ разработки TCP-сервера, который представлен в следующем подразделе.

Разработка TCP-сервера, использующего `net.ListenTCP()`

На сей раз эта альтернативная версия TCP-сервера реализует сервис echo. Проще говоря, TCP-сервер отправляет обратно клиенту данные, которые были получены клиентом.

Код `otherTCPserver.go` выглядит следующим образом:

```
package main

import (
    "fmt"
    "net"
    "os"
    "strings"
)

func main() {
    arguments := os.Args
    if len(arguments) == 1 {
        fmt.Println("Please provide a port number!")
        return
    }

    SERVER := "localhost" + ":" + arguments[1]
    s, err := net.ResolveTCPAddr("tcp", SERVER)
    if err != nil {
        fmt.Println(err)
        return
    }
}
```

Код, представленный выше, получает значение номера порта TCP в качестве аргумента командной строки, который используется в `net.ResolveTCPAddr()`, — это требуется для определения номера порта TCP, который будет прослушиваться TCP-сервером.

Эта функция работает только с TCP, отсюда и ее название.

```
l, err := net.ListenTCP("tcp", s)
if err != nil {
    fmt.Println(err)
    return
}
```

Аналогично, `net.ListenTCP()` работает только с TCP и делает эту программу TCP-сервером, готовым принимать входящие соединения.

```
buffer := make([]byte, 1024)
conn, err := l.Accept()
if err != nil {
    fmt.Println(err)
    return
}
```

Как и прежде, из-за места, где вызывается `Accept()`, эта конкретная реализация может работать только с одним клиентом. Такой подход используется по соображениям простоты. Параллельный TCP-сервер, который разрабатывается далее в этой главе, помещает вызов `Accept()` внутри бесконечного цикла `for`.

```
for {
    n, err := conn.Read(buffer)
    if err != nil {
        fmt.Println(err)
        return
    }

    if strings.TrimSpace(string(buffer[0:n])) == "STOP" {
        fmt.Println("Exiting TCP server!")
        conn.Close()
        return
    }
}
```

Нам необходимо использовать `strings.TrimSpace()` для удаления всех символов пробела из ввода и сравнения результата со словом `STOP`, которое имеет особое значение в этой реализации. Когда ключевое слово `STOP` получено от клиента, сервер закрывает соединение с помощью метода `Close()`.

```
fmt.Print("> ", string(buffer[0:n-1]), "\n")
_, err = conn.Write(buffer)
if err != nil {
    fmt.Println(err)
    return
}
}
```

Весь вышеприведенный код предназначен для взаимодействия с TCP-клиентом до тех пор, пока клиент не решит закрыть соединение.

При выполнении `otherTCPserver.go` и взаимодействии с TCP-клиентом мы получаем такой вывод:

```
$ go run otherTCPserver.go 1234
> Hello from the client!
Exiting TCP server!
```

Первая строка, начинающаяся с >, — это сообщение клиента, тогда как вторая строка — это вывод сервера при получении от клиента сообщения STOP. Таким образом, TCP-сервер обрабатывает клиентские запросы в соответствии с планом и завершает работу, когда получает сообщение STOP, что является ожидаемым поведением.

Следующий раздел посвящен разработке UDP-клиентов.

Разработка UDP-клиента

В этом разделе вы узнаете, как разработать UDP-клиент, который может взаимодействовать с сервисами UDP. Код `udpC.go` выглядит следующим образом:

```
package main

import (
    "bufio"
    "fmt"
    "net"
    "os"
    "strings"
)

func main() {
    arguments := os.Args
    if len(arguments) == 1 {
        fmt.Println("Please provide a host:port string")
        return
    }
    CONNECT := arguments[1]
```

Так мы получаем сведения о сервере UDP от пользователя.

```
s, err := net.ResolveUDPAddr("udp4", CONNECT)
c, err := net.DialUDP("udp4", nil, s)
```

В этих двух строках указано, что мы используем UDP и хотим подключиться к серверу UDP, который указан возвращаемым значением `net.ResolveUDPAddr()`.

Фактическое соединение инициируется с помощью `net.DialUDP()`.

```
if err != nil {
    fmt.Println(err)
    return
}

fmt.Printf("The UDP server is %s\n", c.RemoteAddr().String())
defer c.Close()
```

Эта часть программы получает сведения о сервере UDP, вызывая метод `RemoteAddr()`.

```
for {
    reader := bufio.NewReader(os.Stdin)
    fmt.Print("> ")
    text, _ := reader.ReadString('\n')
    data := []byte(text + "\n")
    _, err = c.Write(data)
```

Полученные от пользователя с помощью `bufio.NewReader(os.Stdin)` данные записываются на UDP-сервер методом `Write()`.

```
if strings.TrimSpace(string(data)) == "STOP" {
    fmt.Println("Exiting UDP client!")
    return
}
```

Если полученный от пользователя ввод является ключевым словом `STOP`, то соединение прерывается.

```
if err != nil {
    fmt.Println(err)
    return
}

buffer := make([]byte, 1024)
n, _, err := c.ReadFromUDP(buffer)
```

Данныечитываются из UDP-соединения с помощью метода `ReadFromUDP()`.

```
if err != nil {
    fmt.Println(err)
    return
}
fmt.Printf("Reply: %s\n", string(buffer[0:n]))
```

Цикл `for` будет продолжаться вечно, пока в качестве входных данных не будет получено ключевое слово `STOP` или же программа не завершится каким-либо другим способом.

Работа с `udpC.go` происходит следующим образом (клиентская сторона реализована с помощью `nc(1)`):

```
$ go run udpC.go localhost:1234
The UDP server is 127.0.0.1:1234
```

127.0.0.1:1234 — это значение `c.RemoteAddr().String()`, которое показывает сведения о сервере UDP, к которому мы подключились.

```
>> Hello!  
Reply: Hi from the server
```

Наш клиент прислал `Hello!` на UDP-сервер и получил в ответ `Hi from server`.

```
>> Have to leave now :)  
Reply: OK - bye from nc -l -u 1234
```

Наш клиент отправил `Have to leave now :)` на UDP-сервер и получил ответ `OK - bye from nc -l -u 1234`.

```
>> STOP  
Exiting UDP client!
```

Наконец, после отправки ключевого слова `STOP` на сервер клиент выводит `Exiting UDP client!` и завершается — сообщение определено в Go-коде и может быть любым.

Следующий раздел посвящен программированию UDP-сервера.

Разработка UDP-сервера

В этом разделе показано, как разработать UDP-сервер, который генерирует и возвращает своим клиентам случайные числа. Код для UDP-сервера (`udpS.go`) выглядит следующим образом:

```
package main

import (
    "fmt"
    "math/rand"
    "net"
    "os"
    "strconv"
    "strings"
    "time"
)

func random(min, max int) int {
    return rand.Intn(max-min) + min
}

func main() {
    arguments := os.Args
```

```

if len(arguments) == 1 {
    fmt.Println("Please provide a port number!")
    return
}
PORT := ":" + arguments[1]

```

Номер порта UDP, который сервер собирается прослушивать, предоставляется в качестве аргумента командной строки.

```

s, err := net.ResolveUDPAddr("udp4", PORT)
if err != nil {
    fmt.Println(err)
    return
}

```

Функция `net.ResolveUDPAddr()` создает конечную точку UDP, которая будет использоваться для создания сервера.

```

connection, err := net.ListenUDP("udp4", s)
if err != nil {
    fmt.Println(err)
    return
}

```

Вызов функции `net.listenUDP("udp4", s)` делает этот процесс сервером для протокола udp4, используя подробности, указанные во втором параметре.

```

defer connection.Close()
buffer := make([]byte, 1024)

```

Переменная `buffer` хранит байтовый срез и используется для считывания данных из UDP-соединения.

```

rand.Seed(time.Now().Unix())

for {
    n, addr, err := connection.ReadFromUDP(buffer)
    fmt.Print("-> ", string(buffer[0:n-1]))
}

```

Методы `ReadFromUDP()` и `WriteToUDP()` используются для чтения данных из UDP-соединения и записи их в него соответственно. Кроме того, благодаря принципу работы UDP UDP-сервер может обслуживать несколько клиентов.

```

if strings.TrimSpace(string(buffer[0:n])) == "STOP" {
    fmt.Println("Exiting UDP server!")
    return
}

```

UDP-сервер завершает работу, когда любой из клиентов отправляет сообщение `STOP`. Если этого не произойдет, то цикл `for` будет выполняться вечно.

```

data := []byte(strconv.Itoa(random(1, 1001)))
fmt.Printf("data: %s\n", string(data))

```

Байтовый срез хранится в переменной `data` и используется для передачи желаемых данных клиенту.

```
    _, err = connection.WriteToUDP(data, addr)
    if err != nil
        fmt.Println(err)
        return
    }
}
}
```

Работать с программой `udpS.go` очень просто:

```
$ go run udpS.go 1234
-> Hello from client!
data: 395
```

Строки, начинающиеся с `->`, показывают данные, поступающие от клиента. Строки, начинающиеся с `data`, показывают случайные числа, генерированные UDP-сервером, — в данном случае 395.

```
-> Going to terminate the connection now.
data: 499
```

Предыдущие две строки показывают еще одно взаимодействие с UDP-клиентом.

```
-> STOP
Exiting UDP server!
```

Как только UDP-сервер получает от клиента ключевое слово `STOP`, он закрывает соединение и завершает работу.

На стороне клиента, использующего `udpC.go`, у нас имеется следующее взаимодействие:

```
$ go run udpC.go localhost:1234
The UDP server is 127.0.0.1:1234
>> Hello from client!
Reply: 395
```

Клиент передает сообщение `Hello from client!` на сервер и получает 395.

```
>> Going to terminate the connection now.
Reply: 499
```

Клиент отправляет `Going to terminate the connection now.` на сервер и получает случайное число 499.

```
>> STOP
Exiting UDP client!
```

Когда клиент получает STOP в качестве пользовательского ввода, он завершает UDP-соединение и заканчивает работу.

В следующем разделе показано, как разработать параллельный TCP-сервер, который использует горутины для обслуживания своих клиентов.

Разработка параллельных TCP-серверов

В этом разделе описывается шаблон для разработки параллельных TCP-серверов, которые представляют собой серверы, использующие отдельные горутины для обслуживания своих клиентов после успешного вызова `Accept()`. Следовательно, такие серверы умеют обслуживать несколько TCP-клиентов одновременно. Именно так реализуются промышленные серверы в реальном мире.

Код `constTCP.go` выглядит следующим образом:

```
package main

import (
    "bufio"
    "fmt"
    "net"
    "os"
    "strconv"
    "strings"
)

var count = 0

func handleConnection(c net.Conn) {
    fmt.Println(".")

    for {
        netData, err := bufio.NewReader(c).ReadString('\n')
        if err != nil {
            fmt.Println(err)
            return
        }

        temp := strings.TrimSpace(string(netData))
        if temp == "STOP" {
            break
        }
        fmt.Println(temp)
    }
}
```

Этот оператор необязателен — он просто информирует нас о том, что был подключен новый клиент.

```
for {
    netData, err := bufio.NewReader(c).ReadString('\n')
    if err != nil {
        fmt.Println(err)
        return
    }

    temp := strings.TrimSpace(string(netData))
    if temp == "STOP" {
        break
    }
    fmt.Println(temp)
}
```

```

        counter := "Client number: " + strconv.Itoa(count) + "\n"
        c.Write([]byte(string(counter)))
    }
}

```

Цикл `for` гарантирует, что `handleConnection()` не завершится автоматически. Повторюсь, ключевое слово `STOP` останавливает текущее клиентское соединение, однако серверный процесс, а также все другие активные клиентские соединения будут продолжать выполняться.

```

    c.Close()
}

```

Это конец функции, которая выполняется как горутина для обслуживания клиентов. Все, что вам нужно для того, чтобы обслуживать клиента, — это параметр `net.Conn` с данными клиента. После считывания клиентских данных сервер отправляет клиенту сообщение с указанием номера клиента, который обслуживается на данный момент.

```

func main() {
    arguments := os.Args
    if len(arguments) == 1 {
        fmt.Println("Please provide a port number!")
        return
    }

    PORT := ":" + arguments[1]
    l, err := net.Listen("tcp4", PORT)
    if err != nil {
        fmt.Println(err)
        return
    }
    defer l.Close()

    for {
        c, err := l.Accept()
        if err != nil {
            fmt.Println(err)
            return
        }
        go handleConnection(c)
        count++
    }
}

```

Каждый раз, когда новый клиент подключается к серверу, переменная `count` увеличивается. Каждый TCP-клиент обслуживается отдельной горутиной, которая выполняет функцию `handleConnection()`. Это освобождает серверный процесс и позволяет ему принимать новые подключения. Проще говоря, пока обслуживается несколько TCP-клиентов, TCP-сервер может свободно взаимодействовать с большим количеством TCP-клиентов. Как и прежде, новые TCP-клиенты подключаются с помощью функции `Accept()`.

Работа с программой `concTCP.go` приводит к следующему выводу:

```
$ go run concTCP.go 1234
.Hello
.Hi from nc localhost 1234
```

Первая строка вывода поступает от первого TCP-клиента, тогда как вторая — уже от второго. Это означает, что параллельный TCP-сервер работает должным образом. Так что если вам нужна возможность обслуживать несколько TCP-клиентов в своих TCP-сервисах, вы можете использовать технику и код, представленные в качестве шаблона для разработки ваших TCP-серверов.

В следующем разделе показано, как работать с доменными сокетами UNIX, которые исключительно быстры при взаимодействии только на локальном компьютере.

Работа с доменными сокетами UNIX

Доменный сокет UNIX, или сокет межпроцессной связи (Inter-Process Communication, IPC), — это конечная точка передачи данных, которая позволяет обмениваться данными процессам, запущенным *на одном компьютере*. Вы можете спросить, а зачем вообще использовать доменные сокеты UNIX вместо соединений TCP/IP для процессов, которые обмениваются данными на одном компьютере? Во-первых, потому что доменные сокеты UNIX работают быстрее, чем соединения TCP/IP, а во-вторых, потому что доменные сокеты UNIX требуют меньше ресурсов. Таким образом, вы можете использовать доменные сокеты UNIX, когда и клиенты, и сервер расположены на одном компьютере.

Сервер на сокетах домена UNIX

В этом подразделе показано, как разработать сервер на сокетах домена UNIX. Хотя нам и не придется иметь дело с TCP-портами и сетевыми подключениями, приведенный код очень похож на код TCP-сервера в файлах `tcpS.go` и `concTCP.go`. Представленный сервер реализует сервис echo.

Исходный код программы `SocketServer.go` выглядит следующим образом:

```
package main

import (
    "fmt"
    "net"
    "os"
)

func echo(c net.Conn) {
```

Функция `echo()` применяется для обслуживания клиентских запросов, отсюда и использование параметра `net.Conn`, содержащего сведения о клиенте:

```
for {
    buf := make([]byte, 128)
    n, err := c.Read(buf)
    if err != nil {
        fmt.Println("Read:", err)
        return
    }
}
```

Мы считываем данные из сокет-соединения с помощью функции `Read()` внутри цикла `for`.

```
data := buf[0:n]
fmt.Println("Server got: ", string(data))
_, err = c.Write(data)
if err != nil {
    fmt.Println("Write:", err)
    return
}
}
```

В этой второй части `echo()` мы возвращаем клиенту данные, которые он отправил нам. Нотация `buf[0:n]` гарантирует, что мы отправим обратно тот же объем данных, который был прочитан, даже если размер буфера больше.

Данная функция обслуживает все клиентские подключения. Как вы скоро увидите, она выполняется как горутина, и это является основной причиной того, что она не возвращает никаких значений.

Вы не сможете определить, обслуживает ли эта функция соединения TCP/IP или же соединения сокетов домена UNIX; в основном это происходит потому, что UNIX обрабатывает все соединения как файлы.

```
func main() {
    if len(os.Args) == 1 {
        fmt.Println("Need socket path")
        return
    }
    socketPath := os.Args[1]
```

Это та часть, где мы указываем файл сокета, который будет использоваться сервером и его клиентами. В нашем случае путь к файлу сокета задается в качестве аргумента командной строки.

```
_, err := os.Stat(socketPath)
if err == nil {
    fmt.Println("Deleting existing", socketPath)
    err := os.Remove(socketPath)
    if err != nil {
```

```

        fmt.Println(err)
        return
    }
}

```

Если файл сокета уже существует, то вы должны *удалить его* до продолжения работы программы — `net.Listen()` снова создает этот файл.

```

l, err := net.Listen("unix", socketPath)
if err != nil {
    fmt.Println("listen error:", err)
    return
}

```

Что делает все это сервером на сокетах домена UNIX, так это использование `net.Listen()` с параметром "unix". В этом случае нам нужно предоставить `net.Listen()` с указанием пути к файлу сокета.

```

for {
    fd, err := l.Accept()
    if err != nil {
        fmt.Println("Accept error:", err)
        return
    }
    go echo(fd)
}
}

```

Каждое клиентское соединение обрабатывается горутиной — в этом смысле это параллельный сервер на сокетах домена UNIX, который умеет работать с несколькими клиентами. Следовательно, если вам нужно обслуживать тысячи клиентов доменных сокетов на реальном сервере, то вы на верном пути.

В следующем подразделе мы увидим работу сервера, поскольку он будет взаимодействовать с клиентом сокета домена UNIX, который мы собираемся создать.

Клиент сокета домена UNIX

В этом подразделе показана реализация клиента доменных сокетов UNIX, которая может использоваться для связи с сервером доменных сокетов (например, с разработанным в предыдущем подразделе). Соответствующий код находится в файле `SocketClient.go`:

```

package main

import (

```

```

    "bufio"
    "fmt"
    "net"
    "os"
    "strings"
    "time"
)

func main() {
    if len(os.Args) == 1 {
        fmt.Println("Need socket path")
        return
    }
    socketPath := os.Args[1]
}

```

Это та часть, где мы получаем от пользователя файл сокета для использования, — данный файл уже должен существовать и обрабатываться сервером сокетов домена UNIX.

```

c, err := net.Dial("unix", socketPath)
if err != nil {
    fmt.Println(err)
    return
}
defer c.Close()

```

Функция `net.Dial()` используется для подключения к сокету.

```

for {
    reader := bufio.NewReader(os.Stdin)
    fmt.Print("> ")
    text, _ := reader.ReadString('\n')

    _, err = c.Write([]byte(text))
}

```

Здесь мы преобразуем пользовательский ввод в байтовый срез и отправляем его на сервер с помощью `Write()`.

```

if err != nil {
    fmt.Println("Write:", err)
    break
}

buf := make([]byte, 256)

n, err := c.Read(buf[:])
if err != nil {
    fmt.Println(err, n)
}

```

```

        return
    }
    fmt.Println("Read:", string(buf[0:n]))
}

```

Используя нотацию `buf[0:n]`, оператор `fmt.Println()` выводит столько символов из среза `buf`, сколько символов было считано методом `Read()`.

```

if strings.TrimSpace(string(text)) == "STOP" {
    fmt.Println("Exiting UNIX domain socket client!")
    return
}

```

Если в качестве входных данных приходит слово `STOP`, то клиент возвращается и, следовательно, закрывает соединение с сервером. Вообще говоря, всегда хорошо иметь способ изящно завершить такую утилиту.

```

    time.Sleep(5 * time.Second)
}
}

```

Вызов `time.Sleep()` используется для задержки цикла `for` и эмуляции работы реальной программы.

Работа как с `SocketServer.go`, так и с `SocketClient.go` при условии, что сервер выполняется первым, приводит к следующим выходным данным:

```
$ go run socketServer.go /tmp/packt.socket
Server got: Hello!
Server got: STOP
Read: EOF
```

Несмотря на то что клиентское соединение завершилось, сервер продолжает работать и ожидает новых клиентских запросов.

На стороне клиента происходит следующее:

```
$ go run socketClient.go /tmp/packt.socket
>> Hello!
Read: Hello!
>> STOP
Read: STOP
Exiting UNIX domain socket client!
```

В предыдущих двух разделах мы узнали, как создавать клиенты и серверы на сокетах домена UNIX, которые более быстрые, чем серверы TCP/IP, но работают только на одном компьютере.

В следующих разделах поговорим о протоколе WebSocket.

Создание сервера WebSocket

Протокол WebSocket — это протокол компьютерной связи, который обеспечивает *полнодуплексные* (в двух направлениях одновременно) каналы связи по одному TCP-соединению. Протокол WebSocket определен в RFC 6455 (<https://tools.ietf.org/html/rfc6455>) и использует `ws://` и `wss://` вместо `http://` и `https://` соответственно. Следовательно, клиент должен начать подключение к WebSocket, используя URL, начинающийся с `ws://`.

В этом разделе мы собираемся разработать небольшой, но полностью функциональный сервер WebSocket, использовав для этого модуль `gorilla/websocket` (<https://github.com/gorilla/websocket>). Сервер реализует *сервис Echo*; это значит, он автоматически возвращает свои входные данные клиенту.

В пакете `golang.org/x/net/websocket` содержится другой способ разработки клиентов и серверов WebSocket. Однако, согласно его документации, в `golang.org/x/net/websocket` отсутствуют некоторые функции и рекомендуется использовать <https://godoc.org/github.com/gorilla/websocket> (используется здесь) или же <https://godoc.org/nhooyr.io/websocket>.

Преимущества протокола WebSocket описаны ниже.

- Соединение WebSocket — это полнодуплексный двунаправленный канал связи. Это значит, серверу не нужно ждать операции чтения от клиента, чтобы отправить данные клиенту, и наоборот.
- Соединения WebSocket — это необработанные TCP-сокеты. Это значит, они не влекут накладных расходов, необходимых для установления HTTP-соединения.
- Соединения WebSocket также могут использоваться для отправки HTTP-данных. Однако обычные HTTP-соединения не могут работать как соединения WebSocket.
- Соединения WebSocket работают до тех пор, пока не будут прерваны, поэтому нет необходимости постоянно открывать их заново.
- Соединения WebSocket можно использовать для веб-приложений реального времени.
- Данные могут быть отправлены с сервера клиенту в любое время, даже если клиент их не запрашивал.
- WebSocket является частью спецификации HTML5, а это значит, что он поддерживается всеми современными браузерами.

Прежде чем обратиться к реализации сервера, будет нeliшним узнать, что метод `websocket.Upgrader` пакета `gorilla/websocket` обновляет подключение

HTTP-сервера к протоколу WebSocket и позволяет определять параметры обновления. После этого HTTP-соединение становится подключением WebSocket, и это значит, вам более не разрешается выполнять операторы, которые работают с протоколом HTTP.

В следующем подразделе показана реализация сервера.

Реализация сервера

В этом подразделе представлена реализация сервера WebSocket, который реализует сервис echo, очень удобный при тестировании сетевых подключений.

Репозиторий GitHub, используемый для хранения кода, можно найти по адресу <https://github.com/mactsouk/ws>. Если вы решите опробовать материал этого подраздела, то вам следует скачать данный репозиторий и поместить его в `~/go/src`. В моем случае он был помещен внутрь `~/go/src/github.com/mactsouk`, в папке `ws`.



Репозиторий GitHub содержит файл Dockerfile для создания образа Docker из исходного файла сервера WebSocket.

Реализацию сервера WebSocket можно найти в `ws.go`, где содержится следующий код:

```
package main

import (
    "fmt"
    "log"
    "net/http"
    "os"
    "time"

    "github.com/gorilla/websocket"
)
```

Это внешний пакет, используемый для работы с протоколом WebSocket.

```
var PORT = ":1234"

var upgrader = websocket.Upgrader{
    ReadBufferSize: 1024,
    WriteBufferSize: 1024,
    CheckOrigin: func(r *http.Request) bool {
        return true
    },
}
```

Именно здесь определяются параметры `websocket.Upgrader`. Мы используем их уже очень скоро.

```
func rootHandler(w http.ResponseWriter, r *http.Request) {
    fmt.Fprintf(w, "Welcome!\n")
    fmt.Fprintf(w, "Please use /ws for WebSocket!")
}
```

Это обычная функция HTTP-обработчика.

```
func wsHandler(w http.ResponseWriter, r *http.Request) {
    log.Println("Connection from:", r.Host)

    ws, err := upgrader.Upgrade(w, r, nil)
    if err != nil {
        log.Println("upgrader.Upgrade:", err)
        return
    }
    defer ws.Close()
```

Серверное приложение WebSocket вызывает метод `Upgrader.Upgrade` для того, чтобы получить подключение к WebSocket от обработчика HTTP-запроса. После успешного вызова `Upgrader.Upgrade` сервер начинает работать с подключением WebSocket и клиентом WebSocket.

```
for {
    mt, message, err := ws.ReadMessage()
    if err != nil {
        log.Println("From", r.Host, "read", err)
        break
    }
    log.Print("Received: ", string(message))
    err = ws.WriteMessage(mt, message)
    if err != nil {
        log.Println("WriteMessage:", err)
        break
    }
}
```

Цикл `for` в `wsHandler()` обрабатывает все входящие сообщения для `/ws` — вы можете использовать любую желаемую технику. Кроме того, в представленной реализации *только клиенту* разрешено закрывать существующее подключение к WebSocket, если нет проблем с сетью или серверный процесс не остановлен.

Наконец, помните, что в соединении с WebSocket вы не можете использовать операторы `fmt.Fprintf()` для отправки данных клиенту WebSocket. Если вы используете любой другой вызов, который может реализовать ту же функциональность, то соединение с WebSocket завершится сбоем, а вы не сможете

отправлять или получать данные. Таким образом, отправлять и получать данные при соединении WebSocket, реализованном с помощью `gorilla/websocket`, можно только с помощью вызовов `WriteMessage()` и `ReadMessage()` соответственно. Конечно, вы всегда можете реализовать желаемую функциональность самостоятельно, работая с необработанными сетевыми данными, но реализация этого выходит за рамки данной книги.

```
func main() {
    arguments := os.Args
    if len(arguments) != 1 {
        PORT = ":" + arguments[1]
    }
}
```

Если аргумент командной строки отсутствует, то используем номер порта по умолчанию, хранящийся в `PORT`.

```
mux := http.NewServeMux()
s := &http.Server{
    Addr:     PORT,
    Handler:  mux,
    IdleTimeout: 10 * time.Second,
    ReadTimeout: time.Second,
    WriteTimeout: time.Second,
}
```

Это сведения о HTTP-сервере, который также обрабатывает подключения к WebSocket.

```
mux.Handle("/", http.HandlerFunc(rootHandler))
mux.Handle("/ws", http.HandlerFunc(wsHandler))
```

Конечная точка, используемая для WebSocket, может быть любой, и в данном случае это `/ws`. Кроме того, у вас может быть несколько конечных точек, работающих с протоколом WebSocket.

```
log.Println("Listening to TCP Port", PORT)
err := s.ListenAndServe()
if err != nil {
    log.Println(err)
    return
}
}
```

Представленный выше код использует `log.Println()` вместо `fmt.Println()` для вывода сообщений. Поскольку это серверный процесс, то использование `log.Println()` — гораздо лучший выбор, нежели `fmt.Println()`, поскольку информация о протоколировании отправляется в файлы, которые можно просмотреть позже. Однако во время разработки вы можете предпочесть вызов `fmt.Println()` и избегать записи в ваши файлы журналов, так как сможете сразу видеть свои данные на экране, не заглядывая куда-то еще.

Реализация сервера короткая, но полностью функциональная. Единственным наиболее важным вызовом в коде выступает `Upgrader.Upgrade`, поскольку это как раз то, что обновляет HTTP-соединение до соединения WebSocket.

Для получения и запуска кода из GitHub требуются следующие шаги (большинство связано с инициализацией модуля и скачиванием необходимых пакетов):

```
$ cd ~/go/src/github.com/mactsouk/
$ git clone https://github.com/mactsouk/ws.git
$ cd ws
$ go mod init
$ go mod tidy
$ go mod download
$ go run ws.go
```

Чтобы протестировать сервер, у вас должен быть клиент. Поскольку мы еще не разрабатывали собственный клиент, мы протестируем сервер WebSocket, используя два других средства.

Использование websocat

`websocat` — это утилита командной строки, которая может помочь протестировать подключения к WebSocket. Однако поскольку `websocat` не устанавливается по умолчанию, необходимо установить ее на свой компьютер с помощью выбранного вами менеджера пакетов. Вы можете использовать ее следующим образом при условии, что по нужному адресу есть сервер WebSocket:

```
$ websocat ws://localhost:1234/ws
Hello from websocat!
```

Это то, что мы вводим и отправляем на сервер.

```
Hello from websocat!
```

Это то, что мы получаем обратно с сервера WebSocket, который реализует `echo`. Разные серверы WebSocket реализуют разную функциональность.

```
Bye!
```

Опять же, предыдущая строка — это пользовательский ввод, предоставленный `websocat`.

```
Bye!
```

И последняя строка — это данные, отправленные обратно с сервера. Соединение было закрыто путем нажатия сочетания клавиш `Ctrl+D` на клиенте `websocat`.

Если вам нужен более подробный вывод из `websocat`, то можете выполнить его с флагом `-v`:

```
$ websocat -v ws://localhost:1234/ws
[INFO websocat::lints] Auto-inserting the line mode
[INFO websocat::stdio_threaded_peer] get_stdio_peer (threaded)
[INFO websocat::ws_client_peer] get_ws_client_peer
[INFO websocat::ws_client_peer] Connected to ws
Hello from websocat!
Hello from websocat!
Bye!
Bye!
[INFO websocat::sessionserve] Forward finished
[INFO websocat::ws_peer] Received WebSocket close message
[INFO websocat::sessionserve] Reverse finished
[INFO websocat::sessionserve] Both directions finished
```

В обоих случаях выходные данные с нашего сервера WebSocket должны быть похожи на эти:

```
$ go run ws.go
2021/04/10 20:54:30 Listening to TCP Port :1234
2021/04/10 20:54:42 Connection from: localhost:1234
2021/04/10 20:54:57 Received: Hello from websocat!
2021/04/10 20:55:03 Received: Bye!
2021/04/10 20:55:03 From localhost:1234 read websocket: close 1005 (no
status)
```

Далее показано, как протестировать сервер WebSocket, используя HTML и JavaScript.

Использование JavaScript

Второй способ тестирования сервера WebSocket — это создание веб-страницы с кое-каким HTML и JavaScript-кодом. Такой метод дает больше контроля над происходящим, но требует и большего объема кода, а также знакомства с HTML и JavaScript.

HTML-страница с кодом JavaScript, который заставляет ее работать в качестве клиента WebSocket, выглядит следующим образом:

```
<!DOCTYPE html>
<meta charset="utf-8">

<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initialscale=1.0" />
    <meta http-equiv="X-UA-Compatible" content="ie=edge" />
```

```
<title>Testing a WebSocket Server</title>
</head>
<body>
    <h2>Hello There!</h2>

    <script>
        let ws = new WebSocket("ws://localhost:1234/ws");
```

Это единственная наиболее важная инструкция JavaScript, поскольку именно здесь вы указываете адрес сервера WebSocket, номер порта и конечную точку, к которой хотите подключиться.

```
console.log("Trying to connect to server.");

ws.onopen = () => {
    console.log("Connected!");
    ws.send("Hello From the Client!")
};
```

Событие `ws.onopen` используется для проверки того, что соединение с WebSocket открыто, тогда как метод `send()` — для отправки сообщений на сервер WebSocket.

```
ws.onmessage = function(event) {
    console.log(`[message] Data received from server: ${event.
data}`);
    ws.close(1000, "Work complete");
};
```

Событие `onmessage` срабатывает каждый раз, когда сервер WebSocket отправляет новое сообщение. Однако в нашем случае соединение закрывается, как только получено первое же сообщение от сервера.

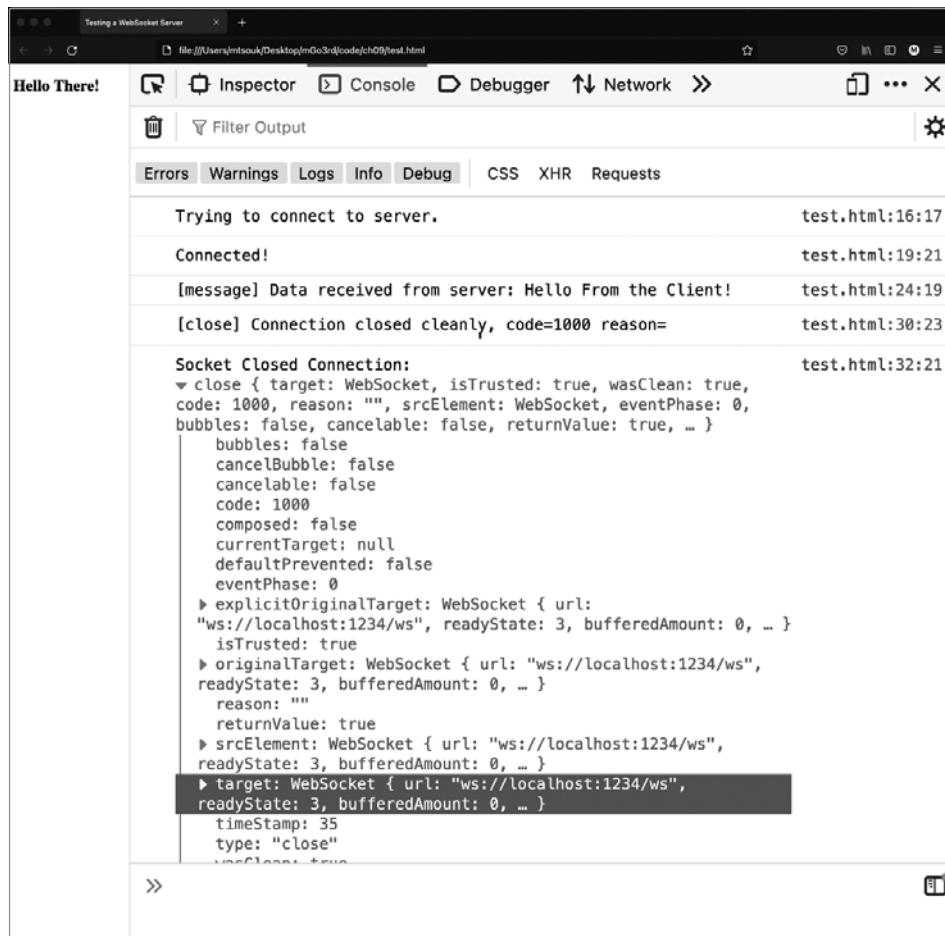
Наконец, метод `close()` JavaScript используется для закрытия соединения с WebSocket — в нашем случае вызов `close()` включен в событие `onmessage`. Вызов `close()` запускает событие `onclose`, которое содержит следующий код:

```
ws.onclose = event => {
    if (event.wasClean) {
        console.log(`[close] Connection closed cleanly, code=${event.
code} reason=${event.reason}`);
    }
    console.log("Socket Closed Connection: ", event);
};

ws.onerror = error => {
    console.log("Socket Error: ", error);
};

</script>
</body>
</html>
```

Вы можете увидеть вывод кода JavaScript, открыв консоль JavaScript в браузере, которым в нашем случае является Google Chrome. На рис. 9.1 показан такой результат.



The screenshot shows the Google Chrome DevTools interface with the 'Console' tab selected. The title bar says 'Testing a WebSocket Server'. The console output is as follows:

```
Trying to connect to server. test.html:16:17
Connected! test.html:19:21
[message] Data received from server: Hello From the Client! test.html:24:19
[close] Connection closed cleanly, code=1000 reason=
test.html:30:23
test.html:32:21
Socket Closed Connection:
  ▼ close { target: WebSocket, isTrusted: true, wasClean: true,
    code: 1000, reason: "", srcElement: WebSocket, eventPhase: 0,
    bubbles: false, cancelable: false, returnValue: true, ... }
    bubbles: false
    cancelBubble: false
    cancelable: false
    code: 1000
    composed: false
    currentTarget: null
    defaultPrevented: false
    eventPhase: 0
    explicitOriginalTarget: WebSocket { url:
      "ws://localhost:1234/ws", readyState: 3, bufferedAmount: 0, ... }
    isTrusted: true
    originalTarget: WebSocket { url: "ws://localhost:1234/ws",
      readyState: 3, bufferedAmount: 0, ... }
    reason: ""
    returnValue: true
    srcElement: WebSocket { url: "ws://localhost:1234/ws",
      readyState: 3, bufferedAmount: 0, ... }
    target: WebSocket { url: "ws://localhost:1234/ws",
      readyState: 3, bufferedAmount: 0, ... }
    timestamp: 35
    type: "close"
```

Рис. 9.1. Взаимодействие с сервером WebSocket с помощью JavaScript

При взаимодействии с WebSocket, определенном в `test.html`, сервер WebSocket сгенерировал следующий вывод в командной строке:

```
2021/04/10 21:43:22 Connection from: localhost:1234
2021/04/10 21:43:22 Received: Hello From the Client!
2021/04/10 21:43:22 From localhost:1234 read websocket: close 1000
(normal): Work complete
```

Оба способа проверяют, что сервер WebSocket работает должным образом: клиент может подключиться к серверу, тот отправляет полученные клиентом данные, и клиент успешно закрывает соединение с сервером. Итак, пришло время разработать наш собственный клиент WebSocket на Go.

Создание клиента WebSocket

В этом разделе показано, как запрограммировать клиент WebSocket на Go. Клиент считывает пользовательские данные, отправляет их на сервер, после чего считывает ответ сервера. Каталог `client` в <https://github.com/mactsouk/ws> содержит реализацию клиента WebSocket — я посчитал более удобным поместить обе реализации в один и тот же репозиторий.

Как и в случае с сервером WebSocket, пакет `gorilla/websocket` поможет нам в разработке клиента WebSocket.



Мы изучим `gorilla` в следующей главе при работе с сервисами RESTful.

Код `./client/client.go` выглядит следующим образом:

```
package main

import (
    "bufio"
    "fmt"
    "log"
    "net/url"
    "os"
    "os/signal"
    "syscall"
    "time"

    "github.com/gorilla/websocket"
)

var SERVER = ""
var PATH = ""
var TIMESWAIT = 0
var TIMESWAITMAX = 5
var in = bufio.NewReader(os.Stdin)
```

Переменная `in` — это просто сокращение для `bufio.NewReader(os.Stdin)`.

```
func getInput(input chan string) {
    result, err := in.ReadString('\n')
```

```

    if err != nil {
        log.Println(err)
        return
    }
    input <- result
}

```

Функция `GetInput()`, выполняемая в виде горутины, получает пользовательский ввод, который передается функции `main()` через канал `input`. Каждый раз, когда программа считывает какой-либо пользовательский ввод, старая горутина завершается и запускается новая горутина `GetInput()` для получения нового ввода.

```

func main() {
    arguments := os.Args
    if len(arguments) != 3 {
        fmt.Println("Need SERVER + PATH!")
        return
    }

    SERVER = arguments[1]
    PATH = arguments[2]
    fmt.Println("Connecting to:", SERVER, "at", PATH)

    interrupt := make(chan os.Signal, 1)
    signal.Notify(interrupt, os.Interrupt)
}

```

Клиент WebSocket обрабатывает прерывания UNIX с помощью канала `interrupt`. Когда будет получен соответствующий сигнал (`syscall.SIGINT`), соединение WebSocket с сервером закрывается с помощью сообщения `websocket.CloseMessage`. Вот так работают профессиональные инструменты!

```

input := make(chan string, 1)
go getInput(input)

URL := url.URL{Scheme: "ws", Host: SERVER, Path: PATH}
c, _, err := websocket.DefaultDialer.Dial(URL.String(), nil)
if err != nil {
    log.Println("Error:", err)
    return
}
defer c.Close()

```

Подключение к WebSocket начинается с вызова `websocket.DefaultDialer.Dial()`. Все, что поступает на канал `input`, передается на сервер WebSocket с помощью метода `WriteMessage()`.

```

done := make(chan struct{})
go func() {

```

```

    defer close(done)
    for {
        _, message, err := c.ReadMessage()
        if err != nil {
            log.Println("ReadMessage() error:", err)
            return
        }
        log.Printf("Received: %s", message)
    }
}()
```

Другаяgorутина, которая на этот раз реализована путем применения анонимной Go-функции, отвечает за чтение данных из соединения WebSocket с помощью метода ReadMessage().

```

for {
    select {
    case <-time.After(4 * time.Second):
        log.Println("Please give me input!", TIMESWAIT)
        TIMESWAIT++
        if TIMESWAIT > TIMESWAITMAX {
            syscall.Kill(syscall.Getpid(), syscall.SIGINT)
        }
    }
```

Оператор `syscall.Kill(syscall.Getpid(), syscall.SIGINT)` отправляет сигнал прерывания в программу с помощью Go-кода. Согласно логике `client.go`, данный сигнал заставляет программу закрыть соединение WebSocket с сервером и прекратить его выполнение. Это происходит только в том случае, если текущее количество периодов ожидания больше предопределенного глобального значения.

```

case <-done:
    return
case t := <-input:
    err := c.WriteMessage(websocket.TextMessage, []byte(t))
    if err != nil {
        log.Println("Write error:", err)
        return
    }
    TIMESWAIT = 0
```

Если вы получаете пользовательский ввод, то текущее количество периодов ожидания (`TIMESWAIT`) сбрасывается и считывается новый ввод.

```

go getInput(input)
case <-interrupt:
    log.Println("Caught interrupt signal - quitting!")
    err := c.WriteMessage(websocket.CloseMessage,
        websocket.FormatCloseMessage(websocket.CloseNormalClosure, ""))
```

Непосредственно перед тем, как закрыть клиентское соединение, мы отправляем на сервер `websocket.CloseMessage`, чтобы выполнить закрытие должным образом.

```
if err != nil {
    log.Println("Write close error:", err)
    return
}
select {
case <-done:
case <-time.After(2 * time.Second):
}
return
}
}
```

Поскольку `./client/client.go` находится в отдельном каталоге относительно `ws.go`, нам придется выполнить следующие команды, чтобы собрать необходимые зависимости и осуществить запуск:

```
$ cd client
$ go mod init
$ go mod tidy
$ go mod download
```

Взаимодействие с сервером WebSocket приводит к следующему выводу:

```
$ go run client.go localhost:1234 ws
Connecting to: localhost:1234 at ws
Hello there!
2021/04/10 21:30:33 Received: Hello there!
```

В двух вышеприведенных строках показан пользовательский ввод, а также ответ сервера.

```
2021/04/10 21:30:37 Please give me input! 0
2021/04/10 21:30:41 Please give me input! 1
2021/04/10 21:30:45 Please give me input! 2
2021/04/10 21:30:49 Please give me input! 3
2021/04/10 21:30:53 Please give me input! 4
2021/04/10 21:30:57 Please give me input! 5
2021/04/10 21:30:57 Caught interrupt signal - quitting!
2021/04/10 21:30:57 ReadMessage() error: websocket: close 1000 (normal)
```

В последних строках показано, как работает процесс автоматического тайм-аута.

Сервер WebSocket сгенерировал такой вывод для вышеописанного взаимодействия:

```
2021/04/10 21:30:29 Connection from: localhost:1234
2021/04/10 21:30:33 Received: Hello there!
2021/04/10 21:30:57 From localhost:1234 read websocket: close 1000
(normal)
```

Но если сервер WebSocket не может быть найден по указанному адресу, то клиент WebSocket выводит следующее:

```
$ go run client.go localhost:1234 ws
Connecting to: localhost:1234 at ws
2021/04/09 10:29:23 Error: dial tcp [::1]:1234: connect: connection
refused
```

Сообщение `connection refused` указывает на то, что не существует процесса, прослушивающего порт 1234 на `localhost`.

WebSocket предоставляет альтернативный способ создания сервисов. Как правило, WebSocket лучше подходит, когда требуется обмениваться большим количеством данных и мы хотим, чтобы соединение оставалось открытым все время, позволяя обмен данными в полнодуплексном режиме. Но если вы не уверены в том, что использовать, то начните с сервиса TCP/IP и посмотрите, как он работает, прежде чем обновлять его до протокола WebSocket.

Упражнения

- Разработайте параллельный TCP-сервер, который генерирует случайные числа в заранее определенном диапазоне.
- Разработайте параллельный TCP-сервер, который генерирует случайные числа в диапазоне, заданном TCP-клиентом. Его можно использовать как способ случайного выбора значений из набора.
- Добавьте обработку сигналов UNIX к параллельному TCP-серверу, разработанному в этой главе, чтобы красиво остановить процесс сервера при получении заданного сигнала.
- Разработайте сервер на сокетах домена UNIX, который генерирует случайные числа. После этого запрограммируйте клиент для этого сервера.
- Разработайте сервер WebSocket, который создает переменное количество случайных целых чисел, которые отправляются клиенту. Количество случайных целых чисел указывается клиентом в первоначальном сообщении клиента.

Резюме

Эта глава была посвящена пакету `net`, TCP/IP, TCP, UDP, сокетам UNIX и WebSocket, которые реализуют довольно низкоуровневые соединения. TCP/IP – это то, что правит Интернетом. Кроме того, WebSocket удобен, когда необходимо передавать много данных. Наконец, доменные сокеты UNIX предпочтительнее, когда обмен данными между сервером и различными его клиентами происходит на одной машине. Со может помочь вам создать все виды параллельных серверов и клиентов. Теперь вы готовы приступить к разработке и развертыванию собственных сервисов!

Следующая глава посвящена REST API, обмену данными JSON по протоколу HTTP и разработке клиентов и серверов RESTful, для которой широко используется Go.

Дополнительные ресурсы

- Протокол WebSocket: <https://tools.ietf.org/rfc/rfc6455.txt>.
- WebSocket в «Википедии»: <https://en.wikipedia.org/wiki/WebSocket>.
- Пакет Gorilla WebSocket: <https://github.com/gorilla/websocket>.
- Документы Gorilla WebSocket: <https://www.gorillatoolkit.org/pkg/websocket>.
- Пакет `websocket`: <https://pkg.go.dev/golang.org/x/net/websocket>.

10

Работа с REST API

Предметом обсуждения в этой главе будут разработка и эксплуатация простых серверов и клиентов RESTful с помощью языка программирования Go. REST является аббревиатурой от *REpresentational State Transfer* и в первую очередь представляет собой архитектуру для проектирования веб-сервисов. Они обмениваются информацией в HTML, а сервисы RESTful обычно используют формат JSON, который отлично поддерживается Go. REST не привязан ни к какой операционной системе или системной архитектуре и не является протоколом. Тем не менее, чтобы реализовать RESTful-сервис, вам придется использовать такой протокол, как HTTP. При разработке серверов RESTful вам нужно будет создать соответствующие структуры Go и выполнить необходимые операции маршалинга и демаршалинга для обмена данными JSON.

В этой поистине важной и практической главе представлены следующие темы:

- введение в REST;
- разработка серверов и клиентов RESTful;
- создание функционального сервера RESTful;
- создание клиента RESTful;
- загрузка и скачивание двоичных файлов;
- использование Swagger для документации REST API.

Введение в REST

Большинство современных веб-приложений работают, предоставляя свои API и позволяя клиентам использовать эти API для взаимодействия. REST не призован к HTTP, но большинство веб-сервисов используют HTTP в качестве базового протокола. Кроме того, хотя REST может работать с любым форматом данных, обычно его использование означает использование *JSON через HTTP*, поскольку большую часть времени обмен данными в сервисах RESTful осуществляется в формате JSON. Бывают также случаи, когда обмен данными происходит в обычном текстовом формате (как правило, когда этот обмен прост и нет необходимости использовать записи JSON). Из-за того, как работает сервис RESTful, он должен содержать архитектуру, которая отвечает следующим принципам:

- дизайн типа «клиент-сервер»;
- реализация без состояния — это означает, что каждое взаимодействие не зависит от других;
- кэшируемость;
- единый интерфейс;
- многоуровневая система.

В соответствии с протоколом HTTP вы можете выполнять на HTTP-сервере следующие операции:

- **POST** — используется для создания новых ресурсов;
- **GET** — применяется для чтения (*получения*) существующих ресурсов;
- **PUT** — служит для обновления существующих ресурсов. По соглашению запрос PUT должен содержать полную и обновленную версию существующего ресурса;
- **DELETE** — используется для удаления существующих ресурсов;
- **PATCH** — применяется для обновления существующих ресурсов. Запрос PATCH содержит только изменения существующего ресурса.

Здесь важно подчеркнуть, что все ваши действия (особенно когда это не обычное поведение) должны быть хорошо задокументированы. В качестве справки можно иметь в виду, что HTTP-методы, поддерживаемые Go, определены как константы в пакете `net/http`:

```
const (
    MethodGet      = "GET"
    MethodHead     = "HEAD"
    MethodPost     = "POST"
    MethodPut      = "PUT"
```

```
MethodPatch = "PATCH" // RFC 5789
MethodDelete = "DELETE"
MethodConnect = "CONNECT"
MethodOptions = "OPTIONS"
MethodTrace = "TRACE"
)
```

Кроме того, существуют соглашения, касающиеся возврата *кода состояния HTTP* для каждого клиентского запроса. Наиболее популярные коды состояния HTTP, а также их значения представлены ниже.

- **200** — все прошло хорошо и указанное действие было выполнено успешно.
- **201** — желаемый ресурс был *создан*.
- **202** — запрос был *принят* и в настоящее время обрабатывается. Обычно этот код используется, когда выполнение действия занимает слишком много времени.
- **301** — запрошенный ресурс был перемещен на постоянной основе, новый URI должен быть частью ответа. Он редко используется в сервисах RESTful, поскольку обычно вы используете управление *версиями API*.
- **400** — поступил *неверный запрос* и вам следует изменить свой первоначальный запрос, прежде чем отправлять его снова.
- **401** — клиент попытался получить доступ к защищенному запросу без авторизации.
- **403** — у клиента нет необходимых разрешений для доступа к ресурсу, даже если он авторизован должным образом. В терминологии UNIX **403** означает, что пользователь не обладает необходимыми привилегиями для выполнения действия.
- **404** — ресурс не был найден.
- **405** — клиент использовал метод, который не разрешен типом ресурса.
- **500** — внутренняя ошибка сервера — вероятно, указывающая на его сбой.

Если вы хотите узнать больше о протоколе HTTP, то вам следует обратиться к RFC 7231 по адресу <https://datatracker.ietf.org/doc/html/rfc7231>.

Пока я пишу эту главу, я разрабатываю небольшой клиент RESTful для проекта, над которым работаю. Клиент подключается к заданному серверу и получает список имен пользователей. Для каждого имени пользователя я должен получить список времени входа и выхода из системы, используя другую конечную точку.

По своему опыту могу вам сказать, что большая часть кода предназначена не для взаимодействия с сервером RESTful, а для обработки данных, преобразования их в желаемый формат и сохранения в базе данных MySQL. Две самые сложные

задачи, которые мне нужно было выполнить, — это получение даты и времени в формате эпохи UNIX и усечение информации о минутах и секундах от времени этой эпохи, а также вставка новой записи в таблицу базы данных после проверки того, что запись еще не была сохранена в этой базе. Таким образом, стоит ожидать, что логика программы будет отвечать за большую часть кода, что верно не только для сервисов RESTful, но и для всех сервисов вообще.

Первый раздел этой главы содержит общую, но важную информацию о программировании серверов и клиентов RESTful.

Разработка серверов и клиентов RESTful

В этом разделе мы разработаем сервер RESTful и клиент для него, используя функциональность стандартной библиотеки Go, чтобы понять, как все работает на самом деле. Функциональность сервера описана в следующем списке конечных точек:

- `/add` предназначена для добавления новых записей на сервер;
- `/delete` используется для удаления существующей записи;
- `/get` предназначена для получения информации об уже существующей записи;
- `/time` возвращает текущую дату и время и в основном используется для тестирования работы сервера RESTful;
- `/` используется для обслуживания любого запроса, который не соответствует какой-либо другой конечной точке.

Это мой предпочтительный способ структурирования сервиса RESTful. Альтернативным способом конечные точки можно задать так:

- `/users/` с помощью метода `GET`: получить список всех пользователей;
- `/users/:id` с помощью метода `GET`: получить информацию о пользователе с заданным значением идентификатора;
- `/users/:id` с помощью метода `DELETE`: удалить пользователя с заданным идентификатором;
- `/users/` с помощью метода `POST`: создать нового пользователя;
- `/users/:id` с помощью метода `PATCH` или `PUT`: обновить пользователя с заданным значением ID.

Реализация альтернативного способа оставлена в качестве упражнения для читателя. Реализовать этот способ не так сложно, учитывая, что код Go для об-

работчиков будет тем же самым, а вам будет нужно переопределить только ту часть, где мы задаем обработку конечных точек. В следующем подразделе представлена реализация сервера RESTful.

Сервер RESTful

Цель представленной реализации — понять, как все работает, так как принципы, лежащие в основе сервисов REST, не меняются. Логика, лежащая в основе каждой функции обработчика, проста: считать пользовательский ввод и определить, являются ли данный ввод и HTTP-метод желаемыми. Принципы взаимодействия каждого клиента не менее просты: сервер должен вернуть клиенту соответствующие сообщения об ошибках и HTTP-коды, чтобы все точно знали, что произошло на самом деле. Наконец, все должно быть задокументировано, чтобы все участники говорили на одном языке.

Код сервера, расположенный в файле `rServer.go`, выглядит следующим образом:

```
package main

import (
    "encoding/json"
    "fmt"
    "io"
    "log"
    "net/http"
    "os"
    "time"
)

type User struct {
    Username string `json:"user"`
    Password string `json:"password"`
}
```

Это структура, которая содержит пользовательские данные. Использование тегов JSON имеет здесь ключевое значение.

```
var user User
```

Эта глобальная переменная содержит пользовательские данные — входные данные для конечных точек `/add`, `/get` и `/delete` и их упрощенных реализаций. Поскольку эта переменная является общей для всей программы, наш код не является *параллельно безопасным*, что вполне нормально для сервера RESTful, используемого в качестве доказательства концепции.

Реальный сервер RESTful, который реализован в разделе «Создание функционального сервера RESTful» (см. далее в этой главе), будет параллельно безопасным.

```
// PORT – это то, что прослушивает веб-сервер
var PORT = ":1234"
```

Сервер RESTful – это лишь HTTP-сервер, поэтому мы определяем номер TCP-порта, который прослушивает сервер.

```
// DATA – это карта, хранящая записи пользователей
var DATA = make(map[string]string)
```

Это еще одна глобальная переменная, которая содержит данные сервиса.

```
func defaultHandler(w http.ResponseWriter, r *http.Request) {
    log.Println("Serving:", r.URL.Path, "from", r.Host)
    w.WriteHeader(http.StatusNotFound)
    Body := "Thanks for visiting!\n"
    fmt.Fprintf(w, "%s", Body)
}
```

Это обработчик по умолчанию. На рабочем сервере обработчик по умолчанию может выводить инструкции о работе сервера, а также список доступных конечных точек.

```
func timeHandler(w http.ResponseWriter, r *http.Request) {
    log.Println("Serving:", r.URL.Path, "from", r.Host)
    t := time.Now().Format(time.RFC1123)
    Body := "The current time is: " + t + "\n"
    fmt.Fprintf(w, "%s", Body)
}
```

Это еще один простой обработчик, который возвращает текущую дату и время. Такие обработчики обычно используются для проверки работоспособности сервера и, как правило, удаляются из рабочей версии.

```
func addHandler(w http.ResponseWriter, r *http.Request) {
    log.Println("Serving:", r.URL.Path, "from", r.Host, r.Method)
    if r.Method != http.MethodPost {
        http.Error(w, "Error:", http.StatusMethodNotAllowed)
        fmt.Fprintf(w, "%s\n", "Method not allowed!")
        return
    }
}
```

Здесь мы впервые сталкиваемся с функцией `http.Error()`. Она отправляет ответ на запрос клиента, в который входит указанное сообщение об ошибке (в виде обычного текста), а также желаемый HTTP-код. Однако вам все равно нужно записать данные, которые вы хотите отправить обратно клиенту, используя оператор `fmt.Fprintf()`.

```

d, err := io.ReadAll(r.Body)
if err != nil {
    http.Error(w, "Error:", http.StatusBadRequest)
    return
}

```

Мы пытаемся прочитать все данные с клиента сразу, используя `io.ReadAll()`, и убеждаемся, что прочитали данные без каких-либо ошибок, путем проверки значения переменной `error`, возвращаемой `io.ReadAll(r.Body)`.

```

err = json.Unmarshal(d, &user)
if err != nil {
    log.Println(err)
    http.Error(w, "Error:", http.StatusBadRequest)
    return
}

```

После получения данных от клиента мы помещаем их в глобальную переменную `user`. Где хранить данные и что с ними делать, решает сервер. Нет никаких правил касательно того, как интерпретировать данные. Однако клиент должен взаимодействовать с сервером в соответствии с пожеланиями сервера.

```

if user.Username != "" {
    DATA[user.Username] = user.Password
    log.Println(DATA)
    w.WriteHeader(http.StatusOK)
}

```

Если указанное поле `Username` не пустое, то нужно добавить новую структуру в карту `DATA`. Сохранение данных не реализовано — каждый раз, когда вы перезапускаете сервер RESTful, `DATA` инициализируется заново.

```

} else {
    http.Error(w, "Error:", http.StatusBadRequest)
    return
}

```

Если значение поля `username` пустое, то мы не можем добавить его в карту `DATA` и операция завершается с ошибкой `http.StatusBadRequest`.

```

func getHandler(w http.ResponseWriter, r *http.Request) {
    log.Println("Serving:", r.URL.Path, "from", r.Host, r.Method)
    if r.Method != http.MethodGet {
        http.Error(w, "Error:", http.StatusMethodNotAllowed)
        fmt.Fprintf(w, "%s\n", "Method not allowed!")
        return
    }
}

```

Для конечной точки `/get` нам нужно использовать `http.MethodGet`, поэтому мы должны убедиться, что это условие выполнено (`if r.Method != http.MethodGet`).

```
d, err := io.ReadAll(r.Body)
if err != nil {
    http.Error(w, "ReadAll - Error", http.StatusBadRequest)
    return
}
```

И снова нам нужно убедиться, что мы без проблем можем считывать данные из клиентского запроса.

```
err = json.Unmarshal(d, &user)
if err != nil {
    log.Println(err)
    http.Error(w, "Unmarshal - Error", http.StatusBadRequest)
    return
}
fmt.Println(user)
```

Затем мы используем клиентские данные и помещаем их в структуру `User` (глобальную переменную `user`).

```
_, ok := DATA[user.Username]
if ok && user.Username != "" {
    log.Println("Found!")
    w.WriteHeader(http.StatusOK)
    fmt.Fprintf(w, "%s\n", d)
```

Если желаемая пользовательская запись найдена, то мы возвращаем ее клиенту, используя данные, хранящиеся в переменной `d`. Помните, что `d` была инициализирована в вызове `io.ReadAll(r.Body)` и уже содержит маршиализируемую запись JSON.

```
} else {
    log.Println("Not found!")
    w.WriteHeader(http.StatusNotFound)
    http.Error(w, "Map - Resource not found!", http.StatusNotFound)
}
return
```

В противном случае мы сообщаем клиенту, что желаемая запись не была найдена.

```
func deleteHandler(w http.ResponseWriter, r *http.Request) {
    log.Println("Serving:", r.URL.Path, "from", r.Host, r.Method)
    if r.Method != http.MethodDelete {
        http.Error(w, "Error:", http.StatusMethodNotAllowed)
        fmt.Fprintf(w, "%s\n", "Method not allowed!")
        return
    }}
```

Метод HTTP `DELETE` выглядит как рациональный выбор при удалении ресурса, отсюда и `r.Method != http.MethodDelete` в teste.

```
d, err := io.ReadAll(r.Body)
if err != nil {
    http.Error(w, "ReadAll - Error", http.StatusBadRequest)
    return
}
```

Повторюсь, мы считываем входные данные клиента и сохраняем их в переменной `d`.

```
err = json.Unmarshal(d, &user)
if err != nil {
    log.Println(err)
    http.Error(w, "Unmarshal - Error", http.StatusBadRequest)
    return
}
log.Println(user)
```

Считается хорошей практикой сохранять дополнительную информацию в журнале при удалении ресурсов.

```
, ok := DATA[user.Username]
if ok && user.Username != "" {
    if user.Password == DATA[user.Username] {
```

В процессе удаления, прежде чем удалять соответствующую запись, мы удостоверяемся, что оба заданных значения (имя пользователя и пароль) совпадают с существующими в карте `DATA`.

```
        delete(DATA, user.Username)
        w.WriteHeader(http.StatusOK)
        fmt.Fprintf(w, "%s\n", d)
        log.Println(DATA)
    }
} else {
    log.Println("User", user.Username, "Not found!")
    w.WriteHeader(http.StatusNotFound)
    http.Error(w, "Delete - Resource not found!", http.StatusNotFound)
}
log.Println("After:", DATA)
```

После процесса удаления мы выводим содержимое карты `DATA`, чтобы убедиться, что все прошло ожидаемо. Обычно на рабочем сервере это не делается.

```
    return
}

func main() {
    arguments := os.Args
```

```

if len(arguments) != 1 {
    PORT = ":" + arguments[1]
}

```

В коде выше представлен метод определения номера TCP-порта веб-сервера при наличии значения по умолчанию. Итак, если нет аргументов командной строки, то используем значение по умолчанию. В противном случае применяем значение, заданное в качестве аргумента командной строки.

```

mux := http.NewServeMux()
s := &http.Server{
    Addr:         PORT,
    Handler:      mux,
    IdleTimeout:  10 * time.Second,
    ReadTimeout:   time.Second,
    WriteTimeout: time.Second,
}

```

Выше приведены подробные сведения и параметры для веб-сервера.

```

mux.Handle("/time", http.HandlerFunc(timeHandler))
mux.Handle("/add", http.HandlerFunc(addHandler))
mux.Handle("/get", http.HandlerFunc(getHandler))
mux.Handle("/delete", http.HandlerFunc(deleteHandler))
mux.Handle("/", http.HandlerFunc(defaultHandler))

```

Здесь мы определяем конечные точки веб-сервера — тут нет ничего особенного, поскольку скрытым образом сервер RESTful реализует HTTP-сервер.

```

fmt.Println("Ready to serve at", PORT)
err := s.ListenAndServe()
if err != nil {
    fmt.Println(err)
    return
}

```

Последний шаг касается запуска веб-сервера с предопределенными параметрами, что является обычной практикой. После этого мы тестируем сервер RESTful с помощью утилиты `curl(1)`, которая очень удобна, когда у вас нет клиента и вы хотите протестировать работу сервера RESTful. Хорошо то, что `curl(1)` может отправлять и получать данные JSON.

```
$ curl localhost:1234/ Thanks for visiting!
```

Первое взаимодействие с сервером RESTful предназначено лишь для того, чтобы убедиться, что сервер работает должным образом. Следующее взаимодей-

ствие заключается в добавлении нового пользователя на сервер — сведения о пользователе находятся в записи {"user": "mtsouk", "password": "admin"} JSON:

```
$ curl -H 'Content-Type: application/json' -d '{"user": "mtsouk", "password": "admin"}' http://localhost:1234/add -v
*   Trying ::1...
* TCP_NODELAY set
* Connected to localhost (::1) port 1234 (#0)
```

В этом выводе показано, что `curl(1)` успешно подключился к серверу (`localhost`), используя нужный TCP-порт (1234).

```
> POST /add HTTP/1.1
> Host: localhost:1234
> User-Agent: curl/7.64.1
> Accept: /*
> Content-Type: application/json
> Content-Length: 40
```

В этом выводе показано, что `curl(1)` собирается отправлять данные с использованием метода `POST`, а длина данных составляет 40 байт.

```
>
* upload completely sent off: 40 out of 40 bytes
< HTTP/1.1 200 OK
< Date: Tue, 27 Apr 2021 09:41:38 GMT
< Content-Length: 0
```

В этом выводе мы получаем сообщение, что данные были отправлены и *тело* ответа сервера составляет 0 байт.

```
<
* Connection #0 to host localhost left intact
* Closing connection 0
```

В последней части выходных данных мы получаем сообщение, что после отправки данных на сервер соединение было закрыто.



При работе с сервером RESTful нам нужно добавить `-H 'Content-Type: application/json'` в `curl(1)`, чтобы указать, что мы планируем работать в формате JSON. Параметр `-d` используется для передачи данных на сервер и эквивалентен параметру `--data`, тогда как параметр `-v` генерирует более подробный вывод, если нам нужно больше подробностей.

Если мы попытаемся добавить того же пользователя, то сервер RESTful не будет возвращать:

```
$ curl -H 'Content-Type: application/json' -d '{"user": "mtsouk",  
"password" : "admin"}' http://localhost:1234/add
```

Хотя такое поведение может быть неидеальным, хорошо, если оно задокументировано. Такое запрещено на реальном сервере, но допустимо при экспериментировании.

```
$ curl -H 'Content-Type: application/json' -d '{"user": "mihalis",  
"password" : "admin"}' http://localhost:1234/add
```

С помощью этой команды мы добавляем другого пользователя, как указывает `{"user": "mihalis", "password" : "admin"}`.

```
$ curl -H -d '{"user": "admin"}' http://localhost:1234/add  
curl: (3) URL using bad/illegal format or missing URL  
Error:  
Method not allowed!
```

В этом выводе показано ошибочное взаимодействие, где за значением `-H` не следует значение. Хотя запрос и отправляется на сервер, он отклоняется, поскольку `/add` не использует HTTP-метод по умолчанию.

```
$ curl -H 'Content-Type: application/json' -d '{"user": "admin",  
"password" : "admin"}' http://localhost:1234/get  
Error:  
Method not allowed!
```

На этот раз команда `curl` верна, но используемый HTTP-метод установлен неправильно. Следовательно, запрос не будет удовлетворен.

```
$ curl -X GET -H 'Content-Type: application/json' -d '{"user": "admin",  
"password" : "admin"}' http://localhost:1234/get  
Map - Resource not found!  
$ curl -X GET -H 'Content-Type: application/json' -d '{"user":  
"mtsouk", "password" : "admin"}' http://localhost:1234/get  
{"user": "mtsouk", "password" : "admin"}
```

В этих двух взаимодействиях для получения информации о существующем пользователе используется `/get`. Однако находится только второй пользователь.

```
$ curl -H 'Content-Type: application/json' -d '{"user": "mtsouk",  
"password" : "admin"}' http://localhost:1234/delete -X DELETE  
{"user": "mtsouk", "password" : "admin"}
```

Последнее взаимодействие успешно удаляет пользователя, указанного с помощью `{"user": "mtsouk", "password": "admin"}`.

Выходные данные, сгенерированные серверным процессом для всех вышеописанных взаимодействий, будут выглядеть следующим образом:

```
$ go run rServer.go Ready to serve at :1234
Ready to serve at :1234
2021/04/27 12:41:31 Serving: / from localhost:1234
2021/04/27 12:41:38 Serving: /add from localhost:1234 POST
2021/04/27 12:41:38 map[mtsouk:admin]
2021/04/27 12:41:41 Serving: /add from localhost:1234 POST
2021/04/27 12:41:41 map[mtsouk:admin]
2021/04/27 12:41:58 Serving: /add from localhost:1234 POST
2021/04/27 12:41:58 map[mihalis:admin mtsouk:admin]
2021/04/27 12:43:02 Serving: /add from localhost:1234 GET
2021/04/27 12:43:13 Serving: /get from localhost:1234 POST
2021/04/27 12:43:30 Serving: /get from localhost:1234 GET
{admin admin}
2021/04/27 12:43:30 Not found!
2021/04/27 12:43:30 http: superfluous response.WriteHeader call from
main.getHandler (rServer.go:101)
2021/04/27 12:43:41 Serving: /get from localhost:1234 GET
{mtsouk admin}
2021/04/27 12:43:41 Found!
2021/04/27 12:44:00 Serving: /delete from localhost:1234 DELETE
2021/04/27 12:44:00 {mtsouk admin}
2021/04/27 12:44:00 map[mihalis:admin]
2021/04/27 12:44:00 After: map[mihalis:admin]
```

На данный момент у нас имеется работающий сервер RESTful, протестированный с помощью утилиты `curl(1)`. Следующий подраздел посвящен разработке клиента командной строки для сервера RESTful.

Клиент RESTful

В текущем подразделе показана разработка клиента для сервера RESTful, созданного ранее. Однако в этом случае клиент действует как тестовая программа, которая пробует возможности сервера RESTful. Позже в этой главе вы узнаете, как писать правильные клиенты, используя `cobra`. Итак, код клиента, который можно найти в файле `rClient.go`, выглядит следующим образом:

```
package main

import (
    "bytes"
    "encoding/json"
    "fmt"
```

```

    "io"
    "net/http"
    "os"
    "time"
)

type User struct {
    Username string `json:"user"`
    Password string `json:"password"`
}

```

Эта же структура присутствует в серверной реализации и используется для обмена данными.

```

var u1 = User{"admin", "admin"}
var u2 = User{"tsoukalos", "pass"}
var u3 = User{"", "pass"}

```

Здесь мы предварительно определяем три переменные `User`, которые будут использоваться во время тестирования.

```

const addEndPoint = "/add"
const getEndPoint = "/get"
const deleteEndPoint = "/delete"
const timeEndPoint = "/time"

```

Вышеприведенные константы определяют конечные точки, которые будут использоваться.

```

func deleteEndpoint(server string, user User) int {
    userMarshall, _ := json.Marshal(user)
    u := bytes.NewReader(userMarshall)

    req, err := http.NewRequest("DELETE", server+deleteEndPoint, u)

```

Мы готовим запрос, который собирается получить доступ `/delete`, используя метод HTTP `DELETE`.

```

    if err != nil {
        fmt.Println("Error in req: ", err)
        return http.StatusInternalServerError
    }
    req.Header.Set("Content-Type", "application/json")

```

Это правильный способ указать, что при взаимодействии с сервером мы хотим использовать *данные JSON*.

```

c := &http.Client{
    Timeout: 15 * time.Second,
}

resp, err := c.Do(req)
defer resp.Body.Close()

```

Затем мы отправляем запрос и ждем ответа сервера, используя метод `Do()` с 15-секундным тайм-аутом.

```
if err != nil {
    fmt.Println("Error:", err)
}
if resp == nil {
    return http.StatusNotFound
}

data, err := io.ReadAll(resp.Body)
fmt.Println("/delete returned: ", string(data))
```

Причина появления здесь `fmt.Println()` заключается в том, что мы хотим знать ответ сервера, даже если во взаимодействии произошла ошибка.

```
if err != nil {
    fmt.Println("Error:", err)
}
return resp.StatusCode
}
```

Значение `resp.StatusCode` указывает на результат, полученный из конечной точки `/delete`.

```
func getEndpoint(server string, user User) int {
    userMarshall, _ := json.Marshal(user)
    u := bytes.NewReader(userMarshall)

    req, err := http.NewRequest("GET", server+getEndPoint, u)
```

Мы собираемся обратиться к `/get`, используя метод HTTP GET.

```
if err != nil {
    fmt.Println("Error in req: ", err)
    return http.StatusInternalServerError
}
req.Header.Set("Content-Type", "application/json")
```

Мы указываем, что будем взаимодействовать с сервером, используя формат JSON, с помощью `Header.Set()`.

```
c := &http.Client{
    Timeout: 15 * time.Second,
}
```

Мы определяем период ожидания для HTTP-клиента на случай, если сервер отвечает слишком долго.

```
resp, err := c.Do(req)
defer resp.Body.Close()

if err != nil {
```

```

        fmt.Println("Error:", err)
    }
    if resp == nil {
        return http.StatusNotFound
    }
}

```

Здесь мы отправляем клиентский запрос на сервер, используя `c.Do (req)`, и сохраняем ответ сервера в `resp`, а значение ошибки — в `err`. Если значение `resp` равно `nil`, то ответ сервера был пустым, что служит условием ошибки.

```

data, err := io.ReadAll(resp.Body)
fmt.Print("/get returned: ", string(data))
if err != nil {
    fmt.Println("Error:", err)
}
return resp.StatusCode
}

```

Значение `resp.StatusCode`, которое указывается и передается сервером RESTful, определяет, было ли взаимодействие успешным с точки зрения HTTP (логически) или нет.

```

func addEndpoint(server string, user User) int {
    userMarshall, _ := json.Marshal(user)
    u := bytes.NewReader(userMarshall)

    req, err := http.NewRequest("POST", server+addEndPoint, u)
}

```

Мы обращаемся к `/add`, используя метод HTTP `POST`. Вы можете использовать `http.MethodPost` вместо `POST`. Как указывалось ранее в этой главе, в `http` существуют соответствующие глобальные переменные для остальных методов HTTP (`http.MethodGet`, `http.MethodDelete`, `http.MethodPut` и др.). Рекомендуется использовать именно их, поскольку это помогает портированию.

```

if err != nil {
    fmt.Println("Error in req: ", err)
    return http.StatusInternalServerError
}
req.Header.Set("Content-Type", "application/json")
}

```

Как и прежде, мы указываем, что будем взаимодействовать с сервером, используя формат JSON.

```

c := &http.Client{
    Timeout: 15 * time.Second,
}

```

И снова мы определяем период ожидания для клиента на случай, если сервер слишком занят.

```

resp, err := c.Do(req)
defer resp.Body.Close()
}

```

```

if resp == nil || (resp.StatusCode == http.StatusNotFound) {
    return resp.StatusCode
}

return resp.StatusCode
}

```

Функция `addEndpoint()` предназначена для тестирования конечной точки `/add` с помощью метода POST и конечной точки `/add`.

```
func timeEndpoint(server string) (int, string) {
    req, err := http.NewRequest("POST", server+timeEndPoint, nil)
```

Мы обратимся к конечной точке `/time`, используя метод HTTP POST.

```

if err != nil {
    fmt.Println("Error in req: ", err)
    return http.StatusInternalServerError, ""
}

c := &http.Client{
    Timeout: 15 * time.Second,
}

```

Как и прежде, мы определяем период ожидания для клиента на случай, если сервер слишком затягивает с ответом.

```

resp, err := c.Do(req)
defer resp.Body.Close()

if resp == nil || (resp.StatusCode == http.StatusNotFound) {
    return resp.StatusCode, ""
}

data, _ := io.ReadAll(resp.Body)
return resp.StatusCode, string(data)
}
```

Функция `timeEndpoint()` предназначена для тестирования конечной точки `/time`. Обратите внимание, что эта точка не требует никаких данных от клиента, поэтому клиентский запрос пуст. Сервер возвращает строку времени и даты.

```
func slashEndpoint(server, URL string) (int, string) {
    req, err := http.NewRequest("POST", server+URL, nil)
```

Мы собираемся обратиться к `/`, используя метод HTTP POST.

```

if err != nil {
    fmt.Println("Error in req: ", err)
    return http.StatusInternalServerError, ""
}
```

```
c := &http.Client{
    Timeout: 15 * time.Second,
}
```

Считается хорошей практикой устанавливать тайм-аут на стороне клиента на случай задержек в ответе сервера.

```
resp, err := c.Do(req)
defer resp.Body.Close()

if resp == nil {
    return resp.StatusCode, ""
}

data, _ := io.ReadAll(resp.Body)
return resp.StatusCode, string(data)
}
```

Функция `slashEndpoint()` предназначена для тестирования конечной точки по умолчанию на сервере. Обратите внимание, что эта точка не требует никаких данных от клиента.

Далее следует реализация функции `main()`, которая использует все вышеприведенные функции для посещения конечных точек сервера RESTful:

```
func main() {
    if len(os.Args) != 2 {
        fmt.Println("Wrong number of arguments!")
        fmt.Println("Need: Server")
        return
    }
    server := os.Args[1]
```

Переменная `server` содержит как адрес сервера, так и номер используемого порта.

```
fmt.Println("/add")
HTTPcode := addEndpoint(server, u1)
if HTTPcode != http.StatusOK {
    fmt.Println("u1 Return code:", HTTPcode)
} else {
    fmt.Println("u1 Data added:", u1, HTTPcode)
}

HTTPcode = addEndpoint(server, u2)
if HTTPcode != http.StatusOK {
    fmt.Println("u2 Return code:", HTTPcode)
} else {
    fmt.Println("u2 Data added:", u2, HTTPcode)
}
```

```

HTTPcode = addEndpoint(server, u3)
if HTTPcode != http.StatusOK {
    fmt.Println("u3 Return code:", HTTPcode)
} else {
    fmt.Println("u3 Data added:", u3, HTTPcode)
}

```

Весь вышеприведенный код используется для тестирования конечной точки `/add` с помощью различных типов данных.

```

fmt.Println("/get")
HTTPcode = getEndpoint(server, u1)
fmt.Println("/get u1 return code:", HTTPcode)
HTTPcode = getEndpoint(server, u2)
fmt.Println("/get u2 return code:", HTTPcode)
HTTPcode = getEndpoint(server, u3)
fmt.Println("/get u3 return code:", HTTPcode)

```

Весь этот код используется для тестирования конечной точки `/get` с помощью различных типов входных данных. Мы проверяем только код возврата, поскольку HTTP-код указывает на успех или неудачу операции.

```

fmt.Println("/delete")
HTTPcode = deleteEndpoint(server, u1)
fmt.Println("/delete u1 return code:", HTTPcode)
HTTPcode = deleteEndpoint(server, u1)
fmt.Println("/delete u1 return code:", HTTPcode)
HTTPcode = deleteEndpoint(server, u2)
fmt.Println("/delete u2 return code:", HTTPcode)
HTTPcode = deleteEndpoint(server, u3)
fmt.Println("/delete u3 return code:", HTTPcode)

```

Вышеприведенный код используется для тестирования конечной точки `/delete` с помощью различных типов входных данных. И снова мы выводим HTTP-код взаимодействия, поскольку значение HTTP-кода определяет успех или неудачу операции.

```

fmt.Println("/time")
HTTPcode, myTime := timeEndpoint(server)
fmt.Print("/time returned: ", HTTPcode, " ", myTime)
time.Sleep(time.Second)
HTTPcode, myTime = timeEndpoint(server)
fmt.Print("/time returned: ", HTTPcode, " ", myTime)

```

Здесь мы проверяем конечную точку `/time` — она выводит HTTP-код, а также остальную часть ответа сервера.

```

fmt.Println("/")
URL := "/"

```

```
HTTPcode, response := slashEndpoint(server, URL)
fmt.Println("/ returned: ", HTTPcode, " with response: ", response)

fmt.Println("/what")
URL = "/what"
HTTPcode, response = slashEndpoint(server, URL)
fmt.Println(URL, " returned: ", HTTPcode, " with response: ", response)
}
```

В последней части программы мы пытаемся подключиться к несуществующей конечной точке, чтобы проверить правильность работы функции обработчика по умолчанию.

При выполнении `rClient.go` при запущенном `rServer.go` мы получаем такой вывод:

```
$ go run rClient.go http://localhost:1234
/add
u1 Data added: {admin admin} 200
u2 Data added: {tsoukalos pass} 200
u3 Return code: 400
```

Эта часть связана с тестированием конечной точки `/add`. Первые два пользователя были успешно добавлены, тогда как третий (`var u3 = User{"", "pass"}`) не был, поскольку он не содержит всей необходимой информации.

```
/get
/get returned: {"user":"admin","password":"admin"}
/get u1 return code: 200
/get returned: {"user":"tsoukalos","password":"pass"}
/get u2 return code: 200
/get returned: Map - Resource not found!
/get u3 return code: 404
```

Эта часть связана с тестированием конечной точки `/get`. Данные первых двух пользователей `admin` и `tsoukalos` были успешно возвращены, тогда как пользователь, сохраненный в переменной `u3`, не был найден.

```
/delete
/delete returned: {"user":"admin","password":"admin"}
/delete u1 return code: 200
/delete returned: Delete - Resource not found!
/delete u1 return code: 404
/delete returned: {"user":"tsoukalos","password":"pass"}
/delete u2 return code: 200
/delete returned: Delete - Resource not found!
/delete u3 return code: 404
```

Этот вывод связан с тестированием конечной точки `/delete`. Пользователи `admin` и `tsoukalos` были удалены. Однако попытка удалить `admin` во второй раз завершилась неудачей.

```
/time
/time returned: 200 The current time is: Tue, 20 Apr 2021 10:23:04 EEST
/time returned: 200 The current time is: Tue, 20 Apr 2021 10:23:05 EEST
```

Аналогично эта часть связана с тестированием конечной точки `/time`.

```
/
/ returned: 404 with response: Thanks for visiting!
/what
/what returned: 404 with response: Thanks for visiting!
```

Последняя часть выходных данных связана с работой обработчика по умолчанию.

Пока что как сервер RESTful, так и клиент могут взаимодействовать друг с другом. Однако ни один из них не выполняет реальную работу. В следующем разделе показано, как разработать реальный сервер RESTful с помощью `gorilla/mux` и серверной части БД для хранения данных.

Создание функционального сервера RESTful

В этом разделе показано, как разработать сервер RESTful в Go с помощью REST API. Самое большое различие между представленным сервисом RESTful и приложением телефонной книги, созданным в главе 8, заключается в том, что сервис RESTful в этой главе повсеместно использует сообщения JSON, тогда как приложение взаимодействует и работает с использованием обычных текстовых сообщений. Если вы уже задумались об использовании `net/http` для реализации сервера RESTful, то, пожалуйста, остановитесь! Эта реализация использует пакет `gorilla/mux`, который является гораздо лучшим выбором, поскольку поддерживает подпрограммы; подробнее об этом поговорим в подразделе «Использование пакета `gorilla/mux`» (см. далее в текущей главе).

Цель сервера RESTful — внедрение системы логина/аутентификации. Цель системы логина — отслеживание пользователей, которые вошли в систему, а также их разрешения. Система поставляется с пользователем-администратором по умолчанию `admin` — пароль по умолчанию также `admin`, и вы должны изменить его. Приложение хранит свои данные в базе данных (PostgreSQL); это значит, при перезапуске список существующих пользователей считывается из нее и не теряется.

REST API

API приложения помогает вам реализовать ту функциональность, которую вы задумали. Однако это работа для клиента, а не для сервера. Задача сервера состоит в том, чтобы максимально облегчить работу своих клиентов, поддерживая простую, но полностью работающую функциональность с помощью правильно определенного и реализованного REST API. Убедитесь, что вы осознаете это, прежде чем пытаться разработать и использовать сервер RESTful.

Мы собираемся определить конечные точки, которые будут использоваться, возвращаемые HTTP-коды, а также разрешенный метод или методы. Создание рабочего сервера RESTful на основе REST API — серьезная работа, к которой не следует относиться легкомысленно. Создание *прототипа* для тестирования и подтверждения ваших идей и планов в долгосрочной перспективе сэкономит вам массу времени. Всегда начинайте с прототипа.

Поддерживаемые конечные точки, а также поддерживаемые методы HTTP и параметры представлены ниже.

- Конечная точка / используется для отлавливания и обработки всего, что не подходит под остальные запросы. Работает со всеми HTTP-методами.
- Конечная точка /`getall` служит для получения полного содержимого базы данных. Для использования требуется пользователь с правами администратора. Эта точка может возвращать несколько записей JSON и работает с методом GET.
- Команда /`getid`/`username` применяется для получения ID пользователя, найденного по его имени пользователя, который передается конечной точке. Команда должна подаваться пользователем с правами администратора и поддерживать метод HTTP GET.
- Конечная точка /`username`/`ID` используется для удаления или получения информации о пользователе с идентификатором, равным ID, в зависимости от применяемого метода HTTP. Следовательно, фактическое действие, которое будет выполнено, зависит от используемого метода HTTP. Метод DELETE удаляет пользователя, в то время как метод GET возвращает информацию о пользователе. Эта конечная точка должна применяться пользователям с правами администратора.
- Конечная точка /`logged` служит для получения списка всех вошедших в систему пользователей. Может возвращать несколько записей JSON и требует использования метода HTTP GET.
- Конечная точка /`update` применяется для обновления имени пользователя, пароля или статуса администратора пользователя — идентификатор пользователя в базе данных остается прежним. Работает только с методом HTTP PUT, и поиск пользователя осуществляется на основе имени пользователя.

- Конечная точка `/login` служит для входа пользователя в систему с указанием имени пользователя и пароля. Работает с методом HTTP `POST`.
- Конечная точка `/logout` используется для выхода пользователя из системы с указанием имени пользователя и пароля. Работает с методом HTTP `POST`.
- Конечная точка `/add` служит для добавления нового пользователя в базу данных. Работает с методом `POST` и применяется пользователем с правами администратора.
- Конечная точка `/time` используется в основном для целей тестирования. Это единственная конечная точка, которая не работает с данными JSON, не требует действительной учетной записи и работает со всеми методами HTTP.

Теперь обсудим возможности и функциональность пакета `gorilla/mux`.

Использование пакета `gorilla/mux`

Пакет `gorilla/mux` (<https://github.com/gorilla/mux>) — популярная и эффективная альтернатива маршрутизатору Go по умолчанию, который позволяет сопоставлять входящие запросы с соответствующим обработчиком. Существует много различий между маршрутизатором Go по умолчанию (`http.ServeMux`) и `mux.Router` (маршрутизатор `gorilla/mux`), но основное заключается в том, что `mux.Router` поддерживает *несколько условий* при сопоставлении маршрута с функцией обработчика. Это означает, что вам потребуется писать меньше кода для обработки некоторых параметров, таких как используемый метод HTTP. Начнем с представления некоторых подходящих примеров — эта функциональность не поддерживается маршрутизатором Go по умолчанию:

- `r.HandleFunc("/url", UrlHandlerFunction)` — эта команда вызывает функцию `UrlHandlerFunction` при каждом посещении `/url`;
- `r.HandleFunc("/url", UrlHandlerFunction).Methods(http.MethodPut)` — в этом примере показано, как вы можете дать Gorilla указание распознавать определенный метод HTTP (в данном случае PUT, который определяется использованием `http.MethodPut`), что избавляет от необходимости писать код для выполнения этого вручную;
- `mux.NotFoundHandler = http.HandlerFunc(handlers.DefaultHandler)` — в Gorilla правильный способ распознать все, что не соответствует никакому другому пути, — это использовать `mux.NotFoundHandler`;
- `mux.MethodNotAllowedHandler = notAllowed` — если метод не разрешен для существующего маршрута, то обрабатывается с помощью `MethodNotAllowedHandler`. Это характерно для `gorilla/mux`;

- `s.HandleFunc("/users/{id:[0-9]+}", HandlerFunction)` — в этом последнем примере показано, что вы можете определить переменную в пути, используя имя (`id`) и шаблон, и Gorilla выполнит сопоставление за вас! Если нет регулярного выражения, то совпадение будет любым, начиная с начальной косой черты и заканчивая следующей косой чертой в пути.

Теперь поговорим о другой возможности пакета `gorilla/mux`, а именно о подмаршрутизаторах.

Использование подмаршрутизаторов

Серверная реализация использует подмаршрутизаторы. *Подмаршрутизатор* — это *вложенный маршрут*, который будет проверяться на наличие потенциальных совпадений только в том случае, если родительский маршрут соответствует параметрам подмаршрутизатора. Хорошо то, что родительский маршрут может содержать условия, общие для всех путей, определенных в подмаршрутизаторе, включая хосты, префиксы путей и, как это происходит в нашем случае, методы HTTP-запроса. В результате наши подмаршрутизаторы разделены на основе общего метода запроса последующих конечных точек. Это не только оптимизирует сопоставления запросов, но и облегчает понимание структуры кода.

В качестве примера подпрограмма для метода HTTP `DELETE` выглядит очень просто:

```
deleteMux := mux.Methods(http.MethodDelete).Subrouter()
deleteMux.HandleFunc("/username/{id:[0-9]+}", handlers.DeleteHandler)
```

Первый оператор предназначен для определения общих характеристик подпрограммы, в данном случае HTTP-метод `http.MethodDelete`, тогда как оставшийся оператор (`deleteMux.HandleFunc(...)`) предназначен для определения поддерживаемых путей.

Да, `gorilla/mux` может показаться более сложным, чем маршрутизатор Go по умолчанию, но вы уже должны понимать преимущества этого пакета для работы с HTTP-сервисами.

Работа с базой данных

В этом подразделе мы разрабатываем Go-пакет для работы с базой данных PostgreSQL, которая поддерживает функциональность сервера RESTful. Пакет называется `restdb` и хранится в <https://github.com/mactsouk/restdb>. Благодаря использованию *Go-модулей* его разработка может происходить в любом месте вашей файловой системы — в данном случае в папке `~/code/restdb`. Итак, мы

запускаем `go mod init github.com/mactsouk/restdb`, тем самым включая Go-модули для пакета `restdb` во время его разработки.



Сам по себе сервер RESTful ничего не знает о сервере PostgreSQL. Вся связанная функциональность хранится в пакете `restdb`; это значит, если вы измените базу данных, то функции-обработчики не должны об этом знать.

Чтобы облегчить задачу читателям, мы запустим базу данных с помощью Docker и конфигурацией, которую можно найти в `docker-compose.yml`, внутри GitHub-репозитория `restdb`, который поставляется со следующим содержимым:

```
version: '3.1'

services:
  postgres:
    image: postgres
    container_name: postgredb
    environment:
      POSTGRES_USER: mtsouk
      POSTGRES_PASSWORD: pass
      POSTGRES_DB: restapi
    volumes:
      - ./postgres:/var/lib/postgresql/data/
    ports:
      - 5432:5432

volumes:
  postgres_data:
    driver: local
```

Таким образом, сервер PostgreSQL прослушивает порт с номером 5432, как внутренний, так и внешний. Что действительно важно, так это номер внешнего порта, поскольку это именно то, что будут использовать все клиенты. Имя базы данных, ее таблица и пользователь `admin` создаются с помощью следующего SQL-файла `create_db.sql`:

```
DROP DATABASE IF EXISTS restapi;
CREATE DATABASE restapi;
\c restapi;

/*
Пользователи
*/
CREATE TABLE users (
  id SERIAL PRIMARY KEY,
  username VARCHAR NOT NULL,
```

```

password VARCHAR NOT NULL,
lastlogin INT,
admin INT,
active INT
);

INSERT INTO users (username, password, lastlogin, admin, active) VALUES
('admin', 'admin', 1620922454, 1, 1);

```

При условии, что PostgreSQL выполняется с помощью представленного docker-compose.yml, вы можете использовать `create_db.sql` следующим образом:

```

$ psql -U mtsouk postgres -h 127.0.0.1 < create_db.sql
Password for user mtsouk:
DROP DATABASE
CREATE DATABASE
You are now connected to database "restapi" as user "mtsouk".
CREATE TABLE
INSERT 0 1

```

Поскольку большинство команд в `restdb` работают аналогичным образом, мы представим здесь лишь наиболее важные функции, начиная с `ConnectPostgres()`:

```

func ConnectPostgres() *sql.DB {
    conn := fmt.Sprintf("host=%s port=%d user=%s password=%s dbname=%s
sslmode=disable",
        Hostname, Port, Username, Password, Database)

```

`Hostname`, `Port`, `Username`, `Password` и `Database` — это глобальные переменные, определенные в другом месте пакета. Они содержат сведения о подключении.

```

    db, err := sql.Open("postgres", conn)
    if err != nil {
        log.Println(err)
        return nil
    }

    return db
}

```

Поскольку нам нужно постоянно подключаться к PostgreSQL, мы создали вспомогательную функцию, возвращающую переменную `*sql.DB`, которую можно использовать для взаимодействия с PostgreSQL.

Далее мы переходим к функции `deleteUser()`:

```

func DeleteUser(ID int) bool {
    db := ConnectPostgres()
    if db == nil {

```

```

        log.Println("Cannot connect to PostgreSQL!")
        db.Close()
        return false
    }
    defer db.Close()
}

```

В коде выше показано, как мы используем `ConnectPostgres()`, чтобы получить подключение к базе данных для работы.

```

t := FindUserID(ID)
if t.ID == 0 {
    log.Println("User", ID, "does not exist.")
    return false
}

```

Здесь мы используем вспомогательную функцию `FindUserID()`, чтобы убедиться, что пользователь с заданным идентификатором есть в базе данных. Если его не существует, то функция останавливается и возвращает значение `false`.

```

stmt, err := db.Prepare("DELETE FROM users WHERE ID = $1")
if err != nil {
    log.Println("DeleteUser:", err)
    return false
}

```

Это фактическое заявление об удалении пользователя. Мы используем `Prepare()` для построения требуемого SQL-оператора, который выполним с помощью `Exec()`. Элемент `$1` в `Prepare()` обозначает параметр, который будет задан в `Exec()`. Если бы мы хотели иметь больше параметров, то должны были бы называть их `$2`, `$3` и т. д.

```

_, err = stmt.Exec(ID)
if err != nil {
    log.Println("DeleteUser:", err)
    return false
}

return true
}

```

На этом реализация функции `deleteUser()` заканчивается. Оператор `stmt.Exec(ID)` удаляет пользователя из базы данных.

Функция `ListAllUsers()`, представленная далее, возвращает срез элементов `User`, содержащий всех пользователей, найденных на сервере RESTful:

```

func ListAllUsers() []User {
    db := ConnectPostgres()
}

```

```

if db == nil {
    fmt.Println("Cannot connect to PostgreSQL!")
    db.Close()
    return []User{}
}
defer db.Close()

rows, err := db.Query("SELECT * FROM users \n")
if err != nil {
    log.Println(err)
    return []User{}
}

```

Поскольку запрос `SELECT` не требует параметров, вместо `Prepare()` и `Exec()` для его запуска мы используем `Query()`. Имейте в виду, что это запрос, который, скорее всего, возвращает несколько записей.

```

all := []User{}
var c1 int
var c2, c3 string
var c4 int64
var c5, c6 int

for rows.Next() {
    err = rows.Scan(&c1, &c2, &c3, &c4, &c5, &c6)

```

Так мы считываем значения из одной записи, возвращаемой SQL-запросом. Сначала мы определяем несколько переменных для каждого из возвращаемых значений, а затем передаем их указатели в `Scan()`. Метод `rows.Next()` продолжает возвращать записи до тех пор, пока есть результаты.

```

    temp := User{c1, c2, c3, c4, c5, c6}
    all = append(all, temp)
}

log.Println("All:", all)
return all
}

```

Как упоминалось ранее, срез структур `User` возвращается из `ListAllUsers()`.

И наконец, представляем реализацию `IsUserValid()`:

```

func IsUserValid(u User) bool {
    db := ConnectPostgres()
    if db == nil {
        fmt.Println("Cannot connect to PostgreSQL!")
        db.Close()
        return false
    }
    defer db.Close()
}

```

Это самая обычная схема: мы вызываем `ConnectPostgres()` и ждем получения соединения для использования.

```
rows, err := db.Query("SELECT * FROM users WHERE Username = $1 \n", u.Username)
if err != nil {
    log.Println(err)
    return false
}
```

Здесь мы передаем наш параметр в `Query()` без использования `Prepare()` и `Exec()`.

```
temp := User{}
var c1 int
var c2, c3 string
var c4 int64
var c5, c6 int
```

Здесь мы создаем необходимые параметры для сохранения выходных данных SQL-запроса.

```
// если существует несколько пользователей с одним и тем же именем
// пользователя,
// то мы получим только ПЕРВОГО
for rows.Next() {
    err = rows.Scan(&c1, &c2, &c3, &c4, &c5, &c6)
    if err != nil {
        log.Println(err)
        return false
    }
    temp = User{c1, c2, c3, c4, c5, c6}
}
```

Повторюсь, цикл `for` продолжает выполняться до тех пор, пока `rows.Next()` возвращает новые записи.

```
if u.Username == temp.Username && u.Password == temp.Password {
    return true
}
```

Отмечу важный момент: чтобы данный пользователь был действительным, не только он должен существовать, но и указанный пароль должен совпадать с тем, который хранится в базе данных.

```
return false
}
```

Вы можете просмотреть остальную часть реализации `restdb` самостоятельно. Большинство функций похожи на те, которые представлены здесь. Код `restdb.go` будет использоваться в реализации сервера RESTful, который представлен далее. Но, как вы сейчас увидите, сначала мы протестируем `restdb`.

Тестирование пакета `restdb`

Сервер RESTful разработан в `~/go/src/github.com/mactsouk/rest-api`, и если вы не планируете делать его доступным для всего мира, то вам не нужно создавать для него отдельный репозиторий GitHub. Тем не менее я хочу иметь возможность поделиться им с вами, поэтому репозиторий GitHub для сервера находится на <https://github.com/mactsouk/rest-api>.

Прежде чем продолжить реализацию сервера, мы используем пакет `restdb` и убедимся, что он работает должным образом, после чего используем его в реализации сервера RESTful. Представленная утилита использует функции из `restdb`. Вам не придется тестировать все возможные варианты использования пакета `restdb` — просто убедитесь, что он работает и подключается к базе данных PostgreSQL. По этой причине мы собираемся создать отдельную утилиту командной строки под названием `useRestdb.go`, которая находится в каталоге `test-db`. Наиболее важной деталью в этом файле является использование в коде `restdb.User`, поскольку именно эта структура необходима пакету `restdb`. Мы не можем передать другую структуру в качестве параметра функциям `restdb`.

Инициализация модулей и запуск `useRestdb.go` приводят к следующему выводу:

```
$ go mod init
go: creating new go.mod: module github.com/mactsouk/rest-api/test-db
go: to add module requirements and sums:
    go mod tidy
$ go mod tidy
go: finding module for package github.com/mactsouk/restdb
go: finding module for package github.com/lib/pq
go: downloading github.com/mactsouk/restdb v0.0.0-20210510205310-
63ba9fa172df
go: found github.com/lib/pq in github.com/lib/pq v1.10.1
go: found github.com/mactsouk/restdb in github.com/mactsouk/restdb
v0.0.0-20210510205310-63ba9fa172df
$ go run useRestdb.go
&{host=localhost port=5432 user=mtsouk password=pass dbname=restapi
sslmode=disable 0x856110} 0 {0 0} [] map[] 0 0 0xc00007c240 false map[]
map[] 0 0 0 <nil> 0 0 0 0x4dd260}
{0 0 0}
mike
packt
admin
2021/05/17 09:40:23 Populating PostgreSQL
User inserted successfully.
2021/05/17 09:40:23 Found user: {9 mtsouk admin 1621233623 1 1}
mtsouk: {9 mtsouk admin 1621233623 1 1}
```

```
2021/05/17 09:40:23 Found user: {9 mtsouk admin 1621233623 1 1}
User Deleted.
mtsouk: {0 0 0 0}
2021/05/17 09:40:23 User 0 does not exist.
User not Deleted.
2021/05/17 09:40:23 User 0 does not exist.
User not Deleted.
```

Отсутствие сообщений об ошибках говорит нам о том, что пока пакет `restdb` работает ожидаемым образом: пользователи добавляются и удаляются из базы данных, а запросы выполняются. Просто помните, что это быстрый и грязный способ тестирования пакета.



Все пакеты улучшены, и новая функциональность добавлена почти ко всем существующим пакетам. Если вы хотите обновить пакет `restdb` или любой другой внешний пакет и использовать более новую версию при разработке собственных утилит, то можете выполнить команду `go get -u -v` в каталоге, в котором находятся файлы вашей утилиты `go.sum` и `go.mod`.

Реализация сервера RESTful

Теперь, когда мы убедились, что пакет `restdb` работает должным образом, мы готовы перейти к реализации сервера RESTful. Код сервера разделен на два файла, оба принадлежат пакету `main` и называются `main.go` и `handlers.go`. Это сделано в основном для того, чтобы избежать наличия огромных файлов кода для работы и логически разделить функциональность сервера.

Наиболее важной частью `main.go`, которая относится к функции `main()`, является следующий код:

```
rMux.NotFoundHandler = http.HandlerFunc(DefaultHandler)
```

Для начала нам нужно определить функцию обработчика по умолчанию (в этом нет необходимости, однако наличие такого обработчика является хорошей практикой).

```
notAllowed := notAllowedHandler{}
rMux.MethodNotAllowedHandler = notAllowed
```

Обработчик `MethodNotAllowedHandler` выполняется, когда вы пытаетесь посетить конечную точку, используя неподдерживаемый HTTP-метод. Фактическая реализация обработчика находится в файле `handlers.go`.

```
rMux.HandleFunc("/time", TimeHandler)
```

Конечная точка `/time` поддерживается всеми HTTP-методами, поэтому не при- надлежит ни к одному подмаршрутизатору.

```
// определяем функции обработчика
// регистрируем GET
getMux := rMux.Methods(http.MethodGet).Subrouter()

getMux.HandleFunc("/getall", GetAllHandler)
getMux.HandleFunc("/getid/{username}", GetIDHandler)
getMux.HandleFunc("/logged", LoggedUsersHandler)
getMux.HandleFunc("/username/{id:[0-9]+}", GetUserDataHandler)
```

Сначала мы определяем подмаршрутизатор для HTTP-метода `GET` вместе с под-держиваемыми конечными точками. Помните, что пакет `gorilla/mux` отвечает за то, чтобы подмаршрутизатор `getMux` обслуживал только запросы `GET`.

```
// регистрируем PUT
// обновляем пользователя
putMux := rMux.Methods(http.MethodPut).Subrouter()
putMux.HandleFunc("/update", UpdateHandler)
```

После этого мы определяем подмаршрутизатор для запросов `PUT`.

```
// регистрируем POST
// Add User + Login + Logout
postMux := rMux.Methods(http.MethodPost).Subrouter()
postMux.HandleFunc("/add", AddHandler)
postMux.HandleFunc("/login", LoginHandler)
postMux.HandleFunc("/logout", LogoutHandler)
```

Затем мы определяем подмаршрутизатор для запросов `POST`.

```
// регистрация DELETE
// удалить пользователя
deleteMux := rMux.Methods(http.MethodDelete).Subrouter()
deleteMux.HandleFunc("/username/{id:[0-9]+}", DeleteHandler)
```

Последний подмаршрутизатор предназначен для HTTP-методов `DELETE`. Код в пакете `gorilla/mux` отвечает за выбор правильного подмаршрутизатора на основе запроса клиента.

```
go func() {
    log.Println("Listening to", PORT)
    err := s.ListenAndServe()
    if err != nil {
        log.Printf("Error starting server: %s\n", err)
        return
    }
}()
```

HTTP-сервер выполняется как горутина, поскольку программа поддерживает обработку сигналов (подробности см. в главе 7).

```
sigs := make(chan os.Signal, 1)
signal.Notify(sigs, os.Interrupt)
sig := <-sigs
log.Println("Quitting after signal:", sig)
time.Sleep(5 * time.Second)
sShutdown(nil)
```

Наконец, мы добавляем обработку сигналов для корректного завершения работы HTTP-сервера. Оператор `sig := <-sigs` предотвращает выход функции `main()`, если только не получен сигнал `os.Interrupt`.

Файл `handlers.go` содержит реализации функций обработчика и также является частью пакета `main`. Его наиболее важные части выглядят так:

```
// AddHandler предназначен для добавления нового пользователя
func AddHandler(rw http.ResponseWriter, r *http.Request) {
    log.Println("AddHandler Serving:", r.URL.Path, "from", r.Host)
    d, err := io.ReadAll(r.Body)
    if err != nil {
        rw.WriteHeader(http.StatusBadRequest)
        log.Println(err)
        return
    }
```

Этот обработчик предназначен для конечной точки `/add`. Сервер считывает вводимые клиентом данные с помощью `io.ReadAll()`.

```
if len(d) == 0 {
    rw.WriteHeader(http.StatusBadRequest)
    log.Println("No input!")
    return
}
```

Затем код проверяет, что тело клиентского запроса не является пустым.

```
// Мы считываем две структуры как массив:
// 1. Пользователь, который выполняет команду
// 2. Пользователь, который будет добавлен
var users = []restdb.User{}
err = json.Unmarshal(d, &users)
if err != nil {
    log.Println(err)
    rw.WriteHeader(http.StatusBadRequest)
    return
}
```

Поскольку конечная точка `/add` требует двух структур `User`, код выше использует `json.Unmarshal()`, чтобы поместить их в переменную `[]restdb.User`. Это означает, что клиент должен отправить эти две записи JSON с помощью массива. Причина использования `restdb.User` заключается в том, что все функции, связанные с базой данных, работают с переменными `restdb.User`. Даже если бы у нас была структура с тем же определением, что и `restdb.User`, Go рассматривал бы их как различающиеся. Это не относится к клиенту, поскольку клиент отправляет данные без связанного с ними типа данных.

```
log.Println(users)

if !restdb.IsUserAdmin(users[0]) {
    log.Println("Issued by non-admin user:", users[0].Username)
    rw.WriteHeader(http.StatusBadRequest)
    return
}
```

Если пользователь, выполняющий команду, не является администратором, то запрос завершается ошибкой. `restdb.IsUserAdmin()` реализован в пакете `restdb`.

```
result := restdb.InsertUser(users[1])
if !result {
    rw.WriteHeader(http.StatusBadRequest)
}
}
```

В противном случае `restdb.InsertUser()` вставляет нужного пользователя в базу данных.

Наконец, рассмотрим обработчик для конечной точки `/getall`.

```
// GetAllHandler предназначен для получения всех данных из пользовательской
базы данных
func GetAllHandler(rw http.ResponseWriter, r *http.Request) {
    log.Println("GetAllHandler Serving:", r.URL.Path, "from", r.Host)
    d, err := io.ReadAll(r.Body)
    if err != nil {
        rw.WriteHeader(http.StatusBadRequest)
        log.Println(err)
        return
    }
}
```

Повторюсь, мы читаем данные от клиента, используя `io.ReadAll(r.Body)`, после чего убеждаемся, что процесс безошибочен, проверяя переменную `err`.

```
if len(d) == 0 {
    rw.WriteHeader(http.StatusBadRequest)
    log.Println("No input!")
```

```

        return
    }

    var user = restdb.User{}
    err = json.Unmarshal(d, &user)
    if err != nil {
        log.Println(err)
        rw.WriteHeader(http.StatusBadRequest)
        return
    }
}

```

Здесь мы помещаем клиентские данные в переменную `restdb.User`. Для конечной точки `/getall` требуется одна запись `restdb.User` в качестве входных данных.

```

if !restdb.IsUserAdmin(user) {
    log.Println("User", user, "is not an admin!")
    rw.WriteHeader(http.StatusBadRequest)
    return
}

```

Только пользователи с правами администратора могут посетить `/getall` и получить список всех пользователей.

```

err = SliceToJSON(restdb.ListAllUsers(), rw)
if err != nil {
    log.Println(err)
    rw.WriteHeader(http.StatusBadRequest)
    return
}
}

```

Последняя часть кода посвящена получению желаемых данных из базы данных и отправке их клиенту с помощью вызова `SliceToJSON(restdb.ListAllUsers(), rw)`.

Не стесняйтесь помещать каждый обработчик в отдельный Go-файл. Общая идея заключается в том, что если у вас много функций-обработчиков, то правильнее использовать отдельный файл для каждой функции-обработчика.

В следующем подразделе выполняется тестирование сервера RESTful с помощью утилиты `curl(1)` перед разработкой надлежащего клиента.

Тестирование сервера RESTful

В этом подразделе показано, как протестировать сервер RESTful с помощью утилиты `curl(1)`. Стоит тестировать сервер RESTful максимально часто и как можно более тщательно, чтобы выявить ошибки или нежелательное поведение. Поскольку мы используем два файла для реализации сервера, нам нужно запустить

его так: `go run main.go handlers.go`. Кроме того, не забудьте установить и запустить PostgreSQL. Мы начнем с тестирования обработчика `/time`, который работает со всеми HTTP-методами:

```
$ curl localhost:1234/time  
The current time is: Mon, 17 May 2021 09:14:00 EEST
```

Далее мы тестируем обработчик по умолчанию:

```
$ curl localhost:1234/  
/ is not supported. Thanks for visiting!  
$ curl localhost:1234/doesNotExist  
/doesNotExist is not supported. Thanks for visiting!
```

Наконец, мы видим, что произойдет, если мы используем неподдерживаемый HTTP-метод с поддерживаемой конечной точкой — в данном случае конечной точкой `/getall`, которая работает только с GET:

```
$ curl -s -X PUT -H 'Content-Type: application/json' localhost:1234/  
getall  
Method not allowed!
```

Хотя для работы конечной точки `/getall` требуется действительный пользователь, тот факт, что мы используем HTTP-метод, который не поддерживается этой конечной точкой, имеет приоритет, и вызов завершается ошибкой по правильным причинам.



Важно просмотреть выходные данные сервера RESTful и записи журнала, которые он генерирует во время тестирования. Не вся информация может быть отправлена обратно клиенту, но серверному процессу разрешено выводить все что угодно. Это может быть очень полезно для отладки серверного процесса, такого как наш RESTful server.

Далее мы протестируем все обработчики, которые поддерживают HTTP-метод GET.

Тестирование обработчиков GET

Сначала мы тестируем конечную точку `/getall`:

```
$ curl -s -X GET -H 'Content-Type: application/json' -d '{"username":  
"admin", "password" : "newPass"}' localhost:1234/getall  
[{"ID":1,"Username":"admin","Password":"newPass","LastLogin":1620922454  
,"Admin":1,"Active":1},{"ID":6,"Username":"mihalis","Password":"admin",  
"LastLogin":1620926815,"Admin":1,"Active":0},{"ID":7,"Username":"mike",  
"Password":"admin","LastLogin":1620926862,"Admin":1,"Active":0}]
```

Этот вывод представляет собой список всех существующих пользователей, найденных в базе данных, в формате JSON.

Затем мы тестируем конечную точку /logged:

```
$ curl -X GET -H 'Content-Type: application/json' -d '{"username": "admin", "password" : "newPass"}' localhost:1234/logged
[{"ID":1,"Username":"admin","Password":"newPass","LastLogin":1620922454,"Admin":1,"Active":1}]
```

После этого мы тестируем конечную точку /username/{id}:

```
$ curl -X GET -H 'Content-Type: application/json' -d '{"username": "admin", "password" : "newPass"}' localhost:1234/username/7
{"ID":7,"Username":"mike","Password":"admin","LastLogin":1620926862,"Admin":1,"Active":0}
```

Наконец, мы тестируем конечную точку /getid/{username}:

```
$ curl -X GET -H 'Content-Type: application/json' -d '{"username": "admin", "password" : "newPass"}' localhost:1234/getid/admin
User admin has ID: 1
```

На данный момент мы можем получить список существующих пользователей и список вошедших в систему пользователей, а также получить информацию о конкретных пользователях — все эти конечные точки используют метод GET.

Далее мы протестируем все обработчики, которые поддерживают HTTP-метод POST.

Тестирование обработчиков POST

Сначала мы тестируем конечную точку /add, добавляя пользователя packt, у которого нет прав администратора:

```
$ curl -X POST -H 'Content-Type: application/json' -d '[{"username": "admin", "password" : "newPass", "admin":1}, {"username": "packt", "password" : "admin", "admin":0} ]' localhost:1234/add
```

Этот вызов передает массив записей JSON на сервер для добавления нового пользователя packt. Команда выдается пользователем admin.

Если мы попытаемся добавить одно и то же имя пользователя более одного раза, то процесс завершится неудачей — это можно наблюдать с помощью -v в команде curl(1). Соответствующее сообщение — это HTTP/1.1 400 Bad Request.

Кроме того, если мы попытаемся добавить нового пользователя, используя учетные данные пользователя, который не является администратором, сервер генерирует сообщение Command issued by non-admin user: packt.

Затем мы тестируем конечную точку /login:

```
$ curl -X POST -H 'Content-Type: application/json' -d '{"username": "packt", "password" : "admin"}' localhost:1234/login
```

Эта команда используется для входа в систему пользователя packt.

Наконец, мы тестируем конечную точку /logout:

```
$ curl -X POST -H 'Content-Type: application/json' -d '{"username": "packt", "password" : "admin"}' localhost:1234/logout
```

Эта команда используется для выхода из системы пользователя packt. Вы можете использовать конечную точку /logged для проверки результатов предыдущих двух взаимодействий.

Теперь протестируем единственную конечную точку, которая поддерживает HTTP-метод PUT.

Тестирование обработчика PUT

Сначала протестируем конечную точку /update:

```
$ curl -X PUT -H 'Content-Type: application/json' -d '[{"username": "admin", "password" : "newPass", "admin":1}, {"username": "admin", "password" : "justChanged", "admin":1}]' localhost:1234/update
```

Эта команда изменяет пароль пользователя admin с newPass на justChanged.

Затем мы попытаемся изменить пароль, используя учетные данные пользователя, не являющегося администратором (packt):

```
$ curl -X PUT -H 'Content-Type: application/json' -d '[{"Username": "packt", "Password": "admin"}, {"username": "admin", "password" : "justChanged", "admin":1}]' localhost:1234/update
```

Сгенерированное сообщение журнала — Command issued by non-admin user: packt.

Можно считать недостатком тот факт, что пользователь, не являющийся администратором, не может изменить даже собственный пароль. Возможно, так оно и есть, но именно так реализован сервер RESTful. Идея заключается в том, что пользователи, не являющиеся администраторами, не должны напрямую выполнять опасные команды. Кроме того, этот недостаток легко обойти. Не предполагается, что обычные пользователи будут взаимодействовать с сервером таким образом, для этого им будет предоставлен веб-интерфейс. После этого пользователь-администратор может отправить пользовательский

запрос на сервер. Таким образом, все просто реализуется другим способом, который является более безопасным и не дает ненужных привилегий обычным пользователям.

Наконец, мы собираемся протестировать HTTP-метод `DELETE`.

Тестирование обработчика `DELETE`

Для HTTP-метода `DELETE` нам нужно протестировать конечную точку `/username/{id}`. Поскольку она не возвращает никаких выходных данных, использование параметра `-v` в утилите `curl(1)` покажет возвращенный код состояния HTTP:

```
$ curl -X DELETE -H 'Content-Type: application/json' -d '{"username": "admin", "password" : "justChanged"}' localhost:1234/username/6 -v
```

Код состояния HTTP/1.1 200 OK подтверждает, что пользователь был успешно удален. Если мы попытаемся удалить того же пользователя снова, то запрос завершится неудачей и возвращенное сообщение будет уже HTTP/1.1 404 Not Found.

Пока что сервер RESTful работает ожидаемым образом. Однако утилиты `curl(1)` не подходит для постоянной работы с сервером RESTful. В следующем разделе показано, как разработать клиент командной строки для созданного нами сервера RESTful.

Создание клиента RESTful

Создать клиент RESTful намного проще, чем программировать сервер, главным образом потому, что на стороне клиента не нужно работать с базой данных. Единственное, что требуется делать клиенту, — это отправлять необходимое количество и вид данных на сервер и получать обратно ответ сервера. Клиент RESTful будет разрабатываться в `~/go/src/github.com/mactsouk/rest-cli-if`, и если вы не планируете делать его доступным для всего мира, то вам не нужно создавать для него отдельный репозиторий GitHub. Однако для того, чтобы вы могли иметь возможность увидеть код клиента, я создал репозиторий GitHub, который находится по адресу <https://github.com/mactsouk/rest-cli>.

Вот поддерживаемые команды `cobra` первого уровня:

- `list` — обращается к конечной точке `/getall` и возвращает список пользователей;
- `time` — предназначена для посещения конечной точки `/time`;
- `update` — предназначена для обновления записей пользователей — идентификатор пользователя не может измениться;

- `logged` — перечисляет всех зарегистрированных пользователей;
- `delete` — удаляет существующего пользователя;
- `login` — предназначена для входа пользователя в систему;
- `logout` — служит для выхода пользователя из системы;
- `add` — предназначена для добавления в систему нового пользователя;
- `getid` — возвращает ID пользователя, основанный на его имени пользователя;
- `search` — отображает информацию о данном пользователе по его ID.



Клиент, который мы собираемся представить, работает намного лучше, чем `curl(1)`, так как может обрабатывать полученную информацию и, что наиболее важно, интерпретировать коды возврата HTTP и предварительно обрабатывать данные перед отправкой их на сервер. Цена, которую вы за это платите, — дополнительное время, необходимое для разработки и отладки клиента RESTful.

Существуют два основных флага командной строки для передачи имени пользователя и пароля пользователя, выдающего команду: `user` и `pass`. Как вы увидите в их реализациях, у них имеются сокращения `-u` и `-p` соответственно.

Кроме того, поскольку запись JSON, содержащая информацию о пользователе, содержит небольшое количество полей, все поля будут указаны в записи JSON в виде обычного текста с использованием флага `data` и сокращения `-d` — это реализовано в `root.go`. Каждая команда будет считывать только нужные флаги и нужные поля входной записи JSON — это реализовано в файле исходного кода каждой команды. Наконец, утилита будет возвращать записи JSON, когда это имеет смысл, или текстовое сообщение, относящееся к посещенной конечной точке. Теперь перейдем к структуре клиента и реализации команд.

Создание структуры клиента командной строки

В этом подразделе утилита `cobra` используется для создания структуры утилиты командной строки. Но сначала мы собираемся создать надлежащий проект `cobra` и Go-модуль:

```
$ cd ~/go/src/github.com/mactsouk
$ git clone git@github.com:mactsouk/rest-cli.git
$ cd rest-cli
$ ~/go/bin/cobra init --pkg-name github.com/mactsouk/rest-cli
$ go mod init
$ go mod tidy
$ go run main.go
```

Вам не нужно выполнять последнюю команду, но она гарантирует, что пока все в порядке. После этого мы готовы определить команды, которые утилита будет поддерживать, выполнив следующие команды `cobra`:

```
$ ~/go/bin/cobra add add
$ ~/go/bin/cobra add delete
$ ~/go/bin/cobra add list
$ ~/go/bin/cobra add logged
$ ~/go/bin/cobra add login
$ ~/go/bin/cobra add logout
$ ~/go/bin/cobra add search
$ ~/go/bin/cobra add getid
$ ~/go/bin/cobra add time
$ ~/go/bin/cobra add update
```

Теперь, когда у нас есть желаемая структура, мы можем приступить к реализации команд и, возможно, удалить некоторые комментарии, вставленные `cobra`, что и является предметом обсуждения в следующем подразделе.

Реализация клиентских команд RESTful

Поскольку нет смысла представлять весь код, который можно найти в репозитории GitHub, мы ограничимся наиболее показательным кодом некоторых команд. Начнем мы с `root.go`, где определяются следующие глобальные переменные:

```
var SERVER string
var PORT string
var data string
var username string
var password string
```

Эти глобальные переменные содержат значения параметров командной строки утилиты и доступны из любого места в коде утилиты.

```
type User struct {
    ID      int     `json:"id"`
    Username string `json:"username"`
    Password string `json:"password"`
    LastLogin int64  `json:"lastlogin"`
    Admin    int     `json:"admin"`
    Active   int     `json:"active"`
}
```

Мы определяем структуру `User` для отправки и получения данных.

```
func init() {
    rootCmd.PersistentFlags().StringVarP(&username, "username", "u",
    "username", "The username")
```

```

rootCmd.PersistentFlags().StringVarP(&password, "password", "p", "admin",
    "The password")
rootCmd.PersistentFlags().StringVarP(&data, "data", "d", "{}",
    "JSON Record")

rootCmd.PersistentFlags().StringVarP(&SERVER, "server", "s",
    "http://localhost", "RESTful server hostname")
rootCmd.PersistentFlags().StringVarP(&PORT, "port", "P", ":1234",
    "Port of RESTful Server")
}

```

Мы представляем реализацию функции `init()`, которая содержит определения параметров командной строки. Значения флагов командной строки автоматически сохраняются в переменных, которые передаются в качестве первого аргумента в `rootCmd.PersistentFlags().StringVarP()`. Итак, флаг `username` (его псевдоним `-u`) сохраняет свое значение в глобальной переменной `username`.

Далее следует реализация команды `list`, находящаяся в `list.go`:

```

var listCmd = &cobra.Command{
    Use:   "list",
    Short: "List all available users",
    Long:  `The list command lists all available users.`,
}

```

Эта часть посвящена справочным сообщениям, которые отображаются для этой команды. Хотя они и необязательны, всегда полезно иметь точное описание команды. Продолжаем реализацию:

```

Run: func(cmd *cobra.Command, args []string) {
    endpoint := "/getall"
    user := User{Username: username, Password: password}
}

```

Сначала мы создаем переменную `User` для хранения имени пользователя, выполняющего команду, и его пароля — эта переменная будет передана на сервер.

```

buf := new(bytes.Buffer)
err := user.ToJSON(buf)
if err != nil {
    fmt.Println("JSON:", err)
    return
}

```

Нам нужно закодировать переменную `user` перед передачей ее на сервер RESTful, что и является целью метода `toJSON()`. Реализация метода `toJSON()` находится в `root.go`.

```

req, err := http.NewRequest(http.MethodGet,
                           SERVER+PORT+endpoint, buf)
if err != nil {
}

```

```

        fmt.Println("GetAll - Error in req: ", err)
        return
    }
    req.Header.Set("Content-Type", "application/json")
}

```

Здесь мы создаем запрос, используя для этого глобальные переменные SERVER и PORT, за которыми следует конечная точка и желаемый HTTP-метод (`http.MethodGet`), и объявляем, что собираемся отправить данные в формате JSON с помощью `Header.Set()`.

```

c := &http.Client{
    Timeout: 15 * time.Second,
}

resp, err := c.Do(req)
if err != nil {
    fmt.Println("Do:", err)
    return
}

```

После этого мы отправляем наши данные на сервер с помощью `Do()` и получаем его ответ.

```

if resp.StatusCode != http.StatusOK {
    fmt.Println(resp)
    return
}

```

Если код состояния ответа — это не `http.StatusOK`, то запрос не выполнен.

```

var users = []User{}
SliceFromJSON(&users, resp.Body)
data, err := PrettyJSON(users)
if err != nil {
    fmt.Println(err)
    return
}

fmt.Print(data)
},
}

```

Если код состояния `http.StatusOK`, то мы готовимся прочитать срез переменных `User`. Поскольку эти переменные содержат записи JSON, нам нужно декодировать их с помощью `SliceFromJSON()`, определенной в `root.go`.

Последним будет код команды `add`, находящийся в `add.go`. Разница между `add` и `list` заключается в том, что команда `add` должна отправить две записи JSON на сервер RESTful: первая содержит данные пользователя, выдавшего команду, а вторая — пользователя, который должен быть добавлен в систему. Флаги

`username` и `password` содержат данные для полей `Username` и `Password` первой записи, в то время как флаг командной строки `data` содержит данные для второй записи.

```
var addCmd = &cobra.Command{
    Use:   "add",
    Short: "Add a new user",
    Long:  `Add a new user to the system.`,
    Run: func(cmd *cobra.Command, args []string) {
        endpoint := "/add"
        u1 := User{Username: username, Password: password}
```

Как и прежде, мы получаем информацию о пользователе, выдающем команду, и помещаем ее в структуру.

```
// преобразование строки данных в пользовательскую структуру
var u2 User
err := json.Unmarshal([]byte(data), &u2)
if err != nil {
    fmt.Println("Unmarshal:", err)
    return
}
```

Поскольку флаг командной строки `data` содержит строковое значение, нам нужно преобразовать данное значение в структуру `User` — это и есть цель вызова `json.Unmarshal()`.

```
users := []User{}
users = append(users, u1)
users = append(users, u2)
```

Затем мы создаем срез переменных `User` для отправки на сервер. Важен порядок размещения структур в этом фрагменте: сначала пользователь, выдающий команду, а затем данные пользователя, который будет создан.

```
buf := new(bytes.Buffer)
err = SliceToJSON(users, buf)
if err != nil {
    fmt.Println("JSON:", err)
    return
}
```

Затем мы кодируем этот фрагмент перед отправкой его на сервер RESTful через HTTP-запрос.

```
req, err := http.NewRequest(http.MethodPost,
                           SERVER+PORT+endpoint, buf)
if err != nil {
    fmt.Println("GetAll - Error in req: ", err)
```

```
        return
    }
    req.Header.Set("Content-Type", "application/json")

    c := &http.Client{
        Timeout: 15 * time.Second,
    }

    resp, err := c.Do(req)
    if err != nil {
        fmt.Println("Do:", err)
        return
    }
```

Мы готовим запрос и отправляем его на сервер. Сервер отвечает за декодирование предоставленных данных и соответствующие действия, в данном случае добавление нового пользователя в систему. Клиенту просто нужно посетить правильную конечную точку, используя соответствующий метод HTTP (`http.MethodPost`), и проверить возвращенный код состояния.

```
if resp.StatusCode != http.StatusOK {
    fmt.Println("Status code:", resp.Status)
} else {
    fmt.Println("User", u2.Username, "added.")
}
},
```

Команда `add` не возвращает никаких данных клиенту — нас интересует код состояния HTTP, поскольку именно он определяет успех или неудачу команды.

Использование клиента RESTful

Теперь мы используем утилиту командной строки для взаимодействия с сервером RESTful. Этот тип утилиты может использоваться для администрирования сервера RESTful, создания автоматизированных задач и выполнения заданий CI/CD. По соображениям простоты клиент и сервер находятся на одном компьютере, и мы в основном работаем с пользователем по умолчанию (`admin`) — это делает представленные команды короче. Кроме того, мы выполняем `go build` для создания двоичного исполняемого файла, чтобы избежать постоянного использования `go main.go`.

Сначала мы получаем время с сервера:

```
$ ./rest-cli time
The current time is: Tue, 25 May 2021 08:38:04 EEST
```

Далее перечислим всех пользователей. Поскольку выходные данные зависят от содержимого базы данных, мы выводим лишь небольшую часть данных. Обратите внимание, что для выполнения команды `list` требуется пользователь с правами администратора:

```
$ ./rest-cli list -u admin -p admin
[
  {
    "id": 7,
    "username": "mike",
    "password": "admin",
    "lastlogin": 1620926862,
    "admin": 1,
    "active": 0
  },
]
```

Далее мы тестируем команду `logged` с неверным паролем:

```
$ ./rest-cli logged -u admin -p notPass
&{400 Bad Request 400 HTTP/1.1 1 1 map[Content-Length:[0] Date:[Tue, 25 May 2021 05:42:36 GMT]] 0xc000190020 0 [] false false map[] 0x00000fc800 <nil>}
```

Как и ожидалось, команда завершается с ошибкой — этот вывод используется для целей отладки. Убедившись, что команда работает должным образом, вы можете выводить более подходящее сообщение об ошибке.

После этого мы тестируем команду `add`:

```
$ ./rest-cli add -u admin -p admin --data '{"Username":"newUser", "Password":"aPass"}'
User newUser added.
```

Попытка еще раз добавить того же пользователя завершится неудачей:

```
$ ./rest-cli add -u admin -p admin --data '{"Username":"newUser", "Password":"aPass"}'
Status code: 400 Bad Request
```

Далее мы собираемся удалить `newUser`, но сначала нам требуется ID пользователя `newUser`:

```
$ ./rest-cli getid -u admin -p admin --data '{"Username":"newUser"}'
User newUser has ID: 15
$ ./rest-cli delete -u admin -p admin --data '{"ID":15}'
User with ID 15 deleted.
```

Не стесняйтесь самостоятельно тестировать RESTful клиента и дайте мне знать, если обнаружите какие-либо ошибки!

Работа с несколькими версиями REST API

REST API может меняться и эволюционировать с течением времени. Существуют различные подходы к реализации управления версиями REST API:

- добавить пользовательский заголовок HTTP (`version-used`), чтобы задать используемую версию;
- применять разные поддомены для каждой версии (`v1.servername` и `v2.servername`);
- использовать комбинации заголовков `Accept` и `Content-Type` — этот метод основан на *согласовании содержимого*;
- применять различные пути для каждой версии (`/v1` и `/v2`, если сервер RESTful поддерживает две версии REST API);
- использовать параметр запроса для ссылки на желаемую версию (`.../endpoint?version=v1` или `.../endpoint?v=1`).

Не существует единственного ответа на вопрос, как именно реализовать управление версиями REST API. Используйте то, что кажется более естественным вам и вашим пользователям. Что важно, так это быть последовательным и везде использовать один и тот же подход. Лично я предпочитаю использовать `/v1/` для поддержки конечных точек версии 1, `/v2/` для поддержки конечных точек версии 2 и т. д.

На этом мы завершаем разработку серверов и клиентов RESTful. В следующем разделе показано, как загружать и скачивать двоичные файлы с помощью пакета `gorilla/mux`, если вы хотите добавить эту функцию.

Загрузка и скачивание двоичных файлов

Нередко возникает необходимость хранить двоичные файлы на сервере RESTful и иметь возможность скачивать их впоследствии — например, для разработки библиотек фотографий или документов. В этом разделе показано, как реализовать эту функциональность.

По соображениям простоты пример будет включен в репозиторий `mactsouk/rest-api` на GitHub, который ранее использовался для реализации сервера

RESTful. Для этого подраздела мы собираемся использовать каталог `file` для хранения соответствующего кода, который сохраняется как `binary.go`. На самом деле `binary.go` — это небольшой сервер RESTful, который поддерживает только загрузку и скачивание двоичных файлов через конечную точку `/files/`.

Существуют три основных способа сохранения загружаемых вами файлов:

- в локальной файловой системе;
- в системе управления базами данных, которая поддерживает хранение двоичных файлов;
- в облаке с помощью облачного провайдера.

В нашем случае мы сохраним файлы в файловой системе, в которой работает сервер. Более конкретно: во время тестирования мы сохраним загруженные файлы в каталоге `/tmp/files`.

Код `binary.go` выглядит следующим образом:

```
package main

import (
    "errors"
    "io"
    "log"
    "net/http"
    "os"
    "time"

    "github.com/gorilla/mux"
)

var PORT = ":1234"
var IMAGESPATH = "/tmp/files"
```

Два вышеприведенных глобальных параметра содержат соответственно TCP-порт, который сервер собирается прослушивать, и локальный путь, по которому будут сохранены загруженные файлы. Обратите внимание, что в большинстве систем UNIX `/tmp` автоматически очищается после перезагрузки системы.

```
func uploadFile(rw http.ResponseWriter, r *http.Request) {
    filename, ok := mux.Vars(r)["filename"]
    if !ok {
        log.Println("filename value not set!")
        rw.WriteHeader(http.StatusNotFound)
        return
    }
    log.Println(filename)
    saveFile(IMAGESPATH+"/"+filename, rw, r)
}
```

Функция `uploadFile()` отвечает за загрузку новых файлов в заранее определенный каталог. Наиболее важной частью этого процесса является использование `mux.Vars(r)` для получения значения ключа `filename`. Обратите внимание, что параметром для `mux.Vars()` служит переменная `http.Request`. Если нужный ключ существует, то функция продолжается и вызывает `SaveFile()`. В противном случае она возвращается без сохранения каких-либо файлов.

```
func saveFile(path string, rw http.ResponseWriter, r *http.Request) {
    log.Println("Saving to", path)
    err := saveToFile(path, r.Body)
    if err != nil {
        log.Println(err)
        return
    }
}
```

Цель `SaveFile()` — сохранение загруженного файла путем вызова `SaveToFile()`. Вы можете спросить, почему бы не объединить `SaveFile()` и `SaveToFile()` в одну функцию? Ответ состоит в том, что таким образом код `SaveToFile()` становится универсальным и может быть повторно использован другими утилитами.

```
func saveToFile(path string, contents io.Reader) error {
    _, err := os.Stat(path)
    if err == nil {
        err = os.Remove(path)
        if err != nil {
            log.Println("Error deleting", path)
            return err
        }
    } else if !os.IsNotExist(err) {
        log.Println("Unexpected error:", err)
        return err
    }

    f, err := os.Create(path)
    if err != nil {
        log.Println(err)
        return err
    }
    defer f.Close()

    n, err := io.Copy(f, contents)
    if err != nil {
        return err
    }
    log.Println("Bytes written:", n)

    return nil
}
```

Весь вышеприведенный код связан с файловым вводом-выводом и используется для сохранения содержимого `io.Reader` по нужному пути.

```
func createImageDirectory(d string) error {
    _, err := os.Stat(d)
    if os.IsNotExist(err) {
        log.Println("Creating:", d)
        err = os.MkdirAll(d, 0755)
        if err != nil {
            log.Println(err)
            return err
        }
    } else if err != nil {
        log.Println(err)
        return err
    }

    fileInfo, _ := os.Stat(d)

    mode := fileInfo.Mode()
    if !mode.IsDir() {
        msg := d + " is not a directory!"
        return errors.New(msg)
    }

    return nil
}

func main() {
    err := createImageDirectory(IMAGESPATH)
    if err != nil {
        log.Println(err)
        return
    }

    mux := mux.NewRouter()
    putMux := mux.Methods(http.MethodPut).Subrouter()
    putMux.HandleFunc("/files/{filename:[a-zA-Z0-9][a-zA-Z0-9\\.]*"
        "[a-zA-Z0-9]}", uploadFile)
}
```

Цель `createImageDirectory()` — создание каталога, в котором будут сохранены файлы, если этого каталога еще нет. Если путь существует и не является каталогом, то у нас проблема, поэтому функция возвращает пользовательское сообщение об ошибке.

```
return nil
}

func main() {
    err := createImageDirectory(IMAGESPATH)
    if err != nil {
        log.Println(err)
        return
    }

    mux := mux.NewRouter()
    putMux := mux.Methods(http.MethodPut).Subrouter()
    putMux.HandleFunc("/files/{filename:[a-zA-Z0-9][a-zA-Z0-9\\.]*"
        "[a-zA-Z0-9]}", uploadFile)
```

Для загрузки файлов на сервер поддерживается только HTTP-метод `PUT`. Регулярное выражение сообщает нам, что нужны имена файлов, которые начинаются с одной буквы или цифры и заканчиваются буквой или цифрой. Это значит, имена файлов не должны начинаться с `.` или `..`, что позволит избежать посещения подкаталогов и, следовательно, не поставить под угрозу безопасность системы. Как мы увидели ранее, код сохраняет значение имени файла в карте,

используя ключ `filename`, — доступ к этой карте осуществляется функцией `uploadFile()`.

```
getMux := mux.Methods(http.MethodGet).Subrouter()
getMux.Handle("/files/{filename:[a-zA-Z0-9][a-zA-Z0-9\\.]*[a-zA-Z0-9]}",
    http.StripPrefix("/files/", http.FileServer(http.Dir(IMAGESPATH))))
```

Часть процесса скачивания является собой комбинацию функциональности `gorilla/mux` и обработчика Go для файловых серверов. Итак, внешний пакет предоставляет поддержку регулярных выражений и простой способ определить, что мы хотим использовать HTTP-метод `GET`, в то время как Go предлагает функциональность `http.FileServer()` для обслуживания файлов. В основном это так, поскольку мы обслуживаем файлы из локальной файловой системы. Однако ничто не запрещает нам написать собственную функцию-обработчик для скачивания двоичных файлов, когда нужно отклониться от поведения по умолчанию.

```
s := http.Server{
    Addr:     PORT,
    Handler:  mux,
    ErrorLog: nil,
    ReadTimeout: 5 * time.Second,
    WriteTimeout: 5 * time.Second,
    IdleTimeout: 10 * time.Second,
}

log.Println("Listening to", PORT)

err = s.ListenAndServe()
if err != nil {
    log.Printf("Error starting server: %s\n", err)
    return
}
}
```

Последняя часть утилиты посвящена запуску сервера с желаемыми параметрами. Поистине удивительно, что `main()` такая короткая, но делает так много полезного.

Теперь нам нужно инициализировать функциональность Go-модуля для запуска `binary.go`:

```
$ go mod init
$ go mod tidy
$ go mod download
```

Для работы с `binary.go` мы используем `curl(1)`:

```
$ curl -X PUT localhost:1234/files/packt.png --data-binary @packt.png
```

Итак, сначала мы загружаем файл `packt.png` на сервер, и он сохраняется как `packt.png` на стороне сервера. Следующая команда сохраняет на сервере тот же файл как `1.png`:

```
$ curl -X PUT localhost:1234/files/1.png --data-binary @packt.png
```

Скачать `1.png` на локальный компьютер как `downloaded.png` можно с помощью этой команды:

```
$ curl -X GET localhost:1234/files/1.png --output downloaded.png
```

Если вы забудете использовать `--output`, то `curl(1)` сгенерирует следующее сообщение об ошибке:

```
Warning: Binary output can mess up your terminal. Use "--output -" to tell
Warning: curl to output it to your terminal anyway, or consider
"--output"
Warning: <FILE>" to save to a file.
```

Наконец, если вы попытаетесь скачать файл, который не может быть найден, то `curl(1)` выдаст сообщение `404 page not found`.

Для вышеописанных взаимодействий `binary.go` сгенерировал следующий вывод:

```
2021/05/25 09:06:46 Creating: /tmp/files
2021/05/25 09:06:46 Listening to :1234
2021/05/25 09:10:21 packt.png
2021/05/25 09:10:21 Saving to /tmp/files/packt.png
2021/05/25 09:10:21 Bytes written: 733
```

Теперь, когда мы знаем, как создавать серверы RESTful, скачивать и выгружать файлы и определять REST API, пришло время научиться документировать REST API с помощью Swagger.

Использование Swagger для документации REST API

В этом разделе мы обсуждаем документацию REST API. Мы собираемся использовать спецификацию OpenAPI для документирования REST API. Данная спецификация, которая также называется спецификацией Swagger, представляет собой спецификацию для описания, создания, потребления и визуализации веб-сервисов RESTful.

Проще говоря, Swagger — это представление вашего RESTful API. Swagger считывает соответствующие *аннотации кода* и создает файл OpenAPI. Доку-

ментировать REST API с помощью Swagger вы можете, используя два варианта. Первый — написать файл спецификации OpenAPI самостоятельно (вручную); второй — добавить в исходный код аннотации, которые помогут Swagger сгенерировать файл спецификации OpenAPI автоматически.

Мы собираемся использовать `go-swagger`, который дает Go возможность работать с Swagger API. Дополнительный контент для создания документации для REST API помещается в исходные Go-файлы в виде комментариев Go. Утилита считывает эти комментарии и генерирует документацию! Однако все комментарии должны следовать определенным правилам и соответствовать поддерживаемой грамматике и соглашениям.

Сначала нам нужно установить двоичный файл `go-swagger`, следуя инструкциям на <https://goswagger.io/install.html>. Поскольку инструкции и версии время от времени меняются, не забывайте проверять наличие обновлений. Инструкции с этой веб-страницы устанавливают двоичный файл `swagger` в каталог `/usr/local/bin`, который подходит для внешних двоичных файлов. Однако вы можете указать и другое место, если каталог, в который вы его поместили, находится в вашей переменной `PATH`. После успешной установки запуск Swagger в командной строке должен сгенерировать следующее сообщение, в котором указаны команды, поддерживаемые `swagger`:

```
Please specify one command of: diff, expand, flatten, generate, init,
mixin, serve, validate or version
```

Вы также можете получить дополнительную справку для каждой команды `swagger` с помощью флага `--help`. Например, получить справку для команды `generate` можно путем простого запуска `swagger generate --help`:

```
$ swagger generate --help
Usage:
  swagger [OPTIONS] generate <command>

  generate go code for the swagger spec file

Application Options:
  -q, --quiet           silence logs
  --log-output=LOG-FILE redirect logs to file

Help Options:
  -h, --help             Show this help message

Available commands:
  cli      generate a command line client tool from the swagger spec
  client   generate all the files for a client library
  markdown generate a markdown representation from the swagger spec
  model    generate one or more models from the swagger spec
  operation generate one or more server operations from the swagger spec
```

```
server      generate all the files for a server application
spec        generate a swagger spec document from a go application
support     generate supporting files like the main function and the
            api builder
```

Далее вы узнаете, как документировать REST API, добавляя в исходный файл метаданные, относящиеся к Swagger.

Документирование REST API

В этом подразделе мы выясним, как документировать существующий REST API. По соображениям простоты мы используем относительно короткий файл, содержащий функции-обработчики. Итак, создадим в качестве репозитория новую папку `swagger` на <https://github.com/mactsouk/rest-api>, где будем хранить Go-файл с дополнительной информацией Swagger. В нашем случае мы создадим новую копию `handlers.go` в каталоге `swagger` и изменим ее. Имейте в виду: поскольку дело касается Go, то новая версия по-прежнему является действительным Go-пакетом, который может быть скомпилирован и использован без каких-либо проблем. Он просто содержит в Go-комментариях дополнительную информацию, связанную со Swagger.

Нет смысла отображать весь код новой версии `handlers.go`, который содержиться в Go-пакете `handlers`, — мы просто представим наиболее важные его части, начав с преамбулы исходного файла:

```
// Package handlers for the RESTful Server
//
// Documentation for REST API
//
// Schemes: http
// BasePath: /
// Version: 1.0.7
//
// Consumes:
// - application/json
//
// Produces:
// - application/json
//
// swagger:meta
```

Мы заявляем, что взаимодействуем с данными в формате JSON (`Consumes` и `Produces`), определяем версию и помещаем некоторые комментарии, описывающие общее назначение пакета. Тег `swagger:meta` сообщает двоичному файлу `swagger`, что это исходный файл с метаданными касательно API. Просто убеди-

тесь, что не забыли этот конкретный тег. Затем мы представляем документацию для структуры `User`, которая необходима для работы сервиса и будет *косвенно* использоваться дополнительными структурами, которые нам необходимо определить для целей генерации документации.

```
// User defines the structure for a Full User Record
//
// swagger:model
type User struct {
    // The ID for the user
    // in: body
    //
    // required: false
    // min: 1
    ID int `json:"id"`
```

Идентификатор пользователя предоставляется базой данных, что делает его необязательным полем, и имеет минимальное значение 1.

```
// The Username of the user
// in: body
//
// required: true
Username string `json:"username"`
```

Поле `Username` является обязательным.

```
// The Password of the user
//
// required: true
Password string `json:"password"`
```

Аналогично, поле `Password` является обязательным.

```
// The Last Login time of the User
//
// required: true
// min: 0
LastLogin int64 `json:"lastlogin"`

// Is the User Admin or not
//
// required: true
Admin int `json:"admin"`

// Is the User Logged In or Not
//
// required: true
Active int `json:"active"`
}
```

В конце концов вам нужно добавить комментарии ко всем полям структуры. Далее мы документируем конечную точку `/delete` вместе с ее функцией-обработчиком:

```
// swagger:route DELETE /delete/{id} DeleteUser deleteID
// Delete a user given their ID.
// The command should be issued by an admin user
```

В этой первой части мы говорим, что данная конечная точка работает с HTTP-методом `DELETE`, использует путь `/delete`, требует параметра `id` и будет отображаться на экране как `DeleteUser`. Последняя часть (`deleteID`) позволяет нам определить детали параметра `id`, который будет представлен через некоторое время.

```
// 
// responses:
// 200: noContent
// 404: ErrorMessage
```

В коде выше мы определяем два возможных ответа конечной точки. Оба варианта будут реализованы позже.

```
// DeleteHandler is for deleting users based on user ID
func DeleteHandler(rw http.ResponseWriter, r *http.Request) { }
```

Реализация обработчика для `/delete` опущена для краткости.

После этого мы задокументируем конечную точку `/logged` вместе с ее функцией-обработчиком:

```
// swagger:route GET /logged logged getUsersInfo
// Returns a list of logged in users
//
// responses:
// 200: UsersResponse
// 400: BadRequest
```

На этот раз первый ответ функции-обработчика называется `UsersResponse` и будет представлен через некоторое время. Если вы помните, этот обработчик возвращает срез элементов `User`.

```
// LoggedUsersHandler returns the list of all logged in users
func LoggedUsersHandler(rw http.ResponseWriter, r *http.Request) { }
```

Наконец, нам нужно определить структуры Go для представления различных входных данных и результатов взаимодействия — это в основном необходимо

для работы Swagger (обычно вы помещаете эти определения в отдельный файл, как правило, `docs.go`). Двумя наиболее важными такими Go-структурами являются следующие:

```
// swagger:parameters deleteID
type idParamWrapper struct {
    // The user id to be deleted
    // in: path
    // required: true
    ID int `json:"id"`
}
```

Это относится к конечной точке `/delete` и определяет переменную `ID`, которая указана в пути, следовательно, в строке `in: path`. Обратите внимание на использование `swagger:parameters`, за которым следует `deleteID`, который связывает эту конкретную структуру с документацией функции обработчика `/delete`.

```
// A User
// swagger:parameters getUserInfo loggedInfo
type UserInputWrapper struct {
    // A list of users
    // in: body
    Body User
}
```

На этот раз структура связана с двумя конечными точками, следовательно, используется как `getUserInfo`, так и `loggedInfo`. Каждая конечная точка должна быть связана с уникальным именем.

Эти вспомогательные структуры определяются один раз и являются той небольшой ценой, которую приходится платить за автоматическое создание документации. В следующем подразделе показано, как создать файл OpenAPI с учетом измененной версии `handlers.go`.

Создание файла документации

Теперь, когда у нас есть Go-файл с метаданными Swagger, мы готовы сгенерировать файл OpenAPI:

```
$ swagger generate spec --scan-models -o ./swagger.yaml
```

Эта команда просит `swagger` сгенерировать документ спецификации Swagger из Go-приложения, которое находится в каталоге, в котором мы и запускаем `swagger`. Параметр `--scan-models` дает `swagger` указание включать модели, которые были помечены с помощью `swagger:model`. Результатом команды будет файл

`swagger.yaml`, что указано с помощью параметра `-o`. Часть содержимого этого файла выглядит так (нет смысла пытаться понять все в представленном выводе):

```
/delete/{id}:
  delete:
    description: |-
      Delete a user given their ID
      The command should be issued by an admin user
    operationId: deleteID
    parameters:
      - description: The user id to be deleted
        format: int64
        in: path
        name: id
        required: true
        type: integer
        x-go-name: ID
    responses:
      "200":
        $ref: '#/responses/noContent'
      "404":
        $ref: '#/responses/ErrorMessage'
    tags:
      - DeleteUser
```

Вышеприведенная часть связана с конечной точкой `/delete`. В выходных данных говорится, что для конечной точки `/delete` требуется один параметр, который служит идентификатором удаляемого пользователя. Сервер возвращает HTTP-код `200` при успешном выполнении и `404` при сбое.

Не поленитесь взглянуть и на полную версию `swagger.yaml`. Однако мы еще не закончили. Нам нужно иметь возможность обслуживать этот генерированный файл с помощью веб-сервера. Этот процесс показан в следующем подразделе.

Обслуживание файла документации

Прежде чем мы начнем говорить об обслуживании файла Swagger, необходимо обсудить использование *функций промежуточного программного обеспечения* на простом, но полностью функциональном примере. Причина этого проста: инструмент `swagger` генерирует файл YAML, который необходимо правильно отобразить перед выводом на экран. Так что мы для этого используем ReDoc (<https://github.com/Redocly/redoc>). Однако нам нужен пакет `middleware` для размещения сайтов ReDoc — хотя работа выполняется прозрачно с помощью пакета `middleware`, полезно знать, что такое функции промежуточного программного обеспечения и что они делают. Это функции с небольшим объемом кода, которые получают запрос, делают с ним что-нибудь и передают его другому

промежуточному ПО или последней функции обработчика. Пакет `gorilla/mux` позволяет подключать одну или несколько таких функций к маршрутизатору с помощью `Router.Use()`. Если найдено совпадение, то соответствующие функции промежуточного программного обеспечения выполняются в том порядке, в котором они были добавлены к маршрутизатору (или подмаршрутизатору).

Важным кодом в файле `middleware.go` можно считать следующий:

```
mux := mux.NewRouter()
mux.Use(middleWare)

putMux := mux.Methods(http.MethodPut).Subrouter()
putMux.HandleFunc("/time", timeHandler)

getMux := mux.Methods(http.MethodGet).Subrouter()
getMux.HandleFunc("/add", addHandler)
getMux.Use(anotherMiddleWare)
```

Поскольку функция `middleWare()` добавлена к основному маршрутизатору (`mux.Use(middleWare)`), она всегда выполняется *перед* любой функцией промежуточного программного обеспечения подмаршрутизатора. Кроме того, `middleWare()` выполняется со всеми запросами, тогда как `anotherMiddleWare()` выполняется только для подмаршрутизатора `getMux`.

Теперь, когда у вас есть представление о функциях промежуточного программного обеспечения, вернемся к файлу `swagger.yaml`. Как уже говорилось ранее, его обслуживание требует добавления обработчика, который мы можем найти во внешнем пакете, что избавит нас от необходимости писать все с нуля. Мы стремимся к простоте, поэтому будем обслуживать `swagger.yaml` сам по себе в `./swagger/serve/swagger.go`, используя путь `/docs`. В приведенном ниже коде показана эта техника в реализации функции `main()`:

```
func main() {
    mux := mux.NewRouter()

    getMux := mux.Methods(http.MethodGet).Subrouter()
    opts := middleware.RedocOpts{SpecURL: "/swagger.yaml"}
```

Здесь мы определяем параметры функции промежуточного программного обеспечения, которая будет использоваться при обслуживании `/swagger.yaml`. Как обсуждалось ранее, эта функция отображает код YAML.

```
sh := middleware.Redoc(opts, nil)
```

Так мы определяем функцию-обработчик, основанную на функции промежуточного программного обеспечения. Данная функция промежуточного ПО не требует использования метода `Use()`.

```
getMux.Handle("/docs", sh)
```

Теперь нам осталось связать обработчик с `/docs`, и мы закончили!

```
getMux.Handle("/swagger.yaml", http.FileServer(http.Dir("./")))

s := http.Server{
    .
    .
}

log.Println("Listening to", PORT)
err := s.ListenAndServe()
if err != nil {
    log.Printf("Error starting server: %s\n", err)
    return
}
}
```

Остальная часть кода предназначена для определения параметров сервера и его запуска.

На рис. 10.1 показана визуализация файла `swagger.yaml`, отображаемая в браузере Firefox, а также информация о конечной точке `/getid`, требуемой полезной нагрузке и ожидаемых ответах.

The screenshot shows the Swagger UI interface. On the left, there is a detailed schema for the `loggedInfo` endpoint, which returns the ID of a User given their username and password. The schema includes fields for `active`, `admin`, `id`, `lastlogin`, `password`, and `username`. On the right, there is a detailed view of the `GET /getid` endpoint, showing the request samples section. It specifies the `Content type` as `application/json` and provides a JSON payload example:

```
{
  "active": 0,
  "admin": 0,
  "id": 1,
  "lastlogin": 0,
  "password": "string",
  "username": "string"
}
```

Рис. 10.1. Создание документации для RESTful API

К сожалению, дальнейшее обсуждение Swagger и `go-swagger` выходит за рамки этой книги. Как и ранее, вам придется поэкспериментировать с инструментами, чтобы добиться желаемых результатов. Основная мысль здесь состоит в том, что наличие сервиса RESTful без надлежащей документации будет не самым приятным сюрпризом для потенциальных разработчиков.

Упражнения

- Включите функциональность `binary.go` в собственный RESTful-сервер.
- Измените пакет `restdb` так, чтобы он поддерживал SQLite вместо PostgreSQL.
- Измените пакет `restdb` так, чтобы он поддерживал MySQL вместо PostgreSQL.
- Поместите функции обработчика из `handlers.go` в отдельные файлы.

Резюме

Го широко используется для разработки клиентов и серверов RESTful. В этой главе показано, как программировать профессиональные клиенты и серверы RESTful на Go и документировать REST API с помощью Swagger. Помните, что определение надлежащего REST API и внедрение сервера и клиентов для него — это процесс, который требует времени, а также небольших подстроек и изменений.

Глава 11 посвящена тестированию кода, бенчмаркингу, профилированию, кросс-компиляции и созданию примеров функций. Среди прочего, мы напишем код, который позволяет тестировать HTTP-обработчики, разработанные в текущей главе, и создавать случайные входные данные в целях тестирования.

Дополнительные ресурсы

- Больше информации о пакете `gorilla/mux`: <https://github.com/gorilla/mux> и <https://www.gorillatoolkit.org/pkg/mux>.
- Библиотека `go-querystring` предназначена для кодирования Go-структур в параметры запроса URL: <https://github.com/google/go-querystring>.
- Больше информации о Swagger: <https://swagger.io/>.
- Go Swagger 2.0: <https://goswagger.io/>.

- Спецификация OpenAPI: <https://www.openapis.org/>.
- Если вам требуется проверять входные данные в формате JSON, то ознакомьтесь с Go-пакетом `validator` по адресу <https://github.com/go-playground/validator>.
- Возможно, утилита командной строки `jq(1)` покажется вам довольно удобной при работе с записями в формате JSON: <https://stedolan.github.io/jq/> и <https://jqplay.org/>.
- Вы можете просмотреть файлы OpenAPI онлайн по адресу <https://editor.swagger.io/>.

11

Тестирование и профилирование кода

Темы этой главы полезны и важны, особенно для тех, кто заинтересован в повышении производительности программ на Go и обнаружении ошибок. В этой главе в основном рассматриваются *оптимизация, тестирование и профилирование* кода.

Оптимизация кода — это процесс, при котором один или несколько разработчиков пытаются заставить определенные части программы работать быстрее, эффективнее или же использовать меньше ресурсов. Проще говоря, оптимизация кода заключается в устранении узких мест, которые влияют на производительность.

Во время тестирования проверяется, делает ли ваш код то, что вы от него хотели. В этой главе вы познакомитесь с методом тестирования Go-кода. Добавлять тестовый код лучше всего во время разработки, так как это поможет выявить ошибки на самых ранних этапах. Термин «профилирование кода» относится к измерению определенных аспектов программы в целях получения точного понимания того, как работает код. Результаты профилирования помогут вам решить, какие части вашего кода необходимо изменить.

Имейте в виду, что при написании кода следует сосредоточиться на его *корректности*, а также на других желаемых свойствах, таких как удобочитаемость, простота и удобство сопровождения, а не на его *производительности*. Только удостоверившись, что код правильный, возможно, стоит обратить

внимание на его производительность. Хороший способ повысить производительность — выполнять код на более медленных машинах, чем те, которые будут использоваться в реальности.

В этой главе:

- оптимизация кода;
- оценка производительности;
- профилирование кода;
- утилита трассировки `go tool`;
- трассировка веб-сервера;
- тестирование Go-кода;
- кросскомпиляция;
- использование `go:generate`;
- создание примеров функций.

Оптимизация кода

Оптимизация кода — одновременно и искусство, и наука. Это означает, что не существует определенного способа оптимизировать ваш код и что вам придется напрячь мозги и испробовать множество подходов, если вы хотите сделать свой код более быстрым. Однако общий принцип, касающийся оптимизации кода, заключается в том, чтобы *сначала сделать его правильным, а затем уже быстрым*. Всегда помните, что сказал об оптимизации Дональд Кнут:

«Реальная проблема заключается в том, что программисты тратят слишком много времени, беспокоясь об эффективности в неподходящих местах и в неподходящее время; преждевременная оптимизация — источник всех бед (или, по крайней мере, большей их части) в программировании».

Кроме того, помните, что говорил об оптимизации покойный Джо Армстронг, один из разработчиков Erlang:

«Сделай так, чтобы код работал, затем сделай его красивым, а затем, если тебе это очень-очень нужно, сделай его быстрым. В 90 % случаев, сделав код красивым, ты сделаешь его быстрым. Так что просто сделай его красивым!»

Если вы действительно увлекаетесь оптимизацией кода, то, возможно, вам стоит прочитать книгу *Compilers: Principles, Techniques and Tools* Альфреда В. Ахо, Моники С. Лам, Рави Сети и Джонни Д. Ульмана (Pearson Education Limited,

2014)¹, основное внимание в которой уделяется созданию компилятора. Кроме того, все тома серии «Искусство программирования» Дональда Кнута послужат отличным источником информации по всем аспектам программирования, если у вас найдется время их прочитать.

Следующий раздел посвящен оценке производительности Go-кода, которая поможет вам определить, какая часть вашего кода быстрее, а какая медленнее — и это идеальная точка старта.

Оценка производительности

Бенчмаркинг измеряет производительность функции или программы, позволяя вам сравнивать реализации и оценивать влияние изменений кода на производительность. Используя эту информацию, вы можете легко выявить ту часть кода, которую необходимо переписать, чтобы повысить его общую производительность. Само собой разумеется, что вам не следует измерять производительность Go-кода на загруженной машине, которая в настоящее время используется для других, более важных целей (если, конечно, у вас нет очень веской причины для этого)! В противном случае вы можете помешать процессу бенчмаркинга и получить неточные результаты, но самое главное, вызвать проблемы с производительностью на компьютере.



В большинстве случаев загрузка операционной системы играет ключевую роль в производительности вашего кода. Позвольте рассказать вам историю: Java-утилита, которую я разработал для проекта, выполняет множество вычислений и завершает работу за 6,242 секунды при самостоятельном запуске. Потребовалось около дня, чтобы четыре экземпляра одной и той же утилиты командной строки Java отработали на одной и той же машине Linux. Если задуматься, то запускать их одну за другой было бы быстрее, чем запускать одновременно.

Go следует определенным соглашениям, касающимся анализа производительности. Наиболее важным соглашением является то, что имя контрольной функции должно начинаться со слова `Benchmark`. После него мы можем поставить знак подчеркивания или заглавную букву. Следовательно, как `BenchmarkFunctionName()`, так и `Benchmark_functionName()` являются допустимыми контрольными функциями, в то время как `Benchmarkfunctionname()` таковой не является. То же правило применяется к функциям тестирования, которые начинаются

¹ Альфред В. Ахо и др. Компиляторы: принципы, технологии и инструментарий.

с `Test`. Хотя и разрешается помещать код тестирования и бенчмаркинга в один файл с другим кодом, все же этого следует избегать. По соглашению такие функции помещаются в файлы, которые заканчиваются на `_test.go`. Если код бенчмаркинга или тестовый код верны, то подкоманда `go test` выполнит за вас всю грязную работу, которая включает сканирование всех файлов `*_test.go` на наличие специальных функций, создание надлежащего временного пакета `main`, вызов этих специальных функций, получение результатов и генерацию окончательного вывода.

Начиная с Go 1.17 мы можем перетасовывать порядок выполнения *как тестов, так и оценок* с помощью параметра `shuffle` (`go test -shuffle=on`). Он принимает значение, которое является исходным для генератора случайных чисел, и может оказаться полезным, если нужно воспроизвести порядок выполнения. Его значение по умолчанию — `off`. Логика, лежащая в основе этой возможности, заключается в том, что иногда порядок выполнения тестов и бенчмарков влияет на их результаты.

Переписывание функции `main()` для более качественного тестирования

Существует умный способ, с помощью которого можно переписать каждую функцию `main()` так, чтобы значительно упростить тестирование и бенчмаркинг. Она содержит ограничение, которое заключается в том, что вы не можете вызвать ее из тестового кода. Этот метод представляет собой решение данной проблемы с помощью кода, который находится в файле `main.go`. Блок `import` опущен для экономии места.

```
Func main() {
    err := run(os.Args, os.Stdout)
    if err != nil {
        fmt.Printf("%s\n", err)
        return
    }
}
```

Поскольку у нас не может быть исполняемой программы без функции `main()`, нам придется написать минимальную программу. Что делает `main()`, так это вызывает `run()`, которая является нашей настроенной версией `main()`, отправляет туда `os.Args` и получает из `run()` возвращаемое значение.

```
Func run(args []string, stdout io.Writer) error {
    if len(args) == 1 {
        return errors.New("No input!")
    }
}
```

```
// продолжаем реализацию run(),
// как будто это main()

return nil
}
```

Как обсуждалось ранее, функция `run()` или любая другая функция, подобным образом вызываемая из `main()`, заменяет `main()`, добавляя преимущество, заключающееся в возможности вызова тестовыми функциями. Проще говоря, функция `run()` содержит код, который в ином случае был бы расположен в `main()`. Единственное различие состоит в том, что `run()` возвращает переменную ошибки, что невозможно с `main()`, которая может возвращать операционной системе только коды выхода. Можно заметить, что это создает немного больший стек из-за дополнительного вызова функции, но преимущества здесь более важны, чем это дополнительное использование памяти. Вы можете опустить второй параметр (`stdout io.Writer`), используемый для перенаправления сгенерированного вывода, но первый все же важен, поскольку позволяет передавать аргументы командной строки в `run()`.

При выполнении `main.go` мы получаем такой вывод:

```
$ go run main.go
No input!
$ go run main.go some input
```

В том, как работает `main.go`, нет ничего особенного. Хорошо то, что вы можете вызвать функцию `run()` из любого места, где захотите, включая код, который пишете для тестирования, и передать в нее нужные параметры! Хорошо иметь в виду этот метод, поскольку однажды он может вас спасти.

Предметом обсуждения в следующем подразделе является оценка производительности буферизованной записи.

Анализ производительности буферизованной записи и чтения

В этом подразделе мы будем выяснять, играет ли размер буфера ключевую роль в производительности операций записи. Это также даст нам возможность обсудить *табличные тесты*, а также использование папки `testdata`, которая зарезервирована Go для хранения файлов, использующихся во время бенчмаркинга. Как табличные тесты, так и папка `testdata` могут использоваться и в функциях тестирования.



Функции бенчмарка используют переменные `testing.B`, тогда как функции тестирования — переменные `testing.T`. Очень легко запомнить.

Код `table_test.go` следующий:

```
package table

import (
    "fmt"
    "os"
    "path"
    "strconv"
    "testing"
)

var ERR error
var countChars int

func benchmarkCreate(b *testing.B, buffer, filesize int) {
    filename := path.Join(os.TempDir(), strconv.Itoa(buffer))
    filename = filename + "-" + strconv.Itoa(filesize)
    var err error
    for I := 0; I < b.N; i++ {
        err = Create(filename, buffer, filesize)
    }
    ERR = err
}
```

А теперь немного *важной информации*, касающейся бенчмаркинга. По умолчанию каждая такая функция выполняется *не менее* одной секунды — в эту продолжительность также входит время выполнения функций, вызываемых функцией бенчмаркинга. Если контрольная функция возвращается за время, которое меньше одной секунды, то значение `b.N` увеличивается и функция выполняется снова в общей сложности столько раз, сколько сохранено в переменной `b.N`. В первый раз значение `b.N` равно 1, затем оно становится 2, потом 5, 10, 20, 50 и т. д. Это происходит потому, что чем быстрее функция, тем больше раз Go должен ее запустить, чтобы получить *точные результаты*.

Причину сохранения возвращаемого значения `Create()` в переменной `err` и последующего использования другой глобальной переменной `ERR` объяснить не просто. Мы хотим запретить компилятору выполнять какие-либо оптимизаций, которые могут исключить выполнение функции, которую мы хотим измерить, поскольку ее результаты никогда не используются.

```
Err = os.Remove(filename)
if err != nil {
    fmt.Println(err)
}
ERR = err
}
```

Ни сигнатура, ни название `benchmarkCreate()` не делают ее функцией бенчмаркинга. Это вспомогательная функция,зывающая `Create()`, которая создает

новый файл на диске, а ее реализацию можно найти в файле `table.go` с соответствующими параметрами. Ее реализация действительна, и она может быть использована бенчмарк-функциями.

```
Func BenchmarkBuffer4Create(b *testing.B) {
    benchmarkCreate(b, 4, 1000000)
}

func BenchmarkBuffer8Create(b *testing.B) {
    benchmarkCreate(b, 8, 1000000)
}

func BenchmarkBuffer16Create(b *testing.B) {
    benchmarkCreate(b, 16, 1000000)
}
```

Это три правильно определенные бенчмарк-функции, каждая из которых вызывает `benchmarkCreate()`. Такие функции требуют единой переменной `*testing.B` и не возвращают никаких значений. В этом случае цифры в конце имени функции указывают на размер буфера.

```
Func BenchmarkRead(b *testing.B) {
    buffers := []int{1, 16, 96}
    files := []string{"10.txt", "1000.txt", "5k.txt"}
```

Это код, определяющий структуры массивов, которые будут использоваться в табличных тестах.

```
For _, filename := range files {
    for _, bufSize := range buffers {
        name := fmt.Sprintf("%s-%d", filename, bufSize)
        b.Run(name, func(b *testing.B) {
            for I := 0; I < b.N; i++ {
                t := CountChars("./testdata/"+filename,
                                bufSize)
                countChars = t
            }
        })
    }
}
```

Метод `b.Run()`, который позволяет запускать один или несколько *субтестов в рамках бенчмарк-функции*, принимает два параметра. Во-первых, название суббенчмарка, которое отображается на экране, и во-вторых, функцию, реализующую суббенчмарк. Это правильный способ запуска нескольких бенчмарков с помощью табличных тестов. Просто не забудьте определить правильное имя для каждого суббенчмарка, поскольку оно будет отображаться на экране.

При выполнении бенчмарков мы получаем такой вывод:

```
$ go test -bench=. *.go
```

Здесь есть два важных момента. Во-первых, значение параметра `-bench` определяет бенчмарк-функции, которые будут выполняться. Использующееся значение `.` — это регулярное выражение, которое соответствует всем допустимым бенчмарк-функциям. Во-вторых, если вы опустите параметр `-bench`, то никакие бенчмарк-функции выполнятся не будут.

```
Goos: darwin
goarch: amd64
cpu: Intel CoreI i7-4790K CPU @ 4.00GHz
BenchmarkBuffer4Create-8           78212    12862 ns/op
BenchmarkBuffer8Create-8          145448    7929 ns/op
BenchmarkBuffer16Create-8         222421   5074 ns/op
```

Три вышеприведенные строки представляют собой результаты бенчмарк-функций `BenchmarkBuffer4Create()`, `BenchmarkBuffer8Create()` и `BenchmarkBuffer16Create()` соответственно и показывают их производительность.

<code>BenchmarkRead/10.txt-1-8</code>	78852	17268 ns/op
<code>BenchmarkRead/10.txt-16-8</code>	84225	14161 ns/op
<code>BenchmarkRead/10.txt-96-8</code>	92056	14966 ns/op
<code>BenchmarkRead/1000.txt-1-8</code>	2821	395419 ns/op
<code>BenchmarkRead/1000.txt-16-8</code>	21147	56148 ns/op
<code>BenchmarkRead/1000.txt-96-8</code>	58035	20362 ns/op
<code>BenchmarkRead/5k.txt-1-8</code>	600	1901952 ns/op
<code>BenchmarkRead/5k.txt-16-8</code>	4893	239557 ns/op
<code>BenchmarkRead/5k.txt-96-8</code>	19892	57309 ns/op

Эти результаты взяты из табличных тестов с девятью суббенчмарками.

О чём же нам говорит этот вывод? Для начала `-8` в конце каждой бенчмарк-функции означает количествоgorутин, используемых для ее выполнения, что, по сути, является значением переменной среды `GOMAXPROCS`. Аналогично вы можете видеть значения `GOOS` и `GOARCH`, которые показывают операционную систему и архитектуру машины. Во втором столбце выходных данных отображается количество выполнений соответствующей функции. Более быстрые функции выполняются больше раз, чем более медленные. Например, `BenchmarkBuffer4Create()` выполнилась 78212 раз, в то время как `BenchmarkBuffer16Create()` — 222421 раз, поскольку она быстрее! Третий столбец в выходных данных показывает среднее время каждого запуска, которое изменяется в наносекундах, затраченных на выполнение бенчмарк-функции (`ns/op`). Чем больше значение третьего столбца, тем медленнее работает функция. Боль-

шое значение в третьем столбце указывает на то, что функцию, возможно, потребуется оптимизировать.

PASS	ok	command-line-arguments	44.756s
------	----	------------------------	---------

Если вы хотите включить в выходные данные статистику распределения памяти, то можете включить в команду `-benchmem`:

BenchmarkBuffer4Create-8	91651	11580 ns/op	304 B/op	5 allocs/op
BenchmarkBuffer8Create-8	170814	6202 ns/op	304 B/op	5 allocs/op

Сгенерированный вывод аналогичен полученному без параметра командной строки `-benchmem`, но включает в себя два дополнительных столбца. Четвертый столбец показывает объем памяти, который был выделен в среднем при каждом выполнении бенчмарк-функции. В пятом столбце показано количество выделений, использованных для выделения объема памяти в четвертом столбце.

На данный момент мы научились создавать бенчмарк-функции для тестирования производительности наших собственных функций, чтобы лучше понять потенциальные узкие места, которые, возможно, стоит оптимизировать. Вы можете спросить: *как часто нам нужно писать бенчмарк-функции?* Ответ прост: когда что-то работает медленнее, чем необходимо, и/или когда нужно выбрать между двумя или более реализациями.

В следующем подразделе показано, как сравнивать результаты бенчмарка.

Утилита `benchstat`

Теперь представьте, что у вас есть данные бенчмаркинга и вы хотите сравнить их с результатами, полученными на другом компьютере или на другой конфигурации. Здесь вам и пригодится утилита `benchstat`. Ее можно найти в пакете `golang.org/x/perf/cmd/benchstat` и скачать с помощью `go get -u golang.org/x/perf/cmd/benchstat`. Go помещает все двоичные файлы в `~/go/bin`, и `benchstat` не является исключением.



Утилита `benchstat` заменяет утилиту `benchcmp`, которую можно найти по адресу <https://pkg.go.dev/golang.org/x/tools/cmd/benchcmp>.

Итак, представьте, что у нас есть два контрольных результата для `table_test.go`, сохраненных в `r1.txt` и `r2.txt`. Вы должны удалить все строки из выходных

данных `go test`, которые не содержат результатов бенчмаркинга, в итоге останутся все строки, начинаяющиеся с `Benchmark`. Вы можете использовать `benchstat` следующим образом:

```
$ ~/go/bin/benchstat r1.txt r2.txt
name      old time/op  new time/op  delta
Buffer4Create-8  10.5µs ± 0%  0.8µs ± 0%  ~   (p=1.000 n=1+1)
Buffer8Create-8  6.88µs ± 0%  0.79µs ± 0%  ~   (p=1.000 n=1+1)
Buffer16Create-8 5.01µs ± 0%  0.78µs ± 0%  ~   (p=1.000 n=1+1)
Read/10.txt-1-8  15.0µs ± 0%  4.0µs ± 0%  ~   (p=1.000 n=1+1)
Read/10.txt-16-8 12.2µs ± 0%  2.6µs ± 0%  ~   (p=1.000 n=1+1)
Read/10.txt-96-8 11.9µs ± 0%  2.6µs ± 0%  ~   (p=1.000 n=1+1)
Read/1000.txt-1-8 381µs ± 0%  174µs ± 0%  ~   (p=1.000 n=1+1)
Read/1000.txt-16-8 54.0µs ± 0%  22.6µs ± 0%  ~   (p=1.000 n=1+1)
Read/1000.txt-96-8 19.1µs ± 0%  6.2µs ± 0%  ~   (p=1.000 n=1+1)
Read/5k.txt-1-8  1.81ms ± 0%  0.89ms ± 0%  ~   (p=1.000 n=1+1)
Read/5k.txt-16-8 222µs ± 0%  108µs ± 0%  ~   (p=1.000 n=1+1)
Read/5k.txt-96-8 51.5µs ± 0%  21.5µs ± 0%  ~   (p=1.000 n=1+1)
```

Если значение столбца `delta` равно `~` (как здесь), то это говорит о том, что существенных изменений в результатах не произошло. В этом выводе не показываются различия между двумя результатами. Подробное обсуждение `benchstat` выходит за рамки данной книги. Введите `benchstat -h`, чтобы узнать больше о поддерживаемых параметрах.

В следующем подразделе мы затронем чувствительную тему, а именно *неправильное определение бенчмарк-функций*.

Неправильно определенные бенчмарк-функции

Вы должны быть очень осторожны при определении бенчмарк-функций, поскольку существует возможность определить их неправильно. Посмотрите на Go-код следующей функции:

```
func BenchmarkFiboI(b *testing.B) {
    for I := 0; I < b.N; i++ {
        _ = fibo1(i)
    }
}
```

Функция `BenchmarkFibo()` имеет допустимое имя и правильную сигнатуру. Плохая новость состоит в том, что эта бенчмарк-функция логически неверна и не даст никаких результатов. Причина же заключается в том, что по мере

увеличения значения `b.N` способом, описанным ранее, время выполнения функции также увеличивается из-за цикла `for`. Этот факт не позволяет функции `BenchmarkFiboI()` сходиться к стабильному числу, что препятствует ее завершению и, следовательно, возврату каких-либо результатов. По сходным причинам следующая бенчмарк-функция также реализована неправильно:

```
func BenchmarkFiboII(b *testing.B) {
    for I := 0; I < b.N; i++ {
        _ = fibo1(b.N)
    }
}
```

Вместе с тем в реализации следующих двух бенчмарк-функций никаких проблем нет:

```
func BenchmarkFiboIV(b *testing.B) {
    for I := 0; I < b.N; i++ {
        _ = fibo1(10)
    }
}

func BenchmarkFiboIII(b *testing.B) {
    _ = fibo1(b.N)
}
```

Правильные бенчмарк-функции — это инструмент для выявления узких мест в вашем коде, который вы должны применять в собственных проектах, особенно при работе с файловым вводом-выводом или операциями с интенсивным использованием процессора. На момент написания этих строк я уже *три дня* жду завершения работы программы на Python, проверяющей производительность математического алгоритма методом грубой силы. Ну что же, хватит об анализе производительности. В следующем разделе мы обсудим профилирование кода.

Профилирование кода

Профилирование — это процесс динамического анализа программы, который измеряет различные значения, связанные с выполнением программы, что позволяет вам лучше понять ее поведение. В этом разделе мы собираемся узнать, как профилировать Go-код, чтобы лучше понять его и повысить производительность. Иногда профилирование кода может даже выявить ошибки в коде, такие как бесконечный цикл или функции, которые никогда не возвращаются.

Стандартный Go-пакет `runtime/pprof` служит для профилирования всех видов приложений, кроме HTTP-серверов. Высокоуровневый пакет `net/http/pprof` следует использовать, когда необходимо профилировать веб-приложение, написанное на Go. Вы можете просмотреть страницу справки инструмента `pprof`, выполнив команду `go tool pprof -help`.

В следующем подразделе будет показано, как профилировать приложение командной строки, а далее продемонстрировано профилирование HTTP-сервера.

Профилирование приложения командной строки

Код приложения сохранен как файл `profileCla.go` и собирает данные профилирования процессора и памяти. Что в нем интересно, так это реализация функции `main()`. Именно здесь происходит сбор данных профилирования:

```
func main() {
    cpuFilename := path.Join(os.TempDir(), "cpuProfileCla.out")
    cpuFile, err := os.Create(cpuFilename)
    if err != nil {
        fmt.Println(err)
        return
    }
    pprof.StartCPUProfile(cpuFile)
    defer pprof.StopCPUProfile()
```

Код выше производит сбор данных профилирования процессора. `pprof.StartCPUProfile()` запускает сбор, который затем останавливается с помощью вызова метода `pprof.StopCPUProfile()`. Все данные сохраняются в файле `cpuProfileCla.out` в каталоге `os.TempDir()`. Это поведение зависит от используемой операционной системы и делает код *переносимым*. Использование `defer` означает, что метод `pprof.StopCPUProfile()` будет вызван непосредственно перед выходом `main()`.

```
Total := 0
for I := 2; I < 100000; i++ {
    n := N1(i)
    if n {
        total = total + 1
    }
}
fmt.Println("Total primes:", total)

total = 0
for I := 2; I < 100000; i++ {
```

```
n := N2(i)
if n {
    total = total + 1
}
}
fmt.Println("Total primes:", total)

for I := 1; I < 90; i++ {
    n := fibo1(i)
    fmt.Print(n, " ")
}
fmt.Println()

for I := 1; I < 90; i++ {
    n := fibo2(i)
    fmt.Print(n, " ")
}
fmt.Println()
runtime.GC()
```

Весь вышеприведенный код выполняет множество вычислений, интенсивно используя ЦП, чтобы профилировщик ЦП мог собирать данные — именно сюда обычно осуществляется переход из вашего кода.

```
// Профилирование памяти!
memoryFilename := path.Join(os.TempDir(), "memoryProfileCla.out")
memory, err := os.Create(memoryFilename)
if err != nil {
    fmt.Println(err)
    return
}
defer memory.Close()
```

Мы создаем второй файл для сбора данных профилирования, связанных с памятью.

```
For I := 0; I < 10; i++ {
    s := make([]byte, 50000000)
    if s == nil {
        fmt.Println("Operation failed!")
    }
    time.Sleep(50 * time.Millisecond)
}

err = pprof.WriteHeapProfile(memory)
if err != nil {
    fmt.Println(err)
    return
}
}
```

Функция `pprof.WriteHeapProfile()` записывает данные из памяти в указанный файл. Повторюсь, мы выделяем много памяти для профилировщика памяти, чтобы иметь данные для сбора.

При запуске программы `profileCla.go` создаются два файла в папке, возвращаемой `os.TempDir()`, — обычно мы сохраняем их в другой папке. Смело меняйте код `profileCla.go` и помещайте файлы профилирования в другое место. Итак, что же нам делать дальше? Для обработки полученных файлов мы должны использовать `go tool pprof`:

```
$ go tool pprof /path/ToTemporary/Directory/cpuProfileCla.out
(pprof) top
Showing nodes accounting for 5.65s, 98.78% of 5.72s total
Dropped 47 nodes (cum <= 0.03s)
Showing top 10 nodes out of 18
      flat  flat%    sum%      cum   cum%
  3.27s 57.17% 57.17%    3.65s 63.81% main.N2 (inline)
```

Команда `top` возвращает сводку из десяти лучших записей.

```
(pprof) top10 -cum
Showing nodes accounting for 5560ms, 97.20% of 5720ms total
Dropped 47 nodes (cum <= 28.60ms)
Showing top 10 nodes out of 18
      flat  flat%    sum%      cum   cum%
  80ms  1.40%  1.40%    5660ms 98.95% main.main
      0     0%  1.40%    5660ms 98.95% runtime.main
```

Команда `top10 -cum` возвращает совокупное время для каждой функции.

```
(pprof) list main.N1
list main.N1
Total: 5.72s
ROUTINE ===== main.N1 in /Users/mtsouk/ch11/
profileCla.go
  1.72s    1.83s (flat, cum) 31.99% of Total
    .        .    35:func N1(n int) bool {
    .        .    36:    k := math.Floor(float64(n/2 + 1))
  50ms    60ms    37:    for I := 2; I < int(k); i++ {
  1.67s    1.77s    38:        if (n % i) == 0 {
```

Наконец, команда `list` отображает информацию о данной функции. В этом выводе показано, что оператор `if (n % i) == 0` ответствен за большую часть времени, которое требуется для выполнения `N1()`.

Для краткости мы не показываем полный вывод этих команд. Попробуйте команды профилирования самостоятельно в собственном коде, чтобы увидеть полные результаты их работы. Посетите блог Go по адресу <https://blog.golang.org/pprof>, чтобы узнать больше о профилировании.



Вы также можете создать выходные данные профилирования в формате PDF из командной строки профилировщика Go с помощью команды `pdf`. Лично я в большинстве случаев начинаю с этой команды, поскольку она предоставляет мне полный обзор собранных данных.

В следующем подразделе обсудим, как профилировать HTTP-сервер.

Профилирование HTTP-сервера

Как обсуждалось ранее, пакет `net/http/pprof` следует использовать, когда вы хотите собрать данные профилирования для Go-приложения, которое запускает HTTP-сервер. С этой целью при импорте `net/http/pprof` устанавливаются различные обработчики по пути `/debug/pprof/`. Вскоре мы поговорим об этом подробнее. На данный момент достаточно помнить, что пакет `net/http/pprof` следует использовать для профилирования веб-приложений, тогда как `runtime/pprof` — для профилирования всех других типов приложений.

Этот метод показан в файле `profileHTTP.go`, где содержится следующий код:

```
package main

import (
    "fmt"
    "net/http"
    "net/http/pprof"
    "os"
    "time"
)
```

Как уже обсуждалось ранее, необходимо импортировать пакет `net/http/pprof`.

```
func myHandler(w http.ResponseWriter, r *http.Request) {
    fmt.Fprintf(w, "Serving: %s\n", r.URL.Path)
    fmt.Printf("Served: %s\n", r.Host)
}

func timeHandler(w http.ResponseWriter, r *http.Request) {
    t := time.Now().Format(time.RFC1123)
    Body := "The current time is:"
    fmt.Fprintf(w, "%s %s", Body, t)
    fmt.Fprintf(w, "Serving: %s\n", r.URL.Path)
    fmt.Printf("Served time for: %s\n", r.Host)
}
```

Две вышеприведенные функции реализуют два обработчика, которые будут использоваться на нашем наивном HTTP-сервере. Функция `MyHandler()`

является функцией обработчика по умолчанию, тогда как `timeHandler()` возвращает текущее время и дату на сервере.

```
func main() {
    PORT := ":8001"
    arguments := os.Args
    if len(arguments) == 1 {
        fmt.Println("Using default port number: ", PORT)
    } else {
        PORT = ":" + arguments[1]
        fmt.Println("Using port number: ", PORT)
    }

    r := http.NewServeMux()
    r.HandleFunc("/time", timeHandler)
    r.HandleFunc("/", myHandler)
}
```

До этого момента не происходит ничего особенного, поскольку мы просто регистрируем функции обработчика.

```
r.HandleFunc("/debug/pprof/", pprof.Index)
r.HandleFunc("/debug/pprof/cmdline", pprof.Cmdline)
r.HandleFunc("/debug/pprof/profile", pprof.Profile)
r.HandleFunc("/debug/pprof/symbol", pprof.Symbol)
r.HandleFunc("/debug/pprof/trace", pprof.Trace)
```

Все вышеприведенные операторы устанавливают обработчики для профилировщика HTTP — вы можете получить к ним доступ, используя имя хоста и номер порта веб-сервера. Вам не обязательно использовать все обработчики.

```
err := http.ListenAndServe(PORT, r)
if err != nil {
    fmt.Println(err)
    return
}
}
```

Наконец, мы запускаем HTTP-сервер обычным способом.

Что дальше? Сначала мы запускаем HTTP-сервер (`go run profileHTTP.go`). После этого запускаем следующую команду для сбора данных профилирования *при взаимодействии с HTTP-сервером*:

```
$ go tool pprof http://localhost:8001/debug/pprof/profile
Fetching profile over HTTP from http://localhost:8001/debug/pprof/
profile
Saved profile in /Users/mtsouk/pprof/pprof.samples.cpu.004.pb.gz
Type: cpu
Time: Jun 18, 2021 at 12:30pm (EEST)
Duration: 30s, Total samples = 10ms (0.033%)
Entering interactive mode (type "help" for commands, "o" for options)
(pprof) %
```

В этом выводе показан начальный экран HTTP-профилировщика — доступные команды те же, что и при профилировании приложения командной строки.

Вы можете либо выйти из командной оболочки и проанализировать свои данные позже с помощью `go tool pprof`, либо продолжить отдавать команды профилировщику. Это общая идея, лежащая в основе профилирования HTTP-серверов в Go.

В следующем подразделе мы обсудим веб-интерфейс профилировщика Go.

Веб-интерфейс профилировщика Go

Хорошая новость в том, что начиная с версии Go 1.10 `go tool pprof` поставляется с веб-интерфейсом пользователя, который вы можете запустить как `go tool pprof -http = [host]:[port] aProfile.out` (не забудьте указать правильные значения `-http`).

Часть веб-интерфейса профилировщика видна на рис. 11.1. Там показано, на что было потрачено время выполнения программы.

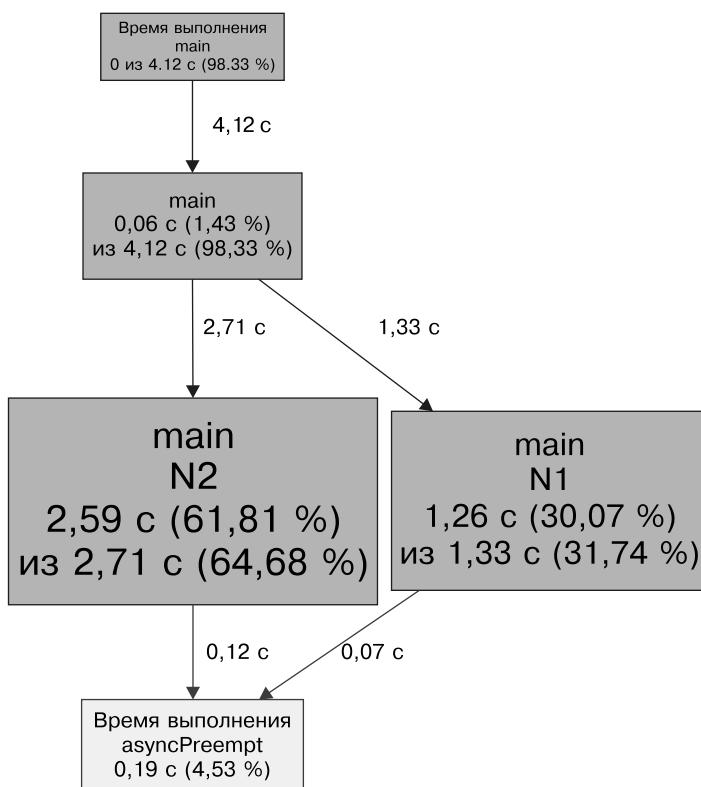


Рис. 11.1. Веб-интерфейс профилировщика Go

Изучите веб-интерфейс и различные предлагаемые опции. К сожалению, более подробное обсуждение профилирования выходит за рамки этой главы. Как всегда, если вы действительно заинтересованы в профилировании кода, то экспериментируйте с ним как можно больше.

Следующий раздел посвящен трассировке кода.

Утилита `go tool trace`

Трассировка кода — это процесс, который позволяет изучать такую информацию, как работа сборщика мусора, время жизни подпрограмм, активность каждого логического процессора и количество используемых потоков операционной системы. Утилита `go tool trace` — это инструмент для просмотра данных, хранящихся в файлах трассировки, которые могут быть сгенерированы любым из таких трех способов:

- с помощью пакета `runtime/trace`;
- с помощью пакета `net/http/pprof`;
- с помощью команды `go test -trace`.

В этом разделе показано использование первого способа с помощью кода в `traceCLA.go`:

```
package main

import (
    "fmt"
    "os"
    "path"
    "runtime/trace"
    "time"
)
```

Пакет `runtime/trace` необходим для сбора всех видов данных трассировки — нет смысла отбирать конкретные данные, поскольку все они взаимосвязаны.

```
func main() {
    filename := path.Join(os.TempDir(), "traceCLA.out")
    f, err := os.Create(filename)
    if err != nil {
        panic(err)
    }
    defer f.Close()
```

Как и в случае профилирования, нам нужно создать файл для хранения данных трассировки. Наш файл называется `traceCLA.out` и расположен во временном каталоге вашей операционной системы.

```
err = trace.Start(f)
if err != nil {
    fmt.Println(err)
    return
}
defer trace.Stop()
```

Эта часть посвящена сбору данных для `go tool trace` и никак не связана с назначением программы. Мы запускаем процесс трассировки с помощью `trace.Start()`. После окончания мы вызываем функцию `trace.Stop()`. Вызов `defer` означает, что мы хотим завершить трассировку, когда функция `main()` вернется.

```
for i := 0; i < 3; i++ {
    s := make([]byte, 50000000)
    if s == nil {
        fmt.Println("Operation failed!")
    }
}

for i := 0; i < 5; i++ {
    s := make([]byte, 100000000)
    if s == nil {
        fmt.Println("Operation failed!")
    }
    time.Sleep(time.Millisecond)
}
```

Весь этот код посвящен выделению памяти для запуска работы сборщика мусора и генерации большего количества данных трассировки. (Больше информации о сборщике мусора Go вы можете узнать в приложении.) Программа выполняется в обычном режиме. Однако когда она завершается, в файле `traceCLA.out` появляются данные трассировки. После этого мы должны обработать их следующим образом:

```
$ go tool trace /path/ToTemporary/Directory/traceCLA.out
```

Команда автоматически запускает веб-сервер и открывает веб-интерфейс средства трассировки в вашем браузере по умолчанию. Вы можете запустить его на своем компьютере, чтобы поэкспериментировать с веб-интерфейсом средства трассировки.

Ссылка `View trace` показывает информацию о горутинах вашей программы и работе сборщика мусора.

Имейте в виду: хотя `go tool trace` очень удобная и эффективная утилита, она не может решить все проблемы с производительностью. Бывают случаи, когда `go tool pprof` более уместен, особенно если требуется узнать, где наш код работает большую часть своего времени.

Как и в случае с профилированием, сбор данных отслеживания для HTTP-сервера представляет собой несколько иной процесс, который объясняется в следующем подразделе.

Трассировка веб-сервера со стороны клиента

В этом разделе показано, как осуществлять трассировку приложения веб-сервера с помощью `net/http/httptrace`. Пакет позволяет трассировать этапы выполнения HTTP-запроса от клиента. Код в файле `traceHTTP.go`, который взаимодействует с веб-серверами, выглядит следующим образом:

```
package main

import (
    "fmt"
    "net/http"
    "net/http/httptrace"
    "os"
)
```

Прежде чем мы сможем включить трассировку HTTP, нам ожидаемо нужно импортировать пакет `net/http/httptrace`.

```
func main() {
    if len(os.Args) != 2 {
        fmt.Printf("Usage: URL\n")
        return
    }

    URL := os.Args[1]
    client := http.Client{}

    req, _ := http.NewRequest("GET", URL, nil)
```

До этого момента мы подготавливаем клиентский запрос к веб-серверу как обычно.

```
trace := &httptrace.ClientTrace{
    GotFirstResponseByte: func() {
        fmt.Println("First response byte!")
    },
    GotConn: func(connInfo httptrace.GotConnInfo) {
        fmt.Printf("Got Conn: %v\n", connInfo)
    },
    DNSDone: func(dnsInfo httptrace.DNSDoneInfo) {
        fmt.Printf("DNS Info: %v\n", dnsInfo)
    },
    ConnectStart: func(network, addr string) {
```

```

        fmt.Println("Dial start")
    },
    ConnectDone: func(network, addr string, err error) {
        fmt.Println("Dial done")
    },
    WroteHeaders: func() {
        fmt.Println("Wrote headers")
    },
}

```

Код выше полностью посвящен отслеживанию HTTP-запросов. Структура `httptrace.ClientTrace` определяет интересующие нас события, которыми являются `GotFirstResponseByte`, `GotConn`, `DNSDone`, `connectStart`, `ConnectDone` и `WriteHeaders`. Когда происходит такое событие, выполняется соответствующий код. Более подробную информацию о поддерживаемых событиях и их назначении вы можете найти в документации пакета `net/http/httptrace`.

```

req = req.WithContext(httptrace.WithClientTrace(req.Context(), trace))
fmt.Println("Requesting data from server!")
_, err := http.DefaultTransport.RoundTrip(req)
if err != nil {
    fmt.Println(err)
    return
}

```

Функция `httptrace.WithClientTrace()` возвращает новое значение `context` на основе заданного родительского контекста, тогда как `http.DefaultTransport.RoundTrip()` оборачивает запрос значением `context`, чтобы отследить запрос.

Имейте в виду, что трассировка HTTP в Go была разработана для отслеживания событий одного `http.Transport.RoundTrip`.

```

_, err = client.Do(req)
if err != nil {
    fmt.Println(err)
    return
}

```

Последняя часть отправляет клиентский запрос на сервер для начала трассировки.

При выполнении `traceHTTP.go` мы получаем такой вывод:

```
$ go run traceHTTP.go https://www.golang.org/
Requesting data from server!
DNS Info: {Addrs:[{IP:2a00:1450:4001:80e::2011 Zone:}
{IP:142.250.185.81 Zone:}] Err:<nil> Coalesced:false}
```

В этой первой части мы видим, что IP-адрес сервера был разрешен; это значит, клиент готов начать взаимодействие с HTTP-сервером.

```
Dial start
Dial done
Got Conn: {Conn:0xc000078000 Reused:false WasIdle:false IdleTime:0s}
Wrote headers
First response byte!
Got Conn: {Conn:0xc000078000 Reused:true WasIdle:false IdleTime:0s}
Wrote headers
First response byte!
DNS Info: {Addrs:[{IP:2a00:1450:4001:80e::2011 Zone:}
{IP:142.250.185.81 Zone:}] Err:<nil> Coalesced:false}
Dial start
Dial done
Got Conn: {Conn:0xc0000a1180 Reused:false WasIdle:false IdleTime:0s}
Wrote headers
First response byte!
```

Этот вывод помогает более подробно понять ход подключения и полезен при устранении неполадок. К сожалению, более подробное обсуждение трассировки выходит за рамки этой книги. В следующем подразделе показано, как просмотреть все маршруты веб-сервера, чтобы убедиться, что они определены правильно.

Посещение всех маршрутов веб-сервера

Пакет `gorilla/mux` предлагает функцию `Walk`, которую можно использовать для посещения всех зарегистрированных маршрутов маршрутизатора. Это может очень пригодиться, когда необходимо убедиться, что каждый маршрут зарегистрирован и работает.

Код `walkAll.go` содержит множество пустых функций-обработчиков, поскольку его целью является не тестирование функций обработки, а их посещение. Выглядит он следующим образом (ничто не запрещает вам использовать ту же технику на полностью реализованном веб-сервере):

```
package main

import (
    "fmt"
    "net/http"
    "strings"

    "github.com/gorilla/mux"
)
```

Поскольку мы используем внешний пакет, запуск `walkAll.go` должен происходить где-то в `~/go/src`.

```
func handler(w http.ResponseWriter, r *http.Request) {
    return
}
```

Эта пустая функция обработчика является общей для всех конечных точек по соображениям простоты.

```
func (h notAllowedHandler) ServeHTTP(rw http.ResponseWriter, r *http.Request)
{
    handler(rw, r)
}
```

Обработчик `notAllowedHandler` также вызывает функцию `handler()`.

```
type notAllowedHandler struct{}

func main() {
    r := mux.NewRouter()

    r.NotFoundHandler = http.HandlerFunc(handler)
    notAllowed := notAllowedHandler{}
    r.MethodNotAllowedHandler = notAllowed

    // регистрация GET
    getMux := r.Methods(http.MethodGet).Subrouter()
    getMux.HandleFunc("/time", handler)
    getMux.HandleFunc("/getall", handler)
    getMux.HandleFunc("/getid", handler)
    getMux.HandleFunc("/logged", handler)
    getMux.HandleFunc("/username/{id:[0-9]+}", handler)

    // регистрация PUT
    // обновить пользователя
    putMux := r.Methods(http.MethodPut).Subrouter()
    putMux.HandleFunc("/update", handler)

    // регистрация POST
    // добавить пользователя + вход в систему + выход из системы
    postMux := r.Methods(http.MethodPost).Subrouter()
    postMux.HandleFunc("/add", handler)
    postMux.HandleFunc("/login", handler)
    postMux.HandleFunc("/logout", handler)

    // регистрация DELETE
    // удалить пользователя
    deleteMux := r.Methods(http.MethodDelete).Subrouter()
    deleteMux.HandleFunc("/username/{id:[0-9]+}", handler)
```

Вышеприведенная часть посвящена определению маршрутов и методов HTTP, которые требуется поддерживать.

```
err := r.Walk(func(route *mux.Route, router *mux.Router, ancestors
[]*mux.Route) error {
```

Вот как мы вызываем метод `Walk()`.

```
    pathTemplate, err := route.GetPathTemplate()
    if err == nil {
        fmt.Println("ROUTE:", pathTemplate)
    }
    pathRegexp, err := route.GetPathRegexp()
    if err == nil {
        fmt.Println("Path regexp:", pathRegexp)
    }
    qT, err := route.GetQueriesTemplates()
    if err == nil {
        fmt.Println("Queries templates:", strings.Join(qT, ","))
    }
    qRegexps, err := route.GetQueriesRegexp()
    if err == nil {
        fmt.Println("Queries regexps:", strings.Join(qRegexps, ","))
    }
    methods, err := route.GetMethods()
    if err == nil {
        fmt.Println("Methods:", strings.Join(methods, ","))
    }
    fmt.Println()
    return nil
})
```

Для каждого посещенного маршрута программа собирает нужную информацию. Можете удалить часть вызовов `fmt.Println()`, если требуется уменьшить объем вывода.

```
if err != nil {
    fmt.Println(err)
}

http.Handle("/", r)
}
```

Итак, общая идея `walkAll.go` заключается в том, что вы назначаете пустой обработчик каждому маршруту на вашем сервере, а затем вызываете `mux.Walk()` для посещения их всех. Включение Go-модулей и запуск `walkAll.go` генерируют следующий вывод:

```
$ go mod init
$ go mod tidy
$ go run walkAll.go
Queries templates:
Queries regexps:
```

```
Methods: GET

ROUTE: /time
Path regexp: ^/time$
Queries templates:
Queries regexps:
Methods: GET
```

Выходные данные показывают HTTP-методы, которые поддерживает каждый маршрут, а также формат пути. Таким образом, конечная точка `/time` работает с `GET`, и ее путь соответствует `/time`, поскольку значение `Path regexp` означает, что `/time` находится между началом (^) и концом пути (\$).

```
ROUTE: /getall
Path regexp: ^/getall$
Queries templates:
Queries regexps:
Methods: GET

ROUTE: /getid
Path regexp: ^/getid$
Queries templates:
Queries regexps:
Methods: GET

ROUTE: /logged
Path regexp: ^/logged$
Queries templates:
Queries regexps:
Methods: GET

ROUTE: /username/{id:[0-9]+}
Path regexp: ^/username/(?P<v0>[0-9]+)$
Queries templates:
Queries regexps:
Methods: GET
```

В случае `/username` выходные данные включают регулярные выражения, связанные с той конечной точкой, которая используется для выбора значения переменной `id`.

```
Queries templates:
Queries regexps:
Methods: PUT

ROUTE: /update
Path regexp: ^/update$
Queries templates:
Queries regexps:
Methods: PUT
Queries templates:
```

```
Queries regexps:  
Methods: POST  
  
ROUTE: /add  
Path regexp: ^/add$  
Queries templates:  
Queries regexps:  
Methods: POST  
  
ROUTE: /login  
Path regexp: ^/login$  
Queries templates:  
Queries regexps:  
Methods: POST  
  
ROUTE: /logout  
Path regexp: ^/logout$  
Queries templates:  
Queries regexps:  
Methods: POST  
  
Queries templates:  
Queries regexps:  
Methods: DELETE  
  
ROUTE: /username/{id:[0-9]+}  
Path regexp: ^/username/(?P<v0>[0-9]+)$  
Queries templates:  
Queries regexps:  
Methods: DELETE
```

Хотя посещение маршрутов веб-сервера — своего рода тестирование, это все же «неофициальный» способ тестирования в Go. Главное, на что следует обратить внимание в таком выводе, — это отсутствие конечной точки, использование неправильного HTTP-метода или отсутствие параметра из конечной точки.

В следующем разделе мы более подробно рассмотрим тестирование Go-кода.

Тестирование Go-кода

Предметом обсуждения в этом разделе является тестирование Go-кода путем *написания тестовых функций*.

Тестирование программного обеспечения — очень обширная тема, и ее нельзя описать в одном разделе главы книги. Так что здесь мы постараемся представить как можно больше практической информации.

Go дает вам возможность писать тесты для вашего Go-кода, позволяющие обнаруживать ошибки. Однако тестирование программного обеспечения может

показать только *наличие* одной или нескольких ошибок, но *не их отсутствие*. Это означает, что вы никогда не можете быть на 100 % уверены в том, что в вашем коде нет ошибок!

Строго говоря, этот раздел посвящен *автоматизированному тестированию*, которое включает в себя написание дополнительного кода для проверки того, работает ли реальный (производственный) код так, как ожидалось. Следовательно, результатом тестовой функции является либо `PASS` (прохождение), либо `FAIL` (сбой). Вскоре вы увидите, как это работает. Подход Go к тестированию на первый взгляд может показаться простым (особенно если сравнить его с практикой тестирования других языков программирования), но является очень эффективным, поскольку не требует от разработчика слишком больших временных затрат.

Всегда следует помещать тестовый код в отдельный исходный файл. Нет необходимости создавать огромный исходный файл, который трудно читать и поддерживать. Теперь представим тестирование, еще раз обратившись к функции `matchInt()` из главы 3.

Написание тестов для `./ch03/intRE.go`

В этом подразделе мы пишем тесты для функции `matchInt()`, которая была реализована в файле `intRE.go` в главе 3. Сначала создадим новый файл `intRE_test.go`, который и будет содержать все тесты. Затем переименуем пакет из `main` в `testRE` и удалим функцию `main()` — это необязательное действие. После этого мы должны решить, что именно собираемся тестировать и как. В основные этапы тестирования входит написание тестов для ожидаемого ввода, неожиданного ввода, пустого ввода и пограничных случаев. Все это будет показано в коде. Кроме того, мы будем генерировать случайные целые числа, преобразовывать их в строки и применять в качестве входных данных для `matchInt()`. Вообще говоря, хорошим способом протестировать функции, которые работают с числовыми значениями, будет использование случайных чисел или случайных значений в целом в качестве входных данных и оценка того, как ваш код ведет себя и обрабатывает эти значения.

Две тестовые функции в файле `intRE_test.go` выглядят следующим образом:

```
func Test_matchInt(t *testing.T) {
    if matchInt("") {
        t.Error(`matchInt("") != true`)
    }
}
```

Вызов `matchInt("")` должен возвращать значение `false`, поэтому, если он вернет `true`, это будет означать, что функция сработала не так, как ожидалось.

```
if matchInt("00") == false {
    t.Error(`matchInt("00") != true`)
}
```

Вызов `matchInt("00")` также должен возвращать значение `true`, поскольку `00` является допустимым целым числом. Поэтому, если вызов возвращает `false`, это означает, что функция работает не так, как ожидалось.

```
if matchInt("-00") == false {
    t.Error(`matchInt("-00") != true`)
}

if matchInt("+00") == false {
    t.Error(`matchInt("+00") != true`)
}
}
```

Эта первая тестовая функция использует статический ввод для проверки правильности `matchInt()`. Как обсуждалось ранее, функция тестирования принимает один параметр `*testing.T` и не возвращает никаких значений.

```
func Test_with_random(t *testing.T) {
    SEED := time.Now().Unix()
    rand.Seed(SEED)
    n := strconv.Itoa(random(-100000, 19999))

    if matchInt(n) == false {
        t.Error("n = ", n)
    }
}
```

Вторая тестовая функция использует случайные, но допустимые входные данные для проверки `matchInt()`. Следовательно, данный ввод всегда должен проверяться. При выполнении двух тестовых функций с помощью `go test` мы получаем такой вывод:

```
$ go test -v *.go
== RUN  Test_matchInt
--- PASS: Test_matchInt (0.00s)
== RUN  Test_with_random
--- PASS: Test_with_random (0.00s)
PASS
ok    command-line-arguments    0.410s
```

Итак, все тесты пройдены, а это значит, что с `matchInt()` все в порядке.

Следующий подраздел посвящен использованию метода `TempDir()`.

Функция TempDir()

Метод `TempDir()` работает как при тестировании, так и при бенчмаркинге. Его цель — создать временный каталог, который будет использоваться во время тестирования или бенчмаркинга. Go автоматически удаляет этот временный каталог, когда тест (и его подтесты) или бенчмарки готовы завершиться с помощью метода `CleanUp()`. За это отвечает Go, и вам не нужно использовать и реализовывать данный метод самостоятельно. Точное место, где будет создан временный каталог, зависит от используемой операционной системы. В macOS он находится в папке `/var/folders`, тогда как в Linux — в папке `/tmp`. Мы собираемся проиллюстрировать работу `TempDir()` в следующем подразделе, где также поговорим о `Cleanup()`.

Функция Cleanup()

Хотя мы показываем метод `Cleanup()` в сценарии тестирования, `Cleanup()` работает в случае бенчмаркинга. Его название раскрывает его назначение, которое состоит в том, чтобы очищать то, что мы создали при тестировании или бенчмаркинге пакета. Однако именно нам нужно указать `Cleanup()`, что делать. Параметр `Cleanup()` — это функция, которая выполняет очистку. Эта функция обычно реализуется встроенно как анонимная, но вы также можете создать ее в другом месте и вызвать по имени.

Файл `cleanup.go` содержит фиктивную функцию с именем `Foo()` — поскольку в ней нет кода, представлять ее смысла нет. С другой стороны, весь важный код находится в файле `cleanup_test.go`:

```
func myCleanUp() func() {
    return func() {
        fmt.Println("Cleaning up!")
    }
}
```

Функция `myCleanUp()` будет использоваться в качестве параметра для метода `CleanUp()` и должна иметь эту конкретную сигнатуру. Помимо сигнатуры, вы можете поместить в реализацию `myCleanUp()` любой код.

```
func TestFoo(t *testing.T) {
    t1 := path.Join(os.TempDir(), "test01")
    t2 := path.Join(os.TempDir(), "test02")
```

Это пути к двум каталогам, которые мы планируем создать.

```
err := os.Mkdir(t1, 0755)
if err != nil {
    t.Error("os.Mkdir() failed:", err)
    return
}
```

Мы создаем каталог с помощью `os.Mkdir()` и указываем путь к нему. Поэтому наша ответственность — удалить этот каталог, когда он больше не нужен.

```
    defer t.Cleanup(func() {
        err = os.Remove(t1)
        if err != nil {
            t.Error("os.Mkdir() failed:", err)
        }
    })
}
```

После завершения `TestFoo()` `t1` будет удален кодом анонимной функции, который передается в качестве параметра в `t.CleanUp()`.

```
err = os.Mkdir(t2, 0755)
if err != nil {
    t.Error("os.Mkdir() failed:", err)
    return
}
}
```

Мы создаем другой каталог с помощью `os.Mkdir()`, однако в данном случае не удаляем его. Следовательно, когда `TestFoo()` завершится, `t2` удален не будет.

```
func TestBar(t *testing.T) {
    t1 := t.TempDir()
```

Из-за использования метода `t.TempDir()` значение (путь к каталогу) `t1` присваивается операционной системой. В добавок этот путь к каталогу будет автоматически удален перед самым завершением тестовой функции.

```
fmt.Println(t1)
t.Cleanup(myCleanUp())
}
```

Здесь мы используем функцию `myCleanUp()` в качестве параметра для метода `Cleanup()`. Это удобно, если нужно выполнить одну и ту же очистку несколько раз. Выполнение тестов показывает следующий результат:

```
$ go test -v *.go
===[ RUN   TestFoo
--- PASS: TestFoo (0.00s)
===[ RUN   TestBar
/var/folders/sk/ltk8cnw501zdtr2hxcj5sv2m0000gn/T/TestBar2904465158/01
```

Это временный каталог, который был создан с помощью функции `TempDir()` на компьютере с macOS.

```
Cleaning up!
--- PASS: TestBar (0.00s)
PASS
ok      command-line-arguments      0.096s
```

Проверка наличия каталогов, созданных с помощью `TempDir()`, показывает, что они были успешно удалены. В то же время каталог, хранящийся в переменной `t2` функции `TestFoo()`, удален не был. Повторный запуск тех же тестов завершится неудачей, поскольку файл `test02` уже существует и не может быть создан повторно:

```
$ go test -v *.go
=== RUN TestFoo
    cleanup_test.go:33: os.Mkdir() failed: mkdir /var/folders/sk/ltk8cnw50lzdtr2hxcj5sv2m000gn/T/test02: file exists
--- FAIL: TestFoo (0.00s)
=== RUN TestBar
/var/folders/sk/ltk8cnw50lzdtr2hxcj5sv2m000gn/T/TestBar2113309096/01
Cleaning up!
--- PASS: TestBar (0.00s)
FAIL
FAIL command-line-arguments          0.097s
FAIL
```

Сообщение об ошибке `/var/folders/sk/ltk8cnw50lzdtr2hxcj5sv2m000gn/T/test02: file exists` указывает на корень проблемы. Решение состоит в том, чтобы проводить очистку после тестов. В следующем подразделе мы обсудим использование пакета `testing/quick`.

Пакет `testing/quick`

Бывают случаи, когда необходимо создать тестовые данные без вмешательства человека. Стандартная библиотека Go предлагает для этого пакет `testing/quick`, который можно использовать для *тестирования методом «черного ящика»* (метод тестирования программного обеспечения, который проверяет функциональность приложения или функции, не имея каких-либо предварительных знаний о его внутренней работе). Он в некоторой степени связан с пакетом `QuickCheck` из языка программирования *Haskell* – оба пакета реализуют служебные функции, которые помогают провести тестирование методом «черного ящика». С помощью `testing/quick` Go генерирует случайные значения встроенных типов, которые можно использовать для тестирования, что избавляет нас от необходимости генерировать все эти значения вручную.

Код `quickT.go` следующий:

```
package quickT

type Point2D struct {
    X, Y int
}

func Add(x1, x2 Point2D) Point2D {
    temp := Point2D{}
```

```

    temp.X = x1.X + x2.X
    temp.Y = x1.Y + x2.Y
    return temp
}

```

Этот код реализует единственную функцию, которая добавляет две переменные `Point2D`, — это функция, которую мы собираемся протестировать.

Код `quickT_test.go` выглядит следующим образом:

```

package quickT

import (
    "testing"
    "testing/quick"
)

var N = 1000000

func TestWithItself(t *testing.T) {
    condition := func(a, b Point2D) bool {
        return Add(a, b) == Add(b, a)
    }

    err := quick.Check(condition, &quick.Config{MaxCount: N})
    if err != nil {
        t.Errorf("Error: %v", err)
    }
}

```

Вызов `quick.Check()` автоматически генерирует случайные числа на основе сигнатуры своего первого аргумента, который является функцией, определенной ранее. Таким образом, исчезает необходимость создавать эти случайные числа самостоятельно, что облегчает чтение и запись кода. Фактические тесты выполняются в функции `condition`.

```

func TestThree(t *testing.T) {
    condition := func(a, b, c Point2D) bool {
        return Add(Add(a, b), c) == Add(a, b)
    }
}

```

Эта реализация *намеренно неверна*. Чтобы исправить ее, мы должны заменить `Add(Add(a, b), c) == Add(a, b)` на `Add(Add(a, b), c) == Add(c, Add(a, b))`. Мы проделали это, чтобы посмотреть на выходные данные, которые генерируются при сбое теста.

```

err := quick.Check(condition, &quick.Config{MaxCount: N})
if err != nil {
    t.Errorf("Error: %v", err)
}

```

При выполнении созданных тестов мы получаем такой вывод:

```
$ go test -v *.go
== RUN TestWithItself
--- PASS: TestWithItself (0.86s)
```

Как и ожидалось, первый тест прошел успешно.

```
== RUN TestThree
quickT_test.go:28: Error: #1: failed on input quickT.
Point2D{X:761545203426276355, Y:-915390795717609627}, quickT.
Point2D{X:-3981936724985737618, Y:2920823510164787684}, quickT.
Point2D{X:-8870190727513030156, Y:-7578455488760414673}
--- FAIL: TestThree (0.00s)
FAIL
FAIL      command-line-arguments  1.153s
FAIL
```

Но, как и ожидалось, второй тест выдал ошибку. Хорошо то, что входные данные, вызвавшие ошибку, отображаются на экране, поэтому вы можете увидеть, что именно привело к сбою вашей функции.

В следующем подразделе рассказывается, как установить тайм-аут для тестов, выполнение которых занимает слишком много времени.

Тайм-аут тестов

Если запуск инструмента `go test` занимает слишком много времени или по какой-то причине никогда не заканчивается, то вам поможет параметр `-timeout`.

Чтобы проиллюстрировать его работу, мы используем код из предыдущего подраздела, а также флаги командной строки `-timeout` и `-count`. В то время как в первом указывается максимально допустимая продолжительность тестов, во втором — количество выполнений тестов.

При запуске `go test -v *.go -timeout 1s` сообщает `go test`, что завершение всех тестов должно занять не более одной секунды, — на моей машине завершение тестов заняло менее секунды. Однако выполнение следующей команды генерирует другой результат:

```
$ go test -v *.go -timeout 1s -count 2
== RUN TestWithItself
--- PASS: TestWithItself (0.87s)
== RUN TestThree
quickT_test.go:28: Error: #1: failed on input quickT.
Point2D{X:-312047170140227400, Y:-5441930920566042029}, quickT.
Point2D{X:7855449254220087092, Y:7437813460700902767}, quickT.
```

```
Point2D{X:4838605758154930957, Y:-7621852714243790655}
--- FAIL: TestThree (0.00s)
==== RUN TestWithItself
panic: test timed out after 1s
```

Выходные данные длиннее, чем представленные, — остальная их часть связана с тем, что горутины прерываются до завершения. Ключевым моментом здесь является то, что команда `go test` прервала процесс из-за использования `-timeout 1s`.

Покрытие тестового кода

В этом подразделе мы узнаем, как найти информацию о покрытии кода наших программ, что дает возможность обнаруживать блоки кода или отдельные операторы, которые не выполняются функциями тестирования.

Помимо прочего, просмотр покрытия кода помогает выявлять проблемы и ошибки в коде, поэтому не стоит недооценивать его полезность. Однако тест покрытия кода дополняет модульное тестирование, не заменяя его. Помните: необходимо убедиться в том, что функции тестирования действительно пытаются охватить все ситуации и, следовательно, запустить весь доступный код. Если функции тестирования не пытаются охватить все ситуации, то проблема может быть в них, а не в тестируемом коде.

Код `coverage.go`, в котором имеются некоторые преднамеренные проблемы (показывающие, как определяется недоступный код), выглядит таким образом:

```
package coverage

import "fmt"

func f1() {
    if true {
        fmt.Println("Hello!")
    } else {
        fmt.Println("Hi!")
    }
}
```

Проблема с этой функцией заключается в том, что первая ветвь `if` всегда имеет значение `true`, и поэтому ветвь `else` не будет выполняться.

```
func f2(n int) int {
    if n >= 0 {
        return 0
    } else if n == 1 {
        return 1
    }
}
```

```
    } else {
        return f2(n-1) + f2(n-2)
    }
}
```

Функция `f2()` содержит две проблемы. Первая заключается в том, что функция плохо работает с отрицательными целыми числами, а вторая — в том, что все положительные целые числа обрабатываются первой ветвью `if`. Покрытие кода может помочь вам решить только вторую проблему. Код `coverage_test.go` содержит обычные тестовые функции, которые пытаются запустить весь доступный код:

```
package coverage

import "testing"

func Test_f1(t *testing.T) {
    f1()
}
```

Эта тестовая функция наивно проверяет работу `f1()`.

```
func Test_f2(t *testing.T) {
    _ = f2(123)
}
```

Вторая тестовая функция проверяет работу `f2()`, выполняя `f2(123)`.

Сначала мы должны запустить `go test` следующим образом (задача покрытия кода выполняется флагом `-cover`):

```
$ go test -cover *.go
ok      command-line-arguments      0.420s      coverage: 50.0% of statements
```

Вывод показывает, что код покрывается на 50%, и это не очень хорошо! Однако мы еще не закончили, так как можно сгенерировать отчет о тестовом покрытии. Следующая команда генерирует отчет о покрытии кода:

```
$ go test -coverprofile=coverage.out *.go
```

Содержимое `coverage.out` выглядит следующим образом (ваш файл при этом может быть немного другим в зависимости от имени пользователя и используемой папки):

```
$ cat coverage.out
mode: set
/Users/mtsouk/Desktop/coverage.go:5.11,6.10 1 1
/Users/mtsouk/Desktop/coverage.go:6.10,8.3 1 1
/Users/mtsouk/Desktop/coverage.go:8.8,10.3 1 0
```

```
/Users/mtsouk/Desktop/coverage.go:13.20,14.12 1 1
/Users/mtsouk/Desktop/coverage.go:14.12,16.3 1 1
/Users/mtsouk/Desktop/coverage.go:16.8,16.19 1 0
/Users/mtsouk/Desktop/coverage.go:16.19,18.3 1 0
/Users/mtsouk/Desktop/coverage.go:18.8,20.3 1 0
```

Формат и поля в каждой строке файла покрытия — `name.go:line.column,line.column numberOfRowsInSectionStatements count`. Последнее поле — это флаг, который сообщает вам, охвачены ли операторы, указанные в `line.column, line.column`. Таким образом, когда вы видите `0` в последнем поле, это означает, что код не охвачен.

Наконец, вывод HTML можно просмотреть в браузере, запустив `go tool cover -html=coverage.out`. Если вы использовали имя файла, отличающееся от `coverage.out`, то измените команду соответствующим образом. На рис. 11.2 показан сгенерированный вывод. Если вы читаете печатную версию книги, то, возможно, не сможете увидеть цвета. Красные линии обозначают код, который не выполняется, в то время как зеленые показывают код, который был выполнен тестами.

```
coverage: Go Coverage Report
file:///private/var/folders/91/5xmhlblx7gq2wr65qv7yht1m0000gn/T/cover34.html
/coverage.go (50.0%) not tracked not covered covered

package coverage

import "fmt"

func f1() {
    if true {
        fmt.Println("Hello!")
    } else {
        fmt.Println("Hi!")
    }
}

func f2(n int) int {
    if n >= 0 {
        return 0
    } else if n == 1 {
        return 1
    } else {
        return f2(n-1) + f2(n-2)
    }
}
```

Рис. 11.2. Отчет о покрытии кода

Часть кода помечена как неотслеживаемая (серым цветом), поскольку это код, который не может быть обработан инструментом покрытия кода. Сгенериро-

ванный вывод четко показывает проблемы с кодом как в `f1()`, так и в `f2()`. Их необходимо исправить прямо сейчас!

В следующем подразделе мы обсудим недостижимый код и способы его обнаружения.

Поиск недостижимого Go-кода

Иногда неправильно реализованный оператор `if` или неуместный оператор `return` могут создать недостижимые блоки кода, которые вообще не будут выполняться. Поскольку это *логическая ошибка* (а это означает, что она не будет обнаружена компилятором), нам нужен способ ее обнаружения.

К счастью, в этом нам помогает инструмент `go vet`, который проверяет исходный код Go и сообщает о подозрительных конструкциях. Мы покажем, как использовать `go vet`, с помощью исходного файла `cannotReach.go`, который содержит следующие две функции:

```
func S2() {
    return
    fmt.Println("Hello!")
}
```

Здесь содержится логическая ошибка, поскольку `S2()` возвращается перед выводом нужного сообщения.

```
func S1() {
    fmt.Println("In S1()")
    return

    fmt.Println("Leaving S1()")
}
```

Аналогично, `S1()` возвращается, не давая оператору `fmt.Println("Leaving S1()")` шанса на выполнение.

При выполнении `go vet` с файлом `cannotReach.go` мы получаем такой вывод:

```
$ go vet cannotReach.go
# command-line-arguments
./cannotReach.go:9:2: unreachable code
./cannotReach.go:16:2: unreachable code
```

Первое сообщение указывает на оператор `fmt.Println()` в `S2()`, а второе — на второй оператор `fmt.Println()` в `S1()`. В этом случае `go vet` сработал отлично. Однако данный инструмент не отличается особой сложностью и не может отлавливать все возможные типы логических ошибок. Если вам нужен более эффективный инструмент, то стоит обратить внимание на `staticcheck`

(<https://staticcheck.io/>), который также может быть интегрирован с кодом Microsoft Visual Studio (<https://code.visualstudio.com/>). На рис. 11.3 показано, как код Visual Studio обозначает недостижимый код с помощью `staticcheck`.

The screenshot shows a code editor window for a file named `cannotReach.go`. The code contains several print statements and function definitions. A specific line of code, `fmt.Println("Leaving S1()")`, is highlighted with a red underline and a tooltip indicating it is "unreachable code". A lightbulb icon is also present near this line, suggesting a quick fix or inspection option.

```
-eo cannotReach.go 2 ×
code > ch11 > -eo cannotReach.go > ⚡ S1
+ |
  5  )
  6
  7  func S2() {
  8    |
  9    fmt.Println("Hello!")
10  }
11
12 func S1() {
13   |
14   |   fmt.Println("In S1()")
15   |   unreachable code unreachable
16   |   View Problem (F8) Quick Fix... (⌘.)
17   |   fmt.Println("Leaving S1()")
18  }
19
20  func main() {
21    |
22 }
```

Рис. 11.3. Просмотр недоступного кода в Visual Studio Code

Как правило, включение `go vet` в ваш рабочий процесс пойдет только на пользу. Более подробную информацию о возможностях этого инструмента вы можете найти, запустив `go doc cmd/vet`.

В следующем разделе показано, как протестировать HTTP-сервер с помощью серверной части базы данных.

Тестирование HTTP-сервера с помощью серверной части базы данных

HTTP-сервер — нечто совсем иное, поскольку он уже должен быть запущен для выполнения тестов. К счастью, в этом вам может помочь пакет `net/http/httptest`. Вам не придется запускать HTTP-сервер самостоятельно, поскольку этот пакет берет на себя всю работу, но вам нужно убедиться, что сервер базы данных запущен. Мы будем тестировать сервер REST API, который мы разработали в главе 10, — скопируем файл `server_test.go` с тестовым кодом сервера в репозиторий <https://github.com/mactsouk/rest-api> на GitHub.



Чтобы создать `server_test.go`, нам не придется менять реализацию сервера REST API.

Код `server_test.go`, который содержит тестовые функции для сервиса HTTP, выглядит следующим образом:

```
package main

import (
    "bytes"
    "net/http"
    "net/http/httptest"
    "strconv"
    "strings"
    "testing"
    "time"

    "github.com/gorilla/mux"
)
```

Единственная причина для включения `github.com/gorilla/mux` заключается в дальнейшем использовании `mux.SetURLVars()`.

```
func TestTimeHandler(t *testing.T) {
    req, err := http.NewRequest("GET", "/time", nil)
    if err != nil {
        t.Fatal(err)
    }

    rr := httptest.NewRecorder()
    handler := http.HandlerFunc(TimeHandler)
    handler.ServeHTTP(rr, req)

    status := rr.Code
    if status != http.StatusOK {
```

```

        t.Errorf("handler returned wrong status code: got %v want %v",
                  status, http.StatusOK)
    }
}

```

Функция `http.NewRequest()` используется для определения метода HTTP-запроса, конечной точки и отправки данных в нее при необходимости. Вызов `http.HandlerFunc(TimeHandler)` определяет тестируемую функцию обработчика.

```

func TestMethodNotAllowed(t *testing.T) {
    req, err := http.NewRequest("DELETE", "/time", nil)
    if err != nil {
        t.Fatal(err)
    }

    rr := httptest.NewRecorder()
    handler := http.HandlerFunc(MethodNotAllowedHandler)

```

В этой тестовой функции мы тестируем `MethodNotAllowedHandler`.

```

    handler.ServeHTTP(rr, req)

    status := rr.Code
    if status != http.StatusNotFound {
        t.Errorf("handler returned wrong status code: got %v want %v",
                  status, http.StatusOK)
    }
}

```

Мы знаем, что это взаимодействие завершится неудачей, поскольку мы тестируем `MethodNotAllowedHandler`. Следовательно, мы ожидаем получить назад ответный код `http.StatusNotFound` — если получим другой код, то тестовая функция завершится неудачей.

```

func TestLogin(t *testing.T) {
    UserPass := []byte(`{"Username": "admin", "Password": "admin"}`)

```

Здесь мы храним нужные поля структуры `User` в байтовом срезе. Чтобы тесты работали, пользователь с правами администратора должен иметь `admin` в качестве пароля, поскольку именно он используется в файле `server_test.go`, чтобы у пользователя `admin` или любого другого с правами администратора был правильный пароль.

```

    req, err := http.NewRequest("POST", "/login", bytes.NewBuffer(UserPass))
    if err != nil {
        t.Fatal(err)
    }
    req.Header.Set("Content-Type", "application/json")

```

В этих строках кода мы создаем желаемый запрос.

```
rr := httptest.NewRecorder()
handler := http.HandlerFunc(LoginHandler)
handler.ServeHTTP(rr, req)
```

`NewRecorder()` возвращает инициализированный `ResponseRecorder`, который используется в `ServeHTTP()` — методе, выполняющем запрос. Ответ сохраняется в переменной `rr`.

Существует также тестовая функция для конечной точки `/logout`, которая здесь не представлена, поскольку почти идентична `testLogin()`. Однако выполнение тестов в случайном порядке может создать проблемы с тестированием, поскольку `testLogin()` всегда должен выполняться перед `TestLogout()`.

```
status := rr.Code
if status != http.StatusOK {
    t.Errorf("handler returned wrong status code: got %v want %v",
        status, http.StatusOK)
    return
}
}
```

Если кодом состояния является `http.StatusOK`, это означает, что взаимодействие сработало ожидаемым образом.

```
func TestAdd(t *testing.T) {
    now := int(time.Now().Unix())
    username := "test_" + strconv.Itoa(now)
    users := `[{"Username": "admin", "Password": "admin"},` +
        {"Username": ` + username + `", "Password": "myPass"}]`
```

Для обработчика `Add()` нам нужно передать массив записей JSON, который здесь и создается. Мы не хотим каждый раз создавать одно и то же имя пользователя, поэтому добавим к строке `_test` текущую временную метку.

```
UserPass := []byte(users)
req, err := http.NewRequest("POST", "/add", bytes.NewBuffer(UserPass))
if err != nil {
    t.Fatal(err)
}
req.Header.Set("Content-Type", "application/json")
```

Здесь мы создаем фрагмент записей JSON (`UserPass`) и запрос.

```
rr := httptest.NewRecorder()
handler := http.HandlerFunc(AddHandler)
handler.ServeHTTP(rr, req)

// проверка ожидаемого кода состояния HTTP
if status := rr.Code; status != http.StatusOK {
```

```
    t.Errorf("handler returned wrong status code: got %v want %v",
             status, http.StatusOK)
    return
}
}
```

Если ответом сервера является `http.StatusOK`, то запрос выполнен успешно и тест пройден.

```
func TestGetUserDataHandler(t *testing.T) {
    UserPass := []byte(`{"Username": "admin", "Password": "admin"}`)
    req, err := http.NewRequest("GET", "/username/1",
        bytes.NewBuffer(UserPass))
```

Хотя мы используем `/username/1` в запросе, это не добавляет значений в карту `Vars`. Следовательно, нам нужно использовать функцию `SetURLVars()` для изменения значений в карте `Vars`. Это показано далее.

```
if err != nil {
    t.Fatal(err)
}
req.Header.Set("Content-Type", "application/json")

vars := map[string]string{
    "id": "1",
}
req = mux.SetURLVars(req, vars)
```

Для целей тестирования пакет `gorilla/mux` содержит функцию `SetURLVars()` — она позволяет добавлять элементы в карту `Vars`. В этом случае нам нужно установить значение `id` ключа равным `1`. Вы можете добавлять столько пар «ключ — значение», сколько нужно.

```
rr := httptest.NewRecorder()
handler := http.HandlerFunc(GetUserDataHandler)
handler.ServeHTTP(rr, req)

if status := rr.Code; status != http.StatusOK {
    t.Errorf("handler returned wrong status code: got %v want %v",
        status, http.StatusOK)
    return
}

expected := `{"ID":1,"Username":"admin","Password":"admin",
"LastLogin":0,"Admin":1,"Active":0}`
```

Это запись, которую мы ожидаем получить в ответ на наш запрос. Мы не можем угадать значение `LastLogin` в ответе сервера, поэтому заменим его на `0`, следовательно, здесь используется `0`.

```
serverResponse := rr.Body.String()

result := strings.Split(serverResponse, "LastLogin")
serverResponse = result[0] + `LastLogin":0,"Admin":1,"Active":0}`
```

Мы не хотим использовать значение `LastLogin` из ответа сервера, поэтому меняем его на `0`.

```
if serverResponse != expected {
    t.Errorf("handler returned unexpected body: got %v but wanted %v",
             rr.Body.String(), expected)
}
}
```

Последняя часть кода содержит стандартный Go-способ проверки того, получили ли мы ожидаемый ответ.

Создавать тесты для HTTP-сервисов станет просто, как только вы разберетесь в представленных примерах. В основном это происходит потому, что большая часть кода в тестовых функциях повторяется.

Выполнение тестов генерирует следующий результат:

```
$ go test -v server_test.go main.go handlers.go
== RUN TestTimeHandler
2021/06/17 08:59:15 TimeHandler Serving: /time from
--- PASS: TestTimeHandler (0.00s)
```

Это результат посещения конечной точки `/time`. Этим результатом является `PASS`.

```
== RUN TestMethodNotAllowed
2021/06/17 08:59:15 Serving: /time from with method DELETE
--- PASS: TestMethodNotAllowed (0.00s)
== RUN TestLogin
```

Результат посещения конечной точки `/time` с помощью HTTP-метода `DELETE`. Этим результатом является `PASS`, поскольку мы ожидали, что этот запрос завершится неудачей, ведь там используется неправильный HTTP-метод.

```
2021/06/17 08:59:15 LoginHandler Serving: /login from
2021/06/17 08:59:15 Input user: {0 admin admin 0 0 0}
2021/06/17 08:59:15 Found user: {1 admin admin 1620922454 1 0}
2021/06/17 08:59:15 Logging in: {1 admin admin 1620922454 1 0}
2021/06/17 08:59:15 Updating user: {1 admin admin 1623909555 1 1}
2021/06/17 08:59:15 Affected: 1
2021/06/17 08:59:15 User updated: {1 admin admin 1623909555 1 1}
--- PASS: TestLogin (0.01s)
```

Это вывод `testLogin()`, который проверяет конечную точку `/login`. Все строки, начинающиеся с даты и времени, генерируются сервером REST API и показывают ход выполнения запроса.

```
==> RUN  TestLogout
2021/06/17 08:59:15 LogoutHandler Serving: /logout from
2021/06/17 08:59:15 Found user: {1 admin admin 1620922454 1 1}
2021/06/17 08:59:15 Logging out: admin
2021/06/17 08:59:15 Updating user: {1 admin admin 1620922454 1 0}
2021/06/17 08:59:15 Affected: 1
2021/06/17 08:59:15 User updated: {1 admin admin 1620922454 1 0}
--- PASS: TestLogout (0.01s)
```

А это вывод из `TestLogout()`, где проверяется конечная точка `/logout`, которая также получает результат `PASS`.

```
==> RUN  TestAdd
2021/06/17 08:59:15 AddHandler Serving: /add from
2021/06/17 08:59:15 [{0 admin admin 0 0 0}
{0 test_1623909555 myPass 0 0 0}]
--- PASS: TestAdd (0.01s)
```

Это вывод тестовой функции `TestAdd()`. Имя нового созданного пользователя — `test_1623909555`, и оно должно быть другим при каждом выполнении.

```
==> RUN  TestGetUserDataHandler
2021/06/17 08:59:15 GetUserDataHandler Serving: /username/1 from
2021/06/17 08:59:15 Found user: {1 admin admin 1620922454 1 0}
--- PASS: TestGetUserDataHandler (0.00s)
PASS
ok    command-line-arguments          (cached)
```

Наконец, рассмотрим выходные данные тестовой функции `TestGetUserDataHandler()`, которая также выполнилась без каких-либо проблем.

В следующем подразделе мы обсудим фаззинг, который предлагает другой способ тестирования.

Фаззинг

Как инженеры-программисты, мы беспокоимся не когда все идет, как ожидалось, а когда происходит что-то неожиданное. Один из способов справиться с неожиданностями — использовать *фаззинг* (или *нечеткое тестирование*).

Это метод тестирования, который предоставляет неверные, неожиданные или случайные данные в программы, требующие ввода.

Преимущества нечеткого тестирования таковы:

- гарантируется, что код может обрабатывать недопустимый или случайный ввод;
- ошибки, обнаруженные с помощью фаззинга, обычно серьезны и могут указывать на угрозы безопасности;
- злоумышленники часто используют фаззинг для обнаружения уязвимостей, так что лучше к этому подготовиться.

Фаззинг официально включен в язык Go в версии 1.18. Ветка `dev.fuzz` на GitHub (<https://github.com/golang/go/tree/dev.fuzz>) содержит последнюю реализацию фаззинга. Эта ветвь будет существовать до тех пор, пока соответствующий код не будет объединен с главной ветвью. В комплекте с фаззингом идет тип данных `testing.F`, похожий на `testing.T` для тестирования и `testing.B` для бенчмаркинга. Если вы хотите попробовать фаззинг в Go, то начните с посещения <https://blog.golang.org/fuzz-beta>.

В следующем разделе мы обсудим удобную функцию Go, а именно кросс-компиляцию.

Кросс-компиляция

Кросс-компиляция — это процесс генерации двоичного исполняемого файла для архитектуры, отличающейся от той, в которой мы работаем, без доступа к другим машинам. Основное преимущество, которое мы получаем благодаря кросс-компиляции, заключается в том, что нам не нужна вторая или третья машина для создания и распространения исполняемых файлов для различных архитектур. Это означает, что в целом нам понадобится только одна машина для нашей разработки. К счастью, Go имеет встроенную поддержку кросс-компиляции.

Чтобы выполнить кросс-компиляцию исходного файла Go, нам нужно установить переменные среды `GOOS` и `GOARCH` для целевой операционной системы и архитектуры соответственно, что не так сложно, как кажется.



Вы можете найти список доступных значений для переменных среды `GOOS` и `GOARCH` по адресу <https://golang.org/doc/install/source>. Однако имейте в виду, что не все комбинации `GOOS` и `GOARCH` допустимы.

Код в файле `crossCompile.go` выглядит следующим образом:

```
package main

import (
    "fmt"
    "runtime"
)

func main() {
    fmt.Println("You are using ", runtime.GOOS, " ")
    fmt.Println("on a(n)", runtime.GOARCH, "machine")
    fmt.Println("with Go version", runtime.Version())
}
```

При выполнении этого кода на компьютере macOS с Go версии 1.16.5 мы получаем такой вывод:

```
$ go run crossCompile.go
You are using darwin on a(n) amd64 machine
with Go version go1.16.5
```

Чтобы скомпилировать файл `crossCompile.go` для *Linux OS*, работающей на машине с процессором `amd64`, достаточно выполнить следующую команду на компьютере с macOS:

```
$ env GOOS=linux GOARCH=amd64 go build crossCompile.go
$ file crossCompile
crossCompile: ELF 64-bit LSB executable, x86-64, version 1
(SYSV), statically linked, Go BuildID=GHF99KZkGUrFADRLsS7l/ty-
Ka44KVhMItrIvMZ61/rdRP5mt_yw2AEox_8uET/HqP0KyUBaOB87LY7gvVu, not
stripped
```

Передача этого файла на компьютер с Arch Linux и его запуск генерируют следующий вывод:

```
$ ./crossCompile
You are using linux on a(n) amd64 machine
with Go version go1.16.5
```

Здесь следует обратить внимание на то, что кросс-скомпилированный двоичный файл `crossCompile.go` выводит Go-версию машины, использовавшейся для его компиляции. Это оправданно, поскольку на целевой машине может даже не быть установлен Go.

Кросс-компиляция — отличная функция Go, которая может пригодиться, когда вы хотите сгенерировать несколько версий ваших исполняемых файлов через систему CI/CD и распространять их.

В следующем разделе мы обсудим директиву `go:generate`.

Использование директивы go:generate

Хотя `go:generate` напрямую не связана с тестированием или профилированием, это удобная расширенная возможность Go, и я считаю, что эта глава — идеальное место, где ее можно обсудить, поскольку она также может помочь вам в тестировании. Директива `go:generate`, связанная с командой `go generate`, была добавлена в Go 1.4 в целях автоматизации и позволяет запускать команды, описанные директивами, в существующих файлах.

Команда `go generate` поддерживает флаги `-v`, `-n` и `-x`. Флаг `-v` выводит имена пакетов и файлов по мере их обработки, в то время как флаг `-n` выводит команды, которые будут выполняться. Наконец, флаг `-x` выводит команды по мере их выполнения — это отлично подходит для отладки команд `go:generate`.

Основными причинами, по которым вам может потребоваться использовать директиву `go:generate`, можно назвать следующие:

- вам требуется скачать динамические данные из Интернета или какого-либо другого источника до выполнения кода Go;
- вы хотите выполнить какой-то код перед запуском кода Go;
- вам нужно сгенерировать номер версии или другие уникальные данные перед выполнением кода;
- вам необходимо убедиться, что у вас имеются образцы данных для работы. Например, вы можете поместить данные в базу данных с помощью `go:generate`.



Использование `go:generate` не считается хорошей практикой, поскольку оно скрывает вещи от разработчика и создает дополнительные зависимости, поэтому я стараюсь избегать этого насколько возможно, и обычно мне это удается. Но если вам это действительно нужно, то вы узнаете то, что вам нужно!

Использование `go:generate` показано в файле `goGenerate.go`:

```
package main

import "fmt"

//go:generate ./echo.sh
```

Строка выполняет скрипт `echo.sh`, который должен быть доступен в текущем каталоге.

```
//go:generate echo GOFILE: $GOFILE
//go:generate echo GOARCH: $GOARCH
```

```
//go:generate echo GOOS: $GOOS
//go:generate echo GOLINE: $GOLINE
//go:generate echo GOPACKAGE: $GOPACKAGE
```

`$GOFILE`, `$GOARCH`, `$GOOS`, `$GOLINE` и `$GOPACKAGE` являются специальными переменными и транслируются во время выполнения.

```
//go:generate echo DOLLAR: $DOLLAR
//go:generate echo Hello!
//go:generate ls -l
//go:generate ./hello.py
```

Выполняется скрипт на Python `hello.py`, который должен быть доступен в текущем каталоге.

```
func main() {
    fmt.Println("Hello there!")
}
```

Команда `go generate` не будет запускать оператора `fmt.Println()` или любые другие операторы, находящиеся в исходном файле Go. Наконец, имейте в виду, что `go generate` не выполняется автоматически и должна быть запущена явно.

В результате работы с файлом `goGenerate.go` из `~/go/src/` генерируется следующий вывод:

```
$ go mod init
$ go mod tidy
$ go generate
Hello world!
GOFILE: goGenerate.go
GOARCH: amd64
GOOS: darwin
GOLINE: 9
GOPACKAGE: main
```

Это выходные данные переменных `$GOFILE`, `$GOARCH`, `$GOOS`, `$GOLINE` и `$GOPACKAGE`, которые показывают значения этих переменных, определенных во время выполнения.

```
DOLLAR: $
```

Существует также специальная переменная с именем `$DOLLAR` для вывода символа доллара в выходных данных, поскольку `$` имеет особое значение в среде операционной системы.

```
Hello!
total 32
-rwxr-xr-x 1 mtsouk staff 32 Jun 2 18:18 echo.sh
```

```
-rw-r--r-- 1 mtsouk staff 45 Jun 2 16:15 go.mod
-rw-r--r-- 1 mtsouk staff 381 Jun 2 18:18 goGenerate.go
-rwxr-xr-x 1 mtsouk staff 52 Jun 2 18:18 hello.py
drwxr-xr-x 5 mtsouk staff 160 Jun 2 17:07 walk
```

Это вывод команды `ls -l`, которая показывает файлы, найденные в текущем каталоге во время выполнения кода. Информация может быть использована для проверки того, присутствуют ли во время выполнения некие необходимые файлы.

```
Hello from Python!
```

Последняя строка — это вывод наивного скрипта на Python.

При выполнении `go generate` с помощью `-n` мы получаем команды, которые будут выполняться:

```
$ go generate -n
./echo.sh
echo GOFILE: goGenerate.go
echo GOARCH: amd64
echo GOOS: darwin
echo GOLINE: 9
echo GOPACKAGE: main
echo DOLLAR: $
echo Hello!
ls -l
./hello.py
```

Таким образом, директива `go:generate` может помочь вам работать с ОС перед выполнением программы. Однако поскольку команда скрывает от разработчика некоторые вещи, ее использование должно быть ограничено.

В последнем разделе этой главы мы поговорим о примерах функций.

Создание примеров функций

Частью процесса документирования является генерация примера кода, в котором показано использование некоторых или всех функций и типов данных пакета. *Примеры функций* имеют много преимуществ, включая тот факт, что они являются исполняемыми тестами, которые выполняются инструментом `go test`. Следовательно, если пример функции содержит строку `// Output:`, то `go test` проверяет, соответствует ли вычисленный вывод значениям после этой строки. Хотя мы должны включать примеры функций в файлы Go, которые заканчиваются на `_test.go`, для примеров функций нам не нужно импортировать Go-пакет `testing`. Более того, название каждой функции примера должно начинаться

с Example. Наконец, примеры функций не принимают входных параметров и не возвращают результатов.

Мы проиллюстрируем примеры функций, используя код файлов `exampleFunctions.go` и `exampleFunctions_test.go`. Содержание `exampleFunctions.go` выглядит следующим образом:

```
package exampleFunctions

func LengthRange(s string) int {
    i := 0
    for _, _ = range s {
        i = i + 1
    }
    return i
}
```

Вышеприведенный код представляет собой обычный пакет, содержащий единственную функцию `LengthRange()`. Содержимое файла `exampleFunctions_test.go`, который включает в себя примеры функций, выглядит так:

```
package exampleFunctions

import "fmt"

func ExampleLengthRange() {
    fmt.Println(LengthRange("Mihalis"))
    fmt.Println(LengthRange("Mastering Go, 3rd edition!"))
    // Output:
    // 7
    // 7
}
```

В строках комментариев говорится, что ожидаемый результат равен 7 и 7, что явно неверно. Это будет видно после того, как мы запустим `go test`:

```
$ go test -v exampleFunctions*
=== RUN   ExampleLengthRange
--- FAIL: ExampleLengthRange (0.00s)
got:
7
26
want:
7
7
FAIL
FAIL      command-line-arguments  0.410s
FAIL
```

Как и ожидалось, в сгенерированном выводе имеется ошибка — второе сгенерированное значение равно 26 вместо ожидаемых 7. Если мы внесем необходимые исправления, то результат будет таким:

```
$ go test -v exampleFunctions*
 === RUN   ExampleLengthRange
 --- PASS: ExampleLengthRange (0.00s)
 PASS
 ok      command-line-arguments 1.157s
```

Примеры функций могут быть отличным инструментом для изучения возможностей пакета, и для проверки корректности функций, поэтому я предлагаю вам включать в ваши Go-пакеты как тестовый код, так и примеры функций. В качестве бонуса ваши тестовые функции *отображаются в документации пакета*, если вы решите ее создать.

Упражнения

- Реализуйте простую версию `ab(1)` (<https://httpd.apache.org/docs/2.4/programs/ab.html>) самостоятельно, используя горутины и каналы для тестирования производительности веб-сервисов.
- Напишите тестовые функции для приложения `phoneBook.go` из главы 3.
- Создайте тестовые функции для пакета, который вычисляет числа в последовательности Фибоначчи. Не забудьте реализовать этот пакет.
- Попробуйте получить значение `os.TempDir()` в различных операционных системах.
- Создайте три разные реализации функции, которая копирует двоичные файлы, и проведите их сравнительный анализ, определив более быструю. Можете ли вы объяснить, почему эта функция работает быстрее?

Резюме

В этой главе мы обсуждали директиву `go:generate`, профилирование и трассировку кода, бенчмаркинг и тестирование кода Go. Возможно, способ тестирования и бенчмаркинга в Go вам покажется скучным, но лишь потому, что *Go в целом скучен и предсказуем*, а это хорошо! Помните, что написание кода без ошибок важно, в то время как написание максимально быстрого кода важно далеко не всегда.

В большинстве случаев вам нужно уметь писать *достаточно быстрый* код. Таким образом, *потратьте больше времени на написание тестов, чем на бенчмарки*, если только ваш код не выполняется крайне медленно. Вы также узнали, как находить недоступный код и как кросс-компилировать Go-код.

Хотя обсуждения Go profiler и `go tool trace` далеки от завершения, вы должны понимать, что в таких темах, как профилирование и трассировка кода, ничто не заменит эксперименты и самостоятельное опробование новых методов!

Следующая глава посвящена созданию сервисов gRPC в Go.

Дополнительные ресурсы

- Пакет `generate`: <https://golang.org/pkg/cmd/go/internal/generate/>.
- Генерация кода: <https://blog.golang.org/generate>.
- Посмотрите на код `testing`: <https://golang.org/src/testing/testing.go>.
- Пакет `net/http/httptrace`: <https://golang.org/pkg/net/http/httptrace/>.
- Представляем трассировку HTTP от Яны Доган: <https://blog.golang.org/http-tracing>.
- GopherCon 2019: Dave Cheney – Two Go Programs, Three Different Profiling Techniques: <https://youtu.be/nok0aYiGiYA>.

12

Работа с gRPC

Эта глава посвящена работе с gRPC в Go. Система *gRPC* представляет собой альтернативу сервисам RESTful, разработанным Google. Главное преимущество gRPC заключается в скорости работы по сравнению с сообщениями REST и JSON. Кроме того, клиенты для сервисов gRPC также создаются быстрее благодаря имеющемуся инструментарию. Наконец, поскольку gRPC использует двоичный формат данных, он легче, чем сервисы RESTful, которые работают с форматом JSON.

Процесс создания сервера и клиента gRPC состоит из трех основных шагов. Первый – создание файла *языка определения интерфейса* (interface definition language, IDL), второй – разработка сервера gRPC, а третий – разработка клиента gRPC, который может взаимодействовать с сервером gRPC.

В этой главе рассматриваются следующие темы:

- введение в gRPC;
- определение файла языка определения интерфейса;
- разработка сервера gRPC;
- разработка клиента gRPC.

Введение в gRPC

gRPC – это система *удаленного вызова процедур* (Remote Procedure Calls) с открытым исходным кодом, которая была разработана в Google еще в 2015 году, построена поверх HTTP/2, позволяет легко создавать сервисы и использует

буферы протокола в качестве IDL, который определяет формат сообщений для обмена и интерфейс сервиса.

Клиенты и серверы gRPC вы можете написать на любом языке программирования, при этом вам не придется писать клиенты на том же языке программирования, что и их серверы. Это означает, что вы можете разработать клиент на Python, даже если сервер gRPC реализован на Go. Список поддерживаемых языков программирования включает следующие языки (но не ограничивается ими): Python, Java, C++, C#, PHP, Ruby и Kotlin.

Преимущества gRPC представлены ниже.

- Использование двоичного формата для обмена данными делает gRPC быстрее, чем сервисы, работающие с данными в обычном текстовом формате.
- Предоставляемые инструменты командной строки упрощают и ускоряют работу.
- Как только вы определили функции и сообщения сервиса gRPC, создавать серверы и клиенты для него становится проще, чем для сервисов RESTful.
- gRPC может использоваться для потоковых сервисов.
- Вам не нужно разбираться с деталями обмена данными, поскольку о них заботится gRPC.



Список преимуществ не должен заставлять вас думать, что gRPC — это панacea, у которой нет никаких недостатков. Всегда используйте для работы наиболее подходящий инструмент или технологию.

В следующем подразделе мы обсудим буферы протокола, которые являются основой сервисов gRPC.

Буферы протокола

Буфер протокола (protobuf) — это, по сути, метод сериализации структурированных данных. Частью буфера протокола является IDL. Поскольку буфер протокола использует двоичный формат для обмена данными, он занимает меньше места, чем обычные текстовые форматы сериализации. Однако чтобы данные были пригодными для машинного использования и удобочитаемыми для человека, их следует соответственно закодировать и декодировать. Буфер протокола имеет собственные типы данных, которые преобразуются в изначально поддерживаемые типы данных используемого языка программирования.

Вообще говоря, файл IDL служит центром каждого сервиса gRPC, поскольку определяет формат данных для обмена, а также интерфейс сервиса. Вы не можете иметь сервис gRPC, не имея под рукой файла буфера протокола. Строго говоря, файл буфера протокола содержит определение сервисов, их методы и формат сообщений, с помощью которых будет осуществляться обмен. Не будет преувеличением сказать, что если вы хотите понять сервис gRPC, то вам следует начать с рассмотрения его файла определения. В следующем разделе показан файл буфера протокола, который будет использоваться в нашем сервисе gRPC.

Определение файла языка определения интерфейса

Сервис gRPC, который мы разрабатываем, будет поддерживать следующую функциональность:

- сервер должен возвращать клиенту свою дату и время;
- сервер должен возвращать клиенту случайно сгенерированный пароль заданной длины;
- сервер должен возвращать клиенту случайные целые числа.

Прежде чем мы начнем разработку gRPC-клиента и сервера для нашего сервиса, нам необходимо определить IDL-файл. Нам понадобится отдельный репозиторий GitHub для размещения файлов, связанных с файлом IDL, и им станет <https://github.com/mactsouk/protoapi>.

Далее мы представим файл IDL, который называется `protoapi.proto`:

```
syntax = "proto3";
```

Представленный файл использует версию `proto3` языка буферов протокола. Существует также более старая версия языка `proto2`, имеющая некоторые незначительные синтаксические различия. Если вы не укажете, что хотите использовать `proto3`, то компилятор буфера протокола предположит использование `proto2`. Версия должна определяться в первой непустой строке без комментариев в файле `.proto`.

```
option go_package = "./;protoapi";
```

Инструменты gRPC генерируют Go-код из этого файла `.proto`. В предыдущей строке указано, что именем создаваемого Go-пакета будет `protoapi`. Выходные

данные будут записаны в текущий каталог как `protoapi.proto` из-за использования `./`.

```
service Random {
    rpc GetDate (RequestDateTime) returns (DateTime);
    rpc GetRandom (RandomParams) returns (RandomInt);
    rpc GetRandomPass (RequestPass) returns (RandomPass);
}
```

В этом блоке указывается имя сервиса gRPC (`Random`), а также поддерживаемые методы. Кроме того, в нем указываются сообщения, которыми необходимо обмениваться для взаимодействия. Итак, для `GetDate` клиенту необходимо отправить сообщение `RequestDateTime` и ожидать получения ответного сообщения `DateTime`.

Эти сообщения определены в одном и том же файле `.proto`.

```
// для случайного числа
```

Все файлы `.proto` поддерживают комментарии типа C и C++. Это означает, что в ваших файлах `.proto` вы можете использовать комментарии типа `// текст` и `/* текст */`.

```
message RandomParams {
    int64 Seed = 1;
    int64 Place = 2;
}
```

Генератор случайных чисел запускается с начального значения, которое в нашем случае задается клиентом и отправляется на сервер с сообщением `RandomParams`. Поле `Place` указывает место случайного числа, которое будет возвращено в последовательности случайно сгенерированных целых чисел.

```
message RandomInt {
    int64 Value = 1;
}
```

Два этих сообщения относятся к методу `GetRandom`. Сообщение `RandomParams` предназначено для установки параметров запроса, тогда как `randomInt` — для хранения случайного числа, которое генерируется сервером. Все поля сообщения имеют тип данных `int64`.

```
// для даты и времени
message DateTime {
    string Value = 1;
}

message RequestDateTime {
    string Value = 2;
}
```

Два этих сообщения предназначены для поддержки работы метода `GetDate`. Сообщение `RequestDateTime` является заглушкой в том смысле, что не содержит никаких полезных данных, — нам просто нужно иметь сообщение, которое клиент отправляет на сервер. Вы можете хранить любую информацию в поле `Value` сообщения `RequestDateTime`. Информация, возвращаемая сервером, сохраняется в виде значения `string` в сообщении `DateTime`.

```
// для случайного пароля
message RequestPass {
    int64 Seed = 1;
    int64 Length = 8;
}

message RandomPass {
    string Password = 1;
}
```

Наконец, два вышеприведенных сообщения предназначены для работы метода `GetRandomPass`. Итак, файл IDL:

- указывает, что мы используем `proto3`;
- определяет имя сервиса, то есть `Random`;
- указывает, что именем сгенерированного Go-пакета будет `protoapi`;
- определяет, что сервис gRPC будет поддерживать три метода: `GetDate`, `GetRandom` и `GetRandomPass`. Он также определяет имена сообщений, которые будут использоваться в этих трех вызовах методов;
- определяет формат шести сообщений, которые используются для обмена данными.

Следующий важный шаг — преобразование этого файла в формат, который может быть использован Go. Возможно, вам потребуется скачать некоторые дополнительные инструменты, чтобы обработать `protoapi.proto` или любой другой файл `.proto` и сгенерировать соответствующие файлы Go `.pb.go`. Двоичный файл компилятора буфера протокола называется `protocol` — на моем компьютере macOS мне пришлось установить `protoc` с помощью команды `brew install protobuf`. Аналогичным образом мне также пришлось установить пакеты `protocol-gen-go-grpc` и `protocol-gen-go`, используя Homebrew. Последние два пакета относятся к Go.

На компьютере с Linux вам необходимо установить `protobuf` с помощью любого менеджера пакетов и `protocol-gen-go` с помощью команды `go install github.com/golang/protobuf/protoc-gen-go@latest`. Аналогичным образом вы должны установить исполняемый файл `protocol-gen-go-grpc`, запустив `go install google.golang.org/grpc/cmd/protoc-gen-go-grpc@latest`.



Начиная с версии Go 1.16 `go install` является рекомендуемым способом сборки и установки пакетов в модульном режиме. Использование `go get` считается устаревшим. Однако при использовании `go install` не забудьте добавлять `@latest` после имени пакета, чтобы установить последнюю версию.

Итак, процесс преобразования требует следующего шага:

```
$ protoc --go_out=. --go_opt=paths=source_relative --go-grpc_out=.  
--go-grpc_opt=paths=source_relative protoapi.proto
```

После этого мы получим файлы `protoapi_grpc.pb.go` и `protoapi.pb.go` — оба расположены в корневом каталоге репозитория GitHub. Исходный файл `protoapi.pb.go` содержит сообщения, тогда как `protoapi_grpc.pb.go` содержит сервисы.

Первые десять строк `protoapi_grpc.pb.go` выглядят следующим образом:

```
// Code generated by protoc-gen-go-grpc. DO NOT EDIT.  
  
package protoapi
```

Как обсуждалось ранее, имя пакета — `protoapi`.

```
import (  
    context "context"  
    grpc "google.golang.org/grpc"  
    codes "google.golang.org/grpc/codes"  
    status "google.golang.org/grpc/status"  
)
```

Это блок `import`. Причина наличия здесь `context "context"` заключается в том, что `context` ранее был внешним пакетом Go, который не являлся частью стандартной библиотеки Go. Первые строки `protoapi.pb.go` следующие:

```
// Code generated by protoc-gen-go. DO NOT EDIT.  
// versions:  
//     protoc-gen-go v1.27.1  
//     protoc      v3.17.3  
// source: protoapi.proto  
  
package protoapi
```

Файлы `protoapi_grpc.pb.go` и `protoapi.pb.go` — часть Go-пакета `protoapi`, а это означает, что нам нужно включить их в наш код только один раз.

Следующий раздел посвящен разработке сервера gRPC.

Разработка сервера gRPC

В этом разделе мы собираемся создать сервер gRPC на основе файла `api.proto`, представленного в предыдущем разделе. Поскольку gRPC нужны внешние пакеты, нам понадобится репозиторий GitHub для размещения файлов. Им станет <https://github.com/mactsouk/grpc>.

Код `gServer.go` (расположенный в каталоге `server`), который относится к gRPC (некоторые функции были опущены для краткости), выглядит следующим образом:

```
package main

import (
    "context"
    "fmt"
    "math/rand"
    "net"
    "os"
    "time"
)
```

Эта программа для генерации случайных чисел использует `math/rand` вместо более безопасного `crypto/rand`, поскольку нам требуется начальное значение, чтобы иметь возможность воспроизводить последовательности случайных чисел.

```
"github.com/mactsouk/protoapi"
"google.golang.org/grpc"
"google.golang.org/grpc/reflection"
)
```

Блок `import` включает в себя внешние пакеты Google, а также `github.com/mactsouk/protoapi`, который мы создали ранее. Пакет `protoapi` содержит структуры, интерфейсы и функции, характерные для разрабатываемого сервиса gRPC, в то время как внешние пакеты Google содержат общий код, относящийся к gRPC.

```
type RandomServer struct {
    protoapi.UnimplementedRandomServer
}
```

Эта структура названа в честь сервиса gRPC. Она будет реализовывать интерфейс, требуемый сервером gRPC. Реализация интерфейса требует использования `protoapi.UnimplementedRandomServer` — это стандартная практика.

```
func (RandomServer) GetDate(ctx context.Context, r *protoapi.
RequestDateTime) (*protoapi.DateTime, error) {
```

```

currentTime := time.Now()
response := &protoapi.DateTime{
    Value: currentTime.String(),
}
return response, nil
}

```

Это первый метод интерфейса, названный в честь функции `GetDate`, находящейся в блоке `service` файла `protoapi.proto`. Метод не требует ввода данных от клиента, поэтому игнорирует параметр `r`.

```

func (RandomServer) GetRandom(ctx context.Context, r *protoapi.
RandomParams) (*protoapi.RandomInt, error) {
    rand.Seed(r.GetSeed())
    place := r.GetPlace()

```

Методы `GetSeed()` и `getPlace()` реализованы в `protoc`, связаны с полями `protoapi.RandomParams` и должны использоваться для считывания данных из клиентского сообщения.

```

temp := random(min, max)
for {
    place--
    if place <= 0 {
        break
    }
    temp = random(min, max)
}

response := &protoapi.RandomInt{
    Value: int64(temp),
}
return response, nil
}

```

Сервер создает переменную `protoapi.RandomInt`, которая будет возвращена клиенту. На этом заканчивается реализация второго метода интерфейса.

```

func (RandomServer) GetRandomPass(ctx context.Context, r *protoapi.Request-
Pass) (*protoapi.RandomPass, error) {
    rand.Seed(r.GetSeed())
    temp := getString(r.GetLength())

```

Методы `GetSeed()` и `getLength()` реализованы в `protoc`, связаны с полями `protoapi.RequestPass` и должны использоваться для чтения данных из клиентского сообщения.

```

response := &protoapi.RandomPass{
    Password: temp,
}
return response, nil
}

```

В последней части `GetRandomPass()` мы создаем ответ (`protoapi.RandomPass`) для отправки клиенту.

```
var port = ":8080"

func main() {
    if len(os.Args) == 1 {
        fmt.Println("Using default port:", port)
    } else {
        port = os.Args[1]
    }
}
```

Первая часть `main()` предназначена для указания TCP-порта, который будет использоваться для сервиса.

```
server := grpc.NewServer()
```

Оператор выше создает новый сервер gRPC, который не привязан ни к какому конкретному сервису gRPC.

```
var randomServer RandomServer
protoapi.RegisterRandomServer(server, randomServer)
```

Эти операторы вызывают `protoapi.RegisterRandomServer()` для создания сервера gRPC для нашего конкретного сервиса.

```
reflection.Register(server)
```

Вызывать `reflection.Register()` не обязательно, но это полезно, когда нужно перечислить доступные сервисы, найденные на сервере, — в нашем случае это можно было бы опустить.

```
listen, err := net.Listen("tcp", port)
if err != nil {
    fmt.Println(err)
    return
}
```

Здесь мы запускаем сервис TCP, который прослушивает нужный TCP-порт.

```
fmt.Println("Serving requests...")
server.Serve(listen)
}
```

Последняя часть программы призвана сообщить серверу gRPC о необходимости начать обслуживание клиентских запросов. Это происходит путем вызова метода `Serve()` и использования сетевых параметров, хранящихся в `listen`.



Утилита `curl(1)` не работает с двоичными данными и поэтому не может быть использована для тестирования сервера gRPC. Однако существует альтернатива для тестирования сервисов gRPC, и ею является утилита `grpcurl` (<https://github.com/fullstorydev/grpcurl>).

Теперь, когда у нас готов сервер gRPC, разработаем клиент, который поможет нам протестировать работу этого сервера.

Разработка клиента gRPC

В этом разделе представлена разработка клиента gRPC на основе файла `api.proto`, размещенного ранее. Основной целью клиента будет тестирование функциональности сервера. Однако что действительно важно, так это реализация трех вспомогательных функций, соответствующих различным вызовам RPC, поскольку эти три функции позволяют нам взаимодействовать с сервером gRPC. Назначение функции `main()` в `gClient.go` состоит в том, чтобы использовать эти три вспомогательные функции.

Итак, код `gClient.go` выглядит следующим образом:

```
package main

import (
    "context"
    "fmt"
    "math/rand"
    "os"
    "time"

    "github.com/mactsouk/protoapi"
    "google.golang.org/grpc"
)

var port = ":8080"

func AskingDateTime(ctx context.Context, m protoapi.RandomClient)
(*protoapi.DateTime, error) {
```

Вы можете назвать функцию `AskingDateTime()` как угодно. Однако, чтобы поже иметь возможность вызвать `GetDate()`, сигнатура функции должна содержать параметр `context.Context`, а также параметр `RandomClient`. Клиенту не нужно реализовывать какие-либо функции IDL — он просто должен их вызывать.

```
request := &protoapi.RequestDateTime{
    Value: "Please send me the date and time",
}
```

Сначала мы создаем переменную `protoapi.RequestDateTime`, которая содержит данные для запроса клиента.

```
return m.GetDate(ctx, request)
}
```

Затем мы вызываем метод `GetDate()`, чтобы отправить клиентский запрос на сервер. Он обрабатывается кодом в модуле `protoapi` — мы просто вызываем `GetDate()` с правильными параметрами. Именно здесь реализация первой вспомогательной функции заканчивается — хотя наличие такой функции не обязательно, она делает код более чистым.

```
func AskPass(ctx context.Context, m protoapi.RandomClient, seed int64,
length int64) (*protoapi.RandomPass, error) {
    request := &protoapi.RequestPass{
        Seed: seed,
        Length: length,
    }
```

Вспомогательная функция `AskPass()` предназначена для вызова метода gRPC `GetRandomPass()`, служащего для получения случайного пароля от серверного процесса. Сначала функция создает переменную `protoapi.RequestPass` с заданными значениями `Seed` и `Length`, которые являются параметрами `AskPass()`.

```
    return m.GetRandomPass(ctx, request)
}
```

Затем мы вызываем `GetRandomPass()`, чтобы отправить клиентский запрос на сервер и получить ответ. Наконец, функция возвращается.

Из-за того, как работает gRPC, и упрощающих работу инструментов реализация `AskPass()` получается короткой. Выполнение того же самого с помощью сервисов RESTful потребует большего количества кода.

```
func AskRandom(ctx context.Context, m protoapi.RandomClient, seed
int64, place int64) (*protoapi.RandomInt, error) {
    request := &protoapi.RandomParams{
        Seed: seed,
        Place: place,
    }

    return m.GetRandom(ctx, request)
}
```

Последняя вспомогательная функция, `AskRandom()`, работает аналогичным образом. Мы создаем клиентское сообщение (`protoapi.RandomParams`), отправляем его на сервер, вызывая `GetRandom()`, и получаем ответ сервера, возвращенный `GetRandom()`.

```
func main() {
    if len(os.Args) == 1 {
        fmt.Println("Using default port:", port)
    } else {
        port = os.Args[1]
    }
```

```

conn, err := grpc.Dial(port, grpc.WithInsecure())
if err != nil {
    fmt.Println("Dial:", err)
    return
}

```

Клиент gRPC должен подключиться к серверу gRPC с помощью параметра `grpc.Dial()`. Однако мы еще не закончили, так как нам нужно указать сервис gRPC, к которому клиент собирается подключиться, и через некоторое время сделаем это. Функция `grpc.Insecure()`, которая передается в качестве параметра `grpc.Dial()`, возвращает значение `DialOption`, отключающее безопасность для клиентского соединения.

```

rand.Seed(time.Now().Unix())
seed := int64(rand.Intn(100))

```

Из-за различных начальных значений, генерируемых и отправляемых на сервер gRPC каждый раз, когда выполняется клиентский код, мы будем получать от сервера gRPC разные случайные значения и пароли.

```
client := protoapi.NewRandomClient(conn)
```

Далее нам нужно создать gRPC-клиент, вызвав `protoapi.NewRandomClient()` и передав TCP-соединение в `protoapi.NewRandomClient()`. Эта переменная `client` будет использоваться для всех взаимодействий с сервером. Имя вызываемой функции зависит от имени сервиса gRPC — это позволяет нам различать разные сервисы gRPC, которые может поддерживать машина.

```

r, err := AskingDateTime(context.Background(), client)
if err != nil {
    fmt.Println(err)
    return
}
fmt.Println("Server Date and Time:", r.Value)

```

Прежде всего мы вызываем вспомогательную функцию `AskingDateTime()`, чтобы получить дату и время с сервера gRPC.

```

length := int64(rand.Intn(20))
p, err := AskPass(context.Background(), client, 100, length+1)
if err != nil {
    fmt.Println(err)
    return
}
fmt.Println("Random Password:", p.Password)

```

Затем мы вызываем `AskPass()`, чтобы получить случайно сгенерированный пароль. Длина пароля определяется оператором `length := int64(rand.Intn(20))`.

```
place := int64(rand.Intn(100))
i, err := AskRandom(context.Background(), client, seed, place)
if err != nil {
    fmt.Println(err)
    return
}
fmt.Println("Random Integer 1:", i.Value)
```

Затем мы тестируем `AskRandom()` с разными параметрами, чтобы убедиться, что он вернет разные значения.

```
k, err := AskRandom(context.Background(), client, seed, place-1)
if err != nil {
    fmt.Println(err)
    return
}
fmt.Println("Random Integer 2:", k.Value)
}
```

Завершив работу как с сервером, так и с клиентом, мы готовы к тестированию.

Тестирование сервера gRPC с клиентом. Теперь, когда мы разработали как сервер, так и клиент, мы готовы их использовать. Для начала мы должны запустить файл `gServer.go`:

```
$ go run gServer.go
Using default port: :8080
Serving requests..
```

Серверный процесс не выдает никаких других выходных данных.

Затем мы выполняем `gClient.go` без каких-либо параметров командной строки. Вывод, который вы получите из `gClient.go`, должен быть похож на следующий:

```
$ go run gClient.go
Using default port: :8080
Server Date and Time: 2021-07-05 08:32:19.654905 +0300 EEST
m=+2.197816168
Random Password: $1!usiz|36
Random Integer 1: 92
Random Integer 2: 78
```

Первая строка представляет собой выполнение клиента из оболочки UNIX, а вторая касается TCP-порта, используемого для подключения к серверу gRPC. Следующая строка выходных данных показывает дату и время, возвращаемые сервером gRPC. Далее представлен случайный пароль, сгенерированный сервером, а также два случайных целых числа.

Если мы выполним `gClient.go` более одного раза, то получим другой результат:

```
$ go run gClient.go
Using default port: :8080
Server Date and Time: 2021-07-05 08:32:23.831445 +0300 EEST
m=+6.374535148
Random Password: $1!usiz|36N}D0*}{
Random Integer 1: 10
Random Integer 2: 68
```

Тот факт, что сервер gRPC вернул разные значения, доказывает, что этот сервер работает так, как ожидалось.

Система gRPC может гораздо больше того, что представлено в этой главе, например, обмениваться массивами сообщений и потоковой передачей — серверы RESTful не могут использоваться для потоковой передачи данных. Однако обсуждение этих вопросов выходит за рамки данной книги.

Упражнения

- Преобразуйте программу `gClient.go` в утилиту командной строки с помощью `cobra`.
- Попробуйте преобразовать программу `gServer.go` в сервер RESTful.
- Создайте сервис RESTful, который действует gRPC для обмена данными. Определите REST API, который вы собираетесь поддерживать, но используйте gRPC для связи между сервером RESTful и сервером gRPC — в этом случае сервер RESTful будет действовать как клиент gRPC для сервера gRPC.
- Создайте собственный сервис gRPC, который реализует сложение и вычисление целых чисел.
- Насколько легко или сложно было бы преобразовать приложение телефонной книги в сервис gRPC?
- Реализуйте сервис gRPC, который вычисляет длину строки.

Резюме

Система gRPC быстра, проста в использовании и понимании и обменивается данными в двоичном формате. В этой главе вы узнали, как определить методы и сообщения сервиса gRPC, перевести их в код Go и разработать сервер и клиент для данного сервиса.

Итак, следует ли вам использовать gRPC или все же стоит остановиться на сервисах RESTful? Только вы можете ответить на этот вопрос. Вы должны выбрать то, что кажется вам более подходящим. Однако если вы все еще сомневаетесь и не можете принять решение, то начните с разработки сервиса RESTful, а затем внедрите тот же сервис с помощью gRPC. После этого вы будете готовы выбрать.

Последняя глава книги посвящена *дженерикам*, которые являются функцией Go, официально включенной в Go в 2022 году. Мы покажем вам код Go, который поможет лучше в них разобраться.

Дополнительные ресурсы

- gRPC: <https://grpc.io/>.
- Руководство по языку протокола Buffers 3: <https://developers.google.com/protocol-buffers/docs/proto3>.
- Утилита `grpcurl`: <https://github.com/fullstorydev/grpcurl>.
- Сайт Йохана Брандхорста: <https://jbrandhorst.com/page/about/>.
- Документация по пакету `google.golang.org/grpc`: <https://pkg.go.dev/google.golang.org/grpc>.
- Руководство по Go и gRPC: <https://grpc.io/docs/languages/go/basics/>.
- Буферы протокола: <https://developers.google.com/protocol-buffers>.

13

Дженерики Go

Эта глава посвящена *дженерикам* и тому, как использовать новый синтаксис для написания универсальных функций и определения универсальных типов данных.



Новый синтаксис дженериков появился в Go 1.18, который, согласно циклу разработки Go, был официально выпущен в марте 2022 года.

Позвольте мне прежде всего кое-что прояснить: вам не обязательно использовать дженерики Go, если вы этого не хотите, и вы все равно можете написать замечательное, эффективное, легкое в сопровождении и корректное программное обеспечение на Go! Вдобавок тот факт, что вы можете использовать дженерики и поддерживать множество типов данных, если не все доступные типы данных, не означает, что вы должны это делать. Всегда *поддерживайте требуемые типы данных*, ни больше ни меньше, но не забывайте следить за будущим ваших данных и возможностью поддержки типов данных, которые не были известны на момент написания вашего кода.

В этой главе:

- введение в дженерики;
- ограничения;
- определение новых типов данных с помощью дженериков;
- интерфейсы и дженерики;
- рефлексия и дженерики;
- заключительные замечания: как выглядит будущее для разработчиков Go?

Введение в дженерики

Дженерики — это функционал, который дает вам возможность не указывать точный тип данных одного или нескольких параметров функции главным образом потому, что вы хотите сделать свои функции как можно более универсальными. Другими словами, дженерики позволяют функциям обрабатывать несколько типов данных, при этом нет необходимости писать специальный код, как в случае с пустым интерфейсом или интерфейсами в целом. Однако при работе с интерфейсами в Go вам приходится писать дополнительный код для определения типа данных переменной интерфейса, с которой вы работаете, чего нельзя сказать о дженериках.

Позвольте мне начать с представления небольшого примера кода, который реализует функцию, ясно показывающую случай, когда дженерики могут быть удобны и избавят вас от необходимости писать много кода:

```
func PrintSlice[T any](s []T) {
    for _, v := range s {
        fmt.Println(v)
    }
}
```

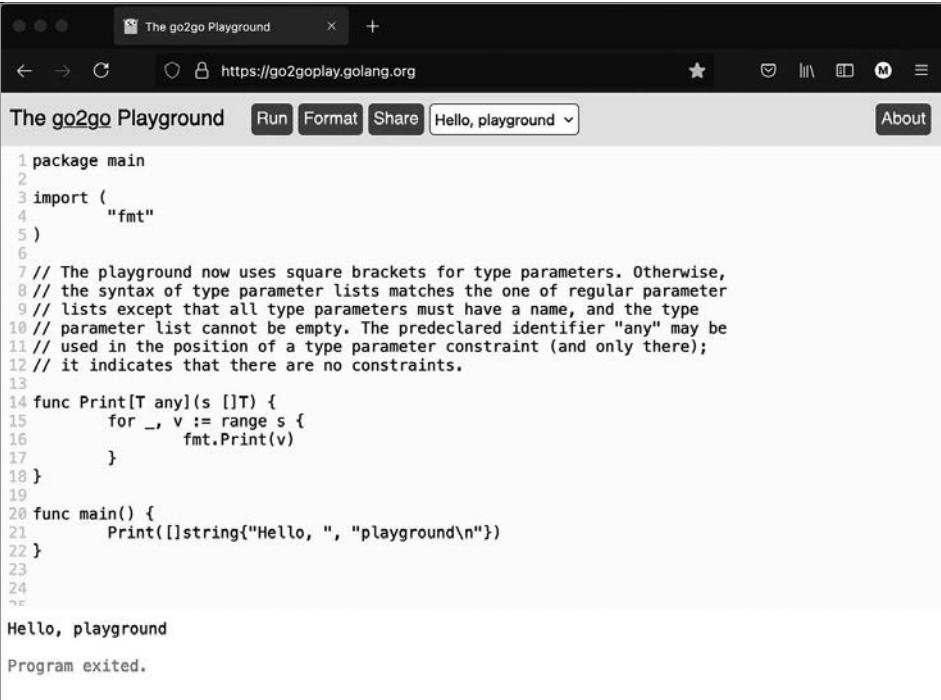
Итак, что мы имеем здесь? Есть функция `PrintSlice()`, которая принимает фрагмент любого типа данных. Это обозначается использованием `[]T` в сигнатуре функции в сочетании с частью `[T any]`. Эта часть сообщает компилятору, что тип данных `T` будет определен во время выполнения. Мы также можем свободно применять несколько типов данных, используя нотацию `[T, U, W any]`, после чего должны использовать типы данных `T, U, W` в сигнатуре функции.

Ключевое слово `any` сообщает компилятору, что нет *никаких ограничений* в отношении типа данных `T`. Мы собираемся обсудить ограничения через некоторое время — пока просто изучите синтаксис дженериков.

Теперь представьте, что пишете отдельные функции для реализации функциональности `PrintSlice()` для срезов целых чисел, строк, чисел с плавающей запятой, комплексных значений и т. д. Итак, есть показательный случай, когда использование дженериков упрощает код и сокращает ваши усилия в плане программирования. Однако не все случаи настолько очевидны, и вы должны быть очень осторожны, чтобы не злоупотреблять `any`.

Но что произойдет, если вы захотите поэкспериментировать с дженериками? Есть решение, которое заключается в том, чтобы посетить <https://gotipplay.golang.org>, после чего разместить и запустить там свой код.

Начальный экран <https://gotipplay.golang.org> представлен на рис. 13.1. В верхней части вы пишете код, а в нижней — получаете результаты своего кода или потенциальные сообщения об ошибках после нажатия кнопки Run.



The screenshot shows a web browser window titled "The go2go Playground". The address bar displays the URL <https://go2goplay.golang.org>. Below the address bar, there's a toolbar with buttons for "Run", "Format", "Share", and a dropdown menu set to "Hello, playground". To the right of the toolbar is an "About" button. The main area contains the following Go code:

```
1 package main
2
3 import (
4     "fmt"
5 )
6
7 // The playground now uses square brackets for type parameters. Otherwise,
8 // the syntax of type parameter lists matches the one of regular parameter
9 // lists except that all type parameters must have a name, and the type
10 // parameter list cannot be empty. The predeclared identifier "any" may be
11 // used in the position of a type parameter constraint (and only there);
12 // it indicates that there are no constraints.
13
14 func Print[T any](s []T) {
15     for _, v := range s {
16         fmt.Println(v)
17     }
18 }
19
20 func main() {
21     Print([]string{"Hello, ", "playground\n"})
22 }
23
24 
```

Below the code, the output is displayed:

```
Hello, playground
Program exited.
```

Рис. 13.1. Gotip Playground для тестирования дженериков

Остальная часть этой главы посвящена выполнению кода Go в <https://gotipplay.golang.org>. Тем не менее я собираюсь представлять код и выходные данные как обычно, скопировав и вставив их из <https://gotipplay.golang.org>.

Ниже приведен код (`hw.go`), использующий дженерики. Он поможет вам лучше понять их, прежде чем мы перейдем к более сложным примерам:

```
package main

import (
    "fmt"
)

func PrintSlice[T any](s []T) {
    for _, v := range s {
```

```
    fmt.Println(v, " ")
}
fmt.Println()
}
```

Функция `PrintSlice()` похожа на функцию, которую мы видели ранее в этой главе. Однако она выводит элементы каждого фрагмента в одной строке и выводит новую строку с помощью `fmt.Println()`.

```
func main() {
    PrintSlice([]int{1, 2, 3})
    PrintSlice([]string{"a", "b", "c"})
    PrintSlice([]float64{1.2, -2.33, 4.55})
}
```

Здесь мы вызываем `PrintSlice()` с тремя различными типами данных: `int`, `string` и `float64`. Компилятор Go не будет на это жаловаться. Вместо этого он будет выполнять код так, как если бы у нас были три отдельные функции, по одной для каждого типа данных.

Следовательно, запуск `hw.go` приводит к следующему результату:

```
1 2 3
a b c
1.2 -2.33 4.55
```

Таким образом, каждый срез выводится, как и ожидалось, с использованием одной общей функции.

Держа в голове эту информацию, начнем с обсуждения дженериков и ограничений.

Ограничения

Допустим, у нас есть функция, которая работает с дженериками и умножает два числовых значения. Должна ли она работать со всеми типами данных? Может ли она работать со всеми ними? Можете ли вы умножить две строки или две структуры? Решением, позволяющим избежать такого рода проблемы, будет использование *ограничений*.

Забудем на время об умножении и подумаем о чем-нибудь более простом. Допустим, мы хотим сравнить переменные на равенство — есть ли способ сообщить Go, что мы хотим работать только со значениями, которые можно сравнивать? Go 1.18 поставляет с предопределенными ограничениями — одно из них называется `comparable` и включает типы данных, которые можно сравнивать на предмет равенства или неравенства.

В коде `allowed.go` показано использование `comparable`:

```
package main

import (
    "fmt"
)

func Same[T comparable](a, b T) bool {
    if a == b {
        return true
    }
    return false
}
```

Функция `Same()` использует *предопределённое* ограничение `comparable` вместо `any`. На самом деле `comparable` — просто предопределенный интерфейс, содержащий все типы данных, которые можно сравнить с помощью `==` или `!=`. Нам не нужно писать никакой дополнительный код для проверки наших входных данных, поскольку сигнатура функции гарантирует, что мы будем иметь дело только с допустимыми типами данных.

```
func main() {
    fmt.Println("4 = 3 is", Same(4,3))
    fmt.Println("aa = aa is", Same("aa","aa"))
    fmt.Println("4.1 = 4.15 is", Same(4.1,4.15))
}
```

Функция `main()` вызывает `Same()` три раза и выводит результаты.

При выполнении `allowed.go` мы получаем такой вывод:

```
4 = 3 is false
aa = aa is true
4.1 = 4.15 is false
```

Поскольку только `Same("aa", "aa")` дает `true`, мы получаем соответствующий вывод.

Если вы попытаетесь запустить оператор, включающий `Same([]int{1,2}, []int{1,3})`, которая пытается сравнить два среза, Gotip Playground выдаст следующее сообщение об ошибке:

```
type checking failed for main
prog.go2:19:31: []int does not satisfy comparable
```

Это происходит потому, что мы не можем напрямую сравнить два среза — такого рода функциональность должна быть реализована вручную.

В следующем подразделе показано, как создать собственные ограничения.

Создание ограничений. В примере, представленном в этом подразделе, мы определяем типы данных, которые разрешено передавать в качестве параметров универсальной функции с помощью *интерфейса*. Код `numeric.go` выглядит следующим образом:

```
package main

import (
    "fmt"
)

type Numeric interface {
    type int, int8, int16, int32, int64, float64
}
```

Здесь мы задаем новый интерфейс под названием `Numeric`, который определяет список поддерживаемых типов данных. Вы можете использовать любой тип данных при условии, что он может быть использован с универсальной функцией, которую вы собираетесь реализовать. В этом случае мы могли бы добавить `string` или `uint` в список поддерживаемых типов данных.

```
func Add[T Numeric](a, b T) T {
    return a + b
}
```

Это определение универсальной функции, которая использует ограничение `Numeric`.

```
func main() {
    fmt.Println("4 + 3 =", Add(4,3))
    fmt.Println("4.1 + 3.2 =", Add(4.1,3.2))
}
```

Этот код представляет собой реализацию функции `main()` с вызовами `Add()`.

При выполнении `numeric.go` мы получаем такой вывод:

```
4 + 3 = 7
4.1 + 3.2 = 7.3
```

Тем не менее правила Go все еще действуют. Поэтому, попытавшись вызвать `Add(4.1,3)`, вы получите следующее сообщение об ошибке:

```
type checking failed for main
prog.go2:16:33: default type int of 3 does not match inferred type
float64 for T
```

Причина этой ошибки заключается в том, что функция `Add()` ожидает два параметра *одного и того же типа данных*. Однако `4.1` — это `float64`, тогда как `3` — это `int`, так что это не один и тот же тип данных.

В следующем разделе показано, как использовать дженерики при определении новых типов данных.

Определение новых типов данных с помощью дженерииков

В этом разделе мы, используя дженерики, собираемся создать новый тип данных, который представлен в файле `newDT.go`. Код `newDT.go` следующий:

```
package main

import (
    "fmt"
    "errors"
)

type TreeLast[T any] []T
```

В этом операторе мы объявляем новый тип данных `TreeLast`, который использует дженерики.

```
func (t TreeLast[T]) replaceLast(element T) (TreeLast[T], error) {
    if len(t) == 0 {
        return t, errors.New("This is empty!")
    }

    t[len(t) - 1] = element
    return t, nil
}
```

Метод `replaceLast()` работает с переменными `TreeLast`. Кроме сигнатуры функции, больше нет ничего, что указывало бы на использование дженерииков.

```
func main() {
    tempStr := TreeLast[string]{“aa”, “bb”}
    fmt.Println(tempStr)
    tempStr.replaceLast(“cc”)
    fmt.Println(tempStr)
```

В этой первой части функции `main()` мы создаем переменную `TreeLast` со строковыми значениями `aa` и `bb`, после чего заменяем значение `bb` на `cc`, используя вызов `replaceLast("cc")`.

```

    tempInt := TreeLast[int]{12, -3}
    fmt.Println(tempInt)
    tempInt.replaceLast(0)
    fmt.Println(tempInt)
}

```

Вторая часть функции `main()` выполняет то же самое, что и первая, используя переменную `TreeLast`, заполненную значениями `int`. Таким образом, `TreeLast` работает как со `string`, так и со значениями `int` без каких-либо проблем.

При выполнении `newDT.go` мы получаем такой вывод:

```
[aa bb]
[aa cc]
```

Выше мы видим выходные данные, относящиеся к переменной `TreeLast[string]`.

```
[12 -3]
[12 0]
```

Конечный вывод относится к переменной `TreeLast[int]`.

Следующий подраздел посвящен использованию дженериков в Go-структуратах.

Использование дженериков в Go-структуратах. Здесь мы собираемся реализовать связанный список, который работает с дженериками. Это один из случаев, когда использование дженериков упрощает работу, поскольку позволяет реализовать связанный список единожды, оставляя возможность работать с несколькими типами данных.

Код `structures.go` выглядит так:

```

package main

import (
    "fmt"
)

type node[T any] struct {
    Data T
    next *node[T]
}

```

Структура `node` использует дженерики для поддержки узлов, которые могут хранить все виды данных. Это не означает, что поле `next` структуры `node` может указывать на другой `node` с полем `Data` другого типа данных. Правило, согласно которому связанный список содержит элементы одного и того же типа данных, по-прежнему применяется. Это просто означает, что если вы хотите создать три

связанных списка: для хранения значений `string`, значений `int` и записей JSON заданного типа данных `struct` — вам не нужно писать для этого какой-либо дополнительный код.

```
type list[T any] struct {
    start *node[T]
}
```

Это определение корневого узла связанныного списка узлов `node`. И `list`, и `node` должны совместно использовать один и тот же тип данных `T`. Однако, как указывалось ранее, это не мешает вам создать несколько связанных списков с различными типами данных.

Вы по-прежнему можете заменить `any` на ограничение как в определении `node`, так и в `list`, если захотите ограничить список разрешенных типов данных.

```
func (l *list[T]) add(data T) {
    n := node[T]{
        Data: data,
        next: nil,
    }
```

Функция `add()` — это дженерик, дающий возможность работать со всеми видами узлов. Помимо сигнатуры `add()`, весь оставшийся код не связан с использованием дженериков.

```
if l.start == nil {
    l.start = &n
    return
}

if l.start.next == nil {
    l.start.next = &n
    return
}
```

Эти два блока `if` относятся к добавлению нового узла в связанный список.

```
temp := l.start
l.start = l.start.next
l.add(data)
l.start = temp
}
```

Последняя часть `add()` связана с определением правильных ассоциаций между узлами при добавлении нового узла в список.

```
func main() {
    var myList list[int]
```

Сначала мы определяем в функции `main()` связанный список значений `int`, который и является тем списком, с которым мы собираемся работать.

```
fmt.Println(myList)
```

Начальное значение `myList` равно `nil`, так как список пуст и не содержит никаких узлов.

```
myList.add(12)
myList.add(9)
myList.add(3)
myList.add(9)
```

В этой первой части мы добавляем четыре элемента в связанный список.

```
// вывести все элементы
for {
    fmt.Println("*", myList.start)
    if myList.start == nil {
        break
    }
    myList.start = myList.start.next
}
```

Последняя часть функции `main()` предназначена для вывода всех элементов списка путем обхода с помощью поля `next`, которое указывает на следующий узел в списке.

При выполнении `structures.go` мы получаем такой вывод:

```
{<nil>}
* &{12 0xc00010a060}
* &{9 0xc00010a080}
* &{3 0xc00010a0b0}
* &{9 <nil>}
* <nil>
```

Немного обсудим результат. Первая строка показывает, что значение пустого списка равно `nil`. Первый узел списка содержит значение `12` и адрес памяти (`0xc00010a060`), который указывает на второй узел. Это продолжается до тех пор, пока мы не достигнем последнего узла, содержащего значение `9` (которое дважды появляется в этом связанном списке и указывает на `nil`, поскольку это последний узел). Таким образом, благодаря использованию дженериков связанный список может работать с несколькими типами данных.

В следующем разделе мы обсудим различия между использованием интерфейсов и дженериков для поддержки нескольких типов данных.

Интерфейсы и дженерики

В этом разделе представлена программа, которая увеличивает числовое значение на единицу, используя интерфейсы и дженерики, чтобы мы могли сравнить детали реализации.

В коде `interfaces.go` показаны эти два метода и содержится следующий код:

```
package main

import (
    "fmt"
)

type Numeric interface {
    type int, int8, int16, int32, int64, float64
}
```

Здесь мы определяем ограничение `Numeric` для ограничения разрешенных типов данных.

```
func Print(s interface{}) {
    // переключатель типа
    switch s.(type) {
```

Функция `Print()` использует пустой интерфейс для получения входных данных и переключатель типа для работы с этим входным параметром.

Проще говоря, мы используем *переключатель типа* для различия поддерживаемых типов данных. В данном случае поддерживаемые типы данных — это просто `int` и `float64`, что связано с реализацией переключателя типа. Однако добавление большего количества типов данных требует изменения кода, что является не самым эффективным решением.

```
    case int:
        fmt.Println(s.(int)+1)
```

Эта ветвь отвечает за обработку случая `int`.

```
    case float64:
        fmt.Println(s.(float64)+1)
```

Эта ветка обрабатывает `float64`.

```
    default:
        fmt.Println("Unknown data type!")
}
```

Ветвь `default` отвечает за обработку всех неподдерживаемых типов данных.

Самая большая проблема с функцией `Print()` заключается в том, что из-за использования пустого интерфейса она принимает все виды входных данных. В результате сигнатура функции не помогает нам ограничить допустимые типы данных. Вторая проблема с `Print()` состоит в том, что нам нужно специально обрабатывать каждый случай, — обработка большего количества случаев означает написание большего количества кода.

С другой стороны, компилятору не нужно ничего угадывать касательно этого кода, чего нельзя сказать о дженериках, где компилятору и среде выполнения предстоит проделать больше работы. Такого рода работа приводит к задержкам во времени выполнения.

```
func PrintGenerics[T any](s T) {  
    fmt.Println(s)  
}
```

Функция `PrintGenerics()` является универсальной и может просто и элегантно обрабатывать все доступные типы данных.

```
func PrintNumeric[T Numeric](s T) {  
    fmt.Println(s+1)  
}
```

Функция `PrintNumeric()` поддерживает все числовые типы данных с помощью ограничения `Numeric`. Тут нет необходимости специально добавлять код для поддержки каждого отдельного типа данных, как это происходит с `Print()`.

```
func main() {  
    Print(12)  
    Print(-1.23)  
    Print("Hi!")
```

Первая часть функции `main()` использует `Print()` с различными типами ввода: значения `int`, `float64` и `string` соответственно.

```
PrintGenerics(1)  
PrintGenerics("a")  
PrintGenerics(-2.33)
```

Как указывалось ранее, функция `PrintGenerics()` работает со всеми типами данных, включая `string`.

```
PrintNumeric(1)  
PrintNumeric(-2.33)  
}
```

Последняя часть функции `main()` использует `PrintNumeric()` только с числовыми значениями из-за применения ограничения `Numeric`.

При выполнении `interfaces.go` мы получаем такой вывод:

```
13  
-0.2299999999999998  
Unknown data type!
```

Эти три строки выходных данных принадлежат функции `Print()`, которая использует пустой интерфейс.

```
1  
a  
-2.33
```

Эти три строки выходных данных принадлежат функции `PrintGenerics()`, которая использует дженерики и поддерживает все доступные типы данных. В результате она не может увеличить значение своих входных данных, поскольку мы не знаем наверняка, что имеем дело с числовым значением. Следовательно, она просто выводит заданные входные данные.

```
2  
-1.33
```

Последние две строки генерируются двумя вызовами функции `PrintNumeric()`, которые работают с использованием ограничения `Numeric`.

Таким образом, на практике, когда вам приходится поддерживать несколько типов данных, использование дженерики может оказаться лучшим выбором, чем использование интерфейсов.

В следующем разделе мы обсудим использование рефлексии как способа избежать применения дженерики.

Рефлексия и дженерики

В этом разделе мы разработаем утилиту, которая выводит элементы среза двумя способами: используя, во-первых, рефлексию, а во-вторых, дженерики.

Код `reflection.go` выглядит следующим образом:

```
package main  
  
import (
```

```

    "fmt"
    "reflect"
)

func PrintReflection(s interface{}) {
    fmt.Println("** Reflection")
    val := reflect.ValueOf(s)

    if val.Kind() != reflect.Slice {
        return
    }

    for i := 0; i < val.Len(); i++ {
        fmt.Print(val.Index(i).Interface(), " ")
    }
    fmt.Println()
}

```

Внутренне функция `PrintReflection()` работает только со срезами. Однако поскольку мы не можем выразить это в сигнатуре функции, нам придется принять пустой параметр интерфейса. Кроме того, мы должны написать больше кода, чтобы получить желаемый результат.

Если вдаваться в детали, то, во-первых, нам нужно убедиться, что мы обрабатываем срез (`reflect.Slice`), и, во-вторых, мы должны вывести элементы среза, используя цикл `for`, что довольно некрасиво.

```

func PrintSlice[T any](s []T) {
    fmt.Println("** Generics")

    for _, v := range s {
        fmt.Print(v, " ")
    }
    fmt.Println()
}

```

Повторюсь, реализация функции-дженерика более проста, и, следовательно, ее легче понять. Более того, сигнатура функции указывает, что в качестве параметров функции принимаются только срезы — нам не нужно выполнять никакие дополнительные проверки, поскольку это работа Go-компилятора. Наконец, мы используем простой цикл `for` с `range` для вывода элементов среза.

```

func main() {
    PrintSlice([]int{1, 2, 3})
    PrintSlice([]string{"a", "b", "c"})
    PrintSlice([]float64{1.2, -2.33, 4.55})

    PrintReflection([]int{1, 2, 3})
}

```

```
PrintReflection([]string{"a", "b", "c"})
PrintReflection([]float64{1.2, -2.33, 4.55})
}
```

Функция `main()` вызывает `PrintSlice()` и `PrintReflection()` с различными типами входных данных для проверки их работы.

При выполнении `reflection.go` мы получаем такой вывод:

```
** Generics
1 2 3
** Generics
a b c
** Generics
1.2 -2.33 4.55
```

Первые шесть строк создаются с использованием дженериков и выводят элементы среза значений `int`, среза значений `string` и среза значений `float64`.

```
** Reflection
1 2 3
** Reflection
a b c
** Reflection
1.2 -2.33 4.55
```

Последние шесть строк выходных данных выдают тот же результат, однако на этот раз с помощью рефлексии. В выходных данных нет никакой разницы — все различия заключаются в коде реализации их вывода, `PrintReflection()` и `PrintSlice()`. Как и ожидалось, код дженерика проще и короче, чем код Go, использующий рефлексию, особенно когда необходимо поддерживать множество различных типов данных.

Упражнения

- Создайте в файле `structures.go` метод `PrintMe()`, который выводит все элементы связанного списка.
- Создайте в файле `reflection.go` две дополнительные функции, чтобы получить возможность вывода строк с использованием рефлексии и дженериков.
- Реализуйте функции `delete()` и `search()`, используя дженерики для связанного списка, находящегося в файле `structures.go`.
- Реализуйте двусвязный список, используя дженерики, начав с кода, находящегося в файле `structures.go`.

Резюме

В этой главе представлены дженерики и дано обоснование их изобретения. Кроме того, в ней представлен синтаксис Go для дженериков, а также некоторые проблемы, которые могут возникнуть, если вы используете дженерики непрородуманно. В стандартную библиотеку Go были внесены изменения, необходимые для поддержки дженериков, и был выпущен новый пакет `slices`, дающий возможность воспользоваться преимуществами новых языковых возможностей.

Хотя функция с дженериками является более гибкой, код с ними обычно выполняется медленнее, чем код, работающий с предопределенными статическими типами данных. Итак, цена, которую вы платите за гибкость, — это скорость выполнения. Аналогично код Go с дженериками компилируется дольше, чем эквивалентный код, который не использует дженерики. В конце концов, программирование — это понимание стоимости ваших решений. Только тогда вы можете считать себя программистом. Таким образом, важно понимать цену использования дженериков вместо интерфейсов, рефлексии или других методов.

Итак, как же выглядит будущее для разработчиков Go? Если коротко, то замечательно! Вам уже должно нравиться программировать на Go, и вы должны продолжать делать это по мере развития языка. Если вы хотите узнать о последних и наилучших разработках в Go, обсуждаемых командой, то вам обязательно следует посетить официальный сайт Go team на GitHub по адресу <https://github.com/golang>.

Go поможет вам создавать отличное программное обеспечение! Дерзайте!

Дополнительные ресурсы

- Google I/O 2012 — знакомьтесь с командой Go: <https://youtu.be/sln-gJaURzk>.
- Познакомьтесь с авторами Go: <https://youtu.be/3yghHvvZQmA>.
- Брайан Керниган берет интервью у Кена Томпсона (видео не имеет прямого отношения к Go): https://youtu.be/EY6q5dv_B-o.
- Брайан Керниган об успешном языковом дизайне (видео не имеет прямого отношения к Go): https://youtu.be/Sg4U4r_AgJU.
- Брайан Керниган: программирование на UNIX, C, AWK, AMPL и Go (из подкаста Лекса Фридмана): <https://youtu.be/O9upVbGSBFo>.
- Почему дженерики? <https://blog.golang.org/why-generics>.
- Следующий шаг для дженериков: <https://blog.golang.org/generics-next-step>.
- Предложение для пакета `slices`: <https://github.com/golang/go/issues/45955>.



Примечание от автора

Быть хорошим программистом сложно, но возможно. Продолжайте совершенствоваться, и кто знает, возможно, станете знаменитым настолько, что и о вас снимут фильм!

Спасибо вам за то, что прочитали эту книгу. Не стесняйтесь обращаться ко мне с предложениями, вопросами или, возможно, идеями для других книг!

Soli Deo gloria

ПРИЛОЖЕНИЕ

Сборщик мусора Go

Это приложение полностью посвящено *сборщику мусора* в Go (Garbage Collector, GC). Эта важнейшая часть Go может повлиять на производительность вашего кода больше, чем любой другой компонент данного языка. Мы начнем с разговора о куче и стеке.

Куча и стек

Куча — место, где языки программирования хранят глобальные переменные, а также в ней происходит сборка мусора. *Стек* — место, где языки программирования хранят временные переменные, используемые функциями. Каждая функция имеет собственный стек. Поскольку горутины расположены в пользовательском пространстве, среда выполнения Go отвечает за правила, которые управляют их работой. Кроме того, каждая горутина имеет собственный стек, в то время как куча является «общей» для всех горутин.

В C++, создавая новые переменные с помощью оператора `new`, вы знаете, что они отправляются в кучу. Это не относится к Go и использованию функций `new()` и `make()`. В Go компилятор решает, куда будет помещена новая переменная, основываясь на ее размере и результате *анализа локальности*. Именно по этой причине вы можете возвращать указатели на локальные переменные из функций Go.



Поскольку мы нечасто встречали функцию `new()` в этой книге, имейте в виду, что она возвращает указатели на инициализированную память.

Если вы хотите знать, где переменные программы распределяются Go, то можете использовать флаг сборщика мусора `-m`. Подход проиллюстрирован в файле `allocate.go`. Это обычная программа, которая не нуждается в изменениях для отображения дополнительных выходных данных, поскольку все детали обрабатываются Go.

```
package main

import "fmt"

const VAT = 24

type Item struct {
    Description string
    Value        float64
}

func Value(price float64) float64 {
    total := price + price*VAT/100
    return total
}

func main() {
    t := Item{Description: "Keyboard", Value: 100}
    t.Value = Value(t.Value)
    fmt.Println(t)

    tP := &Item{}
    *&tP.Description = "Mouse"
    *&tP.Value = 100
    fmt.Println(tP)
}
```

При выполнении `allocate.go` мы получаем такой вывод, являющийся результатом использования `-gcflags '-m'`, который изменяет сгенерированный исполняемый файл. Вы не должны создавать исполняемые двоичные файлы с помощью флагов `-gcflags`, если они после этого отправляются в реальную работу:

```
$ go run -gcflags '-m' allocate.go
# command-line-arguments
./allocate.go:12:6: can inline Value
./allocate.go:19:17: inlining call to Value
./allocate.go:20:13: inlining call to fmt.Println
./allocate.go:25:13: inlining call to fmt.Println
./allocate.go:20:13: t escapes to heap
```

Сообщение `t escapes to heap` означает, что переменная `t` изолирована от функции. Проще говоря, это означает, что `t` используется вне функции и не имеет локальной области видимости (поскольку передается извне функции). Однако это не обязательно означает, что переменная перемещена в кучу.

Есть другое сообщение, которое здесь не показано: `moved to heap`. Это сообщение показывает, что компилятор решил переместить переменную в кучу, поскольку она может использоваться вне функции.

```
./allocate.go:20:13: []interface {}{...} does not escape
./allocate.go:22:8: &Item{} escapes to heap
./allocate.go:25:13: []interface {}{...} does not escape
```

Сообщение `doesn't escape` указывает на то, что интерфейс не изолируется в кучу. Компилятор Go выполняет *анализ локальности*, чтобы выяснить, нужно ли выделять память в куче или переменная должна оставаться в стеке.

```
<autogenerated>:1: .this does not escape
{Keyboard 124}
&{Mouse 100}
```

Последние две строки выходных данных состоят из выходных данных, сгенерированных двумя операторами `fmt.Println()`.

Если вы хотите получить более подробный вывод, то можете использовать флаг `-m` дважды:

```
$ go run -gcflags '-m -m' allocate.go
# command-line-arguments
./allocate.go:12:6: can inline Value with cost 13 as: func(float64)
float64 { total := price + price * VAT / 100; return total }
./allocate.go:17:6: cannot inline main: function too complex: cost 199
exceeds budget 80
./allocate.go:19:17: inlining call to Value func(float64) float64 {
total := price + price * VAT / 100; return total }
./allocate.go:20:13: inlining call to fmt.Println func(...interface {})
(int, error) { var fmt..autotmp_3 int; fmt..autotmp_3 = <N>; var fmt..
autotmp_4 error; fmt..autotmp_4 = <N>; fmt..autotmp_3, fmt..autotmp_4 =
fmt.Fprintln(io.Writer(os.Stdout), fmt.a...); return fmt..autotmp_3,
fmt..autotmp_4 }
./allocate.go:25:13: inlining call to fmt.Println func(...interface {})
(int, error) { var fmt..autotmp_3 int; fmt..autotmp_3 = <N>; var fmt..
autotmp_4 error; fmt..autotmp_4 = <N>; fmt..autotmp_3, fmt..autotmp_4 =
fmt.Fprintln(io.Writer(os.Stdout), fmt.a...); return fmt..autotmp_3,
fmt..autotmp_4 }
./allocate.go:22:8: &Item{} escapes to heap
```

```

./allocate.go:22:8:    flow: tP = &{storage for &Item{}}:
./allocate.go:22:8:        from &Item{} (spill) at ./allocate.go:22:8
./allocate.go:22:8:        from tP := &Item{} (assign) at ./allocate.
go:22:5
./allocate.go:22:8:    flow: ~arg0 = tP:
./allocate.go:22:8:        from tP (interface-converted) at ./allocate.
go:25:13
./allocate.go:22:8:    from ~arg0 := tP (assign-pair) at ./allocate.
go:25:13
./allocate.go:22:8:    flow: {storage for []interface {}{...}} = ~arg0:
./allocate.go:22:8:        from []interface {}{...} (slice-literalelement)
at ./allocate.go:25:13
./allocate.go:22:8:    flow: fmt.a = &{storage for []interface {}{...}}:
./allocate.go:22:8:        from []interface {}{...} (spill) at ./allocate.
go:25:13
./allocate.go:22:8:    from fmt.a = []interface {}{...} (assign) at ./
allocate.go:25:13
./allocate.go:22:8:    flow: {heap} = *fmt.a:
./allocate.go:22:8:        from fmt.Fprintln(io.Writer(os.Stdout),
fmt.a...) (call parameter) at ./allocate.go:25:13
./allocate.go:20:13: t escapes to heap:
./allocate.go:20:13:    flow: ~arg0 = &{storage for t}:
./allocate.go:20:13:        from t (spill) at ./allocate.go:20:13
./allocate.go:20:13:        from ~arg0 := t (assign-pair) at ./allocate.
go:20:13
./allocate.go:20:13:    flow: {storage for []interface {}{...}} = ~arg0:
./allocate.go:20:13:        from []interface {}{...} (slice-literalelement)
at ./allocate.go:20:13
./allocate.go:20:13:    flow: fmt.a = &{storage for []interface {}{...}}:
./allocate.go:20:13:    from []interface {}{...} (spill) at ./
allocate.go:20:13
./allocate.go:20:13:    from fmt.a = []interface {}{...} (assign) at ./
allocate.go:20:13
./allocate.go:20:13:    flow: {heap} = *fmt.a:
./allocate.go:20:13:        from fmt.Fprintln(io.Writer(os.Stdout),
fmt.a...) (call parameter) at ./allocate.go:20:13
./allocate.go:20:13: t escapes to heap
./allocate.go:20:13: []interface {}{...} does not escape
./allocate.go:22:8: &Item{} escapes to heap
./allocate.go:25:13: []interface {}{...} does not escape
< autogenerated >:1: .this does not escape
{Keyboard 124}
&{Mouse 100}

```

Однако я нахожу этот вывод перенасыщенным и сложным. Обычно однократного использования `-m` достаточно, чтобы показать происходящее «за кулисами» в отношении программной кучи и стека.

Теперь, когда вы знаете о куче и стеке, продолжим обсуждение сборки мусора.

Сборка мусора

Сборка мусора — это процесс освобождения неиспользуемого пространства памяти. Другими словами, сборщик видит объекты, которые находятся вне области видимости и на которые больше нельзя ссылаться, после чего освобождает пространство памяти, занимаемое ими. Этот процесс происходит параллельно во время выполнения программы на Go, а не до или после выполнения программы. В документации по реализации сборщика мусора Go указано следующее:

«Сборщик мусора запускается одновременно с потоками-мутаторами, имеет точный тип (также известный как precise) и позволяет нескольким потокам сборщика мусора выполнять параллельно. Это одновременные отслеживание и очистка, в которых используется барьер записи. Он не зависит от поколения и не уплотняется. Распределение выполняется с использованием областей распределения, разделенных по размеру на P, чтобы минимизировать фрагментацию при одновременном устранении блокировок в общем случае».

К счастью, стандартная библиотека Go предлагает функции, которые позволяют изучить работу сборщика мусора и узнать больше о том, как он работает. Эти функции показаны в утилите `gColl.go`, которую можно найти в каталоге `ch03` репозитория книги на GitHub. Исходный код `gColl.go` представлен здесь в виде фрагментов.

```
package main

import (
    "fmt"
    "runtime"
    "time"
)
```

Вам понадобится пакет `runtime`, поскольку он позволяет получать информацию о системе среды выполнения Go, которая, помимо прочего, включает в себя работу сборщика мусора.

```
func printStats(mem runtime.MemStats) {
    runtime.ReadMemStats(&mem)
    fmt.Println("mem.Alloc:", mem.Alloc)
    fmt.Println("mem.TotalAlloc:", mem.TotalAlloc)
    fmt.Println("mem.HeapAlloc:", mem.HeapAlloc)
    fmt.Println("mem.NumGC:", mem.NumGC, "\n")
}
```

Основная цель `printStats()` состоит в том, чтобы избежать многократного написания одного и того же Go-кода. Вызов `runtime.ReadMemStats()` получает для вас последнюю статистику по сборке мусора.

```
func main() {
    var mem runtime.MemStats
    printStats(mem)

    for i := 0; i < 10; i++ {
        // Выделение 50 000 000 байт
        s := make([]byte, 50000000) if
        s == nil {
            fmt.Println("Operation failed!")
        }
    }
    printStats(mem)
}
```

В этой части у нас есть цикл `for`, который создает десятибайтовые срезы по 50 000 000 байт каждый. Причина этого в том, что, выделяя большие объемы памяти, мы можем запустить сборщик.

```
for i := 0; i < 10; i++ {
    // Выделение 100 000 000 байт
    s := make([]byte, 100000000) if
    s == nil {
        fmt.Println("Operation failed!")
    }
    time.Sleep(5 * time.Second)
}
printStats(mem)
}
```

Последняя часть программы выделяет еще больше памяти — на этот раз каждый байтовый срез содержит 100 000 000 байт.

При выполнении `gColl.go` на компьютере macOS Big Sur с 24 Гбайт оперативной памяти мы получаем такой вывод:

```
$ go run gColl.go
mem.Alloc: 124616
mem.TotalAlloc: 124616
mem.HeapAlloc: 124616
mem.NumGC: 0

mem.Alloc: 50124368
mem.TotalAlloc: 500175120
mem.HeapAlloc: 50124368
mem.NumGC: 9
```

```
mem.Alloc: 122536
mem.TotalAlloc: 1500257968
mem.HeapAlloc: 122536
mem.NumGC: 19
```

Значение `mem.Alloc` — это байты выделенных объектов кучи, а *выделены* все объекты, которые сборщик еще не освободил. `mem.TotalAlloc` показывает совокупное количество байтов, выделенных для объектов кучи. Это число не уменьшается при освобождении объектов и, значит, продолжает увеличиваться. Следовательно, показывается общее количество байтов, выделенных для объектов кучи во время выполнения программы. `mem.HeapAlloc` — то же самое, что `mem.Alloc`. Наконец, `mem.NumGC` показывает общее количество завершенных циклов сборки мусора. Чем больше это значение, тем больше вам нужно учитывать, как вы распределяете память в своем коде и есть ли способ оптимизировать этот процесс.

Если вы хотите получить еще более подробный вывод о работе сборщика мусора, то можете объединить `go run gColl.go` с `GODEBUG=gctrace=1`. Помимо обычных выходных данных программы, вы получите некоторые дополнительные показатели, что показано в следующем выводе:

```
$ GODEBUG=gctrace=1 go run gColl.go
gc 1 @0.021s 0%: 0.020+0.32+0.015 ms clock, 0.16+0.17/0.33/0.22+0.12 ms
cpu, 4->4->0 MB, 5 MB goal, 8 P
gc 2 @0.041s 0%: 0.074+0.32+0.003 ms clock, 0.59+0.087/0.37/0.45+0.030
ms cpu, 4->4->0 MB, 5 MB goal, 8 P
.
.
.
gc 18 @40.152s 0%: 0.065+0.14+0.013 ms clock, 0.52+0/0.12/0.042+0.10 ms
cpu, 95->95->0 MB, 96 MB goal, 8 P
gc 19 @45.160s 0%: 0.028+0.12+0.003 ms clock, 0.22+0/0.13/0.081+0.028
ms cpu, 95->95->0 MB, 96 MB goal, 8 P
mem.Alloc: 120672
mem.TotalAlloc: 1500256376
mem.HeapAlloc: 120672
mem.NumGC: 19
```

Как и ранее, у нас такое же количество завершенных циклов сборки мусора (19). Однако мы получаем дополнительную информацию о размере кучи для каждого цикла. Итак, для 19-го цикла сборки мусора (`gc 19`) мы видим следующее:

```
gc 19 @45.160s 0%: 0.028+0.12+0.003 ms clock, 0.22+0/0.13/0.081+0.028
ms cpu, 95->95->0 MB, 96 MB goal, 8 P
```

Разберемся с триплетом `95->95->0` МВ в строке вывода выше. Первое значение (95) — размер кучи непосредственно перед запуском GC. Второе значение (95) — ее размер в тот момент, когда сборщик завершает свою работу. Понятное значение — размер оперативной кучи (0).

Алгоритм трех цветов

Работа сборщика мусора в Go основана на *алгоритме трех цветов*. Обратите внимание, что он не уникален для Go и может использоваться и в других языках программирования.

Строго говоря, официальным названием алгоритма, используемого в Go, является *алгоритм трехцветной маркировки и очистки* (tricolor mark-and-sweep algorithm). Он может работать одновременно с программой и использует *барьер записи*. Это означает, что при запуске программы Go планировщик Go отвечает за планирование приложения, а также за сборщик мусора, который также запускается как горутина. Это как если бы планировщик Go работал с обычным приложением с несколькими *горутинами*!



Основная идея, лежащая в основе этого алгоритма, была предложена Эдсгером В. Дейкстрой, Лесли Лэмпартом, А. Дж. Мартином, К. С. Шолтеном и Э. Ф. М. Стеффенсом и впервые проиллюстрирована в статье под названием «Динамическая сборка мусора: упражнение в сотрудничестве».

Основной принцип, лежащий в основе алгоритма трехцветной разметки и очистки, заключается в том, что он *делит объекты кучи* на три различных набора в соответствии с их цветом (который назначается алгоритмом и может быть черным, серым или белым). Объекты *черного набора* гарантированно не будут иметь указателей ни на один объект *белого*. С другой стороны, объект из белого набора может указывать на объект из черного, поскольку это не влияет на работу сборщика мусора. Объекты *серого набора* могут содержать указатели на некоторые объекты белого. Наконец, объекты *белого набора* являются кандидатами на сборку мусора.

Итак, когда начинается сборка мусора, все объекты становятся белыми, а сборщик мусора посещает все корневые объекты и окрашивает их в серый цвет. *Корни* — это объекты, к которым приложение может получить прямой доступ, включая глобальные переменные и другие объекты в стеке. Эти объекты в основном зависят от кода Go конкретной программы.

После этого сборщик выбирает серый объект, делает его черным и начинает проверять, есть ли у этого объекта указатели на другие объекты из белого набора. Поэтому, когда объект из серого набора сканируется на наличие указателей на другие объекты, он окрашивается в черный цвет. Если при сканировании обнаруживается, что у этого конкретного объекта есть один или несколько указателей на белый объект, то сборщик помещает этот белый объект в набор серых. Процесс продолжается до тех пор, пока существуют объекты в сером наборе. После этого объекты в белом наборе становятся недоступными, и их пространство в памяти может быть использовано повторно. Следовательно, на этот момент считается, что элементы белого набора являются собранным мусором. Пожалуйста, обратите внимание, что ни один объект не может перейти непосредственно из черного набора в белый, и это позволяет алгоритму работать и иметь возможность очищать объекты из белого набора. Как упоминалось ранее, ни один объект из черного набора не может напрямую указывать на объект из белого. Кроме того, если объект из серого набора становится недоступным в какой-то момент цикла сборки мусора, то будет собран не в этом цикле сборки мусора, а в следующем! Хотя это и не оптимальная ситуация, все не так уж плохо.

Во время этого процесса запущенное приложение называется *мутатором* (mutator). Он запускает небольшую функцию с именем *барьер записи*, которая выполняется каждый раз, когда изменяется указатель в куче. Если указатель на объект в куче изменен, это означает, что такой объект теперь доступен, барьер записи окрашивает его в серый цвет и помещает в серый список. Мутатор отвечает за инвариант, заключающийся в том, что ни один элемент черного набора не имеет указателя на элемент белого. Это достигается с помощью функции барьера записи. Невыполнение этого инварианта разрушит процесс сборки мусора и, скорее всего, приведет к сбою вашей программы с довольно плохим и нежелательным исходом!

Итак, есть три цвета: черный, белый и серый. Когда алгоритм запускается, все объекты окрашиваются в белый цвет. По мере продолжения работы алгоритма белые объекты перемещаются в один из двух других наборов. Объекты, оставшиеся в белом наборе, — это те, которые в какой-то момент будут очищены.

На рис. П1 показаны три цветовых набора с включенными в них объектами.

В то время как объект E, который находится в белом наборе, может получить доступ к объекту F, к нему не может получить доступ ни один другой объект, поскольку не указывает на объект E, что делает его идеальным кандидатом для сборки мусора! Кроме того, объекты A, B и C являются корневыми объектами и всегда доступны; следовательно, они не могут быть собраны.

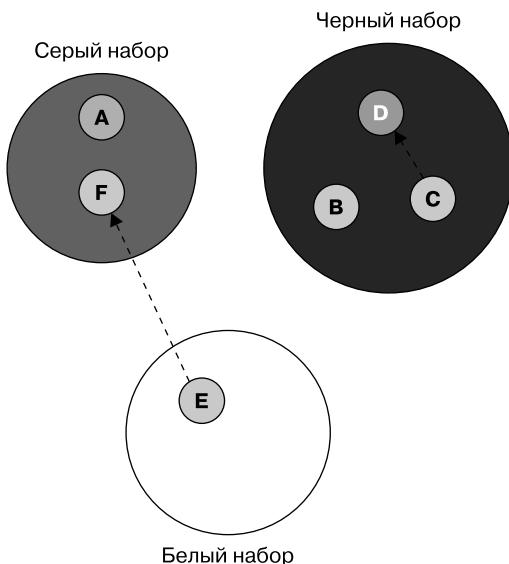


Рис. П1. Сборщик мусора Go представляет кучу программы в виде графика

Можете ли вы предположить, что произойдет дальше? Алгоритму придется обработать оставшиеся элементы серого набора, а это значит, что оба объекта A и F перейдут в черный набор.

Объект A переходит в черный набор, поскольку является корневым элементом, а F – потому, что не указывает ни на какой другой объект, пока он в сером наборе.

После того как объект E будет собран, объект F станет недоступным и будет собран в следующем цикле сборщика мусора, поскольку недоступный объект не может волшебным образом стать доступным на следующей итерации цикла сборки мусора.



Сборка мусора Go также может применяться к таким переменным, как каналы. Обнаружив, что канал недоступен (то есть когда переменная канала больше недоступна), сборщик освобождает ресурсы, даже если тот не был закрыт.

Go позволяет вам вручную инициировать циклы сборки мусора с помощью оператора `runtime.GC()` в вашем коде. Однако имейте в виду, что `runtime.GC()` блокирует вызывающий объект, и это может привести к блокировке всей про-

грамммы, особенно если вы запускаете очень загруженную программу Go с большим количеством объектов. В основном это происходит потому, что невозможно выполнять сборку мусора, в то время как все остальное стремительно меняется. Это не позволит сборщику мусора четко идентифицировать элементы белого, черного и серого наборов. Такой статус сбора мусора также называется *безопасной точкой сбора мусора*.



Обратите внимание, что сборщик мусора постоянно совершенствуется членами команды Go. Они пытаются ускорить процесс, сокращая количество сканирований, необходимых для обработки данных из трех наборов. Однако, несмотря на различные оптимизации, общая идея, лежащая в основе алгоритма, остается неизменной.

Вы можете найти длинный и относительно совершенный Go-код сборщика мусора по адресу <https://github.com/golang/go/blob/master/src/runtime/mgc.go>. Его можно изучить, если хотите узнать еще больше информации об операции сборки мусора. Вы даже можете внести изменения в этот код, если у вас хватит смелости!

Подробнее о работе сборщика мусора в Go

В этом подразделе подробнее рассказывается о сборщике мусора Go и представлена дополнительная информация о его работе. Основной проблемой сборщика мусора в Go является *низкое значение задержки*, что в основном означает короткие паузы в его работе для обеспечения работы в режиме реального времени.

Вместе с тем программа постоянно создает новые объекты и манипулирует существующими с помощью указателей. Этот процесс может закончиться созданием объектов, доступ к которым больше невозможен, поскольку нет указателей на них. Затем эти объекты становятся мусором и ждут, пока сборщик мусора очистит их и освободит место в памяти. После этого освободившееся пространство памяти будет готово к повторному использованию.

Алгоритм маркировки и очистки самый простой из используемых алгоритмов. Алгоритм останавливает выполнение программы (*stop-the-world garbage collector*), чтобы посетить все доступные объекты кучи программы и *маркировать* их. После этого он *очищает* недоступные объекты. На этапе маркировки алгоритма каждый объект помечается как белый, серый или черный. Дочерние элементы серого объекта окрашены в серый цвет, тогда как исходный серый объект теперь окрашен в черный. Фаза очистки начинается, когда больше нет серых объектов

для изучения. Этот метод работает, поскольку нет указателей из черного набора на белый, что является фундаментальным инвариантом алгоритма.

Хотя алгоритм маркировки и очистки прост, он приостанавливает выполнение программы, что означает добавление задержки к фактическому процессу. Go пытается снизить эту задержку, запуская сборщик как параллельный процесс и используя алгоритм трех цветов, описанный в предыдущем разделе. Однако другие процессы могут перемещать указатели или создавать новые объекты, пока сборщик выполняется в параллельном режиме. Этот факт может осложнить работу сборщика мусора.

В результате основной принцип, который позволяет алгоритму трех цветов работать параллельно, сохраняя фундаментальный инвариант алгоритма маркировки и очистки, заключается в том, что *ни один объект из черного набора не может указывать на объект из белого*.

Решением этой проблемы будет исправление всех ситуаций, которые могут вызвать проблемы в алгоритме. Следовательно, новые объекты должны быть помещены в серый набор, поскольку таким образом фундаментальный инвариант алгоритма маркировки и очистки не может быть изменен.

Кроме того, при перемещении указателя программы объект, на который указывает указатель, окрашивается в серый. Серый набор действует как барьер между белым и черным набором. Наконец, каждый раз при перемещении указателя автоматически выполняется код Go, который является барьером записи, упомянутым ранее, выполняющим перекраску. Задержка, возникающая при выполнении кода, препятствующего записи, — это цена, которую мы должны заплатить за возможность параллельного запуска сборщика мусора.

Пожалуйста, обратите внимание, что язык программирования *Java* имеет множество сборщиков мусора, которые легко настраиваются с помощью множества параметров. Один из этих сборщиков в Java называется *G1* и рекомендуется для приложений с низкой задержкой. Хотя в Go нет нескольких сборщиков, в нем все же есть возможности для настройки сборщика мусора для ваших приложений.

В следующем подразделе мы обсудим карты и срезы с точки зрения сборки мусора, поскольку иногда то, как мы обрабатываем переменные, влияет на работу сборщика мусора.

Карты, срезы и сборщик мусора Go

В этом подразделе мы обсудим работу сборщика мусора Go применительно к картам и срезам. Цель данного подраздела — позволить вам писать код, который упрощает работу сборщика мусора.

Использование среза

В представленном ниже примере для хранения большого количества структур используется срез. Каждая структура хранит два целочисленных значения. Это реализовано в программе `sliceGC.go` следующим образом:

```
package main

import (
    "runtime"
)

type data_struct {
    i, j int
}

func main() {
    var N = 80000000
    var structure []data_struct
    for i := 0; i < N; i++ {
        value := int(i)
        structure = append(structure, data_struct{value, value})
    }

    runtime.GC()
    _ = structure[0]
}
```

Последний оператор (`_ = structure[0]`) используется для предотвращения слишком ранней сборки сборщиком мусора переменной `structure`, поскольку на нее не ссылаются и она не используется вне цикла `for`. Та же техника будет использоваться в трех последующих Go-программах.

Помимо этой важной детали, цикл `for` используется для помещения всех значений в структуры, которые хранятся в переменной среза `structure`. Эквивалентным способом проделать это будет использование `runtime.KeepAlive()`. Программа не генерирует никаких выходных данных — она просто запускает сборщик мусора, используя вызов `runtime.GC()`.

Использование карты с указателями

Здесь мы задействуем карту для хранения указателей. На этот раз карта использует целочисленные ключи, которые ссылаются на указатели. Программа называется `mapStar.go` и содержит следующий код Go:

```
package main

import (
    "runtime"
)
```

```
func main() {
    var N = 80000000
    myMap := make(map[int]*int)
    for i := 0; i < N; i++ {
        value := int(i)
        myMap[value] = &value
    }

    runtime.GC()
    _ = myMap[0]
}
```

Работа программы такая же, как и в `sliceGC.go` из предыдущего раздела. Что ее отличает, так это использование карты (`make(map[int]*int)`) для хранения указателей на `int`. Как и прежде, программа не выдает никаких выходных данных.

Использование карты без указателей

Здесь мы используем карту, которая хранит целочисленные значения напрямую вместо указателей на целые числа. Важным кодом программы `mapNoStar.go` будет следующий:

```
func main() {
    var N = 80000000
    myMap := make(map[int]int)
    for i := 0; i < N; i++ {
        value := int(i)
        myMap[value] = value
    }
    runtime.GC()
    _ = myMap[0]
}
```

И снова программа не выдает никаких выходных данных.

Разделение карты

В последней программе мы используем другую технику, называемую сегментированием, когда мы разбиваем одну длинную карту на карту карт. Реализация функции `main()` в программе `mapSplit.go` выглядит следующим образом:

```
func main() {
    var N = 80000000

    split := make([]map[int]int, 2000)
    for i := range split {

```

```
    split[i] = make(map[int]int)
}

for i := 0; i < N; i++ {
    value := int(i)
    split[i%2000][value] = value
}
runtime.GC()
_ = split[0][0]
}
```

Код использует два цикла `for`: один — для создания карты карт, а другой — для сохранения желаемых значений данных в карте карт.

Поскольку все четыре программы используют огромные структуры данных, они потребляют большие объемы памяти. Программы, которые потребляют много места в памяти, чаще запускают сборщик мусора Go. В следующем подразделе мы увидим оценку представленных методов.

Сравнение эффективности представленных техник

В этом подразделе мы сравниваем производительность каждой из четырех реализаций, используя команду `time` из `zsh(1)`, которая очень похожа на команду `time(1)` UNIX.

```
$ time go run sliceGC.go
go run sliceGC.go 2.68s user 1.39s system 127% cpu 3.184 total
$ time go run mapStar.go
go run mapStar.go 55.58s user 3.24s system 209% cpu 28.110 total
$ time go run mapNoStar.go
go run mapNoStar.go 20.63s user 1.88s system 99% cpu 22.684 total
$ time go run mapSplit.go
go run mapSplit.go 20.84s user 1.29s system 100% cpu 21.967 total
```

Похоже, версии с картами работают медленнее, чем версия со срезом. К сожалению для карт, версия с картами *всегда* будет медленнее, чем версия со срезом, из-за выполнения хеш-функции и того факта, что данные не являются непрерывными. В картах данные хранятся в корзине, определяемой выходными данными хеш-функции.

Кроме того, первая программа с картой (`mapStar.go`) может вызвать некоторое замедление сборки мусора, поскольку использование адреса `&value` приведет к его помещению в кучу. Другие программы для этих локальных объектов используют

стек. Когда переменные попадают в кучу, это приводит к большей нагрузке на сборку мусора.



Доступ к элементу карты или среза осуществляется во время выполнения $O(1)$; это значит, время доступа не зависит от количества элементов в карте или срезе. Однако то, как работают эти структуры, влияет на общую скорость.

Дополнительные ресурсы

- Go FAQ: как узнать, размещена переменная в куче или стеке? https://golang.org/doc/faq#stack_or_heap.
- Список доступных опций `-gcflags`: <https://golang.org/cmd/compile/>.
- Если вы хотите узнать больше о сборке мусора, то вам следует посетить <http://gchandbook.org/>.

Михалис Цукалос

Golang для профи: Создаем профессиональные утилиты, параллельные серверы и сервисы

3-е издание

Перевели с английского С. Черников, Р. Чикин

Руководитель дивизиона	<i>Ю. Сергиенко</i>
Руководитель проекта	<i>А. Питиримов</i>
Ведущий редактор	<i>Н. Гринчик</i>
Литературный редактор	<i>Н. Хлебина</i>
Художественный редактор	<i>В. Мостапан</i>
Корректоры	<i>Т. Никифорова, Н. Терех</i>
Верстка	<i>Г. Блинов</i>

Изготовлено в России. Изготовитель: ООО «Прогресс книга».
Место нахождения и фактический адрес: 194044, Россия, г. Санкт-Петербург,
Б. Сампсониевский пр., д. 29А, пом. 52. Тел.: +78127037373.

Дата изготовления: 10.2023. Наименование: книжная продукция. Срок годности: не ограничен.

Налоговая льгота — общероссийский классификатор продукции ОК 034-2014, 58.11.12 — Книги печатные
профессиональные, технические и научные.

Импортер в Беларусь: ООО «ПИТЕР М», 220020, РБ, г. Минск, ул. Тимирязева, д. 121/3, к. 214, тел./факс: 208 80 01.

Подписано в печать 30.08.23. Формат 70×100/16. Бумага офсетная. Усл. п. л. 50,310. Тираж 700. Заказ 0000.