ИНСАЙТ ОТ ЭКСПЕРТА

Golang для профи

Работа с сетью, многопоточность, структуры данных и машинное обучение с Go

Второе издание

Михалис Цукалос

Mastering Go Second Edition

Create Golang production applications using network libraries, concurrency, machine learning, and advanced data structures

Mihalis Tsoukalos

BIRMINGHAM - MUMBAI

Михалис Цукалос

Golang для профи

Работа с сетью, многопоточность, структуры данных и машинное обучение с Go

Второе издание

Санкт-Петербург • Москва • Екатеринбург • Воронеж
Нижний Новгород • Ростов-на-Дону
Самара • Минск

2020

ББК 32.988.02-018.1
УДК 004.43
Ц85

Цукалос Михалис

Ц85 Golang для профи: работа с сетью, многопоточность, структуры данных и машинное обучение с Go. — СПб.: Питер, 2020. — 720 с.: ил. — (Серия «Для профессионалов»).

ISBN 978-5-4461-1617-1

Go является языком высокопроизводительных систем будущего. Эта книга показывает, как заставить Go работать на реальных производственных системах.

Для программистов, которые уже знакомы с основами языка Go, эта книга содержит примеры, шаблоны и четкие объяснения, которые помогут вам глубоко понять возможности Go и применить их в своей работе по программированию.

Книга охватывает нюансы Go с подробными руководствами по типам и структурам, пакетам, параллелизму, сетевому программированию, дизайну компиляторов, оптимизации и многому другому. Каждая глава заканчивается упражнениями и ресурсами, чтобы полностью внедрить ваши новые знания.

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. 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 978-1838559335 англ. © Packt Publishing 2019
First published in the English language under the title 'Mastering Go — Second Edition — (9781838559335)'

ISBN 978-5-4461-1617-1 © Перевод на русский язык ООО Издательство «Питер», 2020
© Издание на русском языке, оформление ООО Издательство «Питер», 2020
© Серия «Для профессионалов», 2020

Краткое содержание

Об авторе 21

О научном редакторе 22

Предисловие 23

Глава 1. Go и операционная система 29

Глава 2. Go изнутри 70

Глава 3. Работа с основными типами данных Go 120

Глава 4. Использование составных типов данных 163

Глава 5. Как улучшить код Go с помощью структур данных 216

Глава 6. Неочевидные знания о пакетах и функциях Go 273

Глава 7. Рефлексия и интерфейсы на все случаи жизни 330

Глава 8. Как объяснить UNIX-системе, что она должна делать 364

Глава 9. Конкурентность в Go: горутины, каналы и конвейеры 429

Глава 10. Конкурентность в Go: расширенные возможности 452

Глава 11. Тестирование, оптимизация и профилирование кода 510

Глава 12. Основы сетевого программирования на Go 580

Глава 13. Сетевое программирование: создание серверов и клиентов 638

Глава 14. Машинное обучение на Go 683

Что дальше? 718

Оглавление

Об авторе 21

О научном редакторе 22

Предисловие 23

Для кого предназначена эта книга 23

О чем эта книга 23

Как получить максимальную пользу от этой книги 26

Загрузите файлы с примерами кода 27

Где скачать цветные иллюстрации 27

Условные обозначения 27

От издательства 28

Глава 1. Go и операционная система 29

История Go 29

Куда движется Go? 30

Преимущества Go 30

Идеален ли Go? 31

Утилита godoc 32

Компиляция Go-кода 33

Выполнение Go-кода 34

Два правила Go 35

Правило пакетов Go: не нужен — не подключай 35

Правильный вариант размещения фигурных скобок — всего один 36

Как скачивать Go-пакеты 37

Стандартные потоки UNIX: stdin, stdout и stderr ... 39
Вывод результатов ... 39
Использование стандартного потока вывода ... 41
Получение данных от пользователя ... 42
Что такое := и = ... 42
Чтение стандартного потока ввода ... 43
Работа с аргументами командной строки ... 45
Вывод ошибок ... 47
Запись в журнальные файлы ... 49
Уровни журналирования ... 49
Средства журналирования ... 50
Серверы журналов ... 50
Пример Go-программы, которая записывает информацию в журнальные файлы ... 51
Функция log.Fatal() ... 54
Функция log.Panic() ... 54
Запись в специальный журнальный файл ... 56
Вывод номеров строк в записях журнала ... 58
Обработка ошибок в Go ... 59
Тип данных error ... 59
Обработка ошибок ... 61
Использование Docker ... 64
Упражнения и ссылки ... 68
Резюме ... 69

Глава 2. Go изнутри ... 70
Компилятор Go ... 71
Сборка мусора ... 72
Трехцветный алгоритм ... 74
Подробнее о работе сборщика мусора Go ... 78
Хеш-таблицы, срезы и сборщик мусора Go ... 79
Небезопасный код ... 82

Пакет unsafe 84
Еще один пример использования пакета unsafe 84
Вызов C-кода из Go 86
Вызов C-кода из Go в одном файле 86
Вызов из Go C-кода в отдельных файлах 87
C-код 87
Go-код 88
Сочетание кода на Go и C 89
Вызов Go-функций из C-кода 90
Go-пакет 90
C-код 91
Ключевое слово defer 92
Использование defer для журналирования 95
Функции panic() и recover() 97
Самостоятельное использование функции panic() 98
Две полезные UNIX-утилиты 99
Утилита strace 100
Утилита dtrace 101
Среда Go 102
Команда go env 104
Go-ассемблер 105
Узловые деревья 106
Хотите знать больше о go build? 111
Создание кода WebAssembly 113
Краткое введение в WebAssembly 113
Почему WebAssembly так важен 113
Go и WebAssembly 114
Пример 114
Использование сгенерированного кода WebAssembly 115
Общие рекомендации по программированию на Go 117
Упражнения и ссылки 118
Резюме 118

Глава 3. Работа с основными типами данных Go ... 120
Числовые типы данных ... 120
Целые числа ... 121
Числа с плавающей точкой ... 121
Комплексные числа ... 121
Числовые литералы в Go 2 ... 123
Циклы Go ... 124
Цикл for ... 124
Цикл while ... 125
Ключевое слово range ... 125
Пример применения нескольких циклов Go ... 125
Массивы в Go ... 127
Многомерные массивы ... 128
Недостатки массивов Go ... 130
Срезы в Go ... 131
Выполнение основных операций со срезами ... 131
Автоматическое расширение срезов ... 133
Байтовые срезы ... 135
Функция copy() ... 135
Многомерные срезы ... 137
Еще один пример использования срезов ... 137
Сортировка срезов с помощью sort.Slice() ... 139
Добавление массива к срезу ... 141
Хеш-таблицы Go ... 142
Запись в хеш-таблицу со значением nil ... 144
Когда использовать хеш-таблицы ... 145
Константы Go ... 145
Генератор констант iota ... 147
Указатели в Go ... 149
Зачем нужны указатели ... 152
Время и дата ... 152
Работа с временем ... 153

Синтаксический анализ времени ... 154
Работа с датами ... 156
Синтаксический анализ дат ... 156
Изменение формата даты и времени ... 157
Измерение времени выполнения программы ... 159
Измерение скорости работы сборщика мусора Go ... 160
Веб-ссылки и упражнения ... 161
Резюме ... 161

Глава 4. Использование составных типов данных ... 163
Составные типы данных ... 164
Структуры ... 164
Указатели на структуры ... 166
Ключевое слово new ... 168
Кортежи ... 169
Регулярные выражения и сопоставление с образцом ... 170
Немного теории ... 171
Простой пример ... 171
Более сложный пример ... 174
Проверка IPv4-адресов ... 176
Строки ... 180
Что такое руны ... 183
Пакет unicode ... 184
Пакет strings ... 185
Оператор switch ... 189
Вычисление числа π с высокой точностью ... 192
Разработка на Go хранилища типа «ключ — значение» ... 195
Go и формат JSON ... 200
Чтение данных из формата JSON ... 200
Сохранение данных в формате JSON ... 202
Использование функций Marshal() и Unmarshal() ... 203
Синтаксический анализ данных в формате JSON ... 205

Go и XML ... 207
Чтение XML-файла ... 210
Настройка вывода данных в формате XML ... 211
Go и формат YAML ... 213
Дополнительные ресурсы ... 213
Упражнения ... 214
Резюме ... 214

Глава 5. Как улучшить код Go с помощью структур данных ... 216
О графах и узлах ... 217
Сложность алгоритма ... 217
Двоичные деревья в Go ... 218
Реализация двоичного дерева в Go ... 219
Преимущества двоичных деревьев ... 221
Пользовательские хеш-таблицы в Go ... 222
Реализация пользовательской хеш-таблицы в Go ... 222
Реализация функции поиска ... 225
Преимущества пользовательских хеш-таблиц ... 226
Связные списки в Go ... 226
Реализация связного списка в Go ... 227
Преимущества связных списков ... 231
Двусвязные списки в Go ... 231
Реализация двусвязного списка в Go ... 232
Преимущества двусвязных списков ... 235
Очереди в Go ... 236
Реализация очереди в Go ... 236
Стеки в Go ... 239
Реализация стека в Go ... 239
Пакет container ... 242
Использование пакета container/heap ... 242
Использование пакета container/list ... 245
Использование пакета container/ring ... 247

Генерация случайных чисел ... 248
Генерация случайных строк ... 251
Генерация безопасной последовательности случайных чисел ... 253
Выполнение матричных вычислений ... 255
Сложение и вычитание матриц ... 255
Умножение матриц ... 258
Деление матриц ... 261
Разгадывание головоломок судоку ... 267
Дополнительные ресурсы ... 270
Упражнения ... 271
Резюме ... 272

Глава 6. Неочевидные знания о пакетах и функциях Go ... 273
Что такое Go-пакеты ... 274
Что такое функции Go ... 274
Анонимные функции ... 275
Функции, которые возвращают несколько значений ... 275
Функции, возвращающие именованные значения ... 277
Функции, принимающие указатели ... 279
Функции, которые возвращают указатели ... 280
Функции, которые возвращают другие функции ... 281
Функции, которые принимают другие функции в качестве параметров ... 282
Функции с переменным числом параметров ... 283
Разработка Go-пакетов ... 285
Компиляция Go-пакета ... 287
Закрытые переменные и функции ... 287
Функция init() ... 287
Go-модули ... 290
Создание и использование Go-модулей ... 290
Использование двух версий одного и того же Go-модуля ... 298
Где хранятся Go-модули ... 299
Команда go mod vendor ... 300

Как писать хорошие Go-пакеты 300
Пакет syscall 302
Как на самом деле работает fmt.Println() 304
Пакеты go/scanner, go/parser и go/token 306
Пакет go/ast 307
Пакет go/scanner 307
Пакет go/parser 309
Практический пример 311
Поиск имен переменных заданной длины 313
Шаблоны для текста и HTML 318
Вывод простого текста 318
Вывод текста в формате HTML 320
Дополнительные ресурсы 327
Упражнения 328
Резюме 328

Глава 7. Рефлексия и интерфейсы на все случаи жизни 330
Методы типов 330
Интерфейсы в Go 332
Операции утверждения типа 333
Как писать свои интерфейсы 335
Использование интерфейса Go 336
Использование переключателей для интерфейсов и типов данных 338
Рефлексия 340
Простой пример рефлексии 340
Более сложный пример рефлексии 343
Три недостатка рефлексии 345
Библиотека reflectwalk 346
Объектно-ориентированное программирование на Go 348
Основы git и GitHub 351
Использование git 351

Отладка с помощью Delve ... 357
Пример отладки ... 358
Дополнительные ресурсы ... 362
Упражнения ... 362
Резюме ... 362

Глава 8. Как объяснить UNIX-системе, что она должна делать ... 364
О процессах в UNIX ... 365
Пакет flag ... 365
Пакет viper ... 370
Простой пример использования viper ... 371
От flag к viper ... 372
Чтение конфигурационных файлов в формате JSON ... 373
Чтение конфигурационных файлов в формате YAML ... 375
Пакет cobra ... 377
Простой пример cobra ... 378
Создание псевдонимов команд ... 382
Интерфейсы io.Reader и io.Writer ... 385
Буферизованный и небуферизованный ввод и вывод в файл ... 385
Пакет bufio ... 386
Чтение текстовых файлов ... 386
Построчное чтение текстового файла ... 386
Чтение текстового файла по словам ... 388
Посимвольное чтение текстового файла ... 390
Чтение из /dev/random ... 392
Чтение заданного количества данных ... 393
Преимущества двоичных форматов ... 395
Чтение CSV-файлов ... 396
Запись в файл ... 399
Загрузка и сохранение данных на диске ... 401
И снова пакет strings ... 404

Пакет bytes ... 406
Полномочия доступа к файлам ... 407
Обработка сигналов в UNIX ... 408
Обработка двух сигналов ... 409
Обработка всех сигналов ... 411
Программирование UNIX-каналов на Go ... 413
Реализация утилиты cat(1) на Go ... 414
Структура syscall.PtraceRegs ... 416
Отслеживание системных вызовов ... 418
Идентификаторы пользователя и группы ... 422
Docker API и Go ... 423
Дополнительные ресурсы ... 426
Упражнения ... 427
Резюме ... 428

Глава 9. Конкурентность в Go: горутины, каналы и конвейеры ... 429
О процессах, потоках и горутинах ... 430
Планировщик Go ... 430
Конкурентность и параллелизм ... 431
Горутины ... 431
Создание горутины ... 432
Создание нескольких горутин ... 433
Как дождаться завершения горутин, прежде чем закончить программу ... 435
Что происходит, если количество вызовов Add() и Done() не совпадает ... 437
Каналы ... 439
Запись в канал ... 439
Чтение из канала ... 440
Прием данных из закрытого канала ... 442
Каналы как аргументы функции ... 443
Конвейеры ... 444

Состояние гонки ... 447
Сравнение моделей конкурентности в Go и Rust ... 449
Сравнение моделей конкурентности в Go и Erlang ... 449
Дополнительные ресурсы ... 450
Упражнения ... 450
Резюме ... 451

Глава 10. Конкурентность в Go: расширенные возможности ... 452
И снова о планировщике Go ... 453
Переменная среды GOMAXPROCS ... 455
Ключевое слово select ... 456
Принудительное завершение горутины ... 459
Принудительное завершение горутины, способ 1 ... 459
Принудительное завершение горутины, способ 2 ... 461
И снова о Go-каналах ... 463
Сигнальные каналы ... 464
Буферизованные каналы ... 464
Нулевые каналы ... 466
Каналы каналов ... 467
Выбор последовательности исполнения горутин ... 470
Как не надо использовать горутины ... 472
Общая память и общие переменные ... 473
Тип sync.Mutex ... 474
Тип sync.RWMutex ... 478
Пакет atomic ... 481
Совместное использование памяти с помощью горутин ... 483
И снова об операторе go ... 485
Распознавание состояния гонки ... 488
Пакет context ... 493
Расширенный пример использования пакета context ... 497
Еще один пример использования пакета context ... 502
Пулы обработчиков ... 503

Дополнительные ресурсы 508
Упражнения 508
Резюме 509

Глава 11. Тестирование, оптимизация и профилирование кода 510
Оптимизация 511
Оптимизация кода Go 512
Профилирование кода Go 513
Стандартный Go-пакет net/http/pprof 513
Простой пример профилирования 513
Удобный внешний пакет для профилирования 521
Веб-интерфейс Go-профилировщика 523
Утилита go tool trace 527
Тестирование кода Go 532
Написание тестов для существующего кода Go 532
Тестовое покрытие кода 536
Тестирование HTTP-сервера с базой данных 539
Пакет testing/quick 545
Бенчмаркинг кода Go 551
Простой пример бенчмаркинга 552
Неправильно определенные функции бенчмаркинга 557
Бенчмаркинг буферизованной записи 558
Обнаружение недоступного кода Go 562
Кросс-компиляция 564
Создание примеров функций 565
От кода Go до машинного кода 567
Использование ассемблера в Go 568
Генерация документации 570
Использование образов Docker 575
Дополнительные ресурсы 577
Упражнения 578
Резюме 579

Глава 12. Основы сетевого программирования на Go 580
Что такое net/http, net и http.RoundTripper 581
Тип http.Response 581
Тип http.Request 582
Тип http.Transport 582
Что такое TCP/IP 583
Что такое IPv4 и IPv6 584
Утилита командной строки nc(1) 584
Чтение конфигурации сетевых интерфейсов 585
Выполнение DNS-поиска 589
Получение NS-записей домена 591
Получение MX-записей домена 593
Создание веб-сервера на Go 594
Использование пакета atomic 597
Профилирование HTTP-сервера 599
Создание веб-сайта на Go 604
HTTP-трассировка 613
Тестирование HTTP-обработчиков 616
Создание веб-клиента на Go 618
Как усовершенствовать наш веб-клиент Go 620
Задержки HTTP-соединений 623
Подробнее о SetDeadline 625
Установка периода ожидания на стороне сервера 625
Еще один способ определить период ожидания 627
Инструменты Wireshark и tshark 629
Go и gRPC 629
Определение файла описания интерфейса 629
gRPC-клиент 632
gRPC-сервер 633
Дополнительные ресурсы 635
Упражнения 636
Резюме 637

Глава 13. Сетевое программирование: создание серверов и клиентов 638
Работа с HTTPS-трафиком 639
Создание сертификатов 639
HTTPS-клиент 640
Простой HTTPS-сервер 642
Разработка TLS-сервера и TLS-клиента 643
Стандартный Go-пакет net 646
TCP-клиент 646
Другая версия TCP-клиента 648
TCP-сервер 650
Другая версия TCP-сервера 652
UDP-клиент 654
Разработка UDP-сервера 656
Конкурентный TCP-сервер 658
Удобный конкурентный TCP-сервер 662
Создание образа Docker для TCP/IP-сервера на Go 668
Дистанционный вызов процедур 670
RPC-клиент 671
RPC-сервер 672
Низкоуровневое сетевое программирование 674
Получение необработанных сетевых данных ICMP 676
Дополнительные ресурсы 680
Упражнения 681
Резюме 682

Глава 14. Машинное обучение на Go 683
Вычисление простых статистических показателей 684
Регрессия 688
Линейная регрессия 688
Реализация линейной регрессии 688
Вывод данных 690
Классификация 694

Кластеризация ... 698
Выявление аномалий ... 700
Нейронные сети ... 702
Анализ выбросов ... 704
Работа с TensorFlow ... 707
Поговорим о Kafka ... 712
Дополнительные ресурсы ... 716
Упражнения ... 717
Резюме ... 717

Что дальше? ... 718

Об авторе

Михалис Цукалос (Mihalis Tsoukalos) — администратор UNIX, программист, администратор баз данных и математик. Любит писать технические книги и статьи, узнавать что-то новое. Помимо этой книги, Михалис написал *Go Systems Programming*, а также более 250 технических статей для многих журналов, включая Sys Admin, MacTech, Linux User and Developer, Usenix ;login:, Linux Format и Linux Journal. Сфера научных интересов Михалиса — базы данных, визуализация, статистика и машинное обучение.

Вы можете связаться с автором через его сайт https://www.mtsoukalos.eu/ или по адресу @mactsouk.

Михалис также увлекается фотографией.

Я благодарю сотрудников Packt Publishing за помощь в написании этой книги, в том числе моего научного редактора Мэта Райера (Mat Ryer), а также Кишора Рита (Kishor Rit) за ответы на все мои вопросы и поддержку в период написания книги.

Эту книгу я посвящаю памяти моих любимых родителей Иоанниса и Аргетты.

О научном редакторе

Мэт Райер (Mat Ryer) пишет компьютерные программы с шести лет: сначала на BASIC для ZX Spectrum, а затем, вместе со своим отцом, — на AmigaBASIC и AMOS для Commodore Amiga. Много времени он потратил на копирование вручную кода из журнала Amiga Format, изменение значений переменных или ссылок операторов GOTO, чтобы посмотреть, что из этого выйдет. Тот же дух исследования и одержимость программированием привели 18-летнего Мэтта к работе в местной организации в Мансфилде (Великобритания), где он начал создавать веб-сайты и другие онлайн-сервисы.

После нескольких лет работы с различными технологиями в разных областях не только в Лондоне, но и по всему миру Мэт обратил внимание на новый язык системного программирования под названием Go, впервые использованный в Google. Поскольку Go решал очень актуальные и остросовременные технические проблемы, Мэт начал использовать этот язык для решения задач, когда Go находился еще на стадии бета-тестирования, и с тех пор продолжает программировать на нем. Мэт работал над разными проектами с открытым исходным кодом, создал несколько пакетов Go, в том числе Testify, Moq, Silk и Is, а также инструментарий для разработчиков в MacOS — BitBar.

С 2018 года Мэт — соучредитель компании Machine Box, но он по-прежнему принимает участие в конференциях, пишет о Go в своем блоге и является активным участником сообщества Go.

Предисловие

Книга «Golang для профи: работа с сетью, многопоточность, структуры данных и машинное обучение с Go», второе издание, которую вы сейчас держите в руках, поможет вам стать самым лучшим разработчиком на Go!

В книге много свежих интересных тем, включая совершенно новую главу об использовании Go для машинного обучения. Вы также найдете здесь описания и примеры кода для пакетов Viper и Cobra Go, gRPC, вы узнаете, как работать с образами Docker, файлами YAML, пакетами `go/scanner` и `go/token`, научитесь генерировать из Go код WebAssembly. В общей сложности второе издание книги «Golang для профи» расширилось более чем на 130 страниц.

Для кого предназначена эта книга

Эта книга рассчитана на начинающих и программистов среднего уровня, работающих с языком Go, которые стремятся перейти на новый уровень знаний Go, а также на опытных разработчиков, пишущих на других языках программирования, которые хотят изучать Go, не возвращаясь к циклу `for`.

Часть информации из второго издания можно найти в другой книге автора — *Go Systems Programming*. Основное различие между этими двумя книгами заключается в том, что *Go Systems Programming* посвящена разработке системных инструментов с использованием возможностей языка Go, а «Golang для профи» — разъяснению возможностей и внутренних особенностей самого Go, что позволит вам стать лучшим разработчиком на Go. Обе книги могут служить справочниками после того, как вы их прочитаете один-два раза.

О чем эта книга

Глава 1 «Go и операционная система» начинается с истории возникновения языка Go и его преимуществ, затем приводится описание утилиты `godoc` и объяснение, как компилировать и выполнять Go-программы. Далее речь пойдет о выводе данных

и получении данных от пользователя, об аргументах командной строки программы и использовании журнальных файлов. Последний раздел первой главы посвящен обработке ошибок, которая в Go играет ключевую роль.

Глава 2 «Go изнутри» посвящена сборщику мусора Go и принципам его работы. Затем поговорим о небезопасном коде и о пакете `unsafe`, а также о том, как вызывать из Go-программы код на C, а из C-программы — код на Go.

Вы узнаете, как использовать ключевое слово `defer`, а также познакомитесь с утилитами `strace(1)` и `dtrace(1)`. В оставшихся разделах этой главы вы изучите, как получить информацию о вашей среде Go, как использовать ассемблер и как генерировать из Go код WebAssembly.

Глава 3 «Работа с основными типами данных Go» посвящена типам данных, предоставляемым в Go: массивам, срезам и хеш-таблицам, а также указателям, константам, циклам, функциям для работы с датами и временем. Вы точно не захотите пропустить эту главу!

Глава 4 «Использование составных типов данных» начинается с изучения структур Go и ключевого слова `struct`, после чего речь пойдет о кортежах, строках, рунах, байтовых срезах и строковых литералах. Из оставшейся части главы вы узнаете о регулярных выражениях и сопоставлении с образцом, об операторе `switch`, пакетах `strings` и `math/big`, о разработке на Go хранилища типа «ключ — значение» и о работе с файлами форматов XML и JSON.

Глава 5 «Как улучшить код Go с помощью структур данных» посвящена разработке пользовательских структур данных в тех случаях, когда стандартные структуры Go не соответствуют конкретной задаче. Здесь же рассмотрено построение и применение бинарных деревьев, связных списков, пользовательских хеш-таблиц, стеков и очередей, а также их преимущества. В этой главе продемонстрировано использование структур из стандартного пакета Go `container`, а также показано, как можно использовать Go для проверки головоломок судоку и генерации случайных чисел.

Глава 6 «Неочевидные знания о пакетах и функциях Go» посвящена пакетам и функциям, в том числе использованию функции `init()`, стандартного Go-пакета `syscall`, а также пакетов `text/template` и `html/template`. Кроме того, вы узнаете, как применять расширенные пакеты `go/scanner`, `go/parser` и `go/token`. Эта глава определенно улучшит ваши навыки разработки на Go!

В *главе 7 «Рефлексия и интерфейсы на все случаи жизни»* обсуждаются три передовые концепции Go: рефлексия, интерфейсы и методы типов. Кроме того, в этой главе описываются объектно-ориентированные возможности Go и способы отладки Go-программ с помощью Delve.

Глава 8 «Как объяснить UNIX-системе, что она должна делать» посвящена системному программированию на Go. Здесь рассматриваются такие темы, как пакет `flag` для работы с аргументами командной строки, обработка сигналов UNIX, файловый ввод и вывод, пакеты `bytes`, `io.Reader` и `io.Writer`, а также

обсуждается использование пакетов Viper и Cobra Go. Напомню: если вы действительно увлекаетесь системным программированием на Go, то я настоятельно рекомендую вам после прочтения этой книги приобрести и прочесть *Go Systems Programming*!

В *главе 9 «Конкурентность в Go: горутины, каналы и конвейеры»* обсуждаются горутины, каналы и конвейеры — то, что позволяет реализовать конкурентность на Go.

Вы также узнаете, чем различаются между собой процессы, потоки и горутины, познакомитесь с пакетом `sync` и особенностями работы планировщика Go.

Глава 10 «Конкурентность в Go: расширенные возможности» является продолжением предыдущей главы. Прочитав ее, вы станете повелителем горутин и каналов! Вы глубже изучите планировщик Go, научитесь использовать мощное ключевое слово `select`, узнаете о различных типах каналов Go, а также о разделяемой памяти, мьютексах, типах `sync.Mutex` и `sync.RWMutex`. В последней части главы говорится о пакете `context`, пулах обработчиков и о том, как распознать состояние гонки (race conditions).

В *главе 11 «Тестирование, оптимизация и профилирование кода»* обсуждаются тестирование, оптимизация и профилирование кода, а также кросс-компиляция под разные платформы, создание документации, тестирование производительности Go-кода, создание тестовых функций и поиск неиспользуемого Go-кода.

Глава 12 «Основы сетевого программирования на Go» посвящена пакету `net/http` и тому, как разрабатывать веб-клиенты и веб-серверы на Go. Далее рассмотрены структуры `http.Response`, `http.Request` и `http.Transport`, а также тип `http.NewServeMux`. Вы даже узнаете, как разработать на Go целый веб-сайт! Кроме того, в этой главе вы научитесь читать конфигурацию сетевых интерфейсов и выполнять на Go DNS-поиск, а также использовать с Go gRPC.

В *главе 13 «Сетевое программирование: создание серверов и клиентов»* рассказывается о работе с HTTPS-трафиком и создании на Go серверов и клиентов UDP и TCP с использованием функций из пакета `net`. Здесь же рассмотрены такие темы, как построение клиентов и серверов RPC, разработка на Go многопоточного TCP-сервера и чтение «сырых» сетевых пакетов.

В *главе 14 «Машинное обучение на Go»* рассказывается о реализации на Go алгоритмов машинного обучения, включая классификацию, кластеризацию, обнаружение аномалий, выбросы, нейронные сети и TensorFlow, а также работу Go с Apache Kafka.

Эту книгу можно разделить на три логические части. В первой части подробно рассматриваются некоторые важные концепции Go, включая пользовательский ввод и вывод, загрузку внешних Go-пакетов, компиляцию Go-кода, вызов из Go кода на C и создание кода WebAssembly, а также использование основных и составных типов Go.

Вторая часть начинается с главы 5 и включает в себя главы 6 и 7. Эти три главы посвящены организации Go-кода в виде пакетов и модулей, структуре Go-проектов и некоторым дополнительным функциям Go.

Последняя часть включает в себя оставшиеся семь глав и посвящена более практическим темам Go. В главах 8–11 рассказывается о системном программировании на Go, реализации конкурентности на Go, тестировании, оптимизации и профилировании кода. В последних трех главах этой книги речь идет о сетевом программировании и машинном обучении на Go.

Книга включает в себя материалы о Go и WebAssembly, использовании Docker с Go, создании профессиональных утилит работы с командной строкой с помощью пакетов Viper и Cobra, обработке JSON и YAML, выполнении операций с матрицами, работе с головоломками судоку, пакетами `go/scanner` и `go/token`, а также с `git(1)` и GitHub, пакетом `atomic`, об использовании в Go gRPC и HTTPS.

В книге представлены относительно небольшие, но полные Go-программы, которые иллюстрируют описанные концепции. У этого есть два основных преимущества: во-первых, вам не придется просматривать бесконечный листинг, чтобы изучить всего одну методику, а во-вторых, вы сможете использовать этот код как отправную точку при создании собственных приложений и утилит.

Поскольку контейнеры и Docker очень важны, я включил в эту книгу примеры исполняемых Go-файлов, которые используются в образах Docker, ведь образы Docker являются отличным способом развертывания серверного программного обеспечения.

Как получить максимальную пользу от этой книги

Для работы с книгой вам потребуется UNIX-компьютер и установленная на нем относительно новая версия Go. Это может быть любой компьютер с операционной системой Mac OS X, macOS или Linux. Большая часть представленного здесь кода также будет работать и на компьютерах под управлением Microsoft Windows.

Чтобы извлечь максимальную пользу из этой книги, вы должны как можно быстрее применить знания, полученные при прочтении каждой главы, в своих программах и посмотреть, что работает, а что — нет! Как я уже говорил, попробуйте выполнять все упражнения, приведенные в конце каждой главы, или создавайте собственные задачи программирования.

Загрузите файлы с примерами кода

Примеры кода для этой книги размещены на GitHub по адресу https://github.com/PacktPublishing/Mastering-Go-Second-Edition. В случае если код обновится, он будет также размещен в уже существующем репозитории GitHub.

Чтобы скачать файлы с кодом, нужно выполнить следующие действия.

1. Перейдите по указанной ссылке на сайт github.com.
2. Нажмите кнопку Clone or Download.
3. Щелкните кнопкой мыши на ссылке Download ZIP.
4. Скачайте архив с файлами примеров.

Когда загрузится файл с архивом, распакуйте его или извлеките из него папку с упакованными файлами, используя последнюю версию одного из следующих архиваторов:

- WinRAR или 7-Zip для Windows;
- Zipeg, iZip или UnRarX для Mac;
- 7-Zip или PeaZip для Linux.

По адресу https://github.com/PacktPublishing/ вам доступны и другие примеры кода из нашего богатого каталога книг и видео. Все это к вашим услугам!

Где скачать цветные иллюстрации

Мы предоставляем вам PDF-файл с цветными скриншотами и диаграммами, использованными в этой книге. Вы можете его скачать по этому адресу: https://static.packt-cdn.com/downloads/9781838559335.pdf.

Условные обозначения

В этой книге используются следующие текстовые обозначения.

`Код в тексте`: размещенные в тексте книги кодовые слова, имена таблиц из базы данных, папок и файлов; расширения файлов, пути, фиктивные URL-адреса, вводимые пользователем данные и учетные записи Twitter. Например: «Первый способ похож на использование команды `man(1)`, но только для функций и пакетов Go».

Блоки кода представлены следующим образом:

```
package main
import (
    "fmt"
)
```

```
func main() {
    fmt.Println("This is a sample Go program!")
}
```

Когда нужно обратить ваше внимание на определенную часть блока кода, соответствующие строки или элементы выделяются жирным шрифтом:

```
import (
    "fmt"
)
func main() {
    fmt.Println("This is a sample Go program!")
}
```

Все, что вводится или выводится в командной строке, записывается следующим образом:

```
$ date
Sat Oct 21 20:09:20 EEST 2017
$ go version
go version go1.12.7 darwin/amd64
```

Курсивом обозначаются новые термины, **жирным шрифтом** — важные слова. То, что вы видите на экране (например, слова в меню или диалоговых окнах), отображается в тексте так: «Выберите на панели Administration раздел System info».

Так выглядят 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 Go и операционная система

Эта глава — введение в различные аспекты языка Go, которые будут очень полезными для начинающих. Более опытные разработчики на Go также могут использовать эту главу в качестве курса повышения квалификации. Как часто бывает с большинством практических предметов, лучший способ что-то усвоить — поэкспериментировать с этим. В данном случае экспериментировать означает самостоятельно писать Go-код, совершать собственные ошибки и учиться на них. Только не позволяйте этим ошибкам и сообщениям о них отбивать у вас охоту учиться дальше!

В этой главе рассмотрены следующие темы:

- история и будущее языка программирования Go;
- преимущества Go;
- компиляция Go-кода;
- выполнение Go-кода;
- загрузка и использование внешних Go-пакетов;
- стандартный ввод, вывод и сообщения об ошибках в UNIX;
- вывод данных на экран;
- получение данных от пользователя;
- вывод данных о стандартных ошибках;
- работа с лог-файлами;
- использование Docker для компиляции и запуска исходных файлов Go;
- обработка ошибок в Go.

История Go

Go — это современный универсальный язык программирования с открытым исходным кодом, выпуск которого официально состоялся в конце 2009 года. Go планировался как внутренний проект Google — это означает, что сначала он был запущен в качестве эксперимента и с тех пор вдохновлялся многими другими

языками программирования, в том числе C, Pascal, Alef и Oberon. Создателями Go являются профессиональные программисты Роберт Гризмер (Robert Griesemer), Кен Томсон (Ken Thomson) и Роб Пайк (Rob Pike). Они разработали Go как язык для профессионалов, позволяющий создавать надежное, устойчивое и эффективное программное обеспечение. Помимо синтаксиса и стандартных функций, в состав Go входит довольно богатая стандартная библиотека.

На момент публикации этой книги последней стабильной версией Go была версия 1.14. Однако даже если номер вашей версии выше, книга все равно будет актуальной.

Если вы устанавливаете Go впервые, начните с посещения веб-сайта https://golang.org/dl/. Однако с большой вероятностью в вашем дистрибутиве UNIX уже есть готовый к установке пакет для языка программирования Go, поэтому вы можете получить Go с помощью обычного менеджера пакетов.

Куда движется Go?

Сообщество Go уже обсуждает следующую полноценную версию Go, которая будет называться Go 2, но пока еще не появилось ничего определенного.

Цель нынешней команды по разработке Go 1 — сделать так, чтобы Go 2 больше развивался по инициативе сообщества. В целом это неплохая идея, однако всегда есть риск, когда слишком много людей участвуют в принятии важных решений относительно языка программирования, который изначально создавался и разрабатывался как внутренний проект небольшой группы гениальных профессионалов.

Некоторые крупные изменения, рассматриваемые для Go 2, — это дженерики, управление версиями пакетов и улучшенная обработка ошибок. Все новые функции в настоящее время находятся на стадии обсуждения, и вам не стоит о них беспокоиться — однако нужно иметь представление о направлении, в котором движется Go.

Преимущества Go

У языка Go много преимуществ. Некоторые из них уникальны для Go, а другие свойственны и иным языкам программирования.

Среди наиболее значимых преимуществ и возможностей Go можно отметить следующие.

- Go — современный язык программирования, созданный опытными разработчиками. Его код понятен и легко читается.
- Цель Go — счастливые разработчики. Именно счастливые разработчики пишут самый лучший код!
- Компилятор Go выводит информативные предупреждения и сообщения об ошибках, которые помогут вам решить конкретную проблему. Проще говоря,

компилятор Go будет вам помогать, а не портить жизнь, выводя бессмысленные сообщения!

- Код Go является переносимым, особенно между UNIX-машинами.
- Go поддерживает процедурное, параллельное и распределенное программирование.
- Go поддерживает *сборку мусора*, поэтому вам не придется заниматься выделением и освобождением памяти.
- У Go нет *препроцессора*; вместо этого выполняется высокоскоростная компиляция. Вследствие этого Go можно использовать как язык сценариев.
- Go позволяет создавать веб-приложения и предоставляет простой веб-сервер для их тестирования.
- В стандартную библиотеку Go входит множество пакетов, которые упрощают жизнь разработчика. Функции, входящие в стандартную библиотеку Go, предварительно тестируются и отлаживаются людьми, разрабатывающими Go, а значит, в основном работают без ошибок.
- По умолчанию в Go используется *статическая компоновка* — это значит, что создаваемые двоичные файлы легко переносятся на другие компьютеры с той же ОС. Как следствие, после успешной компиляции Go-программы и создания исполняемого файла не приходится беспокоиться о библиотеках, зависимостях и разных версиях этих библиотек.
- Для разработки, отладки и тестирования Go-приложений вам не понадобится *графический интерфейс пользователя* (Graphical User Interface, GUI), так как Go можно использовать из командной строки — именно так, как, мне кажется, предпочитают многие пользователи UNIX.
- Go поддерживает Unicode, а следовательно, вам не понадобится дополнительная обработка для вывода символов на разных языках.
- В Go сохраняется принцип независимости, потому что несколько независимых функций работают лучше, чем много взаимно перекрывающихся.

Идеален ли Go?

Идеального языка программирования не существует, и Go не исключение. Есть языки программирования, которые эффективнее в некоторых других областях программирования, или же мы их просто больше любим. Лично я не люблю Java, и хотя раньше мне нравился C++, сейчас он мне не нравится. C++ стал слишком сложным как язык программирования, а код на Java, на мой взгляд, не очень красиво смотрится.

Вот некоторые из недостатков Go:

- у Go нет встроенной поддержки *объектно-ориентированного программирования*. Это может стать проблемой для тех программистов, которые привыкли писать

объектно-ориентированный код. Однако вы можете имитировать наследование в Go, используя композицию;

- некоторые считают, что Go никогда не заменит C;
- C все еще остается более быстрым, чем любой другой язык системного программирования, главным образом потому, что UNIX написана на C.

Несмотря на это, Go — вполне достойный язык программирования. Он вас не разочарует, если вы найдете время для его изучения и использования.

Что такое препроцессор

Как я уже говорил, в Go нет препроцессора, и это хорошо. Препроцессор — это программа, которая обрабатывает входные данные и генерирует выходные данные, которые будут использоваться в качестве входных для другой программы. В контексте языков программирования входные данные препроцессора — это исходный код программы, который будет обработан препроцессором и затем передан на вход компилятора языка программирования.

Самый большой недостаток препроцессора — он ничего не знает ни о базовом языке, ни о его синтаксисе! Это значит, что, когда используется препроцессор, нельзя гарантировать, что окончательная версия кода будет делать именно то, что вы хотите: препроцессор может изменить и логику, и семантику исходного кода.

Препроцессор используется в таких языках программирования, как C, C++, Ada, PL/SQL. Печально известный препроцессор C обрабатывает строки, которые начинаются с символа # и называются *директивами* или *прагмами*. Таким образом, директивы и прагмы не являются частью языка программирования C!

Утилита godoc

В дистрибутив Go входит множество инструментов, способных значительно упростить жизнь программиста. Одним из таких инструментов является утилита godoc[1], которая позволяет просматривать документацию загруженных функций и пакетов Go без подключения к Интернету.

Утилита godoc может выполняться как обычное приложение командной строки, которое выводит данные на терминал, или же как приложение командной строки, которое запускает веб-сервер. В последнем случае для просмотра документации Go вам понадобится браузер.

Если ввести в командной строке просто godoc, без каких-либо параметров, то получим список параметров командной строки, поддерживаемых godoc.

[1] Если godoc на вашем компьютере не установлена, просто выполните такую команду: $ go get golang.org/x/tools/cmd/godoc. — *Здесь и далее примеч. науч. ред.*

Первый способ аналогичен использованию команды `man(1)`, только для функций и пакетов Go. Например, чтобы получить информацию о функции `Printf()` из пакета `fmt`, необходимо ввести команду:

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

Аналогичным образом можно получить информацию обо всем пакете `fmt`, введя следующую команду:

```
$ go doc fmt
```

Второй способ требует выполнения `godoc` с параметром `-http`:

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

Число в предыдущей команде, в данном случае равное `8001`, — это номер порта, который будет прослушивать HTTP-сервер. Вы можете указать любой доступный номер порта, если у вас есть необходимые привилегии. Однако обратите внимание, что номера портов от 0 до 1023 зарезервированы и могут использоваться только пользователем `root`, поэтому лучше избегать использования одного из этих портов и выбирать какой-нибудь другой, если только он еще не используется другим процессом.

Вместо знака равенства в предыдущей команде можно поставить символ пробела. Следующая команда полностью эквивалентна предыдущей:

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

Если после этого ввести в браузере URL-адрес http://localhost:8001/pkg/, то вы получите список доступных пакетов Go и сможете просмотреть их документацию.

Компиляция Go-кода

Из этого раздела вы узнаете, как скомпилировать код на Go. Хорошая новость: вы можете скомпилировать код Go из командной строки без графического приложения. Более того, для Go не имеет значения имя исходного файла с текстом программы, если именем пакета является `main` и в нем есть только одна функция `main()`. Дело в том, что именно с функции `main()` начинается выполнение программы. Из-за этого в файлах одного проекта не может быть нескольких функций `main()`.

Нашей первой скомпилированной Go-программой будет программа с именем `aSourceFile.go`, которая содержит следующий код Go:

```
package main
import (
    "fmt"
)

func main() {
    fmt.Println("This is a sample Go program!")
}
```

Обратите внимание, что сообщество Go предпочитает называть исходный файл Go `source_file.go`, а не `aSourceFile.go`. В любом случае, что бы вы ни выбрали, будьте последовательны.

Чтобы скомпилировать `aSourceFile.go` и создать *статически скомпонованный* исполняемый файл, нужно выполнить следующую команду:

```
$ go build aSourceFile.go
```

В результате будет создан новый исполняемый файл с именем `aSourceFile`, который теперь нужно выполнить:

```
$ file aSourceFile
aSourceFile: Mach-O 64-bit executable x86_64
$ ls -l aSourceFile
-rwxr-xr-x 1 mtsouk staff 2007576 Jan 10 21:10 aSourceFile
$ ./aSourceFile
This is a sample Go program!
```

Основная причина, по которой файл `aSourceFile` такой большой, заключается в том, что он статически скомпонован, другими словами, для его работы не требуется никаких внешних библиотек.

Выполнение Go-кода

Есть другой способ выполнить Go-код, при котором не создается постоянных исполняемых файлов — генерируется лишь несколько временных файлов, которые впоследствии автоматически удаляются.

Этот способ позволяет использовать Go как язык сценариев, подобно Python, Ruby или Perl.

Итак, чтобы запустить `aSourceFile.go`, не создавая исполняемый файл, необходимо выполнить следующую команду:

```
$ go run aSourceFile.go
This is a sample Go program!
```

Как видим, результат выполнения этой команды точно такой же, как и раньше.

Обратите внимание, что при запуске go run компилятору Go по-прежнему нужно создать исполняемый файл. Только вы его не видите, потому что он автоматически выполняется и так же автоматически удаляется после завершения программы. Из-за этого может показаться, что нет необходимости в исполняемом файле.

В этой книге для выполнения примеров кода в основном будет использоваться `go run`; в первую очередь потому, что так проще, чем сначала запускать `go build`, а затем — исполняемый файл. Кроме того, `go run` после завершения программы не оставляет файлов на жестком диске.

Два правила Go

В Go приняты строгие правила кодирования. Они помогут вам избежать ошибок и багов в коде, а также позволят облегчить чтение кода для сообщества Go. В этом разделе представлены два правила Go, о которых вам необходимо знать.

Как я уже говорил, пожалуйста, помните, что компилятор Go будет помогать вам, а не усложнять жизнь. Основная цель компилятора Go — компилировать код и повышать его качество.

Правило пакетов Go: не нужен — не подключай

В Go приняты строгие правила использования пакетов. Вы не можете просто подключить пакет на всякий случай и в итоге не использовать его.

Рассмотрим следующую простую программу, которая сохраняется как `packageNotUsed.go`:

```
package main

import (
    "fmt"
    "os"
)

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

В этой книге вам встретится множество сообщений об ошибках, ошибочных ситуаций и предупреждений. Считаю, что изучение кода, который не компилируется, не менее (а иногда и более!) полезно, чем просто чтение Go-кода, который компилируется без каких-либо ошибок. Компилятор Go обычно выводит информативные сообщения об ошибках и предупреждения. Эти сообщения, скорее всего, помогут вам устранить ошибочную ситуацию, поэтому не стоит недооценивать их.

Если попытаться выполнить `packageNotUsed.go`, то программа не будет выполнена, а мы получим от Go следующее сообщение об ошибке:

```
$ go run packageNotUsed.go
# command-line-arguments
./packageNotUsed.go:5:2: imported and not used: "os"
```

Если удалить пакет `os` из списка `import` программы, то `packageNotUsed.go` отлично скомпилируется — попробуйте сами.

Сейчас еще не время говорить о том, как нарушать правила Go, однако существует способ обойти такое ограничение. Он показан в следующем Go-коде, который сохраняется в файле `packageNotUsedUnderscore.go`:

```
package main

import (
    "fmt"
    _ "os"
)

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

Как видим, если в списке `import` поставить перед именем пакета символ подчеркивания, то мы не получим сообщение об ошибке в процессе компиляции, даже если этот пакет не используется в программе:

```
$ go run packageNotUsedUnderscore.go
Hello there!
```

Причина, по которой Go позволяет обойти это правило, станет более понятной в главе 6.

Правильный вариант размещения фигурных скобок — всего один

Рассмотрим следующую Go-программу с именем `curly.go`:

```
package main

import (
    "fmt"
)

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

Все выглядит просто отлично, но если вы попытаетесь это выполнить, то будете весьма разочарованы, потому что код не скомпилируется и, соответственно, не запустится, а вы получите следующее сообщение о *синтаксической ошибке*:

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

Официально смысл этого сообщения об ошибке разъясняется так: во многих контекстах Go требует использования точки с запятой как признака завершения оператора и поэтому компилятор автоматически вставляет точки с запятой там, где считает их необходимыми. Поэтому при размещении открывающей фигурной скобки ({) в отдельной строке компилятор Go поставит точку с запятой в конце предыдущей строки (`func main()`) — это и есть причина сообщения об ошибке.

Как скачивать Go-пакеты

Стандартная библиотека Go весьма обширна, однако бывают случаи, когда необходимо загрузить внешние пакеты Go, чтобы использовать их функциональные возможности. В этом разделе вы узнаете, как загрузить внешний Go-пакет и где он будет размещен на вашем UNIX-компьютере.

Имейте в виду, что недавно в Go появился новый функционал — модули, которые все еще находятся в стадии разработки и поэтому могут внести изменения в работу с внешним Go-кодом. Однако процедура загрузки на компьютер отдельного Go-пакета останется прежней.

Вы узнаете намного больше о пакетах и модулях Go из главы 6.

Рассмотрим следующую простую Go-программу, которая сохраняется как `getPackage.go`:

```
package main

import (
    "fmt"
    "github.com/mactsouk/go/simpleGitHub"
)

func main() {
    fmt.Println(simpleGitHub.AddTwo(5, 6))
}
```

В одной из команд `import` указан интернет-адрес — это значит, что в программе используется внешний пакет. В данном случае внешний пакет называется `simpleGitHub` и находится по адресу `github.com/mactsouk/go/simpleGitHub`.

Если вы попытаетесь сразу выполнить getPackage.go, то будете разочарованы:

```
$ go run getPackage.go
getPackage.go:5:2: cannot find package
"github.com/mactsouk/go/simpleGitHub" in any of:
    /usr/local/Cellar/go/1.9.1/libexec/src/github.com/mactsouk/go/simpleGitHub
        (from $GOROOT)
    /Users/mtsouk/go/src/github.com/mactsouk/go/simpleGitHub (from $GOPATH)
```

Как видно, необходимо установить недостающий пакет на ваш компьютер. Чтобы скачать этот пакет, нужно выполнить следующую команду:

```
$ go get -v github.com/mactsouk/go/simpleGitHub
github.com/mactsouk/go (download)
github.com/mactsouk/go/simpleGitHub
```

Загруженные файлы вы найдете в следующем каталоге:

```
$ ls -l ~/go/src/github.com/mactsouk/go/simpleGitHub/
total 8
-rw-r--r--  1 mtsouk  staff  66 Oct 17 21:47 simpleGitHub.go
```

Однако команда go get также компилирует пакет. Соответствующие файлы размещаются здесь:

```
$ ls -l ~/go/pkg/darwin_amd64/github.com/mactsouk/go/simpleGitHub.a
-rw-r--r--  1 mtsouk  staff  1050 Oct 17 21:47
/Users/mtsouk/go/pkg/darwin_amd64/github.com/mactsouk/go/simpleGitHub.a
```

Теперь без проблем выполняйте getPackage.go:

```
$ go run getPackage.go
11
```

Временные файлы загруженного Go-пакета можно удалить:

```
$ go clean -i -v -x github.com/mactsouk/go/simpleGitHub
cd /Users/mtsouk/go/src/github.com/mactsouk/go/simpleGitHub
rm -f simpleGitHub.test simpleGitHub.test.exe
rm -f /Users/mtsouk/go/pkg/darwin_amd64/github.com/mactsouk/go/
simpleGitHub.a
```

Аналогичным образом с помощью UNIX-команды rm(1) можно удалить весь загруженный и размещенный на локальном компьютере Go-пакет, чтобы удалить его исходный Go-код после использования go clean:

```
$ go clean -i -v -x github.com/mactsouk/go/simpleGitHub
$ rm -rf ~/go/src/github.com/mactsouk/go/simpleGitHub
```

После выполнения этих команд, чтобы использовать Go-пакет, вам придется снова его загрузить.

Стандартные потоки UNIX: stdin, stdout и stderr

В любой ОС UNIX есть три файла, которые постоянно открыты для своих процессов. Помните, что для UNIX все является файлом, даже если это принтер или мышь.

В качестве внутреннего представления для доступа ко всем открытым файлам UNIX использует *файловые дескрипторы* — положительные целые числа. Это гораздо красивее, чем длинные пути.

Таким образом, по умолчанию любая UNIX-система поддерживает три специальных стандартных имени файла: `/dev/stdin`, `/dev/stdout` и `/dev/stderr`, к которым также можно получить доступ, используя файловые дескрипторы 0, 1 и 2 соответственно. Эти три файловых дескриптора называются *стандартными потоками ввода, вывода и ошибок* соответственно. Кроме того, на компьютере с MacOS файловый дескриптор 0 может быть доступен как `/dev/fd/0`, а на компьютере с Debian Linux — как `/dev/fd/0` или `/dev/pts/0`.

Для доступа к стандартному потоку ввода Go использует `os.Stdin`, для доступа к стандартному потоку вывода — `os.Stdout` и для доступа к стандартному потоку ошибок — `os.Stderr`. Несмотря на то что для доступа к этим устройствам все равно можно использовать `/dev/stdin`, `/dev/stdout` и `/dev/stderr` или дескрипторы соответствующих файлов, все же лучше, безопаснее и легче придерживаться варианта с `os.Stdin`, `os.Stdout` и `os.Stderr`, предлагаемого Go.

Вывод результатов

Подобно C в UNIX, Go предлагает различные способы вывода результатов на экран. Все функции вывода, использованные в этом разделе, требуют подключения стандартного Go-пакета `fmt`. Функции будут продемонстрированы в программе `printing.go`, которую мы разделим на две части.

Самый простой способ вывести что-нибудь в Go — использовать функцию `fmt.Println()` или `fmt.Printf()`. У функции `fmt.Printf()` много общего с C-функцией `printf(3)`. Вместо `fmt.Print()` также можно использовать `fmt.Println()`. Основное различие между функциями `fmt.Print()` и `fmt.Println()` заключается в том, что последняя при каждом вызове автоматически добавляет в конец вывода символ новой строки.

В то же время наибольшим различием между `fmt.Println()` и `fmt.Printf()` является то, что в `fmt.Printf()` нужно указывать *спецификатор формата* для всего, что вы хотите вывести на экран, точно так же, как в C-функции `printf(3)`. С одной стороны, это означает, что вы лучше контролируете то, что делаете, но с другой — приходится писать больше кода. В Go эти спецификаторы формата называются *глаголами*. Подробнее о глаголах читайте на странице https://golang.org/pkg/fmt/.

Если перед выводом на экран нужно выполнить какое-либо форматирование или разместить несколько переменных, то лучше воспользоваться `fmt.Printf()`. Однако, если вы хотите вывести только одну переменную, то, возможно, лучше выбрать `fmt.Print()` или `fmt.Println()`, в зависимости от того, нужен ли вам в конце символ новой строки или нет.

Первая часть `printing.go` содержит следующий Go-код:

```
package main

import (
    "fmt"
)

func main() {
    v1 := "123"
    v2 := 123
    v3 := "Have a nice day\n"
    v4 := "abc"
```

В этой части мы видим импорт пакета `fmt` и определение четырех Go-переменных. Символ `\n`, использованный в `v3`, является символом разрыва строки. Однако если вы просто хотите вставить разрыв строки в выводимые данные, то вместо чего-то вроде `fmt.Print("\n")` можете просто вызвать `fmt.Println()` без аргументов.

Вторая часть `printing.go` выглядит так:

```
    fmt.Print(v1, v2, v3, v4)
    fmt.Println()
    fmt.Println(v1, v2, v3, v4)
    fmt.Print(v1, " ", v2, " ", v3, " ", v4, "\n")
    fmt.Printf("%s%d %s %s\n", v1, v2, v3, v4)
}
```

В этой части выводятся четыре переменные с использованием функций `fmt.Println()`, `fmt.Print()` и `fmt.Printf()`, чтобы лучше понять, чем эти функции различаются между собой.

Если выполнить `print.go`, то получим следующий вывод:

```
$ go run printing.go
123123Have a nice day
abc
123 123 Have a nice day
abc
123 123 Have a nice day
abc
123123 Have a nice day
abc
```

Как видим, функция `fmt.Println()`, в отличие от `fmt.Print()`, также добавляет пробел между выводимыми параметрами.

В результате вызов наподобие `fmt.Println(v1, v2)` эквивалентен `fmt.Print(v1, " ", v2, "\n")`.

Кроме `fmt.Println()`, `fmt.Print()` и `fmt.Printf()`, которые являются простейшими функциями для вывода данных на экран, существует также семейство S-функций, в которое входят функции `fmt.Sprintln()`, `fmt.Sprint()` и `fmt.Sprintf()`. Они используются для построения строк по заданному формату.

Наконец, существует семейство F-функций, в которое входят функции `fmt.Fprintln()`, `fmt.Fprint()` и `fmt.Fprintf()`. Они используются для записи в файлы с помощью `io.Writer`.

Подробнее об интерфейсах io.Writer и io.Reader вы прочитаете в главе 8.

В следующем разделе вы узнаете, как выводить данные на экран, используя стандартный поток вывода, что является довольно распространенной практикой в мире UNIX.

Использование стандартного потока вывода

Стандартный поток вывода более или менее эквивалентен выводу на экран. Однако его использование иногда требует применения функций, не входящих в пакет `fmt`, именно поэтому стандартному потоку вывода посвящен отдельный раздел.

Рассмотрим соответствующий метод в программе `stdOUT.go`, которую разделим на три части. Первая часть программы выглядит так:

```
package main

import (
    "io"
    "os"
)
```

Итак, в `stdOUT.go` вместо пакета `fmt` будет использоваться пакет `io`. Пакет `os` применяется для чтения программой аргументов командной строки и для доступа к `os.Stdout`.

Вторая часть `stdOUT.go` содержит следующий Go-код:

```
func main() {
    myString := ""
    arguments := os.Args
    if len(arguments) == 1 {
        myString = "Please give me one argument!"
    } else {
        myString = arguments[1]
    }
```

Переменная `myString` содержит текст, который будет выведен на экран. Этот текст является либо первым аргументом командной строки программы, либо, если программа выполнялась без аргументов командной строки, — жестко закодированным текстовым сообщением.

Третья часть программы выглядит следующим образом:

```
    io.WriteString(os.Stdout, myString)
    io.WriteString(os.Stdout, "\n")
}
```

В данном случае функция `io.WriteString()` работает так же, как `fmt.Print()`, однако принимает только два параметра. Первый из них — это файл, в который будут записываться результаты, в данном случае `os.Stdout`, а второй — строковая переменная.

Строго говоря, первый параметр функции io.WriteString() должен иметь тип io.Writer, что требует в качестве второго параметра срез байтов. Однако в данном случае строковая переменная отлично справляется со своей задачей. Подробнее о срезах вы узнаете в главе 3.

Запуск `stdOUT.go` приведет к следующему результату:

```
$ go run stdOUT.go
Please give me one argument!
$ go run stdOUT.go 123 12
123
```

Как видим, функция `io.WriteString()` отправляет содержимое своего второго параметра на экран, если первым параметром является `os.Stdout`.

Получение данных от пользователя

Есть три основных способа получения данных от пользователя: прочитать аргументы командной строки программы, предложить пользователю ввести данные или прочитать внешние файлы. В этом разделе описаны первые два способа. Чтобы узнать, как прочитать внешний файл, обратитесь к главе 8.

Что такое := и =

Прежде чем продолжить, полезно узнать об использовании оператора `:=` и его отличии от `=`. Официальное название `:=` — *сокращенный оператор присваивания*. Сокращенный оператор присваивания можно использовать вместо объявления `var` с неявным типом.

Использование var в Go встречается редко. Ключевое слово var в Go-программах используется главным образом для объявления глобальных переменных, а также для объявления переменных без начального значения. Причина первого варианта заключается в том, что каждое утверждение, существующее вне кода функции, должно начинаться с ключевого слова, такого как func или var. Это означает, что сокращенный оператор присваивания нельзя использовать вне функции, потому что он там недоступен.

Оператор `:=` работает следующим образом:

```
m := 123
```

Результатом выполнения этого оператора станет создание целочисленной переменной с именем `m` и значением `123`.

Но если вы попытаетесь использовать `:=` для уже объявленной переменной, то компиляция завершится неудачно и выдаст следующее предельно ясное сообщение об ошибке:

```
$ go run test.go
# command-line-arguments
./test.go:5:4: no new variables on left side of :=
```

Возможно, у вас возникнет вопрос: что произойдет, если мы ожидаем от функции два значения или более и хотим использовать существующую переменную для одного из них? Какой оператор применять: `:=` или `=`? Ответ прост: оператор `:=`, как в следующем примере кода:

```
i, k := 3, 4
j, k := 1, 2
```

Поскольку переменная `j` впервые встречается во втором выражении, то мы использовали `:=`, даже если в первом выражении значение `k` уже определено.

Возможно, вам покажется скучным обсуждение подобных незначительных деталей, однако впоследствии, зная эти нюансы, вы сможете избежать различных ошибок!

Чтение стандартного потока ввода

Чтение данных из стандартного потока ввода рассмотрим на примере программы `stdIN.go`, которую разделим на две части. Первая часть выглядит так:

```
package main

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

В данном коде нам впервые в этой книге встретился пакет `bufio`.

Подробнее о пакете bufio вы узнаете в главе 8.

Обычно для ввода и вывода файлов используется пакет `bufio`, но в этой книге мы будем использовать в основном пакет `os`, так как в нем содержится много удобных функций; одно из самых востребованных свойств этого пакета — то, что он предоставляет доступ к аргументам командной строки Go-программы (`os.Args`).

Согласно официальному описанию пакета `os` в нем представлены функции, которые выполняют операции ОС. Сюда входят функции создания, удаления, переименования файлов и каталогов, а также функции распознавания полномочий доступа UNIX и других характеристик файлов и каталогов. Основным преимуществом пакета `os` является то, что он не зависит от платформы. Проще говоря, его функции будут работать на компьютерах как с UNIX, так и с Microsoft Windows.

Вторая часть `stdIN.go` содержит следующий Go-код:

```
func main() {
    var f *os.File
    f = os.Stdin
    defer f.Close()

    scanner := bufio.NewScanner(f)
    for scanner.Scan() {
        fmt.Println(">", scanner.Text())
    }
}
```

Во-первых, здесь вызывается функция `bufio.NewScanner()`, которой в качестве параметра передается стандартный поток ввода (`os.Stdin`). Этот вызов возвращает переменную `bufio.Scanner`, которая используется функцией `Scan()` для построчного чтения из `os.Stdin`. Каждая прочитанная строка выводится на экран, после чего считывается следующая строка. Обратите внимание, что каждая строка, которую выводит программа, начинается с символа `>`.

Запуск `stdIN.go` даст нам следующий вывод:

```
$ go run stdIN.go
This is number 21
> This is number 21
This is Mihalis
> This is Mihalis
Hello Go!
> Hello Go!
Press Control + D on a new line to end this program!
> Press Control + D on a new line to end this program!
```

Как принято в UNIX, вы можете прервать чтение программой данных из стандартного потока ввода, нажав Ctrl+D.

Go-программы stdIN.go и stdOUT.go очень просты, однако не стоит их недооценивать: они пригодятся, когда мы будем обсуждать UNIX-каналы в главе 8.

Работа с аргументами командной строки

Способ, описанный в этом разделе, рассмотрим на примере Go-кода `cla.go`, который разделим на три части. Эта программа находит минимальный и максимальный из своих аргументов командной строки.

Первая часть программы выглядит так:

```
package main

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

Важно понимать, что для того, чтобы программа получала аргументы командной строки, необходимо использовать пакет `os`. Кроме того, нам нужен еще один пакет, `strconv`, чтобы была возможность преобразовывать аргументы командной строки, заданные в виде строк, в арифметический тип данных.

Вторая часть программы выглядит так:

```
func main() {
    if len(os.Args) == 1 {
        fmt.Println("Please give one or more floats.")
        os.Exit(1)
    }

    arguments := os.Args
    min, _ := strconv.ParseFloat(arguments[1], 64)
    max, _ := strconv.ParseFloat(arguments[1], 64)
```

Здесь `cla.go` проверяет, есть ли у нас аргументы командной строки, контролируя длину `os.Args`. Дело в том, что для работы программы необходим как минимум один аргумент командной строки. Обратите внимание, что `os.Args` — это срез Go, содержащий значения типа `string`. Первый элемент среза — это имя исполняемой программы. Поэтому, чтобы инициализировать переменные `min` и `max`, нужно использовать второй элемент среза `os.Args`, имеющий тип `string`, индекс которого равен 1.

Важный момент: если вы ожидаете одно или несколько значений с плавающей точкой, это вовсе не значит, что пользователь предоставит вам именно корректные значения с плавающей точкой. Он может этого и не сделать — случайно или преднамеренно. Однако, поскольку мы пока еще не обсуждали обработку ошибок в Go, `cla.go` предполагает, что все аргументы командной строки имеют правильный формат и, следовательно, будут приемлемыми. Поэтому `cla.go` игнорирует значение типа `error`, возвращаемое функцией `strconv.ParseFloat()`, используя следующую инструкцию:

```
n, _ := strconv.ParseFloat(arguments[i], 64)
```

Предыдущий оператор говорит Go, что мы хотим получить только первое значение, возвращаемое `strconv.ParseFloat()`, и что нас не интересует второе значение, которое в данном случае является переменной типа `error`, — мы присваиваем это значение символу подчеркивания. Символ подчеркивания, который называют *пустым идентификатором*, является средством сброса значения в Go. Если Go-функция возвращает несколько значений, пустой идентификатор можно использовать несколько раз.

Игнорирование всех или некоторых значений, возвращаемых Go-функцией, особенно значений типа error, — очень опасный метод, который не следует использовать в продуктивном коде!

Третья часть содержит следующий Go-код:

```
    for i := 2; i < len(arguments); i++ {
        n, _ := strconv.ParseFloat(arguments[i], 64)

        if n < min {
            min = n
        }
        if n > max {
            max = n
        }
    }

    fmt.Println("Min:", min)
    fmt.Println("Max:", max)
}
```

Здесь используется цикл `for`. Он позволяет перебрать все элементы среза `os.Args`, который перед этим присвоен переменной `arguments`.

Выполнение `cla.go` приведет к следующему выводу:

```
$ go run cla.go -10 0 1
Min: -10
Max: 1
```

```
$ go run cla.go -10
Min: -10
Max: -10
```

Как и следовало ожидать, программа ведет себя некорректно, когда на вход подаются ошибочные данные; хуже всего то, что она не выводит никаких предупреждений, сообщающих пользователю об ошибке (или нескольких ошибках), возникших во время обработки аргументов командной строки:

```
$ go run cla.go a b c 10
Min: 0
Max: 10
```

Вывод ошибок

В этом разделе представлен метод передачи данных в *стандартный поток ошибок* UNIX, который помогает различить в UNIX реальные значения и вывод сообщений об ошибках.

Go-код для иллюстрации использования стандартного потока ошибок в Go содержится в файле `stdERR.go`. Мы его рассмотрим, разделив на две части. Поскольку запись в стандартный поток требует использования дескриптора файла, связанного со стандартным потоком ошибок, Go-код `stdERR.go` будет основан на Go-коде из `stdOUT.go`.

Первая часть программы выглядит так:

```
package main

import (
    "io"
    "os"
)

func main() {
    myString := ""
    arguments := os.Args
    if len(arguments) == 1 {
        myString = "Please give me one argument!"
    } else {
        myString = arguments[1]
    }
```

До сих пор `stdERR.go` практически не отличается от `stdOUT.go`.

Вторая часть `stdERR.go` выглядит так:

```
    io.WriteString(os.Stdout, "This is Standard output\n")
    io.WriteString(os.Stderr, myString)
    io.WriteString(os.Stderr, "\n")
}
```

Функция `io.WriteString()` вызывается два раза для записи в стандартный поток ошибок (`os.Stderr`) и еще один раз — для записи в стандартный поток вывода (`os.Stdout`).

Выполнение `stdERR.go` даст следующий результат:

```
$ go run stdERR.go
This is Standard output
Please give me one argument!
```

Этот вывод не поможет нам отличать данные, записанные в стандартный поток вывода, от данных, записанных в стандартный поток ошибок, что иногда очень полезно. Но если использовать оболочку `bash(1)`, то можно прибегнуть к одному приему, позволяющему отличать данные из стандартного потока вывода от данных из стандартного потока ошибок. Почти в каждой UNIX-оболочке эта функциональность реализована по-своему.

При применении `bash(1)` можно перенаправить стандартный поток ошибок в файл:

```
$ go run stdERR.go 2>/tmp/stdError
This is Standard output
$ cat /tmp/stdError
Please give me one argument!
```

Число, стоящее после имени UNIX-программы или после системного вызова, соответствует номеру раздела руководства, к которому относится страница этой программы или вызова. Большинство имен встречается только на одной странице справочника, следовательно, указывать номер раздела не обязательно. Однако есть имена, которые встречаются в нескольких разделах, поскольку имеют несколько значений, такие как crontab(1) и crontab(5). Поэтому если вы попытаетесь получить справочную страницу имени с несколькими значениями, не указав номер раздела, то получите запись с наименьшим номером раздела.

Подобным образом можно отменить вывод ошибок, перенаправив их на устройство `/dev/null`, что соответствует указанию UNIX полностью игнорировать этот вывод:

```
$ go run stdERR.go 2>/dev/null
This is Standard output
```

В двух примерах мы перенаправили файловый дескриптор стандартного потока ошибок в определенный файл и в `/dev/null` соответственно. Если вы хотите сохранить данные стандартного потока вывода и стандартного потока ошибок в одном файле, то можете перенаправить файловый дескриптор стандартного потока

ошибок (`2`) в файловый дескриптор стандартного потока вывода (`1`). В следующей команде показан довольно распространенный в UNIX-системах способ:

```
$ go run stdERR.go >/tmp/output 2>&1
$ cat /tmp/output
This is Standard output
Please give me one argument!
```

Наконец, можно перенаправить и стандартный поток вывода, и стандартный поток ошибок на `/dev/null`:

```
$ go run stdERR.go >/dev/null 2>&1
```

Запись в журнальные файлы

Пакет `log` позволяет отправлять журнальные сообщения в системную службу журналирования UNIX-машины, а Go-пакет `syslog`, который является частью пакета `log`, позволяет определить *уровень журналирования* и *средство журналирования*, которое будет использовать ваша Go-программа.

Обычно большинство системных журнальных файлов в UNIX размещаются в каталоге `/var/log`. Однако журнальные файлы многих популярных сервисов, таких как Apache и Nginx, находятся в других местах, в зависимости от их конфигурации.

Вообще, использование журнальных файлов для записи некоторой информации считается более эффективной практикой, чем вывод тех же данных на экран, по двум причинам: во-первых, потому, что выходные данные, сохраняемые в файле, не теряются, а во-вторых, потому, что вы можете вести поиск и обрабатывать файлы журналов с помощью инструментов UNIX, таких как `grep(1)`, `awk(1)` и `sed(1)`; если же сообщения выводятся в окне терминала, это невозможно.

Пакет `log` предлагает множество функций для перенаправления вывода в системную службу журналирования UNIX-машины. В число этих функций входят `log.Printf()`, `log.Print()`, `log.Println()`, `log.Fatalf()`, `log.Fatalln()`, `log.Panic()`, `log.Panicln()` и `log.Panicf()`.

Обратите внимание, что функции журналирования могут быть очень полезны при отладке программ, особенно написанных на Go серверных процессов. Не стоит недооценивать их возможности.

Уровни журналирования

Уровень журналирования — это значение, которое определяет степень серьезности журнального сообщения. Существуют следующие уровни журналирования (в порядке возрастания серьезности): debug, info, notice, warning, err, crit, alert и emerg.

Средства журналирования

Средство журналирования подобно категории, используемой при регистрации информации. Средство журналирования принимает значения `auth`, `authpriv`, `cron`, `daemon`, `kern`, `lpr`, `mail`, `mark`, `news`, `syslog`, `user`, `UUCP`, `local0`, `local1`, `local2`, `local3`, `local4`, `local5`, `local6` или `local7` и определяется в файле `/etc/syslog.conf`, `/etc/rsyslog.conf` или другом соответствующем файле, в зависимости от процесса сервера, используемого для ведения журнала в операционной системе на данной UNIX-машине.

Это означает, что, если средство журналирования не определено и, соответственно, не поддерживается, отправленные в него журнальные сообщения могут быть проигнорированы и, следовательно, потеряны.

Серверы журналирования

На каждом компьютере с UNIX есть особый серверный процесс, который отвечает за получение журнальных сообщений и запись их в лог-файлы. Существует различные разновидности серверов журналирования. Но только два из них используются в большинстве вариантов UNIX: `syslogd(8)` и `rsyslogd(8)`.

На машинах с macOS процесс ведения журнала называется `syslogd(8)`. На большинстве машин с Linux, напротив, используется `rsyslogd(8)` — улучшенная и более надежная версия `syslogd(8)`, которая была первой системной утилитой UNIX для регистрации журнальных сообщений.

Впрочем, независимо от вашего варианта UNIX или серверного процесса, используемого для ведения журнала, на всех компьютерах с UNIX журналирование работает одинаково и, следовательно, не влияет на Go-код, который вы напишете.

Файл конфигурации `rsyslogd(8)` обычно называется `rsyslog.conf` и находится в `/etc`. Содержимое файла конфигурации `rsyslog.conf`, без учета строк с комментариями и строк, начинающихся с `$`, может выглядеть следующим образом:

```
$ grep -v '^#' /etc/rsyslog.conf | grep -v '^$' | grep -v '^\$'
auth,authpriv.*                 /var/log/auth.log
*.*;auth,authpriv.none          -/var/log/syslog
daemon.*                        -/var/log/daemon.log
kern.*                          -/var/log/kern.log
lpr.*                           -/var/log/lpr.log
mail.*                          -/var/log/mail.log
user.*                          -/var/log/user.log
mail.info                       -/var/log/mail.info
mail.warn                       -/var/log/mail.warn
mail.err                        /var/log/mail.err
news.crit                       /var/log/news/news.crit
```

```
news.err                      /var/log/news/news.err
news.notice                   -/var/log/news/news.notice
*.=debug;\
    auth,authpriv.none;\
    news.none;mail.none       -/var/log/debug
*.=info;*.=notice;*.=warn;\
    auth,authpriv.none;\
    cron,daemon.none;\
    mail,news.none            -/var/log/messages
*.emerg                  :omusrmsg:*
daemon.*;mail.*;\
    news.err;\
    *.=debug;*.=info;\
    *.=notice;*.=warn        |/dev/xconsole
local7.* /var/log/cisco.log
```

Таким образом, для того чтобы передать журнальную информацию в `/var/log/cisco.log`, нужно использовать средство журналирования *local7*. Символ звездочки после имени средства журналирования дает серверу журналирования указание перехватывать все уровни журналирования, которые поступают в средство журналирования local7, и записывать их в `/var/log/cisco.log`.

Сервер `syslogd(8)` имеет довольно похожий файл конфигурации, обычно это `/etc/syslog.conf`. В macOS High Sierra файл `/etc/syslog.conf` практически пуст и заменен на `/etc/asl.conf`. Тем не менее, логика конфигурации у `/etc/syslog.conf`, `/etc/rsyslog.conf` и `/etc/asl.conf` одинакова.

Пример Go-программы, которая записывает информацию в журнальные файлы

Рассмотрим использование пакетов `log` и `log/syslog` для записи в файлы системного журнала на примере Go-кода, представленного в файле `logFiles.go`.

Обратите внимание: пакет log/syslog не реализован в версии Go для Microsoft Windows.

Первая часть `logFiles.go` выглядит так:

```
package main

import (
    "fmt"
    "log"
    "log/syslog"
    "os"
```

```
    "path/filepath"
)

func main() {
    programName := filepath.Base(os.Args[0])
    sysLog, err := syslog.New(syslog.LOG_INFO|syslog.LOG_LOCAL7, programName)
```

Первым параметром функции `syslog.New()` является приоритет, который представляет собой соединение средства журналирования и уровня журналирования. Так, приоритет `LOG_NOTICE | LOG_MAIL`, который выбран в качестве примера, будет отправлять сообщения уровня журналирования `NOTICE` в средство журналирования `MAIL`.

В результате этот код устанавливает такой режим журналирования по умолчанию: средство журналирования *local7* и уровень журналирования *info*. Второй параметр функции `syslog.New()` — это имя процесса, который будет отображаться в журналах в качестве отправителя сообщения. Вообще, рекомендуется использовать настоящее имя исполняемого файла, чтобы впоследствии можно было легко найти нужную информацию в файлах журналов.

Вторая часть программы содержит следующий Go-код:

```
    if err != nil {
        log.Fatal(err)
    } else {
        log.SetOutput(sysLog)
    }
    log.Println("LOG_INFO + LOG_LOCAL7: Logging in Go!")
```

После вызова функции `syslog.New()` нужно проверить переменную типа `error`, которая возвращается этой функцией, чтобы убедиться, что все в порядке. Если это так, тогда значение переменной типа `error` равно `nil` и можно вызвать функцию `log.SetOutput()`, которая задает приемник вывода для записи в журнал по умолчанию, — в данном случае это созданный нами ранее регистратор журнала (`sysLog`). Затем можно использовать функцию `log.Println()` для отправки информации на сервер журнала.

Третья часть `logFiles.go` содержит следующий код:

```
    sysLog, err = syslog.New(syslog.LOG_MAIL, "Some program!")
    if err != nil {
        log.Fatal(err)
    } else {
        log.SetOutput(sysLog)
    }

    log.Println("LOG_MAIL: Logging in Go!")
    fmt.Println("Will you see this?")
}
```

Как видно из последней части кода, мы можем изменять конфигурацию ведения журнала в своих программах столько раз, сколько захотим, и при этом все равно можем использовать `fmt.Println()` для вывода данных на экран.

При выполнении `logFiles.go` на экран компьютера с Debian Linux будут выведены следующие данные:

```
$ go run logFiles.go
Broadcast message from systemd-journald@mail (Tue 2017-10-17 20:06:08 EEST):
logFiles[23688]: Some program![23688]: 2017/10/17 20:06:08 LOG_MAIL:
Logging in Go!
Message from syslogd@mail at Oct 17 20:06:08 ...
Some program![23688]: 2017/10/17 20:06:08 LOG_MAIL: Logging in Go!
Will you see this?
```

Выполнение того же Go-кода на компьютере с macOS High Sierra приведет к следующим результатам:

```
$ go run logFiles.go
Will you see this?
```

Имейте в виду, что на большинстве UNIX-машин информация журналов хранится в нескольких файлах. Это также относится и к машине с Debian Linux, использованной в этом разделе. В результате `logFiles.go` отправляет выходные данные в несколько файлов журнала, в чем можно убедиться с помощью следующих команд оболочки:

```
$ grep LOG_MAIL /var/log/mail.log
Oct 17 20:06:08 mail Some program![23688]: 2017/10/17 20:06:08 LOG_MAIL:
Logging in Go!
$ grep LOG_LOCAL7 /var/log/cisco.log
Oct 17 20:06:08 mail logFiles[23688]: 2017/10/17 20:06:08 LOG_INFO +
LOG_LOCAL7: Logging in Go!
$ grep LOG_ /var/log/syslog
Oct 17 20:06:08 mail logFiles[23688]: 2017/10/17 20:06:08 LOG_INFO +
LOG_LOCAL7: Logging in Go!
Oct 17 20:06:08 mail Some program![23688]: 2017/10/17 20:06:08 LOG_MAIL:
Logging in Go!
```

Как видно из результатов, сообщение оператора `log.Println("LOG_INFO + LOG_LOCAL7: Logging in Go!")` записано в два файла: `/var/log/cisco.log` и `/var/log/syslog`, тогда как сообщение оператора `log.Println("LOG_MAIL: Logging in Go!")` попало в `/var/log/syslog` и `/var/log/mail.log`.

Из этого раздела важно запомнить следующее: если сервер журналирования на UNIX-машине не настроен на перехват всех средств журналирования, то некоторые из отправляемых на него записей журнала могут быть удалены без каких-либо предупреждений.

Функция log.Fatal()

В этом разделе мы познакомимся с тем, как работает функция `log.Fatal()`. Она используется тогда, когда происходит что-то действительно очень плохое и вы просто хотите поскорее выйти из программы, предварительно отправив сообщение о сложной ситуации.

Использование `log.Fatal()` будет проиллюстрировано в программе `logFatal.go`, которая содержит следующий Go-код:

```
package main

import (
    "fmt"
    "log"
    "log/syslog"
)

func main() {
    sysLog, err := syslog.New(syslog.LOG_ALERT|syslog.LOG_MAIL, "Some program!")
    if err != nil {
        log.Fatal(err)
    } else {
        log.SetOutput(sysLog)
    }
    log.Fatal(sysLog)
    fmt.Println("Will you see this?")
}
```

Выполнение `logFatal.go` приведет к следующему выводу:

```
$ go run logFatal.go
exit status 1
```

Легко понять, что при использовании `log.Fatal()` Go-программа завершается в том месте, где была вызвана функция `log.Fatal()`. Именно поэтому вы не увидели вывод оператора `fmt.Println("Will you see this?")`.

Однако в соответствии с параметрами вызова `syslog.New()` при этом в файл журнала `/var/log/mail.log`, связанный с почтой, добавлена запись:

```
$ grep "Some program" /var/log/mail.log
Jan 10 21:29:34 iMac Some program![7123]: 2019/01/10 21:29:34 &{17 Some program!
iMac.local {0 0} 0xc00000c220}
```

Функция log.Panic()

Возможны ситуации, когда программа завершается сбоем и вы хотите получить как можно больше информации о нем.

В такие сложные моменты одним из решений станет использование `log.Panic()` — разновидности функций журналирования, работа которой будет продемонстрирована в этом разделе на примере Go-кода из файла `logPanic.go`.

Go-код `logPanic.go` выглядит следующим образом:

```
package main

import (
    "fmt"
    "log"
    "log/syslog"
)

func main() {
    sysLog, err := syslog.New(syslog.LOG_ALERT|syslog.LOG_MAIL, "Some program!")
    if err != nil {
        log.Fatal(err)
    } else {
        log.SetOutput(sysLog)
    }

    log.Panic(sysLog)
    fmt.Println("Will you see this?")
}
```

Выполнение `logPanic.go` в macOS Mojave приведет к следующим результатам:

```
$ go run logPanic.go
panic: &{17 Some program! iMac.local
{0 0} 0xc0000b21e0}
goroutine 1 [running]:
log.Panic(0xc00004ef68, 0x1, 0x1)
    /usr/local/Cellar/go/1.11.4/libexec/src/log/log.go:326 +0xc0
main.main()
    /Users/mtsouk/Desktop/mGo2nd/Mastering-Go-Second-Edition/ch01/logPanic.go:17 +0xd6
exit status 2
```

Выполнение той же программы в Debian Linux с версией Go 1.3.3 приведет к следующему:

```
$ go run logPanic.go
panic: &{17 Some program! mail
{0 0} 0xc2080400e0}
goroutine 16 [running]:
runtime.panic(0x4ec360, 0xc208000320)
    /usr/lib/go/src/pkg/runtime/panic.c:279 +0xf5
log.Panic(0xc208055f20, 0x1, 0x1)
    /usr/lib/go/src/pkg/log/log.go:307 +0xb6
main.main()
```

```
    /home/mtsouk/Desktop/masterGo/ch/ch1/code/logPanic.go:17 +0x169
goroutine 17 [runnable]:
runtime.MHeap_Scavenger()
    /usr/lib/go/src/pkg/runtime/mheap.c:507
runtime.goexit()
    /usr/lib/go/src/pkg/runtime/proc.c:1445
goroutine 18 [runnable]:
bgsweep()
    /usr/lib/go/src/pkg/runtime/mgc0.c:1976
runtime.goexit()
    /usr/lib/go/src/pkg/runtime/proc.c:1445
goroutine 19 [runnable]:
runfinq()
    /usr/lib/go/src/pkg/runtime/mgc0.c:2606
runtime.goexit()
    /usr/lib/go/src/pkg/runtime/proc.c:1445
exit status 2
```

Таким образом, `log.Panic()` выводит дополнительную низкоуровневую информацию, которая, по идее, должна помочь разрешить сложную ситуацию, возникшую в Go-коде.

Как и `log.Fatal()`, функция `log.Panic()` добавит запись в соответствующий файл журнала и немедленно прекратит работу Go-программы.

Запись в специальный журнальный файл

Иногда нужно просто записать журнальные сообщения в выбранный вами файл. Это может понадобиться по многим причинам. Например, для записи данных отладки, которые бывают слишком большими. Или если вы не хотите записывать их в системные журнальные файлы, а собираетесь хранить данные в собственных журналах, отдельно от системных данных, а потом, допустим, передать их куда-то или сохранить в базе данных. Или же вы хотите хранить свои данные в другом формате. В этом подразделе вы узнаете, как организовать запись в специальный журнальный файл.

Назовем нашу Go-утилиту `customLog.go` и будем записывать данные в журнальный файл `/tmp/mGo.log`.

Разделим Go-код для `customLog.go` на три части. Первая часть выглядит следующим образом:

```
package main

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

var LOGFILE = "/tmp/mGo.log"
```

Путь к журнальному файлу в customLog.go задан жестко как значение глобальной переменной с именем LOGFILE. Для наших целей в примере журнальный файл находится в каталоге /tmp, что не является обычным местом для хранения данных, поскольку, как правило, каталог /tmp очищается после каждой перезагрузки системы. Однако на данном этапе это избавит нас от необходимости запускать customLog.go с полномочиями пользователя root и не потребует размещать лишние файлы в системных каталогах. Если позже вы решите использовать код customLog.go в реальном приложении, вам следует изменить путь на более правильный.

Вторая часть customLog.go выглядит следующим образом:

```
func main() {
    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()
```

Здесь мы создаем новый журнальный файл, используя функцию os.OpenFile() с необходимыми правами доступа к UNIX-файлам (0644).

Последняя часть customLog.go выглядит следующим образом:

```
    iLog := log.New(f, "customLogLineNumber ", log.LstdFlags)

    iLog.SetFlags(log.LstdFlags)
    iLog.Println("Hello there!")
    iLog.Println("Another log entry!")
}
```

Если вы посмотрите на страницу документации пакета log, которую можно найти по адресу https://golang.org/pkg/log/, то увидите, что функция SetFlags позволяет устанавливать выходные флаги (варианты) для текущего средства журналирования. По умолчанию функция предлагает значения LstdFlags: Ldate и Ltime, то есть в каждой записи журнала, которая записывается в журнальный файл, будут указаны текущая дата и время.

При выполнении customLog.go не будет генерироваться вывод на экран. Однако если дважды выполнить эту программу, получим следующее содержимое файла /tmp/mGo.log:

```
$ go run customLog.go
$ cat /tmp/mGo.log
customLog 2019/01/10 18:16:09 Hello there!
CustomLog 2019/01/10 18:16:09 Another log entry!
$ go run customLog.go
$ cat /tmp/mGo.log
customLog 2019/01/10 18:16:09 Hello there!
CustomLog 2019/01/10 18:16:09 Another log entry!
CustomLog 2019/01/10 18:16:17 Hello there!
CustomLog 2019/01/10 18:16:17 Another log entry!
```

Вывод номеров строк в записях журнала

В этом разделе вы узнаете, как указать в журнальном файле номер строки исходного файла, который выполнил инструкцию, сделавшую запись в журнал. Для этого рассмотрим Go-программу из файла `customLogLineNumber.go`, разделив ее на две части. Первая часть выглядит так:

```
package main

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

var LOGFILE = "/tmp/mGo.log"

func main() {
    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()
```

Как видим, здесь пока нет особых отличий от кода `customLog.go`.

Остальной Go-код из файла `customLogLineNumber.go` выглядит так:

```
    iLog := log.New(f, "customLogLineNumber ", log.LstdFlags)
    iLog.SetFlags(log.LstdFlags | log.Lshortfile)
    iLog.Println("Hello there!")
    iLog.Println("Another log entry!")
}
```

Всю магию выполняет оператор `iLog.SetFlags(log.LstdFlags | log.Lshortfile)`, который, кроме `log.LstdFlags`, также активизирует флаг `log.Lshortfile`. Последний добавляет в строку записи журнала полное имя файла, а также номер строки Go-оператора, который создал эту запись.

Выполнение `customLogLineNumber.go` не выводит данные на экран. Но после двух запусков файла `customLogLineNumber.go` содержимое файла журнала `/tmp/mGo.log` будет примерно таким:

```
$ go run customLogLineNumber.go
$ cat /tmp/mGo.log
customLogLineNumber 2019/01/10 18:25:14 customLogLineNumber.go:26: Hello there!
CustomLogLineNumber 2019/01/10 18:25:14 customLogLineNumber.go:27: Another log entry!
CustomLogLineNumber 2019/01/10 18:25:23 customLogLineNumber.go:26: Hello there!
```

```
CustomLogLineNumber 2019/01/10 18:25:23 customLogLineNumber.go:27: Another log entry!
CustomLogLineNumber 2019/01/10 18:25:23 customLogLineNumber.go:26: Hello there!
CustomLogLineNumber 2019/01/10 18:25:23 customLogLineNumber.go:27: Another log entry!
```

Как видим, использование длинных имен для утилит командной строки затрудняет чтение журнальных файлов.

В главе 2 вы узнаете, как использовать ключевое слово defer для того, чтобы Go-функции генерировали более красивые сообщения журнала.

Обработка ошибок в Go

Ошибки и обработка ошибок — две очень важные темы для Go. Go так любит сообщения об ошибках, что имеет особый тип данных для ошибок — `error`. Это также означает, что вы можете легко создавать собственные сообщения об ошибках, если окажется, что то, что уже есть в Go, вам не подходит.

Скорее всего, при разработке Go-пакетов вам придется создавать и обрабатывать собственные сообщения об ошибках.

Обратите внимание, что само по себе ошибочное состояние — это одно, а решение о том, как реагировать на это состояние, — совсем другое. Проще говоря, не все ошибочные состояния одинаковы: для одних ошибочных состояний может потребоваться немедленная остановка программы, а в других ситуациях можно ограничиться выводом предупреждающего сообщения для пользователя и продолжить выполнение программы. Принимая решение о том, что делать с каждым значением типа `error`, которое может получить программа, разработчику следует руководствоваться здравым смыслом.

Ошибки в Go не похожи на исключения или ошибки в других языках программирования; это обычные объекты Go, которые возвращаются из функций или методов, подобно любым другим значениям.

Тип данных error

Существует множество сценариев, в которых возникает необходимость обработать новый случай ошибки при разработке собственных Go-приложений. Тип данных `error` для того и предназначен, чтобы помочь вам создавать собственные ошибки.

В этом разделе вы научитесь создавать переменные типа `error`. Как вы увидите, для создания переменной типа `error` необходимо вызвать функцию `New()` из стандартного Go-пакета `errors`.

Пример Go-кода для иллюстрации этого процесса представлен в файле newError.go. Разделим его на две части. Первая часть программы выглядит так:

```
package main

import (
    "errors"
    "fmt"
)

func returnError(a, b int) error {
    if a == b {
        err := errors.New("Error in returnError() function!")
        return err
    } else {
        return nil
    }
}
```

Здесь происходит много интересного. Прежде всего, впервые в этой книге встречается определение Go-функции, отличной от main(). Имя этой новой функции — returnError(). Кроме того, здесь показано, как работает функция errors.New(), которая принимает в качестве параметра значение типа string. Наконец, если функция должна вернуть переменную типа error, но ошибка не произошла, то возвращается nil.

Подробнее о типах Go-функций вы узнаете в главе 6.

Вторая часть newError.go выглядит так:

```
func main() {
    err := returnError(1, 2)
    if err == nil {
        fmt.Println("returnError() ended normally!")
    } else {
        fmt.Println(err)
    }

    err = returnError(10, 10)
    if err == nil {
        fmt.Println("returnError() ended normally!")
    } else {
        fmt.Println(err)
    }

    if err.Error() == "Error in returnError() function!" {
        fmt.Println("!!")
    }
}
```

Как видно из кода, большая часть времени уходит на проверки того, равна ли переменная типа `error` со значением `nil`, после чего можно действовать соответственно. Здесь также продемонстрировано использование функции `errors.Error()`, которая позволяет преобразовать переменную типа `error` в тип `string`. Эта функция позволяет сравнивать переменную типа `error` с переменной типа `string`.

На компьютерах с UNIX рекомендуется передавать сообщения об ошибках в службу журналирования, особенно если Go-программа является сервером или другим критически важным приложением. Однако примеры кода, представленные в этой книге, не всегда будут следовать данному принципу, чтобы избежать засорения ваших журнальных файлов лишними данными.

Выполнение `newError.go` приведет к следующим результатам:

```
$ go run newError.go
returnError() ended normally!
Error in returnError() function!
!!
```

Если вы попытаетесь сравнить переменную типа `error` с переменной типа `string` без предварительного преобразования переменной типа `error` в `string`, компилятор Go выдаст следующее сообщение об ошибке:

```
# command-line-arguments
./newError.go:33:9: invalid operation: err == "Error in returnError()
function!" (mismatched types error and string)
```

Обработка ошибок

Обработка ошибок является очень важной частью Go. Поскольку почти все функции Go возвращают сообщение об ошибке или `nil`, у Go есть способ сообщить, возникло ли состояние ошибки при выполнении функции. Скорее всего, вам вскоре надоест встречать следующий Go-код не только в этой книге, но и в любой другой Go-программе, найденной в Интернете:

```
if err != nil {
    fmt.Println(err)
    os.Exit(10)
}
```

Пожалуйста, не путайте обработку ошибок, сопровождаемую выводом сообщений, с выводом сообщения в стандартный поток вывода ошибок: это совершенно разные вещи. Первая из них связана с Go-кодом, который обрабатывает ошибки, тогда как вторая — с записью чего-либо в стандартный дескриптор файла ошибок.

Представленный пример кода выводит сгенерированное сообщение об ошибке на экран и выходит из программы посредством функции `os.Exit()`. Обратите внимание: чтобы выйти из программы, можно также поставить ключевое слово `return` внутри функции `main()`. Вообще, вызов `os.Exit()` из функции, если только это не `main()`, считается плохой практикой. Функции, отличные от `main()`, как правило, перед выходом возвращают сообщение об ошибке, которое обрабатывается вызывающей функцией.

Если вы хотите отправить сообщение об ошибке не на экран, а в службу журналирования, нужно изменить предыдущий Go-код следующим образом:

```
if err != nil {
    log.Println(err)
    os.Exit(10)
}
```

Наконец, есть и другой вариант этого кода, который используется, когда случается что-то очень плохое и вы хотите срочно завершить программу:

```
if err != nil {
    panic(err)
    os.Exit(10)
}
```

Panic — это встроенная Go-функция, которая останавливает выполнение программы и начинает паниковать! Если вам приходится слишком часто использовать `panic`, возможно, стоит пересмотреть вашу реализацию Go-кода. Лучше избегать ситуаций паники, по возможности заменяя ее обработкой ошибок.

Как вы узнаете из следующей главы, в Go есть функция `recover`, которая выручит вас в некоторых неприятных ситуациях. Из главы 2 вы узнаете больше о возможностях этого дуэта функций — `panic` и `recover`.

А теперь пора познакомиться с Go-программой, которая не только обрабатывает сообщения об ошибках, генерируемые стандартными Go-функциями, но и создает собственное сообщение об ошибке. Эта программа называется `errors.go`. Разделим ее на пять частей. Как вы увидите, утилита `errors.go` пытается улучшить функциональность программы `cla.go`, о которой уже говорилось в данной главе. Теперь мы добавим проверку того, соответствуют ли ее аргументы командной строки формату чисел с плавающей точкой.

Первая часть программы выглядит так:

```
package main

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

Эта часть `error.go` содержит ожидаемые операторы `import`.

Вторая часть `error.go` содержит следующий Go-код:

```
func main() {
    if len(os.Args) == 1 {
        fmt.Println("Please give one or more floats.")
        os.Exit(1)
    }

    arguments := os.Args
    var err error = errors.New("An error")
    k := 1
    var n float64
```

Здесь создается новая переменная типа `error` с именем `err`, чтобы инициализировать ее собственным значением.

Третья часть программы выглядит так:

```
    for err != nil {
        if k >= len(arguments) {
            fmt.Println("None of the arguments is a float!")
            return
        }
        n, err = strconv.ParseFloat(arguments[k], 64)
        k++
    }

    min, max := n, n
```

Это самая сложная часть программы: если первый аргумент командной строки не является корректным числом с плавающей точкой, нужно проверить следующий аргумент и продолжать проверку до тех пор, пока не встретится аргумент командной строки подходящего формата. Если ни один из аргументов командной строки не соответствует формату числа с плавающей точкой, `error.go` прекращает работу и выводит на экран сообщение об ошибке. Вся проверка выполняется путем исследования значения типа `error`, которое возвращается функцией `strconv.ParseFloat()`. Весь этот код предназначен лишь для точной инициализации переменных `min` и `max`.

Четвертая часть программы содержит следующий Go-код:

```
    for i := 2; i < len(arguments); i++ {
        n, err := strconv.ParseFloat(arguments[i], 64)
        if err == nil {
            if n < min {
                min = n
            }
            if n > max {
                max = n
            }
        }
    }
```

Здесь мы просто обрабатываем все аргументы командной строки правильного формата, чтобы найти среди них минимальное и максимальное значения с плавающей точкой.

Наконец, последняя часть кода программы выводит на экран текущие значения переменных `min` и `max`:

```
    fmt.Println("Min:", min)
    fmt.Println("Max:", max)
}
```

Как видим из Go-кода, представленного в файле `errors.go`, большая часть кода посвящена обработке ошибок, а не выполнению того, для чего, собственно, предназначена программа. К сожалению, это относится к большинству современных программ, разработанных на Go, а также на большинстве других языков программирования.

Если выполнить `error.go`, то получим следующий вывод:

```
$ go run errors.go a b c
None of the arguments is a float!
$ go run errors.go b c 1 2 3 c -1 100 -200 a
Min: -200
Max: 100
```

Использование Docker

В последнем разделе этой главы вы узнаете, как можно использовать образ Docker для компиляции и выполнения Go-кода внутри образа Docker.

Возможно, вы знаете, что в Docker все начинается с образа Docker; вы можете создать собственный образ Docker с нуля или начать с уже существующего. В этом разделе мы загрузим базовый образ Docker с Docker Hub и продолжим сборку Go-программы `Hello World!` внутри этого образа.

Содержимое `Dockerfile` будет использоваться следующим образом:

```
FROM golang:alpine

RUN mkdir /files
COPY hw.go /files
WORKDIR /files
RUN go build -o /files/hw hw.go
ENTRYPOINT ["/files/hw"]
```

Здесь в первой строке определяется образ Docker, который будет использоваться. Остальные три команды создают в образе Docker новый каталог, копируют в образ Docker файл (`hw.go`) из текущего пользовательского каталога и соответственно изменяют текущий рабочий каталог образа Docker. Последние две команды соз-

дают двоичный исполняемый файл из исходного Go-файла и задают путь к этому двоичному файлу, который будет выполняться при запуске образа Docker.

Итак, как использовать этот `Dockerfile`? Если файл с именем `hw.go` существует и находится в текущем рабочем каталоге, можно создать новый образ Docker следующим образом:

```
$ docker build -t go_hw:v1 .
Sending build context to Docker daemon 2.237MB
Step 1/6 : FROM golang:alpine
alpine: Pulling from library/golang
cd784148e348: Pull complete
7e273b0dfc44: Pull complete
952c3806fd1a: Pull complete
ee1f873f86f9: Pull complete
7172cd197d12: Pull complete
Digest:
sha256:198cb8c94b9ee6941ce6d58f29aadb855f64600918ce602cdeacb018ad77d647
Status: Downloaded newer image for golang:alpine
---> f56365ec0638
Step 2/6 : RUN mkdir /files
---> Running in 18fa7784d82c
Removing intermediate container 18fa7784d82c
---> 9360e95d7cb4
Step 3/6 : COPY hw.go /files
---> 680517bc4aa3
Step 4/6 : WORKDIR /files
---> Running in f3f678fcc38d
Removing intermediate container f3f678fcc38d
---> 640117aea82f
Step 5/6 : RUN go build -o /files/hw hw.go
---> Running in 271cae1fa7f9
Removing intermediate container 271cae1fa7f9
---> dc7852b6aeeb
Step 6/6 : ENTRYPOINT ["/files/hw"]
---> Running in cdadf286f025
Removing intermediate container cdadf286f025
---> 9bec016712c4
Successfully built 9bec016712c4
Successfully tagged go_hw:v1
```

Имя нового образа Docker — `go_hw:v1`.

Если на вашем компьютере уже есть образ Docker `golang:alpine`, то результат этой команды будет таким:

```
$ docker build -t go_hw:v1 .
Sending build context to Docker daemon 2.237MB
Step 1/6 : FROM golang:alpine
---> f56365ec0638
Step 2/6 : RUN mkdir /files
```

```
---> Running in 982e6883bb13
Removing intermediate container 982e6883bb13
---> 0632577d852c
Step 3/6 : COPY hw.go /files
---> 68a0feb2e7dc
Step 4/6 : WORKDIR /files
---> Running in d7d4d0c846c2
Removing intermediate container d7d4d0c846c2
---> 6597a7cb3882
Step 5/6 : RUN go build -o /files/hw hw.go
---> Running in 324400d532e0
Removing intermediate container 324400d532e0
---> 5496dd3d09d1
Step 6/6 : ENTRYPOINT ["/files/hw"]
---> Running in bbd24840d6d4
    Removing intermediate container bbd24840d6d4
---> 5a0d2473aa96
    Successfully built 5a0d2473aa96
    Successfully tagged go_hw:v1
```

Чтобы убедиться, что образ Docker `go_hw:v1` действительно существует на вашем компьютере, нужно сделать следующее:

```
$ docker images
REPOSITORY   TAG      IMAGE ID       CREATED              SIZE
go_hw        v1       9bec016712c4   About a minute ago   312MB
golang       alpine   f56365ec0638   11 days ago          310MB
```

В файле `hw.go` содержится следующий код:

```
package main

import (
    "fmt"
)

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

Образ Docker, который находится на локальном компьютере, можно использовать следующим образом:

```
$ docker run go_hw:v1
Hello World!
```

Существуют и другие, более сложные способы выполнения образа Docker, но для столь наивного образа Docker это самый простой способ его использования.

При желании можно передать (push) образ Docker в реестр Docker, размещенный в Интернете, чтобы впоследствии иметь возможность извлекать его оттуда (pull).

Таким местом может быть Docker Hub — при условии, что у вас есть учетная запись Docker Hub, создать которую легко, и это бесплатно. Итак, для того чтобы отправить созданный нами образ в Docker Hub, нужно создать в Docker Hub учетную запись, затем выполнить на вашей UNIX-машине следующие команды:

```
$ docker login
Authenticating with existing credentials...
Login Succeeded
$ docker tag go_hw:v1 "mactsouk/go_hw:v1"
$ docker push "mactsouk/go_hw:v1"
The push refers to repository [docker.io/mactsouk/go_hw]
bdb6946938e3: Pushed
99e21c42e35d: Pushed
0257968d27b2: Pushed
e121936484eb: Pushed
61b145086eb8: Pushed
789935042c6f: Pushed
b14874cfef59: Pushed
7bff100f35cb: Pushed
v1: digest:
sha256:c179d5d48a51b74b0883e582d53bf861c6884743eb51d9b77855949b5d91dd
e1 size: 1988
```

Первая команда необходима для входа в Docker Hub, ее достаточно выполнить один раз. Команда `docker tag` нужна для указания имени, под которым локальный образ будет храниться в Docker Hub. Эту команду нужно выполнить перед командой `push docker`. Последняя команда отправляет желаемый образ Docker в Docker Hub и генерирует расширенное сообщение о результатах. Если вы сделаете свой образ Docker общедоступным, то любой желающий сможет получить его из Docker Hub и использовать.

Есть несколько способов удалить один или несколько образов Docker с локальной UNIX-машины. Один из них — воспользоваться `IMAGE ID` образа Docker:

```
$ docker rmi 5a0d2473aa96 f56365ec0638
Untagged: go_hw:v1
Deleted:
sha256:5a0d2473aa96bcdafbef92751a0e1c1bf146848966c8c971f462eb1eb242d2a6
Deleted:
sha256:5496dd3d09d13c63bf7a9ac52b90bb812690cdfd33cfc3340509f9bfe6215c48
Deleted:
sha256:598c4e474b123eccb84f41620d2568665b88a8f176a21342030917576b9d82a8
Deleted:
sha256:6597a7cb3882b73855d12111787bd956a9ec3abb11d9915d32f2bba4d0e92ec6
Deleted:
sha256:68a0feb2e7dc5a139eaa7ca04e54c20e34b7d06df30bcd4934ad6511361f2cb8
Deleted:
sha256:c04452ea9f45d85a999bdc54b55ca75b6b196320c021d777ec1f766d115aa514
Deleted:
sha256:0632577d852c4f9b66c0eff2481ba06c49437e447761d655073eb034fa0ac333
```

```
Deleted:
sha256:52efd0fa2950c8f3c3e2e44fbc4eb076c92c0f85fff46a07e060f5974c1007a9
Untagged: golang:alpine
Untagged:
golang@sha256:198cb8c94b9ee6941ce6d58f29aadb855f64600918ce602cdeacb018ad77d647
Deleted:
sha256:f56365ec0638b16b752af4bf17e6098f2fda027f8a71886d6849342266cc3ab7
Deleted:
sha256:d6a4b196ed79e7ff124b547431f77e92dce9650037e76da294b3b3aded709bdd
Deleted:
sha256:f509ec77b9b2390c745afd76cd8dd86977c86e9ff377d5663b42b664357c3522
Deleted:
sha256:1ee98fa99e925362ef980e651c5a685ad04cef41dd80df9be59f158cf9e52951
Deleted:
sha256:78c8e55f8cb4c661582af874153f88c2587a034ee32d21cb57ac1fef51c6109e
Deleted:
sha256:7bff100f35cb359a368537bb07829b055fe8e0b1cb01085a3a628ae9c187c7b8
```

Работа с Docker — огромная и действительно важная тема, к которой мы вернемся еще не раз.

Упражнения и ссылки

- Посетите сайт Go: https://golang.org/.
- Посетите сайт Docker: https://www.docker.com/.
- Посетите сайт Docker Hub: https://hub.docker.com/.
- Посетите Go 2 Draft Designs: https://blog.golang.org/go2draft.
- Посетите сайт документации по Go: https://golang.org/doc/.
- Почитайте документацию пакета `log`: https://golang.org/pkg/log/.
- Почитайте документацию пакета `log/syslog`: https://golang.org/pkg/log/syslog/.
- Почитайте документацию пакета `os`: https://golang.org/pkg/os/.
- Посетите https://golang.org/cmd/gofmt/ — страницу документации инструмента `gofmt`, который используется для форматирования Go-кода.
- Напишите Go-программу, которая вычисляет сумму всех аргументов командной строки, которые являются действительными числами.
- Напишите Go-программу, вычисляющую среднее значение всех чисел с плавающей запятой, переданных программе в качестве аргументов командной строки.
- Напишите Go-программу, которая считывает целые числа до тех пор, пока не встретит во входных данных слово `END`.

- Можете ли вы изменить программу `customLog.go` так, чтобы данные журнала записывались одновременно в два файла журнала? Возможно, для этого вам придется обратиться к главе 8.
- Если вы работаете на компьютере Mac, почитайте описание редактора TextMate по адресу http://macromates.com/, а также редактора BBEdit по адресу https://www.barebones.com/products/bbedit/.
- Посетите страницу документации пакета `fmt`: https://golang.org/pkg/fmt/, чтобы больше узнать о «глаголах» форматирования и доступных функциях.
- Приглашаю вас посетить страницу https://blog.Golang.Org/why-generics, чтобы больше узнать о Go и Generics.

Резюме

В этой главе приведена общая информация о языке Go, а также раскрыты многие интересные темы: компиляция Go-кода, работа Go со стандартными потоками ввода, вывода и ошибок, обработка аргументов командной строки, вывод данных на экран и использование службы журналирования UNIX-систем, а также обработка ошибок. Все эти вопросы лежат в основе изучения Go.

Следующая глава посвящена деталям внутренней реализации Go, в том числе сборке мусора, компилятору Go, вызову C-кода из Go, ключевому слову `defer`, Go-ассемблеру и WebAssembly, а также функциям `panic` и `recover`.

2 Go изнутри

Все функции Go, которые вы изучили в предыдущей главе, очень удобны, и вы будете постоянно их использовать. Однако нет ничего более полезного, чем способность видеть и понимать, что при этом происходит внутри Go.

В этой главе вы познакомитесь со сборщиком мусора Go и узнаете, как он работает. Кроме того, вы научитесь вызывать код на C из программ на Go, что иногда незаменимо. Однако вам не придется обращаться к этой функции слишком часто, поскольку Go — язык программирования с большими возможностями.

Кроме того, вы узнаете, как вызывать код на Go из программ на C, научитесь использовать функции `panic()` и `recovery()` и ключевое слово `defer`.

В этой главе рассмотрим следующие темы:

- компилятор Go;
- как работает сборка мусора в Go;
- как проверить работу сборщика мусора;
- вызов кода на C из программы на Go;
- вызов кода на Go из программы на C;
- функции `panic()` и `recovery()`;
- пакет `unsafe`;
- удобное, но коварное ключевое слово `defer`;
- Linux-утилита `strace(1)`;
- утилита `dtrace(1)` из систем FreeBSD, включая macOS Mojave;
- где найти информацию о вашей среде Go;
- построение деревьев узлов (node trees) с помощью Go;
- генерация кода WebAssembly из Go;
- Go-ассемблер.

Компилятор Go

Компилятор Go запускается с помощью инструмента go. Этот инструмент делает гораздо больше, чем просто создает исполняемые файлы.

Используемый в этом разделе файл unsafe.go не содержит никакого особенного кода — представленные здесь команды будут работать с любым корректным исходным файлом Go.

Для того чтобы скомпилировать исходный файл Go, нужно воспользоваться командой[1] `go tool compile`. В результате вы получите *объектный файл* — файл с расширением `.o`. Пример компиляции показан ниже, в виде следующих команд, выполненных на компьютере с macOS Mojave:

```
$ go tool compile unsafe.go
$ ls -l unsafe.o
-rw-r--r--  1 mtsouk  staff  6926 Jan 22 21:39 unsafe.o
$ file unsafe.o
unsafe.o: current ar archive
```

Объектный файл — это файл, в котором содержится *объектный код*, то есть машинный код в переносимом формате, который в большинстве случаев не может быть непосредственно выполнен. Главным преимуществом переносимого формата является то, что на этапе компоновки ему требуется минимум памяти.

Если при выполнении компиляции `go tool` использовать флаг командной строки `-pack`, то вместо объектного файла получим *архивный файл*:

```
$ go tool compile -pack unsafe.go
$ ls -l unsafe.a
-rw-r--r--  1 mtsouk  staff  6926 Jan 22 21:40 unsafe.a
$ file unsafe.a
unsafe.a: current ar archive
```

Архивный файл — это двоичный файл, внутри которого содержится один или несколько других файлов. Как правило, архивные файлы применяются для объединения нескольких файлов в один. В Go используются архивные файлы формата `ar`.

[1] Если вы используете компьютер с Windows, то go tool compile может выполниться с ошибкой can’t find import. В этом случае при вызове команды нужно добавить флаг -I, который будет указывать на место расположения библиотек в GOPATH, например:

$ go tool compile -I C:golang\pkg\windows_amd64 unsafe.go

Для того чтобы просмотреть содержимое архивного файла `.a`, нужно воспользоваться следующей командой:

```
$ ar t unsafe.a
__.PKGDEF
_go_.o
```

Обратите внимание еще на один полезный флаг командной строки для команды `go tool compile`: `-race`. Он позволяет распознавать *состояние гонки*. Подробнее о том, что такое состояние гонки и почему его следует избегать, вы прочитаете в главе 10.

В конце этой главы вы найдете дополнительные примеры применения команды `go tool compile`, когда речь пойдет о языке ассемблера и деревьях узлов (node trees). Сейчас для проверки попробуйте выполнить следующую команду:

```
$ go tool compile -S unsafe.go
```

Эта команда генерирует много выходных данных, которые могут показаться непонятными. Это означает, что Go довольно хорошо скрывает излишние сложности, если только специально не попросить его показать их!

Сборка мусора

Сборка мусора (Garbage Collection, GC) — это процесс освобождения места в памяти, которое больше не используется. Другими словами, сборщик мусора определяет объекты, находящиеся вне области видимости, на которые нельзя больше ссылаться (недостижимые объекты), и освобождает занимаемую ими память. Этот процесс выполняется конкурентно, не до и не после, а во время работы Go-программы. Документация реализации сборщика мусора в Go гласит следующее:

> *«GC выполняется конкурентно (concurrent), одновременно с потоками мутатора (mutator), в точном соответствии с типом (этот принцип также известен как чувствительность к типу), допускается параллельное выполнение нескольких потоков GC. Это конкурентная пометка и очистка (mark-sweep), при которой используется барьер записи (write barrier). При этом процессе ничего не генерируется и не сжимается. Освобождение памяти выполняется на основе размера, выделенного для каждой программы P, чтобы в общем случае минимизировать фрагментацию и избежать блокировок».*

В этом фрагменте документации много терминов, которые станут понятными позже. Но сначала разберемся, как просмотреть некоторые параметры сборки мусора.

К счастью, стандартная библиотека Go включает в себя функции, которые позволяют узнать, какая именно скрытая деятельность происходит внутри сборщика мусора. Необходимый для этого код хранится в файле `gColl.go` и состоит из трех частей.

Первая часть кода `gColl.go` выглядит следующим образом:

```
package main

import (
    "fmt"
    "runtime"
    "time"
)

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)
    fmt.Println("-----")
}
```

Обратите внимание: каждый раз, когда мы хотим получить свежую статистику о сборке мусора, нам нужно заново вызывать функцию `runtime.ReadMemStats()`. Функция `printStats()` нужна для того, чтобы не писать многократно один и тот же код Go.

Вторая часть программы выглядит так:

```
func main() {
    var mem runtime.MemStats
    printStats(mem)

    for i := 0; i < 10; i++ {
        s := make([]byte, 50000000)
        if s == nil {
            fmt.Println("Operation failed!")
        }
    }
    printStats(mem)
```

Цикл `for` создает много больших *срезов* Go, чтобы под них выделялись большие объемы памяти и запускался сборщик мусора.

Последняя часть `gColl.go` содержит следующий код Go, который вызывает дополнительное выделение памяти для срезов Go:

```
    for i := 0; i < 10; i++ {
        s := make([]byte, 100000000)
        if s == nil {
            fmt.Println("Operation failed!")
        }
        time.Sleep(5 * time.Second)
    }
    printStats(mem)
}
```

Выполнение `gColl.go` на macOS Mojave дает следующий результат:

```
$ go run gColl.go
mem.Alloc: 66024
mem.TotalAlloc: 66024
mem.HeapAlloc: 66024
mem.NumGC: 0
-----
mem.Alloc: 50078496
mem.TotalAlloc: 500117056
mem.HeapAlloc: 50078496
mem.NumGC: 10
-----
mem.Alloc: 76712
mem.TotalAlloc: 1500199904
mem.HeapAlloc: 76712
mem.NumGC: 20
-----
```

Вряд ли вы будете постоянно проверять работу сборщика мусора Go, однако возможность наблюдать за тем, как он работает в медленном приложении, в долгосрочной перспективе способна сэкономить уйму времени. Поверьте, вы не пожалеете о том времени, которое потратите на изучение сборки мусора вообще и сборщика мусора Go в частности.

В следующей команде показан прием, позволяющий получить еще более подробный отчет о том, как работает сборщик мусора Go:

```
$ GODEBUG=gctrace=1 go run gColl.go
```

Если перед любой командой `go run` поставить `GODEBUG=gctrace=1`, то Go выводит аналитические данные о работе сборщика мусора. Данные представлены в такой форме:

```
gc 4 @0.025s 0%: 0.002+0.065+0.018 ms clock,
    0.021+0.040/0.057/0.003+0.14 ms cpu, 47->47->0 MB, 48 MB goal, 8 P
gc 17 @30.103s 0%: 0.004+0.080+0.019 ms clock,
    0.033+0/0.076/0.071+0.15 ms cpu, 95->95->0 MB, 96 MB goal, 8 P
```

Здесь приводится подробная информация о размерах кучи в процессе сборки мусора. Возьмем для примера тройку значений `47->47->0 MB`. Первое число — размер кучи перед запуском сборщика мусора; второе значение — размер кучи, когда сборщик завершает работу; последнее значение — актуальный размер кучи.

Трехцветный алгоритм

В основе работы сборщика мусора Go лежит трехцветный алгоритм.

Обратите внимание на то, что трехцветный алгоритм неуникален для Go; он может использоваться и в других языках программирования.

Официальное название алгоритма, используемого в Go, — *трехцветный алгоритм пометки и очистки*. Он может работать конкурентно, одновременно с программой, и использует *барьер записи*. Это означает, что при запуске Go-программы планировщик Go составляет график работы собственно приложения и сборщика мусора — как если бы планировщик Go имел дело с обычным приложением, состоящим из нескольких *горутин*! Подробнее о горутинах и планировщике Go вы узнаете из главы 9.

Основную идею трехцветного алгоритма сформулировали Эдсгер В. Дайкстра (Edsger W. Dijkstra), Лесли Лэмпорт (Leslie Lamport), Эй. Дж. Мартин (A. J. Martin), К. С. Шолтен (C. S. Scholten) и И. Ф. М. Стеффенс (E. F. M. Steffens). Этот алгоритм впервые описан в статье *On-the-Fly Garbage Collection: An Exercise in Cooperation*.

Главный принцип алгоритма трехцветной пометки и очистки состоит в разделении объектов, находящихся в куче, на три набора, в соответствии с цветом, который назначается им алгоритмом. Пора обсудить значение каждого из этих цветов. Объекты *черного цвета* гарантированно не имеют указателей ни на один объект *белого цвета*. Однако объект белого цвета может иметь указатель на объект черного цвета, поскольку это не влияет на работу сборщика мусора. Объекты *серого цвета* могут иметь указатели на некоторые объекты белого цвета. Именно объекты белого цвета являются претендентами на удаление.

Обратите внимание, что ни один объект не может перейти непосредственно из черного множества в белое — именно это обеспечивает работу алгоритма и позволяет освобождать память из-под объектов белого множества. Кроме того, ни один объект из черного множества не может напрямую указывать на объект из белого множества.

Итак, когда начинается сборка мусора, все объекты становятся белыми. Сборщик мусора перебирает все корневые объекты и окрашивает их в серый цвет. *Корневые объекты* — это объекты, к которым приложение может обращаться напрямую, включая глобальные переменные и другие элементы, находящиеся в стеке. Большинство этих объектов зависят от Go-кода конкретной программы.

После этого сборщик мусора выбирает серый объект, помечает его черным и проверяет, есть ли у него указатели на другие объекты из белого множества. Это означает, что при проверке серого объекта на предмет указателей на другие объекты он окрашивается в черный цвет. Если проверка обнаружит, что у данного объекта есть один или несколько указателей на белые объекты, алгоритм поменяет цвет этих белых объектов на серый. Процесс продолжается до тех пор, пока не будут перебраны все объекты серого множества. Затем объекты белого множества считаются недостижимыми, и занимаемая ими память может использоваться повторно. Таким образом, считается, что в этот момент элементы белого множества попали в корзину.

Обратите внимание: если в какой-то момент сборки мусора объект из серого множества станет недостижимым, он будет обработан не в этом, а в следующем цикле сборки мусора! Это не оптимальный, но и не такой уж плохой вариант.

Приложение, работающее во время выполнения сборки мусора, называется *мутатором*. Мутатор запускает небольшую функцию, называемую *барьером записи*. Эта функция выполняется всякий раз, когда изменяется указатель в куче. Если указатель объекта в куче изменился, это означает, что данный объект теперь достижим. Барьер записи помечает этот объект в серый цвет и помещает в серое множество.

Мутатор отвечает за то, чтобы ни один элемент из черного множества не имел указателя на элемент из белого множества. Это достигается с помощью функции барьера записи. Невыполнение этого условия разрушило бы процесс сборки мусора и, скорее всего, привело бы к аварийному завершению работы программы.

В итоге кучу можно представить как граф, состоящий из связанных объектов. Такой граф показан на рис. 2.1, где также продемонстрирован один из этапов сборки мусора.

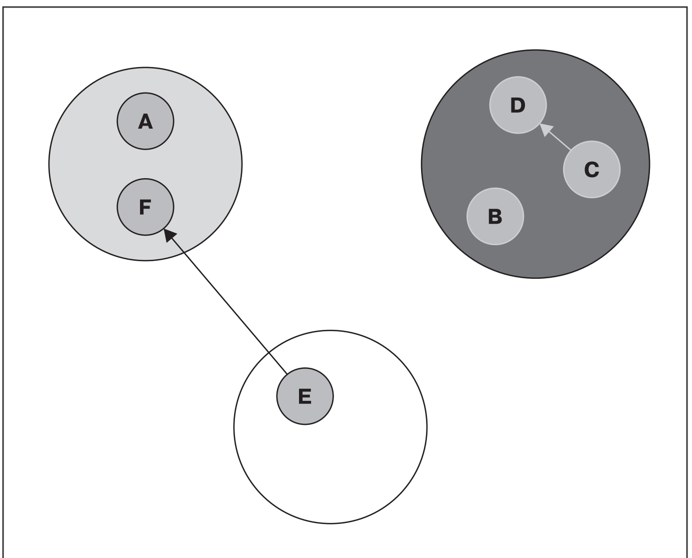


Рис. 2.1. В Go при сборке мусора в корзину область кучи программы можно представить в виде графа

Итак, у нас есть три цвета: черный, белый и серый. Когда алгоритм начинает работу, все объекты окрашены в белый цвет. По мере работы алгоритма белые объекты перемещаются в одно из двух остальных множеств: серое или черное.

Те объекты, которые останутся в белом множестве, в какой-то момент будут удалены.

На представленном графе видно, что, хотя объект E, который относится к белому множеству, может получить доступ к объекту F, он сам недостижим из какого-либо другого объекта, потому что никакой другой объект не указывает на объект E. Это делает объект E идеальным претендентом для сборки мусора! Кроме того, объекты A, B и C являются корневыми объектами и всегда достижимы; следовательно, они не могут быть собраны как мусор.

Угадаете ли вы, что будет дальше на этом графе? Несложно понять, что алгоритм должен будет обработать оставшиеся элементы из серого множества и, следовательно, объекты A и F перейдут в черное множество: объект A — потому что является корневым элементом, а F — потому что не указывает ни на какой другой объект, пока так же будет находиться в сером множестве.

После того как объект A будет удален сборщиком мусора, объект F станет недостижимым и попадет в мусор в следующем цикле сборщика мусора, поскольку недостижимый объект не может волшебным образом стать достижимым на следующей итерации цикла сборки мусора.

Сборщик мусора Go может также применяться к таким переменным, как *каналы*. Когда сборщик мусора обнаруживает, что канал недостижим, то есть недостижима переменная канала, сборщик мусора освобождает ресурсы этой переменной, даже если сам канал не закрыт. Подробнее о каналах вы узнаете в главе 9.

Go позволяет запускать сборку мусора вручную. Для этого нужно вставить в код Go оператор `runtime.GC()`. Однако помните, что `runtime.GC()` заблокирует вызывающую функцию и может даже заблокировать всю программу, особенно если это очень большая Go-программа с огромным количеством объектов. Так происходит главным образом потому, что невозможно выполнять сборку мусора, когда остальное окружение быстро меняется, поскольку тогда сборщик мусора не сможет четко идентифицировать элементы белого, черного и серого множеств. Такой статус сборки мусора также называется *начальной точкой сборки мусора*.

По адресу https://github.com/golang/go/blob/master/src/runtime/mgc.go вы найдете полный исходный код сборщика мусора Go, который стоит изучить, если вы хотите получить больше информации об операции сборки мусора. Вы даже можете внести изменения в этот код, если хватит смелости!

Обратите внимание: команда Go постоянно совершенствует сборщик мусора. Создатели Go пытаются ускорить сборку мусора, уменьшая количество сканирований, необходимых для трех цветовых множеств. Однако, несмотря на все оптимизации, общая идея алгоритма остается прежней.

Подробнее о работе сборщика мусора Go

В этом разделе вы более подробно узнаете о сборщике мусора Go и его деятельности. Основной проблемой сборщика мусора Go является низкая латентность, что означает, в сущности, короткие паузы, чтобы обеспечить его работу в реальном времени. При этом программа создает новые объекты и постоянно манипулирует существующими объектами посредством указателей. Такой процесс может завершиться созданием объектов, доступ к которым больше невозможен, потому что не существует указателей, ссылающихся на эти объекты. Затем эти объекты становятся мусором и ждут, когда сборщик мусора очистит их и освободит занимаемую ими память. После этого освободившаяся память снова готова к использованию.

Алгоритм пометки и очистки является самым простым из используемых алгоритмов. Этот алгоритм останавливает выполнение программы *(сборщик мусора stop-the-world)*, чтобы перебрать все доступные объекты из кучи и **пометить** их. После этого алгоритм **удаляет** все недостижимые объекты. На этапе пометки каждый объект отмечается как белый, серый или черный. Дочерним объектам серых объектов назначается серый цвет, а их родительский серый объект теперь окрашивается в черный цвет. Когда все серые объекты будут перебраны, начинается этап очистки. Эта методика работает потому, что не существует указателей от черных объектов на белые, что является необходимым условием алгоритма.

Несмотря на свою простоту, алгоритм пометки и очистки приостанавливает выполнение работающей программы, следовательно, увеличивает задержку реальных процессов. Go пытается уменьшить эту задержку, запуская сборщик мусора как параллельный процесс и используя трехцветный алгоритм, описанный в предыдущем разделе. Однако другие процессы могут перемещать указатели или создавать новые объекты одновременно с работой сборщика мусора. Для сборщика мусора это может сильно усложнить жизнь. В результате трехцветный алгоритм сможет работать конкурентно при условии сохранения главного требования алгоритма отслеживания и очистки: ни один объект из черного множества не должен указывать на объект белого множества.

Решением этой проблемы является корректная обработка всех ситуаций, которые могут вызвать проблему для алгоритма сборки мусора. В частности, все новые объекты должны заноситься в серое множество, поскольку таким образом сохранится необходимое условие алгоритма пометки и очистки. В дополнение, при перемещении указателя объект, на который ссылается этот указатель, должен помечаться серым цветом. Серое множество играет роль барьера между белым и черным множествами. Наконец, при каждом перемещении указателя автоматически должен выполняться некий Go-код, которым является ранее упомянутый барьер записи, изменяющий цвет объекта. Задержка, вызванная выполнением кода барьера записи, — это та цена, которую приходится платить за возможность конкурентной сборки мусора.

Обратите внимание, что в языке программирования Java есть много сборщиков мусора, которые легко настраиваются с помощью ряда параметров. Один из них — *G1*. Его рекомендуется применять для приложений с низкой задержкой.

Важно помнить, что сборщик мусора Go работает в реальном времени, конкурентно с другими горутинами и оптимизируется только для обеспечения низкой латентности своего запуска.

В главе 11 вы узнаете, как отобразить на графике тайминг работы Go-программы, включая информацию о запусках сборщика мусора в Go.

Хеш-таблицы, срезы и сборщик мусора Go

В этом разделе на нескольких примерах я продемонстрирую, почему следует быть осторожным при сборке мусора. Цель этого раздела — показать, что способ хранения указателей сильно влияет на производительность сборщика мусора, особенно когда мы имеем дело с очень большим количеством указателей.

В представленных примерах используются указатели, срезы (slice) и хеш-таблицы (map), которые являются собственными типами данных Go. Подробнее об указателях, срезах и хеш-таблицах в Go вы узнаете из главы 3.

Использование срезов

Рассмотрим пример использования *среза* для хранения большого количества структур. Каждая структура состоит из двух целочисленных значений. Go-код `sliceGC.go` выглядит следующим образом:

```
package main

import (
    "runtime"
)

type data struct {
    i, j int
}

func main() {
    var N = 40000000
    var structure []data
    for i := 0; i < N; i++ {
        value := int(i)
        structure = append(structure, data{value, value})
    }

    runtime.GC()
    _ = structure[0]
}
```

Последний оператор (`_ = structure[0]`) используется для предотвращения преждевременной очистки сборщиком мусора переменной `structure`, поскольку на нее нет указателей и она не задействована за пределами цикла `for`. Тот же прием будет применяться в следующих трех Go-программах. Кроме этой важной детали, цикл `for` здесь необходим для помещения всех значений в структуры, которые хранятся в срезе.

Использование хеш-таблиц с указателями

В этом я подразделе покажу, как использовать хеш-таблицу для хранения всех указателей в виде целых чисел. Программа называется `mapStar.go` и содержит следующий Go-код:

```
package main

import (
    "runtime"
)

func main() {
    var N = 40000000
    myMap := make(map[int]*int)
    for i := 0; i < N; i++ {
        value := int(i)
        myMap[value] = &value
    }
    runtime.GC()
    _ = myMap[0]
}
```

Хеш-таблица, в которой хранятся целочисленные указатели, носит имя `myMap`. Цикл `for` здесь используется для помещения целочисленных значений в хеш-таблицу.

Использование хеш-таблиц без указателей

Рассмотрим, как использовать хеш-таблицу, в которой хранятся простые значения без указателей. Go-код `mapNoStar.go` выглядит так:

```
package main

import (
    "runtime"
)

func main() {
    var N = 40000000
    myMap := make(map[int]int)
```

```
    for i := 0; i < N; i++ {
        value := int(i)
        myMap[value] = value
    }
    runtime.GC()
    _ = myMap[0]
}
```

Как и раньше, цикл `for` здесь используется для помещения в хеш-таблицу целочисленных значений.

Разделение хеш-таблицы

Программа, представленная в этом подразделе, выполняет преобразование одной хеш-таблицы в хеш-таблицу нескольких хеш-таблиц. Такой процесс называется *шардингом* (sharding). Программа этого подраздела хранится в файле `mapSplit.go`, и мы ее рассмотрим по частям. Первая часть `mapSplit.go` содержит следующий Go-код:

```
package main

import (
    "runtime"
)

func main() {
    var N = 40000000
    split := make([]map[int]int, 200)
```

В этой части программы определяется хэш хэшей.

Вторая часть выглядит так:

```
    for i := range split {
        split[i] = make(map[int]int)
    }
    for i := 0; i < N; i++ {
        value := int(i)
        split[i%200][value] = value
    }
    runtime.GC()
    _ = split[0][0]
}
```

На этот раз мы использовали два цикла `for`: один для создания хеша хешей, а второй — для хранения данных в этом хеше хешей.

Сравнение производительности описанных методик

Поскольку во всех четырех программах используются огромные структуры данных, они потребляют большие объемы памяти. Программы, которые занимают много

места в памяти, чаще запускают сборщик мусора Go. Сравним между собой производительность этих четырех реализаций с помощью команды `time(1)`.

В представленном результате важны не конкретные цифры, а разница во времени между четырьмя подходами. Результат сравнения выглядит так:

```
$ time go run sliceGC.go
real    1.50s
user    1.72s
sys     0.71s
$ time go run mapStar.go
real    13.62s
user    23.74s
sys      1.89s
$ time go run mapNoStar.go
real    11.41s
user    10.35s
sys     1.15s
$ time go run mapSplit.go
real    10.60s
user    10.01s
sys      0.74s
```

Оказывается, хеш-таблицы замедляют сборщик мусора Go, в то время как срезы работают с ним гораздо лучше. Следует отметить, что это не проблема хеш-таблиц, а результат работы сборщика мусора Go. Однако, если не использовать хеш-таблицы, в которых хранятся огромные объемы данных, в ваших программах эта проблема не будет столь очевидной.

Подробнее о тестировании в Go вы прочитаете в главе 11. Кроме того, в главе 3 вы познакомитесь с более профессиональным способом измерения времени, необходимого для выполнения команды или программы в Go.

Пока достаточно о сборщике мусора и его особенностях; тема следующего раздела — небезопасный код и стандартный Go-пакет `unsafe`.

Небезопасный код

Небезопасный код — это Go-код, который обходит безопасность типов и безопасность памяти Go. В большинстве случаев небезопасный код связан с указателями. Однако следует помнить, что использование небезопасного кода может быть опасным для ваших программ, поэтому, если вы не вполне уверены, что вам необходим небезопасный код в вашей программе, не используйте его!

Мы продемонстрируем использование небезопасного кода в программе `unsafe.go`, которую разделим на три части.

Первая часть `unsafe.go` выглядит так:

```
package main

import (
    "fmt"
    "unsafe"
)
```

Как видим, для использования небезопасного кода нужно импортировать стандартный пакет Go `unsafe`.

Вторая часть программы содержит следующий Go-код:

```
func main() {
    var value int64 = 5
    var p1 = &value
    var p2 = (*int32)(unsafe.Pointer(p1))
```

Обратите внимание на то, как здесь используется функция `unsafe.Pointer()`: она позволяет нам на свой страх и риск создать указатель `int32` с именем `p2`. Этот указатель ссылается на переменную `int64` с именем `value`, доступ к которой осуществляется с помощью указателя `p1`. Любой указатель Go можно преобразовать в `unsafe.Pointer`.

Указатель типа unsafe.Pointer позволяет переопределять систему типов Go. Это позволяет существенно повысить производительность, но может быть опасно, если использовать указатели неправильно или небрежно. Кроме того, так разработчики получают больший контроль над данными.

Последняя часть `unsafe.go` содержит следующий Go-код:

```
    fmt.Println("*p1: ", *p1)
    fmt.Println("*p2: ", *p2)
    *p1 = 5434123412312431212
    fmt.Println(value)
    fmt.Println("*p2: ", *p2)
    *p1 = 54341234
    fmt.Println(value)
    fmt.Println("*p2: ", *p2)
}
```

Вы можете разыменовать указатель и получать, использовать или изменять его значение с помощью символа звездочки (*).

Если запустить `unsafe.go`, получим следующие результаты:

```
$ go run unsafe.go
*p1:  5
*p2:  5
5434123412312431212
*p2:  -930866580
54341234
*p2:  54341234
```

О чем говорит нам этот вывод? О том, что указатель на 32-разрядное целое число не может предоставить доступ к 64-разрядному целому числу.

Как будет показано в следующем разделе, функции пакета `unsafe` позволяют делать с памятью еще много интересных вещей.

Пакет unsafe

Теперь, когда вы увидели, как работает пакет `unsafe`, самое время поговорить о том, почему этот пакет такой особенный. Если вы посмотрите на исходный код пакета `unsafe`, то несколько удивитесь. В системе macOS Mojave с версией Go 1.11.4, установленной с сайта Homebrew (https://brew.sh/), исходный код пакета `unsafe` расположен по адресу `/usr/local/Cellar/go/1.11.4/libexec/src/unsafe/unsafe.go`, и его содержимое без учета комментариев выглядит так:

```
$ cd /usr/local/Cellar/go/1.11.4/libexec/src/unsafe/
$ grep -v '^//' unsafe.go | grep -v '^$'
package unsafe
type ArbitraryType int
type Pointer *ArbitraryType
func Sizeof(x ArbitraryType) uintptr
func Offsetof(x ArbitraryType) uintptr
func Alignof(x ArbitraryType) uintptr
```

Где же находится остальной код Go пакета `unsafe`? Ответ на этот вопрос сравнительно прост: компилятор Go генерирует пакет `unsafe`, когда вы импортируете его в программу.

Многие низкоуровневые пакеты, такие как runtime, syscall и os, интенсивно используют пакет unsafe.

Еще один пример использования пакета unsafe

В этом подразделе вы ближе познакомитесь с пакетом `unsafe` и его возможностями на примере небольшой Go-программы с именем `moreUnsafe.go`. Мы разделим ее на три части. Программа `moreUnsafe.go` делает следующее: получает доступ ко всем элементам массива с помощью указателей.

Первая часть программы выглядит так:

```
package main

import (
    "fmt"
    "unsafe"
)
```

Вторая часть moreUnsafe.go содержит следующий Go-код:

```
func main() {
    array := [...]int{0, 1, -2, 3, 4}
    pointer := &array[0]
    fmt.Print(*pointer, " ")
    memoryAddress := uintptr(unsafe.Pointer(pointer)) + unsafe.Sizeof(array[0])

    for i := 0; i < len(array)-1; i++ {
        pointer = (*int)(unsafe.Pointer(memoryAddress))
        fmt.Print(*pointer, " ")
        memoryAddress = uintptr(unsafe.Pointer(pointer)) + unsafe.Sizeof(array[0])
    }
```

Сначала переменная `pointer` указывает на адрес в памяти `array[0]` — первого элемента целочисленного массива. Затем переменная `pointer`, которая указывала на целочисленное значение, преобразуется в `unsafe.Pointer()`, потом — в `uintptr`. Результат сохраняется в `memoryAddress`.

Значение `unsafe.Sizeof(array[0])` переводит нас к следующему элементу массива, а именно — сколько памяти занимает каждый элемент массива. Поэтому данное значение прибавляется к переменной `memoryAddress` на каждой итерации цикла `for`, что позволяет получить адрес в памяти следующего элемента массива. Запись `*pointer` разыменовывает указатель и возвращает сохраненное целочисленное значение.

Третья часть программы выглядит так:

```
    fmt.Println()
    pointer = (*int)(unsafe.Pointer(memoryAddress))
    fmt.Print("One more: ", *pointer, " ")
    memoryAddress = uintptr(unsafe.Pointer(pointer)) + unsafe.Sizeof(array[0])
    fmt.Println()
}
```

В последней части программы мы пытаемся получить доступ к несуществующему элементу массива, используя указатели и адреса ячеек памяти. Вследствие использования пакета `unsafe` компилятор Go не может отследить эту логическую ошибку, в результате которой будет возвращено нечто некорректное.

При выполнении moreUnsafe.go генерируются следующие результаты:

```
$ go run moreUnsafe.go
0 1 -2 3 4
One more: 824634208008
```

Мы только что получили доступ ко всем элементам массива Go с помощью указателей. Однако настоящая проблема заключается в том, что при попытке доступа к недопустимому элементу массива программа, не генерируя ошибку, возвращает случайное число.

Вызов C-кода из Go

Цель Go — сделать программирование приятным, избавив вас от особенностей C. Однако C остается языком программирования с очень большими возможностями и все еще полезен. Это означает, что существуют ситуации, например, при использовании базы данных или драйверов устройств, написанных на C, в которых все еще требуется использование C. Это говорит о том, что вам придется работать с C-кодом в Go-проектах.

Если вы обнаружите, что слишком часто используете эту возможность в одном и том же проекте, вам, вероятно, придется пересмотреть выбранную концепцию или выбор языка программирования.

Вызов C-кода из Go в одном файле

Самый простой способ вызвать C-код из Go-программы — включить код на C в исходный Go-файл. Тут нужен особый подход, но это довольно быстро и не очень сложно.

Рассмотрим исходный Go-файл с именем `cGo.go`. В этом файле содержится код на C и на Go, который мы разделим на три части.

Первая часть выглядит так:

```
package main

// #include <stdio.h>
// void callC() {
//    printf("Calling C code!\n");
// }
import "C"
```

Как видим, C-код включен в Go-программу в виде комментариев. Однако благодаря импорту пакета C go tool знает, что делать с подобными комментариями.

Вторая часть программы содержит следующий Go-код:

```
import "fmt"

func main() {
```

Таким образом, все остальные пакеты должны импортироваться отдельно.

В последней части `cGo.go` содержится следующий код:

```
    fmt.Println("A Go statement!")
    C.callC()
    fmt.Println("Another Go statement!")
}
```

Для того чтобы выполнить С-функцию `callC()`, ее нужно вызвать как `C.callC()`.

Выполнение `cGo.go` приведет к следующим результатам:

```
$ go run cGo.go
A Go statement!
Calling C code!
Another Go statement!
```

Вызов из Go С-кода в отдельных файлах

Теперь попробуем вызывать С-код из Go-программы, если этот С-код находится в отдельном файле.

Рассмотрим следующую задачу, которую мы решим с помощью нашей программы. Будем использовать две С-функции, которые мы якобы реализовали ранее и теперь не хотим или не можем переписать на Go.

С-код

В этом подразделе представлен С-код нашего примера. Он содержится в двух файлах: `callC.h` и `callC.c`. Подключаемый файл (`callC.h`) хранит следующий код:

```
#ifndef CALLC_H
#define CALLC_H

void cHello();
void printMessage(char* message);

#endif
```

Исходный С-файл (`callC.c`) содержит следующий С-код:

```
#include <stdio.h>
#include "callC.h"

void cHello() {
    printf("Hello from C!\n");
}

void printMessage(char* message) {
    printf("Go send me %s\n", message);
}
```

Файлы `callC.c` и `callC.h` хранятся в отдельном каталоге, который в данном случае называется `callClib`. Но вы можете выбрать любое имя каталога по своему желанию.

В C-коде нет ничего, что бы указывало на то, что этот код будет вызываться из Go-программы. Главное — вызывать правильные C-функции с указанием верного количества параметров правильного типа. Посмотрите на Go-код — и тогда все станет ясно.

Go-код

В этом подразделе представлен исходный Go-код для нашего примера. Код находится в файле `callC.go`, и мы его рассмотрим по частям.

Первая часть `callC.go` содержит следующий Go-код:

```
package main

// #cgo CFLAGS: -I${SRCDIR}/callClib
// #cgo LDFLAGS: ${SRCDIR}/callC.a
// #include <stdlib.h>
// #include <callC.h>
import "C"
```

Самый важный оператор Go всего исходного Go-файла — это отдельный оператор `import`, используемый для подключения пакета `C`. Но `C` — это виртуальный Go-пакет, который просто дает инструкцию `go build` сначала обработать входной файл с помощью инструмента `cgo`, и только потом передать файл компилятору Go. Как видим, здесь тоже использованы комментарии, которые сообщают Go-программе о наличии в ней C-кода. В данном случае они указывают программе `callC.go`, где найти файл `callC.h`, а также файл библиотеки `callC.a`, который мы вскоре создадим. Такие строки начинаются с `#cgo`.

Вторая часть программы выглядит так:

```
import (
    "fmt"
    "unsafe"
)

func main() {
    fmt.Println("Going to call a C function!")
    C.cHello()
```

Последняя часть `callC.go` выглядит следующим образом:

```
    fmt.Println("Going to call another C function!")
    myMessage := C.CString("This is Mihalis!")
    defer C.free(unsafe.Pointer(myMessage))
```

```
    C.printMessage(myMessage)

    fmt.Println("All perfectly done!")
}
```

Чтобы передать строку из Go в С-функцию, нужно создать строку в формате С с помощью `C.CString()`. Кроме того, нам понадобится оператор `defer`, чтобы освободить память, занимаемую строкой С, после того как она будет использована. Оператор `defer` включает в себя вызов функции `C.free()` и затем `unsafe.Pointer()`.

В следующем разделе вы узнаете, как скомпилировать и выполнить программу `callC.go`.

Сочетание кода на Go и С

Теперь, когда у нас есть код на С и на Go, пора узнать, что нужно сделать, чтобы выполнить файл на Go, из которого вызывается код на С.

Хорошая новость: нам не придется делать ничего сверхсложного, потому что вся важная информация содержится в Go-файле. Единственное — нужно скомпилировать С-код, чтобы создать библиотеку, для чего следует выполнить такие команды:

```
$ ls -l callClib/
total 16
-rw-r--r--@ 1 mtsouk  staff  162 Jan 10 09:17 callC.c
-rw-r--r--@ 1 mtsouk  staff   89 Jan 10 09:17 callC.h
$ gcc -c callClib/*.c
$ ls -l callC.o
-rw-r--r-- 1  mtsouk  staff  952 Jan 22 22:03 callC.o
$ file callC.o
callC.o: Mach-O 64-bit object x86_64
$ /usr/bin/ar rs callC.a *.o
ar: creating archive callC.a
$ ls -l callC.a
-rw-r--r-- 1  mtsouk  staff 4024 Jan 22 22:03 callC.a
$ file callC.a
callC.a: current ar archive
$ rm callC.o
```

У нас появится файл с именем `callC.a`, расположенный в том же каталоге, что и `callC.go`. Исполняемый файл `gcc` — это имя *С-компилятора*.

Теперь мы готовы скомпилировать файл с Go-кодом и создать новый исполняемый файл:

```
$ go build callC.go
$ ls -l callC
-rwxr-xr-x 1 mtsouk  staff 2403184 Jan 22 22:10 callC
$ file callC
callC: Mach-O 64-bit executable x86_64
```

Запустив исполняемый файл `callC`, получим следующий результат:

```
$ ./callC
Going to call a C function!
Hello from C!
Going to call another C function!
Go send me This is Mihalis!
All perfectly done!
```

Если вам нужно использовать небольшое количество C-кода, то весьма желательно использовать общий Go-файл для кода на C и на Go — так проще. Но если вы собираетесь сделать что-то большое и сложное, лучше создать статическую C-библиотеку.

Вызов Go-функций из C-кода

Аналогичным образом можно вызвать Go-функции из C-кода. В этом разделе представлен небольшой пример, в котором две Go-функции вызываются из C-программы. Для этого Go-пакет преобразуем в библиотеку C (shared library), которая будет использоваться в C-программе.

Go-пакет

Представим код Go-пакета, который будет использоваться в C-программе. Именем Go-пакета должно быть `main`, но имя файла может быть любым; в данном случае это `ByC.go`. Мы рассмотрим код пакета по частям.

Подробнее о Go-пакетах вы прочитаете в главе 6.

Первая часть кода Go-пакета выглядит так:

```
package main

import "C"

import (
    "fmt"
)
```

Как вам известно, Go-пакет обязательно должен называться `main`. Нам также нужно импортировать в Go-код пакет `C`.

Вторая часть пакета содержит следующий Go-код:

```
// export PrintMessage
func PrintMessage() {
    fmt.Println("A Go function!")
```

Каждую функцию Go, которая будет вызываться из C-кода, необходимо сначала экспортировать. Это означает, что перед ее реализацией нужно поставить строку комментария, начинающуюся с `//export`. После `//export` следует указать имя функции, которую будет использовать C-код.

Последняя часть файла `usedByC.go` выглядит так:

```
// export Multiply
func Multiply(a, b int) int {
    return a * b
}

func main() {
}
```

Функция `main()` файла `usedByC.go` не нуждается в коде, поскольку она не будет экспортироваться и, следовательно, не будет использоваться C-программой. Кроме того, поскольку мы хотим экспортировать функцию `Multiply()`, необходимо поставить перед реализацией этой функции строку `//export Multiply`.

Теперь нужно сгенерировать из Go-кода общую библиотеку C, выполнив следующую команду:

```
$ go build -o usedByC.o -buildmode=c-shared usedByC.go
```

Эта команда создаст два файла с именами `usedByC.h` и `usedByC.o`:

```
$ ls -l usedByC.*
-rw-r--r--@ 1 mtsouk  staff      204    Jan 10 09:17 usedByC.go
-rw-r--r--  1 mtsouk  staff     1365    Jan 22 22:14 usedByC.h
-rw-r--r--  1 mtsouk  staff  2329472    Jan 22 22:14 usedByC.o
$ file usedByC.o
usedByC.o: Mach-O 64-bit dynamically linked shared library x86_64
```

После этого не следует вносить никаких изменений в `usedByC.h`.

C-код

Соответствующий C-код содержится в исходном файле `willUseGo.c`, который мы рассмотрим по частям. Первая часть `willUseGo.c` выглядит так:

```
#include <stdio.h>
#include "usedByC.h"

int main(int argc, char **argv) {
    GoInt x = 12;
    GoInt y = 23;

    printf("About to call a Go function!\n");
    PrintMessage();
}
```

Если вам знаком язык C, то вы поймете, зачем нужно подключить файл `usedByC.h`: именно благодаря этому файлу C-код узнает о доступных функциях библиотеки.

Вторая часть C-программы выглядит так:

```
    GoInt p = Multiply(x,y);
    printf("Product: %d\n",(int)p);
    printf("It worked!\n");
    return 0;
}
```

Переменная `GoInt p` необходима для получения целочисленного значения из функции Go; затем посредством операции `(int) p` это значение преобразуется в целое число формата C.

Компиляция и выполнение `willUseGo.c` на машине с macOS Mojave приведет к следующим результатам:

```
$ gcc -o willUseGo willUseGo.c ./usedByC.o
$ ./willUseGo
About to call a Go function!
A Go function!
Product: 276
It worked!
```

Ключевое слово defer

Ключевое слово `defer` позволяет отложить выполнение функции до тех пор, пока не потребуется вернуть значение внешней функции. Этот прием широко используется в операциях ввода и вывода файлов, поскольку избавляет от необходимости запоминать, когда следует закрыть открытый файл. Ключевое слово `defer` позволяет поместить вызов функции, которая закрывает открытый файл, рядом с вызовом функции, которая его открыла. Подробнее об использовании `defer` в операциях, связанных с файлами, вы узнаете из главы 8. В этом же разделе будут представлены два варианта использования `defer`. Как практически использовать `defer`, вы увидите в разделе о встроенных Go-функциях `panic()` и `recover()`, а также в разделе о журналировании.

Очень важно учитывать, что после возвращения из внешней функции *отложенные функции* выполняются в порядке «последним пришел — первым вышел» (Last In, First Out, LIFO). Это означает, что если сначала отложить функцию `f1()`, затем функцию `f2()` и после нее — функцию `f3()` в одной и той же внешней функции, то, когда эта внешняя функция будет возвращать значение, функция `f3()` выполнится первой, функция `f2()` — второй, а функция `f1()` — последней.

Поскольку такое определение `defer` не очень понятно, возможно, изучив Go-код и результаты программы `defer.go`, которые мы рассмотрим по частям, вы лучше разберетесь в применении `defer`.

Первая часть программы `defer.go` выглядит так:

```
package main

import (
    "fmt"
)

func d1() {
    for i := 3; i > 0; i-- {
        defer fmt.Print(i, " ")
    }
}
```

Помимо блока `import`, в этом коде Go реализована функция с именем `d1()`. Эта функция содержит цикл `for` с оператором `defer`, который будет выполнен три раза.

Вторая часть `defer.go` состоит из следующего Go-кода:

```
func d2() {
    for i := 3; i > 0; i-- {
        defer func() {
            fmt.Print(i, " ")
        }()
    }
    fmt.Println()
}
```

В этой части кода мы видим реализацию второй функции с именем `d2()`. Функция `d2()` содержит цикл `for` с оператором `defer`, который также будет выполнен три раза. Однако здесь ключевое слово `defer` применяется не к отдельному выражению `fmt.Print()`, а к *анонимной функции*. Кроме того, эта анонимная функция не принимает параметров.

Последняя часть программы содержит следующий Go-код:

```
func d3() {
    for i := 3; i > 0; i-- {
        defer func(n int) {
            fmt.Print(n, " ")
        }(i)
    }
}

func main() {
    d1()
    d2()
    fmt.Println()
    d3()
    fmt.Println()
}
```

Кроме функции main(), которая вызывает функции d1(), d2() и d3(), мы видим здесь реализацию функции d3(). Данная функция включает в себя цикл for, в котором ключевое слово defer используется для анонимной функции. Однако на этот раз у анонимной функции есть целочисленный параметр с именем n. Из Go-кода видно, что параметр n принимает значение от переменной i, используемой в цикле for.

Выполнение defer.go приводит к следующим результатам:

```
$ go run defer.go
1 2 3
0 0 0
1 2 3
```

Скорее всего, вы посчитаете полученные результаты сложными для понимания, что доказывает: работа и результат применения defer могут быть весьма странными, если не сделать сам код понятным и недвусмысленным. Позвольте мне разъяснить эти результаты, чтобы вы имели представление о том, сколько путаницы может внести defer, если не уделить коду достаточно внимания.

Начнем с первой строки (1 2 3), сгенерированной функцией d1(). Значения i в d1() равны 3, 2 и 1 — именно в таком порядке. Отложенной функцией в d1() является fmt.Print(); в результате, когда функция d1() готова завершить работу, мы получаем три значения переменной i из цикла for в обратном порядке, поскольку отложенные функции выполняются в порядке LIFO.

Теперь о второй строке результатов, которая создается функцией d2(). Странно, что вместо 1 2 3 мы получили на выходе три нуля; однако причина этого относительно проста.

После того как завершился цикл for, значение i равно 0, поскольку именно это значение i привело к завершению цикла for. Но весь фокус в том, что отложенная анонимная функция выполняется после завершения цикла for. А поскольку у нее нет параметров, то она трижды выполняется со значением i, равным 0, и эти результаты выводятся на экран. Такой сбивающий с толку код приводит к появлению в проектах неприятных ошибок, поэтому постарайтесь избежать подобных вещей.

Наконец, рассмотрим третью строку результатов, которую генерирует функция d3(). Благодаря тому что анонимная функция имеет параметр, всякий раз, когда эта анонимная функция откладывается, она получает и использует текущее значение i. В результате при каждом запуске анонимная функция получает разные значения для обработки, что видно по сгенерированным результатам.

Сейчас вам должно быть ясно, что наилучший вариант использования defer — третий, который представлен в функции d3(), потому что здесь мы намеренно передаем в анонимную функцию нужную переменную простым и понятным способом.

Использование defer для журналирования

Рассмотрим способ применения `defer`, связанный с журналированием. Цель этого метода — упростить журналирование информации о выполнении функции, сделать ее более заметной. Go-программа, в которой показано использование ключевого слова `defer` для журналирования, называется `logDefer.go`. Исследуя ее, мы разделим эту программу на три части.

Первая часть `logDefer.go` выглядит так:

```
package main

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

var LOGFILE = "/tmp/mGo.log"

func one(aLog *log.Logger) {
    aLog.Println("-- FUNCTION one ------")
    defer aLog.Println("-- FUNCTION one ------")

    for i := 0; i < 10; i++ {
        aLog.Println(i)
    }
}
```

В функции `one()` ключевое слово `defer` используется для того, чтобы гарантировать, что второй вызов `aLog.Println()` будет выполнен непосредственно перед завершением работы функции `one()`. Поэтому все журнальные сообщения от функции `one()` будут размещаться между открывающим и закрывающим вызовами `aLog.Println()`. Так намного легче обнаружить журнальные сообщения данной функции в журнальных файлах.

Вторая часть `logDefer.go` выглядит так:

```
func two(aLog *log.Logger) {
    aLog.Println("---- FUNCTION two")
    defer aLog.Println("FUNCTION two ------")

    for i := 10; i > 0; i-- {
        aLog.Println(i)
    }
}
```

Для удобства группировки журнальных сообщений функции `two()` также применяется функция `defer`. Однако в функции `two()` используются сообщения,

немного отличающиеся от тех, что выводятся функцией `one()`. Пользователь сам выбирает формат сообщений.

Последняя часть `logDefer.go` содержит следующий Go-код:

```
func main() {
    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()

    iLog := log.New(f, "logDefer ", log.LstdFlags)
    iLog.Println("Hello there!")
    iLog.Println("Another log entry!")

    one(iLog)
    two(iLog)
}
```

При выполнении `logDefer.go` ничего не будет выводиться на экран. Однако просмотр содержимого `/tmp/mGo.log` — журнального файла, который используется программой, — покажет, насколько удобно в данном случае применять `defer`:

```
$ cat /tmp/mGo.log
logDefer 2019/01/19 21:15:11 Hello there!
logDefer 2019/01/19 21:15:11 Another log entry!
logDefer 2019/01/19 21:15:11 -- FUNCTION one ------
logDefer 2019/01/19 21:15:11 0
logDefer 2019/01/19 21:15:11 1
logDefer 2019/01/19 21:15:11 2
logDefer 2019/01/19 21:15:11 3
logDefer 2019/01/19 21:15:11 4
logDefer 2019/01/19 21:15:11 5
logDefer 2019/01/19 21:15:11 6
logDefer 2019/01/19 21:15:11 7
logDefer 2019/01/19 21:15:11 8
logDefer 2019/01/19 21:15:11 9
logDefer 2019/01/19 21:15:11 -- FUNCTION one ------
logDefer 2019/01/19 21:15:11 ---- FUNCTION two
logDefer 2019/01/19 21:15:11 10
logDefer 2019/01/19 21:15:11 9
logDefer 2019/01/19 21:15:11 8
logDefer 2019/01/19 21:15:11 7
logDefer 2019/01/19 21:15:11 6
logDefer 2019/01/19 21:15:11 5
logDefer 2019/01/19 21:15:11 4
logDefer 2019/01/19 21:15:11 3
logDefer 2019/01/19 21:15:11 2
logDefer 2019/01/19 21:15:11 1
logDefer 2019/01/19 21:15:11 FUNCTION two ------
```

Функции panic() и recover()

В этом разделе вы познакомитесь с хитроумной техникой, которая уже упоминалась в предыдущей главе. Эта техника, предполагающая использование функций `panic()` и `recover()`, представлена в файле `panicRecover.go`, который мы рассмотрим по частям.

Строго говоря, `panic()` — это встроенная функция Go, которая завершает текущий поток Go-программы и начинает бить тревогу. Что же касается функции `recover()`, которая также является встроенной функцией Go, то она позволяет вернуть контроль над программой, которая только что запаниковала, запустив `panic()`.

Первая часть программы выглядит так:

```
package main

import (
    "fmt"
)

func a() {
    fmt.Println("Inside a()")
    defer func() {
        if c := recover(); c != nil {
            fmt.Println("Recover inside a()!")
        }
    }()
    fmt.Println("About to call b()")
    b()
    fmt.Println("b() exited!")
    fmt.Println("Exiting a()")
}
```

Помимо блока `import`, эта часть содержит реализацию функции `a()`. Самой важной частью функции `a()` является блок кода `defer`. В нем реализована анонимная функция, которая будет вызываться при вызове `panic()`.

Второй фрагмент кода `panicRecover.go` выглядит так:

```
func b() {
    fmt.Println("Inside b()")
    panic("Panic in b()!")
    fmt.Println("Exiting b()")
}
```

Последняя часть программы, которая иллюстрирует функции `panic()` и `recover()`, выглядит следующим образом:

```
func main() {
    a()
    fmt.Println("main() ended!")
}
```

Выполнение `panicRecover.go` приводит к следующим результатам:

```
$ go run panicRecover.go
Inside a()
About to call b()
Inside b()
Recover inside a()!
main() ended!
```

То, что сейчас произошло, поистине впечатляет. Однако, как видно из вывода программы, функция `a()` не завершилась нормально, потому что два ее последних оператора не были выполнены:

```
fmt.Println("b() exited!")
fmt.Println("Exiting a()")
```

Все же, к счастью, `panicRecover.go` завершилась по нашей воле и без паники благодаря тому, что анонимная функция, используемая в `defer`, взяла ситуацию под контроль. Обратите внимание: функция `b()` ничего не знает о функции `a()`, но функция `a()` содержит Go-код, который обрабатывает состояние паники, возникающее в функции `b()`.

Самостоятельное использование функции panic()

Мы также можем использовать функцию `panic()` самостоятельно, без каких-либо попыток восстановления. В этом подразделе покажем, как это сделать, используя Go-код из файла `justPanic.go`, который мы разделим на две части.

Первая часть `justPanic.go` выглядит так:

```
package main

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

Как видим, использование `panic()` не требует импорта дополнительных Go-пакетов.

Вторая часть `justPanic.go` содержит следующий Go-код:

```
func main() {
    if len(os.Args) == 1 {
        panic("Not enough arguments!")
    }

    fmt.Println("Thanks for the argument(s)!")
}
```

Если Go-программа не имеет хотя бы одного аргумента командной строки, то она вызовет функцию `panic()`. Функция `panic()` принимает один параметр — сообщение об ошибке, которое мы хотим вывести на экран.

Выполнение `justPanic.go` на машине с macOS Mojave приводит к следующим результатам:

```
$ go run justPanic.go
panic: Not enough arguments!
goroutine 1 [running]:
main.main()
    /Users/mtsouk/ch2/code/justPanic.go:10 +0x91
exit status 2
```

Как видим, применение функции `panic()` само по себе приводит к завершению работы Go-программы, не давая возможности восстановить ее. Таким образом, использование пары `panic()` и `recover()` гораздо более практично и профессионально, чем использование одной только функции `panic()`.

Вывод функции panic() похож на вывод функции Panic() из пакета log. Однако panic() ничего не передает в службу журналирования UNIX-компьютера.

Две полезные UNIX-утилиты

Бывают случаи, когда UNIX-программа по неизвестной причине дает сбой или не работает должным образом. Вы хотите выяснить, почему так происходит, но при этом не хотите переписывать код и добавлять кучу операторов отладки.

В этом разделе представлены две утилиты командной строки, которые позволяют увидеть системные вызовы C, генерируемые исполняемым файлом. Эти два инструмента называются `strace(1)` и `dtrace(1)` и позволяют проверять работу программы.

Имейте в виду, что по большому счету любая программа, работающая на компьютере с UNIX, для связи с ядром UNIX и выполнения большинства задач в итоге использует системные вызовы C.

Обе утилиты работают с командой `go run`, но вы получите меньше несвязанных выходных данных, если сначала создадите исполняемый файл, используя `go build`, а затем используете этот файл. В основном так происходит потому, что, как вы уже знаете, `go run` создает несколько временных файлов, прежде чем фактически запустить Go-код. Обе утилиты увидят это и попытаются отобразить информацию о временных файлах, а это не то, что нам нужно.

Утилита strace

Утилита командной строки `strace(1)` позволяет отслеживать системные вызовы и сигналы. Поскольку `strace(1)` работает только на компьютерах с Linux, в этом разделе для демонстрации `strace(1)` будет использоваться компьютер с Debian Linux.

Утилита `strace(1)` выводит на экран следующую информацию:

```
$ strace ls
execve("/bin/ls", ["ls"], [/* 15 vars */]) = 0
brk(0)                                  = 0x186c000
fstat(3, {st_mode=S_IFREG|0644, st_size=35288, ...}) = 0
```

В выходных данных `strace(1)` отображаются все системные вызовы вместе с их параметрами, а также возвращаемыми значениями. Обратите внимание: в мире UNIX, если функция возвращает `0` — это хорошо.

Для того чтобы обработать исполняемый файл, следует поместить команду `strace(1)` перед файлом, который вы хотите обработать. Однако, чтобы сделать полезные выводы из результатов, вам придется их самостоятельно интерпретировать. К счастью, такие утилиты, как `grep(1)`, позволяют получить результат, который нам действительно нужен:

```
$ strace find /usr 2>&1 | grep ioctl
ioctl(0, SNDCTL_TMR_TIMEBASE or SNDRV_TIMER_IOCTL_NEXT_DEVICE or
TCGETS, 0x7ffe3bc59c50) = -1 ENOTTY (Inappropriate ioctl for device)
ioctl(1, SNDCTL_TMR_TIMEBASE or SNDRV_TIMER_IOCTL_NEXT_DEVICE or
TCGETS, 0x7ffe3bc59be0) = -1 ENOTTY (Inappropriate ioctl for device)
```

При использовании параметра командной строки `-c` утилита `strace(1)` позволяет выводить счетчик времени, вызовы и ошибки для каждого системного вызова:

```
$ strace -c find /usr 1>/dev/null
% time    seconds   usecs/call     calls    errors  syscall
------  ---------  -----------  ---------  ---------  -------------
 82.88   0.063223            2      39228             getdents
 16.60   0.012664            1      19587             newfstatat
  0.16   0.000119            0      19618         13  open
```

Поскольку обычно результаты программы попадают в стандартный поток вывода, а результаты `strace(1)` — в стандартный поток ошибок, выполненная выше команда отменяет вывод результатов проверяемой команды `find` и показывает результаты команды `strace(1)`. Как видно из последней строки выходных данных, системный вызов `open(2)` сделан 19 618 раз, при этом возникло 13 ошибок и все это заняло примерно 0,16 % времени выполнения всей команды, или 0,000119 секунды.

Утилита dtrace

Утилиты отладки, такие как `strace(1)` и `truss(1)`, позволяют отслеживать системные вызовы, производимые процессом, однако иногда они бывают медленными и поэтому не подходят для решения проблем производительности в высоконагруженных UNIX-системах. Еще одна утилита, называемая *DTrace*, позволяет увидеть, что происходит внутри всей системы, без необходимости что-либо изменять или перекомпилировать. DTrace также позволяет работать в системах, находящихся в процессе эксплуатации, и динамически наблюдать за работающими программами и серверными процессами, не создавая больших дополнительных нагрузок.

Несмотря на наличие версии dtrace(1) для Linux, утилита dtrace(1) лучше всего работает в macOS и других вариантах FreeBSD.

В этом подразделе используется утилита командной строки `dtruss(1)`, поставляемая с macOS. Это просто сценарий для `dtrace(1)`, который показывает системные вызовы процесса и избавляет нас от необходимости самостоятельно писать скрипт для `dtrace(1)`. Обратите внимание, что для выполнения `dtrace(1)` и `dtruss(1)` необходимы права пользователя `root`.

Утилита `dtruss(1)` выводит на экран следующие данные:

```
$ sudo dtruss godoc
ioctl(0x3, 0x80086804, 0x7FFEEFBFEC20)         = 0 0
close(0x3)    = 0 0
access("/AppleInternal/XBS/.isChrooted\0", 0x0, 0x0)      = -1 Err#2
thread_selfid(0x0, 0x0, 0x0)        = 1895378 0
geteuid(0x0, 0x0, 0x0)    = 0 0
getegid(0x0, 0x0, 0x0)    = 0 0
```

Как видим, `dtruss(1)` работает так же, как утилита `strace(1)`. Подобно `strace(1)`, `dtruss(1)` при использовании с параметром `-c` выводит счетчик системных вызовов:

```
$ sudo dtruss -c go run unsafe.go 2>&1
CALL                                        COUNT
access                                          1
bsdthread_register                              1
getuid                                          1
ioctl                                           1
issetugid                                       1
kqueue                                          1
write                                           1
mkdir                                           2
read                                          244
kevent                                        474
```

```
fcntl                                             479
lstat64                                           553
psynch_cvsignal                                   649
psynch_cvwait                                     654
```

Эти данные сразу сообщат вам о возможных узких местах вашего Go-кода или позволят сравнить производительность двух утилит командной строки.

К утилитам, подобным strace(1), dtrace(1) и dtruss(1), нужно привыкнуть. Однако такие инструменты способны сделать нашу жизнь намного проще и комфортнее, поэтому я настоятельно рекомендую вам прямо сейчас начать изучение хотя бы одного такого инструмента.

Если хотите больше узнать об утилите `dtrace(1)`, обратитесь к публикации *DTrace: Dynamic Tracing in Oracle Solaris, Mac OS X and FreeBSD* Брендана Грегга (Brendan Gregg) и Джима Мауро (Jim Mauro) или посетите сайт http://dtrace.org/.

Учтите, что `dtrace(1)` намного функциональнее, чем `strace(1)`, поскольку у `dtrace(1)` есть собственный язык программирования. Однако `strace(1)` более универсальна, если вам нужно только проследить за системными вызовами исполняемого файла.

Среда Go

В этом разделе речь пойдет о том, как узнать информацию о вашей текущей среде Go с помощью функций и свойств пакета `runtime`. Программа, которая будет разработана в этом разделе, называется `goEnv.go`. Рассмотрим ее по частям.

Первая часть `goEnv.go` выглядит так:

```
package main

import (
    "fmt"
    "runtime"
)
```

Как вы вскоре увидите, в пакет `runtime` входят функции и свойства, которые предоставят нужную информацию. Во второй части кода `goEnv.go` содержится реализация функции `main()`:

```
func main() {
    fmt.Print("You are using ", runtime.Compiler, " ")
    fmt.Println("on a", runtime.GOARCH, "machine")
    fmt.Println("Using Go version", runtime.Version())
    fmt.Println("Number of CPUs:", runtime.NumCPU())
    fmt.Println("Number of Goroutines:", runtime.NumGoroutine())
}
```

Выполнение `goEnv.go` на компьютере с macOS Mojave с версией Go 1.11.4 приведет к следующим результатам:

```
$ go run goEnv.go
You are using gc on a amd64 machine
Using Go version go1.11.4
Number of CPUs: 8
Number of Goroutines: 1
```

На компьютере с Debian Linux и версией Go 1.3.3 эта же программа генерирует следующие выходные данные:

```
$ go run goEnv.go
You are using gc on a amd64 machine
Using Go version go1.3.3
Number of CPUs: 1
Number of Goroutines: 4
```

Однако, чтобы получить по-настоящему полезную информацию о среде Go, мы воспользуемся следующей программой с именем `requiredVersion.go`. Эта программа сообщает, какую версию Go вы используете — 1.8 или выше:

```
package main

import (
    "fmt"
    "runtime"
    "strconv"
    "strings"
)

func main() {
    myVersion := runtime.Version()
    major := strings.Split(myVersion, ".")[0][2]
    minor := strings.Split(myVersion, ".")[1]
    m1, _ := strconv.Atoi(string(major))
    m2, _ := strconv.Atoi(minor)

    if m1 == 1 && m2 < 8 {
        fmt.Println("Need Go version 1.8 or higher!")
        return
    }

    fmt.Println("You are using Go version 1.8 or higher!")
}
```

Стандартный Go-пакет `strings` здесь используется для разбиения строки с версией Go, которую возвращает функция `runtime.Version()`. Из этой строки мы извлекаем две первые части, а функция `strconv.Atoi()` используется для преобразования строки в целое число.

Выполнение `requiredVersion.go` на компьютере с macOS Mojave дает следующий результат:

```
$ go run requiredVersion.go
You are using Go version 1.8 or higher!
```

Если же запустить `requiredVersion.go` на компьютере с Debian Linux, который мы уже использовали ранее в этом разделе, то программа выдаст следующий результат:

```
$ go run requiredVersion.go
Need Go version 1.8 or higher!
```

Таким образом, используя Go-код из файла `requiredVersion.go`, мы можем определить, установлена ли на вашем UNIX-компьютере требуемая версия Go.

Команда go env

Если вам нужно получить список всех переменных среды, поддерживаемых языком и компилятором Go, а также текущие значения этих переменных, то следует выполнить команду `go env`.

На моем компьютере с macOS Mojave с установленной версией Go 1.11.4 `go env` выдает весьма обширные результаты:

```
$ go env
GOARCH="amd64"
GOBIN=""
GOCACHE="/Users/mtsouk/Library/Caches/go-build"
GOEXE=""
GOFLAGS=""
GOHOSTARCH="amd64"
GOHOSTOS="darwin"
GOOS="darwin"
GOPATH="/Users/mtsouk/go"
GOPROXY=""
GORACE=""
GOROOT="/usr/local/Cellar/go/1.11.4/libexec"
GOTMPDIR=""
GOTOOLDIR="/usr/local/Cellar/go/1.11.4/libexec/pkg/tool/darwin_amd64"
GCCGO="gccgo"
CC="clang"
CXX="clang++"
CGO_ENABLED="1"
GOMOD=""
CGO_CFLAGS="-g -O2"
CGO_CPPFLAGS=""
CGO_CXXFLAGS="-g -O2"
CGO_FFLAGS="-g -O2"
CGO_LDFLAGS="-g -O2"
```

```
PKG_CONFIG="pkg-config"
GOGCCFLAGS="-fPIC -m64 -pthread -fno-caret-diagnostics -Qunused-
arguments -fmessage-length=0 -fdebug-prefix-
map=/var/folders/sk/ltk8cnw50lzdtr2hxcj5sv2m0000gn/T/go-
build790367620=/tmp/go-build -gno-record-gcc-switches -fno-common"
```

Обратите внимание, что некоторые из этих переменных среды могут иметь другие значения, если вы используете другую версию Go, или ваше имя пользователя не `mtsouk`, или вы используете другой вариант UNIX на ином оборудовании, или вы используете Go-модули по умолчанию (`GOMOD`).

Go-ассемблер

В этом разделе вкратце изложена информация о языке ассемблера и Go-ассемблере, который является инструментом Go, позволяющим увидеть ассемблерный код, используемый компилятором Go.

В качестве примера рассмотрим код ассемблера для программы `goEnv.go`, описанной в предыдущем разделе этой главы. Для этого мы выполним следующую команду:

```
$ GOOS=darwin GOARCH=amd64 go tool compile -S goEnv.go
```

Значение переменной `GOOS` определяет имя операционной системы, для которой создается программа, а значение переменной `GOARCH` — архитектуру компиляции. Эта команда выполнена на компьютере с macOS Mojave, поэтому переменной `GOOS` было присвоено значение `darwin`.

Даже для такой небольшой программы, как `goEnv.go`, эта команда выводит довольно много данных. Часть ее результатов выглядит так:

```
"".main STEXT size=859 args=0x0 locals=0x118
    0x0000 00000 (goEnv.go:8)       TEXT    "".main(SB), $280-0
    0x00be 00190 (goEnv.go:9)       PCDATA  $0, $1
    0x0308 00776 (goEnv.go:13)      PCDATA  $0, $5
    0x0308 00776 (goEnv.go:13)      CALL    runtime.convT2E64(SB)
"".init STEXT size=96 args=0x0 locals=0x8
    0x0000 00000 (<autogenerated>:1)    TEXT    "".init(SB), $8-0
    0x0000 00000 (<autogenerated>:1)    MOVQ    (TLS), CX
    0x001d 00029 (<autogenerated>:1)    FUNCDATA     $0,
gclocals d4dc2f11db048877dbc0f60a22b4adb3(SB)
    0x001d 00029 (<autogenerated>:1) FUNCDATA        $1,
gclocals 33cdeccccebe80329f1fdbee7f5874cb(SB)
```

Строки, содержащие директивы `FUNCDATA` и `PCDATA`, автоматически генерируются компилятором Go, затем используются сборщиком мусора Go.

У предыдущей команды есть другой вариант:

```
$ GOOS=darwin GOARCH=amd64 go build -gcflags -S goEnv.go
```

Переменная `GOOS` может принимать значения `android`, `darwin`, `dragonfly`, `freebsd`, `linux`, `nacl`, `netbsd`, `openbsd`, `plan9`, `solaris`, `windows` и `zos`; переменная `GOARCH` — значения `386`, `amd64`, `amd64p32`, `arm`, `armbe`, `arm64`, `arm64be`, `ppc64`, `ppc64le`, `mips`, `mipsle`, `mips64`, `mips64le`, `mips64p32`, `mips64p32le`, `ppc`, `s390`, `s390x`, `spar`, `spar` и `sparc64`.

Если вас действительно заинтересовал Go-ассемблер и вы хотите получить дополнительную информацию, посетите сайт https://golang.org/doc/asm.

Узловые деревья

Узлы Go (nodes) — это объекты типа `struct` с большим количеством свойств. Подробнее об определении и использовании *структур* в Go вы узнаете из главы 4. Все элементы Go-программы подвергаются синтаксическому разбору и анализируются модулями компилятора Go в соответствии с грамматикой языка. Конечным результом анализа Go-кода являются соответствующее ему *дерево*. Дерево — это альтернативный способ представления программы, предназначенный не для разработчика, а для компилятора.

Обратите внимание, что команда go tool 6g -W test.go не будет работать в новых версиях Go. Вместо нее следует использовать команду go tool compile -W test.go.

В этом разделе мы сначала используем в качестве примера следующий Go-код, который хранится в файле `nodeTree.go`. Это позволит нам увидеть информацию низкого уровня, предоставляемую `go tool`:

```
package main

import (
    "fmt"
)

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

Go-код `nodeTree.go` довольно понятен, поэтому вы не удивитесь тому, что он выводит следующие данные:

```
$ go run nodeTree.go
Hello there!
```

Теперь пора взглянуть на некоторые внутренние операции Go, для чего выполним следующую команду:

```
$ go tool compile -W nodeTree.go
before walk main
.   CALLFUNC l(8) tc(1) STRUCT-(int, error)
.   .   NAME-fmt.Println a(true) l(263) x(0) class(PFUNC) tc(1) used FUNC-
func(...interface {}) (int, error)
.   .   DDDARG l(8) esc(no) PTR64-*[1]interface {}
.   CALLFUNC-list
.   .   CONVIFACE l(8) esc(h) tc(1) implicit(true) INTER-interface {}
.   .   .   NAME-main.statictmp_0 a(true) l(8) x(0) class(PEXTERN) tc(1) used string
.   VARKILL l(8) tc(1)
.   .   NAME-main..autotmp_0 a(true) l(8) x(0) class(PAUTO) esc(N) used
ARRAY-[1]interface {}
after walk main
.   CALLFUNC-init
.   .   AS l(8) tc(1)
.   .   .   NAME-main..autotmp_0 a(true) l(8) x(0) class(PAUTO) esc(N)
tc(1) addrtaken assigned used ARRAY-[1]interface {}
.   .   AS l(8) tc(1)
.   .   .   NAME-main..autotmp_2 a(true) l(8) x(0) class(PAUTO) esc(N)
tc(1) assigned used PTR64-*[1]interface {}
.   .   .   ADDR l(8) tc(1) PTR64-*[1]interface {}
.   .   .   .   NAME-main..autotmp_0 a(true) l(8) x(0) class(PAUTO) esc(N)
tc(1) addrtaken assigned used ARRAY-[1]interface {}
.   .   BLOCK l(8)
.   .   BLOCK-list
.   .   .   AS l(8) tc(1) hascall
.   .   .   .   INDEX l(8) tc(1) assigned bounded hascall INTER-interface {}
.   .   .   .   .   IND l(8) tc(1) implicit(true) assigned hascall ARRAY-[1]
interface {}
.   .   .   .   .   .   NAME-main..autotmp_2 a(true) l(8) x(0) class(PAUTO)
esc(N) tc(1) assigned used PTR64-*[1]interface {}
.   .   .   .   .   LITERAL-0 l(8) tc(1) int
.   .   .   .   EFACE l(8) tc(1) INTER-interface {}
.   .   .   .   .   ADDR a(true) l(8) tc(1) PTR64-*uint8
.   .   .   .   .   .   NAME-type.string a(true) x(0) class(PEXTERN) tc(1) uint8
.   .   .   .   .   ADDR l(8) tc(1) PTR64-*string
.   .   .   .   .   .   NAME-main.statictmp_0 a(true) l(8) x(0)
class(PEXTERN) tc(1) addrtaken used string
.   .   BLOCK l(8)
.   .   BLOCK-list
.   .   .   AS l(8) tc(1) hascall
.   .   .   .   NAME-main..autotmp_1 a(true) l(8) x(0) class(PAUTO) esc(N)
tc(1) assigned used SLICE-[]interface {}
.   .   .   .   SLICEARR l(8) tc(1) hascall SLICE-[]interface {}
.   .   .   .   .   NAME-main..autotmp_2 a(true) l(8) x(0) class(PAUTO)
```

```
esc(N) tc(1) assigned used PTR64-*[1]interface {}
.   CALLFUNC l(8) tc(1) hascall STRUCT-(int, error)
.   .   NAME-fmt.Println a(true) l(263) x(0) class(PFUNC) tc(1) used FUNC-
func(...interface {}) (int, error)
.   .   DDDARG l(8) esc(no) PTR64-*[1]interface {}
.   CALLFUNC-list
.   .   AS l(8) tc(1)
.   .   .   INDREGSP-SP a(true) l(8) x(0) tc(1) addrtaken main.__ SLICE-[]interface {}
.   .   .   NAME-main..autotmp_1 a(true) l(8) x(0) class(PAUTO) esc(N)
tc(1) assigned used SLICE-[]interface {}
.   VARKILL l(8) tc(1)
.   .   NAME-main..autotmp_0 a(true) l(8) x(0) class(PAUTO) esc(N) tc(1)
addrtaken assigned used ARRAY-[1]interface {}
before walk init
.   IF l(1) tc(1)
.   .   GT l(1) tc(1) bool
.   .   .   NAME-main.initdone a(true) l(1) x(0) class(PEXTERN) tc(1)
assigned used uint8
.   .   .   LITERAL-1 l(1) tc(1) uint8
.   IF-body
.   .   RETURN l(1) tc(1)
.   IF l(1) tc(1)
.   .   EQ l(1) tc(1) bool
.   .   .   NAME-main.initdone a(true) l(1) x(0) class(PEXTERN) tc(1)
assigned used uint8
.   .   .   LITERAL-1 l(1) tc(1) uint8
.   IF-body
.   .   CALLFUNC l(1) tc(1)
.   .   .   NAME-runtime.throwinit a(true) x(0) class(PFUNC) tc(1) used
FUNC-func()
.   AS l(1) tc(1)
.   .   NAME-main.initdone a(true) l(1) x(0) class(PEXTERN) tc(1) assigned
used uint8
.   .   LITERAL-1 l(1) tc(1) uint8
.   CALLFUNC l(1) tc(1)
.   .   NAME-fmt.init a(true) l(1) x(0) class(PFUNC) tc(1) used FUNC-func()
.   AS l(1) tc(1)
.   .   NAME-main.initdone a(true) l(1) x(0) class(PEXTERN) tc(1) assigned
used uint8
.   .   LITERAL-2 l(1) tc(1) uint8
.   RETURN l(1) tc(1)
after walk init
.   IF l(1) tc(1)
.   .   GT l(1) tc(1) bool
.   .   .   NAME-main.initdone a(true) l(1) x(0) class(PEXTERN) tc(1)
assigned used uint8
.   .   .   LITERAL-1 l(1) tc(1) uint8
.   IF-body
.   .   RETURN l(1) tc(1)
```

```
.   IF l(1) tc(1)
.   .   EQ l(1) tc(1) bool
.   .   .   NAME-main.initdone a(true) l(1) x(0) class(PEXTERN) tc(1)
assigned used uint8
.   .   .   LITERAL-1 l(1) tc(1) uint8
.   IF-body
.   .   CALLFUNC l(1) tc(1) hascall
.   .   .   NAME-runtime.throwinit a(true) x(0) class(PFUNC) tc(1) used FUNC-func()
.   AS l(1) tc(1)
.   .   NAME-main.initdone a(true) l(1) x(0) class(PEXTERN) tc(1) assigned
used uint8
.   .   LITERAL-1 l(1) tc(1) uint8
.   CALLFUNC l(1) tc(1) hascall
.   .   NAME-fmt.init a(true) l(1) x(0) class(PFUNC) tc(1) used FUNC-func()
.   AS l(1) tc(1)
.   .   NAME-main.initdone a(true) l(1) x(0) class(PEXTERN) tc(1) assigned used uint8
.   .   LITERAL-2 l(1) tc(1) uint8
.   RETURN l(1) tc(1)
```

Как можно понять, компилятор Go и его инструменты делают многое незаметно для нас, даже в случае столь небольшой программы, как `nodeTree.go`.

Параметр -W дает команде компиляции go tool инструкцию вывести на экран дерево синтаксического анализа отладки (debug parse tree) после проверки соответствия типов.

Посмотрим на результаты следующих двух команд:

```
$ go tool compile -W nodeTree.go | grep before
before walk main
before walk init
$ go tool compile -W nodeTree.go | grep after
after walk main
after walk init
```

Как вы видите, ключевое слово `before` означает начало выполнения функции. Если бы у нашей программы было больше функций, то мы бы получили больше выходных данных, как показано в следующем примере:

```
$ go tool compile -W defer.go | grep before
before d1
before d2
before d3
before main
before d2.func1
before d3.func1
before init
before type..hash.[2]interface {}
before type..eq.[2]interface {}
```

В этом примере использован Go-код из файла `defer.go`. Он намного сложнее, чем `nodeTree.go`. Однако очевидно, что функция `init()` автоматически генерируется Go, поскольку она существует в обоих результатах выполнения команды `go tool compile -W` (для `nodeTree.go` и `defer.go`). Сейчас вы увидите более интересную версию `nodeTree.go` с именем `nodeTreeMore.go`:

```
package main

import (
    "fmt"
)

func functionOne(x int) {
    fmt.Println(x)
}

func main() {
    varOne := 1
    varTwo := 2
    fmt.Println("Hello there!")
    functionOne(varOne)
    functionOne(varTwo)
}
```

В программе `nodeTreeMore.go` определены две переменные с именами `varOne` и `varTwo` и дополнительная функция с именем `functionOne`. Поиск имен `varOne`, `varTwo` и `functionOne` в результатах выполнения команды `go tool compile -W` даст нам следующее:

```
$ go tool compile -W nodeTreeMore.go | grep functionOne | uniq
before walk functionOne
after walk functionOne
.   .   NAME-main.functionOne a(true) l(7) x(0) class(PFUNC) tc(1) used FUNC-func(int)
$ go tool compile -W nodeTreeMore.go | grep varTwo | uniq
.   .   NAME-main.varTwo a(true) g(2) l(13) x(0) class(PAUTO) tc(1) used int
.   .   .   NAME-main.varTwo a(true) g(2) l(13) x(0) class(PAUTO) tc(1) used int
$ go tool compile -W nodeTreeMore.go | grep varOne | uniq
.   .   NAME-main.varOne a(true) g(1) l(12) x(0) class(PAUTO) tc(1) used int
.   .   .   NAME-main.varOne a(true) g(1) l(12) x(0) class(PAUTO) tc(1) used int
```

Как видим, переменная `varOne` представлена здесь как `NAME-main.varOne`, `varTwo` — как `NAME-main.varTwo`, а функция `functionOne()` — как `NAME-main.functionOne`. Соответственно, функция `main()` здесь называется `NAME-main`.

Теперь рассмотрим следующий код дерева синтаксического анализа отладки для программы `nodeTreeMore.go`:

```
before walk functionOne
.   AS l(8) tc(1)
```

```
.   .   NAME-main..autotmp_2 a(true) l(8) x(0) class(PAUTO) esc(N) tc(1) assigned
used int
.   .   NAME-main.x a(true) g(1) l(7) x(0) class(PPARAM) tc(1) used int
```

Эти данные соответствуют определению функции `functionOne()`. Запись `l(8)` говорит о том, что определение этого узла находится в строке 8, то есть после чтения строки 7. Целочисленная переменная `NAME-main..autotmp_2` генерируется компилятором автоматически.

Рассмотрим следующий фрагмент данных дерева синтаксического анализа отладки:

```
.   CALLFUNC l(15) tc(1)
.   .   NAME-main.functionOne a(true) l(7) x(0) class(PFUNC) tc(1) used FUNC-func(int)
.   CALLFUNC-list
.   .   NAME-main.varOne a(true) g(1) l(12) x(0) class(PAUTO) tc(1) used int
```

Первая строка сообщает о том, что в строке 15 программы, на что указывает запись `l(15)`, вызывается функция `NAME-main.functionOne`, которая, судя по записи `l(7)`, определена в строке 7 программы. Запись `FUNC-func(int)` говорит о том, что эта функция принимает один целочисленный параметр. Список параметров функции, который указывается после `CALLFUNC-list`, включает в себя переменную `NAME-main.varOne`, которая, согласно записи `l(12)`, определена в строке 12 программы.

Хотите знать больше о go build?

Если вы хотите больше узнать о том, что происходит внутри компилятора, когда выполняется команда `go build`, добавьте к ней флаг `-x`:

```
$ go build -x defer.go
WORK=/var/folders/sk/ltk8cnw50lzdtr2hxcj5sv2m0000gn/T/go-build254573394
mkdir -p $WORK/b001/
cat >$WORK/b001/importcfg.link << 'EOF' # internal
packagefile command-line-arguments=/Users/mtsouk/Library/Caches/go-
build/9d/9d6ca8651e083f3662adf82bb90a00837fc76f55839e65c7107bb55fcab92458-d
packagefile fmt=/usr/local/Cellar/go/1.11.4/libexec/pkg/darwin_amd64/fmt.a
packagefile
runtime=/usr/local/Cellar/go/1.11.4/libexec/pkg/darwin_amd64/runtime.a
packagefile
errors=/usr/local/Cellar/go/1.11.4/libexec/pkg/darwin_amd64/errors.a
packagefile io=/usr/local/Cellar/go/1.11.4/libexec/pkg/darwin_amd64/io.a
packagefile
math=/usr/local/Cellar/go/1.11.4/libexec/pkg/darwin_amd64/math.a
packagefile os=/usr/local/Cellar/go/1.11.4/libexec/pkg/darwin_amd64/os.a
```

```
packagefile
reflect=/usr/local/Cellar/go/1.11.4/libexec/pkg/darwin_amd64/reflect.a
packagefile
strconv=/usr/local/Cellar/go/1.11.4/libexec/pkg/darwin_amd64/strconv.a
packagefile
sync=/usr/local/Cellar/go/1.11.4/libexec/pkg/darwin_amd64/sync.a
packagefile
unicode/utf8=/usr/local/Cellar/go/1.11.4/libexec/pkg/darwin_amd64/unicode/utf8.a
packagefile
internal/bytealg=/usr/local/Cellar/go/1.11.4/libexec/pkg/darwin_amd64/internal/
bytealg.a
packagefile
internal/cpu=/usr/local/Cellar/go/1.11.4/libexec/pkg/darwin_amd64/internal/cpu.a
packagefile
runtime/internal/atomic=/usr/local/Cellar/go/1.11.4/libexec/pkg/darwin_amd64/
runtime/internal/atomic.a
packagefile
runtime/internal/sys=/usr/local/Cellar/go/1.11.4/libexec/pkg/darwin_amd64/runtime/
internal/sys.a
packagefile
sync/atomic=/usr/local/Cellar/go/1.11.4/libexec/pkg/darwin_amd64/sync/atomic.a
packagefile
internal/poll=/usr/local/Cellar/go/1.11.4/libexec/pkg/darwin_amd64/internal/poll.a
packagefile
internal/syscall/unix=/usr/local/Cellar/go/1.11.4/libexec/pkg/darwin_amd64/
internal/syscall/unix.a
packagefile
internal/testlog=/usr/local/Cellar/go/1.11.4/libexec/pkg/darwin_amd64/internal/
testlog.a
packagefile
syscall=/usr/local/Cellar/go/1.11.4/libexec/pkg/darwin_amd64/syscall.a
packagefile
time=/usr/local/Cellar/go/1.11.4/libexec/pkg/darwin_amd64/time.a
packagefile
unicode=/usr/local/Cellar/go/1.11.4/libexec/pkg/darwin_amd64/unicode.a
packagefile
math/bits=/usr/local/Cellar/go/1.11.4/libexec/pkg/darwin_amd64/math/bits.a
packagefile
internal/race=/usr/local/Cellar/go/1.11.4/libexec/pkg/darwin_amd64/internal/race.a
EOF
mkdir -p $WORK/b001/exe/
cd .
/usr/local/Cellar/go/1.11.4/libexec/pkg/tool/darwin_amd64/link -o
$WORK/b001/exe/a.out -importcfg $WORK/b001/importcfg.link -buildmode=exe
-buildid=nkFdi6n3HGYZXDdCOju1/VfKOjehfe3PSzik3cZom/OthUDj9rThOtZPf-2627/
nkFdi6n3HGYZXDdCOju1 -extld=clang /Users/mtsouk/Library/Caches/go-
build/9d/9d6ca8651e083f3662adf82bb90a00837fc76f55839e65c7107bb55fcab92458-d
```

```
/usr/local/Cellar/go/1.11.4/libexec/pkg/tool/darwin_amd64/buildid -w
$WORK/b001/exe/a.out # internal
mv $WORK/b001/exe/a.out defer
rm -r $WORK/b001/
```

Повторю: очень многое из происходящего скрыто от нас, и хорошо бы об этом помнить. Однако в большинстве случаев вам не придется иметь дело с детальными командами процесса компиляции.

Создание кода WebAssembly

Go позволяет создавать код WebAssembly с помощью инструмента `go`. Прежде чем показать этот процесс, приведу дополнительную информацию о WebAssembly.

Краткое введение в WebAssembly

WebAssembly (Wasm) — это машинная модель и исполняемый формат для виртуальной машины, разработанный для повышения скорости и сокращения размера файла. Это означает, что двоичный файл WebAssembly можно использовать на любой платформе без каких-либо изменений.

WebAssembly поставляется в двух форматах: простом текстовом и двоичном. Файлы WebAssembly в простом текстовом формате имеют расширение `.wat`, а двоичные файлы — `.wasm`. Обратите внимание, что для загрузки и использования двоичного файла WebAssembly необходимо использовать JavaScript API.

Кроме Go, код WebAssembly можно генерировать из других языков программирования, которые поддерживают статическую типизацию, таких как Rust, C и C++.

Почему WebAssembly так важен

WebAssembly важен по следующим причинам:

- код WebAssembly работает со скоростью, очень близкой к «родной» скорости машины, то есть быстро;
- код WebAssembly можно генерировать из многих языков программирования, в том числе, возможно, и тех, которые вы уже знаете;
- большинство современных браузеров изначально поддерживают WebAssembly, не требуя подключения плагина или какого-либо другого программного обеспечения;
- код WebAssembly работает намного быстрее, чем код JavaScript.

Go и WebAssembly

Для Go WebAssembly — это просто еще одна архитектура. Поэтому для создания кода WebAssembly допускается использовать возможности кросс-компиляции Go.

Подробнее о возможностях кросс-компиляции Go вы прочитаете в главе 11. А пока обратите внимание на значения переменных среды `GOOS` и `GOARCH`, которые используются при компиляции Go-кода в WebAssembly, — именно здесь происходит вся магия.

Пример

В этом разделе будет показано, как скомпилировать Go-программу в код WebAssembly. Go-код программы `toWasm.go` выглядит так:

```
package main

import (
    "fmt"
)

func main() {
    fmt.Println("Creating WebAssembly code from Go!")
}
```

Здесь важно отметить, что в данном коде нет признаков WebAssembly, и `toWasm.go` можно скомпилировать и выполнить как отдельную программу. Другими словами, у данной программы нет внешних зависимостей, связанных с WebAssembly.

Последний шаг, который нужно будет сделать для создания кода WebAssembly, — выполнить следующую команду:

```
$ GOOS=js GOARCH=wasm go build -o main.wasm toWasm.go
$ ls -l
total 4760
-rwxr-xr-x  1 mtsouk    staff    2430633   Jan 19 21:00 main.wasm
-rw-r--r--@ 1 mtsouk    staff        100   Jan 19 20:53 toWasm.go
$ file main.wasm
main.wasm: , created: Thu Oct 25 20:41:08 2007, modified: Fri May 28 13:51:43 2032
```

Как видим, значения `GOOS` и `GOARCH`, указанные в первой команде, дают инструкцию Go сгенерировать код WebAssembly. Если не указать правильные значения `GOOS` и `GOARCH`, то при компиляции не будет создан код WebAssembly или же возникнет сбой.

Использование сгенерированного кода WebAssembly

Пока мы создали только двоичный файл WebAssembly. Но это еще не все. Нужно сделать еще кое-что, прежде чем мы сможем использовать этот двоичный файл WebAssembly и увидеть его результаты в окне браузера.

Если вы используете браузер Google Chrome, то можно применить флаг, который позволяет включить Liftoff — компилятор для WebAssembly, что теоретически улучшает время выполнения кода WebAssembly. Попробуйте — это не больно! Чтобы активизировать этот флаг, откройте страницу chrome://flags/#enable-webassembly-baseline.

Прежде всего нужно скопировать `main.wasm` в каталог веб-сервера, а затем выполнить следующую команду:

```
$ cp "$(go env GOROOT)/misc/wasm/wasm_exec.js" .
```

Эта команда скопирует файл `wasm_exec.js` из установочного каталога Go в текущий каталог. Поместите этот файл в тот же каталог веб-сервера, что и файл `main.wasm`.

Мы не будем приводить здесь код JavaScript, который содержится в `wasm_exec.js`. Представим HTML-код файла `index.html`:

```
<HTML>

<head>
    <meta charset="utf-8">
    <title>Go and WebAssembly</title>
</head>

<body>
    <script src="wasm_exec.js"></script>
    <script>
        if (!WebAssembly.instantiateStreaming) { // polyfill
            WebAssembly.instantiateStreaming = async (resp, importObject) => {
                const source = await (await resp).arrayBuffer();
                return await WebAssembly.instantiate(source, importObject);
            };
        }

        const go = new Go();
        let mod, inst;
        WebAssembly.instantiateStreaming(fetch("main.wasm"),
          go.importObject).then((result) => {
```

```
            mod = result.module;
            inst = result.instance;
            document.getElementById("runButton").disabled = false;
        }).catch((err) => {
            console.error(err);
        });

        async function run() {
            console.clear();
            await go.run(inst);
            inst = await WebAssembly.instantiate(mod, go.importObject);
        }
    </script>

    <button onClick="run();" id="runButton" disabled>Run</button>
</body>
</HTML>
```

Обратите внимание, что кнопка `Run`, созданная этим HTML-кодом, не будет активизирована, пока не загрузится код WebAssembly.

На рис. 2.2 показан результат работы кода WebAssembly, представленный в консоли JavaScript браузера Google Chrome. В других браузерах результаты будут аналогичными.

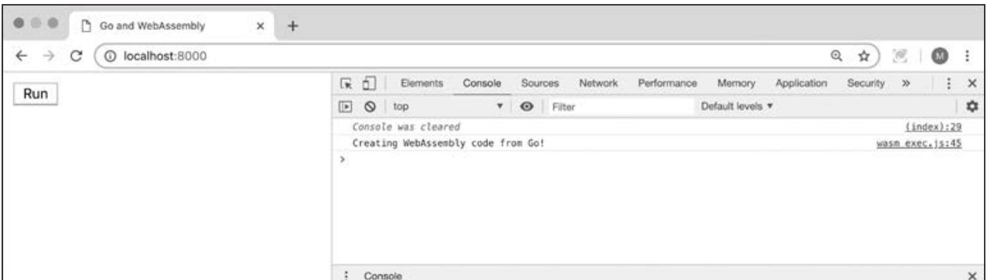


Рис. 2.2. Результат выполнения кода WebAssembly, сгенерированного в Go

В главе 12 вы узнаете, как написать на Go собственный веб-сервер.

Однако я считаю, что есть гораздо более простой и удобный способ тестирования приложений WebAssembly — с помощью *Node.js*. При использовании Node.js отпадает необходимость в веб-сервере, потому что Node.js — это среда выполнения JavaScript, построенная на движке Chrome V8 JavaScript.

Если на вашем локальном компьютере установлен Node.js, то вы можете выполнить следующую команду:

```
$ export PATH="$PATH:$(go env GOROOT)/misc/wasm"
$ GOOS=js GOARCH=wasm go run .
Creating WebAssembly code from Go!
```

Вторая команда показывает, правильно ли был выполнен код WebAssembly, и генерирует желаемое сообщение. Обратите внимание, что первая команда не является обязательной, она лишь меняет текущее значение переменной среды `PATH` на тот каталог, в котором хранятся файлы Go, связанные с WebAssembly.

Общие рекомендации по программированию на Go

Вот список практических советов, которые помогут вам написать более эффективный Go-код.

- Если в Go-функции произошла ошибка, журналируйте ее или передайте ее на уровень выше; не делайте того и другого одновременно, если только у вас нет на это действительно веской причины.
- Интерфейсы Go определяют поведение, а не данные и не структуру данных.
- По возможности используйте интерфейсы `io.Reader` и `io.Writer` — благодаря им получается более расширяемый код.
- Не передавайте в функцию указатель на простую переменную, если этого можно избежать. Лучше передайте значение переменной.
- Переменные ошибок (errors) не то же самое, что строковые переменные; это переменные ошибок!
- Не проверяйте Go-код на компьютерах, относящихся к среде эксплуатации, если только на это нет веских причин.
- Если вы недостаточно хорошо знаете какие-либо особенности Go, протестируйте их, прежде чем использовать, особенно если вы разрабатываете приложение или утилиту, которой будет пользоваться много людей.
- Если вы боитесь делать ошибки, то, скорее всего, в итоге вам не удастся сделать ничего по-настоящему интересного. Экспериментируйте, насколько хватит сил!

Упражнения и ссылки

- Познакомьтесь ближе со стандартным Go-пакетом `unsafe`, посетив страницу его документации https://golang.org/pkg/unsafe/.
- Посетите сайт DTrace по адресу http://dtrace.org/.
- Используя `strace(1)`, проверьте, как на вашем Linux-компьютере работают стандартные UNIX-утилиты, такие как `cp(1)` и `ls(1)`. Каковы оказались результаты проверки?
- Если у вас машина с macOS, воспользуйтесь `dtruss(1)`, чтобы проверить, как работает утилита `sync(8)`.
- Напишите собственный пример использования вашего C-кода из Go-программы.
- Напишите Go-функцию и используйте ее в программе на C.
- Узнайте больше о функциях пакета `runtime`, посетив страницу https://golang.org/pkg/runtime/.
- Читать научные труды бывает непросто, но это очень полезно. Скачайте статью *On-the-Fly Garbage Collection: An Exercise in Cooperation* и изучите ее. Эту статью можно найти на многих ресурсах, включая https://dl.acm.org/citation.cfm?id=359655.
- Посетите страницу https://github.com/gasche/gc-latency-experiment, где вы найдете код для сравнительного анализа производительности сборщиков мусора в различных языках программирования.
- Чтобы больше узнать о Node.js, посетите веб-сайт https://nodejs.org/en/.
- Подробнее о WebAssembly вы можете узнать на сайте https://webassembly.org/.
- Если хотите узнать больше о сборке мусора, обязательно посетите сайт http://gchandbook.org/.
- Посетите страницу документации `cgo` по адресу https://golang.org/cmd/cgo/.

Резюме

В этой главе рассмотрено много интересных тем, касающихся Go, в том числе теоретическая и практическая информация о сборщике мусора в Go. Вы узнали, как вызвать C-код из Go-программ; познакомились с удобным, хоть иногда и непростым, ключевым словом `defer`, функциями `panic()` и `restore()`; UNIX-инструментами `strace(1)`, `dtrace(1)` и `dtruss(1)`; узнали об использовании стандартного Go-пакета `unsafe`; научились генерировать из Go код WebAssembly и читать генерируемый Go ассемблерный код. С помощью пакета `runtime` вы получили информацию о своей Go-среде и узнали, как выявить и интерпретировать дерево узлов Go-программы. И в конце — несколько полезных советов по написанию программ на Go.

Из этой главы вам следует запомнить, что такие инструменты, как Go-пакет `unsafe` и возможность вызова C-кода из Go-программ, обычно используются в одном из трех случаев: если вы хотите добиться максимальной производительности и готовы пожертвовать для этого некоторой безопасностью Go, если вы хотите обмениваться данными с другим языком программирования и если хотите реализовать то, что невозможно реализовать в Go.

Следующая глава посвящена изучению основных типов данных, существующих в Go, включая *массивы*, *срезы* и *хеш-таблицы*. Несмотря на свою простоту, эти типы данных являются строительными блоками практически любого Go-приложения, поскольку служат основой для более сложных структур данных, которые позволяют хранить данные и перемещать информацию внутри Go-проектов.

Кроме того, вы познакомитесь с *указателями*, которые встречаются и в других языках программирования, *циклами* Go, а также с уникальным способом работы Go с датами и временем.

3 Работа с основными типами данных Go

В предыдущей главе рассмотрено много интересных тем, в том числе работа сборщика мусора Go, функции `panic()` и `recover()`, пакет `unsafe`, способы вызова C-кода из программы Go и, наоборот, Go-кода из программы на C, а также деревья узлов, создаваемые компилятором Go при компиляции Go-программы.

Главная тема этой главы — основные типы данных Go. Сюда входят *числовые типы*, *массивы*, *срезы* и *хеш-таблицы*. Несмотря на простоту, эти типы данных помогут вам при выполнении числовых расчетов. Они также предоставляют удобный и быстрый способ хранения, извлечения и изменения данных в Go-программе. Кроме того, вы узнаете, что такое *указатели*, *константы*, *циклы*, а также как работать с датами и временем в Go.

В данной главе рассмотрены следующие темы:

- числовые типы данных;
- массивы в Go;
- срезы Go, и почему они намного лучше, чем массивы;
- как добавить массив в существующий срез;
- хеш-таблицы Go;
- указатели в Go;
- циклы в Go;
- константы в Go;
- работа с временем;
- измерение времени выполнения команд и функций;
- работа с датами.

Числовые типы данных

В Go реализована встроенная поддержка целых чисел, чисел с плавающей точкой, а также комплексных чисел. В следующих подразделах мы ближе познакомимся со всеми числовыми типами, поддерживаемыми в Go.

Целые числа

Go поддерживает четыре размера целых чисел *со знаком* и *без знака*. Эти целочисленные типы со знаком называются `int8`, `int16`, `int32`, `int64`, а без знака — `uint8`, `uint16`, `uint32` и `uint64`. Число в конце каждого названия типа соответствует количеству битов, выделяемых для представления соответствующего числа.

Кроме того, есть типы `int` и `uint`, которые являются наиболее эффективным представлением целых чисел со знаком и без знака соответственно для текущей платформы. Поэтому в случае сомнений используйте `int` и `uint`, однако имейте в виду, что размер этих типов изменяется в зависимости от архитектуры компьютера.

Разница между целыми числами со знаком и без знака заключается в следующем: если целое число занимает 8 бит и не имеет знака, то оно может принимать значения в диапазоне от двоичного 00000000 (`0`) до двоичного 11111111 (`255`). Если же у числа есть знак, то оно может принимать значения от –127 до 127. Таким образом, для хранения числа выделяется семь двоичных разрядов, так как восьмой бит используется для хранения знака целого числа. Это же правило применяется и к другим размерам целых чисел без знака.

Числа с плавающей точкой

Go поддерживает всего два типа чисел с плавающей точкой: `float32` и `float64`. Первый из них обеспечивает точность примерно до шестого знака после запятой, а второй — до 15-го знака.

Комплексные числа

Как и в случае с числами с плавающей запятой, Go предлагает два типа комплексных чисел с именами `complex64` и `complex128`. В первом из них используется два типа `float32`: один для вещественной части комплексного числа, а второй — для мнимой, тогда как в `complex128` используется два типа `float64`.

Комплексные числа выражаются в форме $a + bi$, где a и b — действительные числа, а i — решение уравнения $x^2 = -1$.

Все эти числовые типы данных продемонстрированы в программе `numbers.go`, которую мы рассмотрим, разделив на три части.

Первая часть `numbers.go` выглядит так:

```
package main

import (
    "fmt"
)

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)

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)
```

В этой части программы мы имеем дело с комплексными числами и выполняем над ними некоторые вычисления. Есть два способа создания комплексного числа: напрямую, как в случае с `c1` и `c2`, или косвенно, путем выполнения расчетов с существующими комплексными числами, как в случае с `c3` и `cZero`.

Если вы по ошибке попытаетесь создать комплексное число как aComplex: = 12 + 2 * i, то возможны два варианта, поскольку Go воспринимает данный оператор как сложение и умножение. Если в текущей области видимости нет числовой переменной с именем i, то такой оператор приведет к появлению синтаксической ошибки и код Go не будет скомпилирован. Если же числовая переменная с именем i определена, то вычисление будет успешным, но в результате вы не получите желаемое комплексное число (дефект программы).

Вторая часть `numbers.go` выглядит так:

```
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 имеем дело с целыми числами со знаком. Обратите внимание, что если разделить одно целое число на другое, то Go решит, что вы хотите получить результат целочисленного деления, и вычислит и вернет результат в виде *неполного частного от целочисленного деления*. Таким образом, попытка разделить целое число 11 на целое число 2 даст результат 5, а не 5,5, и это может удивить.

При преобразовании числа с плавающей запятой в целое число дробь отбрасывается путем усечения числа с плавающей запятой до нуля знаков после запятой. Это означает, что в процессе такой операции некоторые данные могут быть потеряны.

Последняя часть numbers.go содержит следующий код:

```
    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)
}
```

В этой части программы обрабатываются числа с плавающей запятой. В представленном коде показано, как использовать функцию `float64()`, чтобы при делении одного целого числа на другое Go создавал число с плавающей точкой. Если просто ввести `divFloat: = float64(x) / k`, то при выполнении кода получим следующее сообщение об ошибке:

```
$ go run numbers.go
# command-line-arguments
./numbers.go:35:25: invalid operation: float64(x) / k (mismatched types
float64 and int)
```

Выполнение numbers.go дает такой результат:

```
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
```

Числовые литералы в Go 2

На момент написания этой книги поступило предложение внести изменения в поддержку Go *числовых литералов*. С помощью числовых литералов можно определять и использовать числа в языке программирования. Предложение связано с представлением двоичных и восьмеричных целочисленных литералов, разделителя между цифрами, а также с поддержкой шестнадцатеричных чисел с плавающей точкой.

Более подробную информацию о предложении, касающемся Go 2 и числовых литералов, вы найдете по адресу https://golang.org/design/19308-number-literals.

У разработчиков Go есть подробная информационная панель управления выпусками, которую вы можете просмотреть по адресу https://dev.golang.org/release.

Циклы Go

В любом языке программирования есть циклы, и Go не исключение. В Go есть цикл `for`, который позволяет осуществлять иттерации по различным типам данных.

В Go нет ключевого слова while, но циклы for в Go вполне заменяют циклы while.

Цикл for

Цикл `for` позволяет выполнить определенный код заранее заданное число раз, пока действительно некое условие или в соответствии со значением, которое вычисляется в начале цикла `for`. Таким значением может быть размер среза или массива либо количество ключей в хеш-таблице. Итак, цикл `for` является самым распространенным способом перебрать все элементы массива, среза или хеш-таблицы.

Далее представлена простейшая форма цикла `for`. Переменная `i` принимает все значения из предопределенного диапазона:

```
for i := 0; i < 100; i++ {
}
```

Вообще, цикл `for` состоит из трех частей: первая называется *инициализацией*, вторая — *условием*, а последняя — *изменением счетчика*. Ни одна из этих частей не является обязательной.

В приведенном выше цикле переменная `i` принимает значения в диапазоне от 0 до 99. Как только `i` станет равной `100`, выполнение цикла `for` остановится. В данном случае `i` — локальная и временная переменная, то есть после завершения цикла `for` в какой-то момент `i` попадет в сборку мусора и исчезнет. Однако, если бы `i` была определена вне цикла `for`, она сохранила бы свое значение после завершения цикла `for`. В таком случае значение `i` после завершения цикла `for` будет равно `100`, так как это последнее значение `i` в этой точке программы.

Для того чтобы выйти из цикла `for` до его завершения, можно использовать ключевое слово `break`. Ключевое слово `break` также позволяет создать цикл `for` без

условия выхода, такого как `i < 100` в предыдущем примере, поскольку тогда условие выхода может быть включено в блок кода цикла `for`. Также разрешается использовать в цикле `for` несколько условий выхода. Кроме того, можно пропустить одну итерацию цикла `for`, используя ключевое слово `continue`.

Цикл while

Как уже отмечалось, в Go нет ключевого слова `while` для реализации циклов `while`, однако язык позволяет использовать вместо цикла `while` цикл `for`. В этом разделе представлены два примера, где цикл `for` выполняет работу цикла `while`.

Сначала рассмотрим типичный случай того, как можно написать нечто аналогичное `while (true)`:

```
for {
}
```

Задача разработчика заключается в том, чтобы поставить в нужном месте ключевое слово `break` для выхода из этого цикла `for`.

Однако цикл `for` также может эмулировать цикл `do ... while`, который встречается в других языках программирования.

Например, следующий код Go эквивалентен циклу `do ... while (anExpression)`:

```
for ok := true; ok; ok = anExpression {
}
```

Как только переменная `ok` примет значение `false`, цикл `for` прекратит работу.

В Go также есть цикл `for condition {}`, который выполняется до тех пор, пока соблюдается условие `condition`.

Ключевое слово range

В Go есть еще ключевое слово `range`, которое используется в циклах `for` и позволяет писать легкочитаемый код для перебора любых типов данных, поддерживаемых в Go, включая *каналы*. Главным преимуществом ключевого слова `range` является то, что с ним не нужно заранее знать *емкость* (количество элементов) среза, хеш-таблицы или канала, чтобы перебрать по одному все его элементы. Далее вы узнаете, как использовать ключевое слово `range`.

Пример применения нескольких циклов Go

В этом разделе показаны несколько примеров использования циклов `for`. Имя файла с программой — `loops.go`, и для его изучения разделим программу на четыре части. Первый фрагмент кода `loops.go` выглядит так:

```
package main

import (
    "fmt"
)

func main() {
    for i := 0; i < 100; i++ {
        if i%20 == 0 {
            continue
        }
        if i == 95 {
            break
        }

        fmt.Print(i, " ")
}
```

В этом коде показан типичный цикл `for`, а также продемонстрировано использование ключевых слов `continue` и `break`.

Следующий фрагмент кода выглядит следующим образом:

```
    fmt.Println()
    i := 10
    for {
        if i < 0 {
            break
        }
        fmt.Print(i, " ")
        i--
    }
    fmt.Println()
```

Этот код эмулирует типичный цикл `while`. Обратите внимание на использование ключевого слова `break` для выхода из цикла `for`.

Третья часть `loops.go` выглядит так:

```
    i = 0
    anExpression := true
    for ok := true; ok; ok = anExpression {
        if i > 10 {
            anExpression = false
        }

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

Здесь вы видите, как цикл `for` выполняет работу цикла `do ... while`, упоминавшегося ранее в этой главе. Заметьте, что этот цикл `for` трудно читается.

Последняя часть loops.go содержит следующий код Go:

```
    anArray := [5]int{0, 1, -1, 2, -2}
    for i, value := range anArray {
        fmt.Println("index:", i, "value: ", value)
    }
}
```

Применение ключевого слова range к переменной массива возвращает два значения: индекс массива и значение элемента, соответствующее этому индексу.

Вы можете использовать оба этих значения, только одно из них или ни одного, если просто хотите сосчитать элементы массива или выполнить какую-либо другую задачу столько раз, сколько элементов содержится в массиве.

Выполнение loops.go даст следующий результат:

```
$ go run loops.go
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 21 22 23 24 25 26 27 28 29
30 31 32 33 34 35 36 37 38 39 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55
56 57 58 59 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 81 82
83 84 85 86 87 88 89 90 91 92 93 94
10 9 8 7 6 5 4 3 2 1 0
0 1 2 3 4 5 6 7 8 9 10 11
index: 0 value: 0
index: 1 value: 1
index: 2 value: -1
index: 3 value: 2
index: 4 value: -2
```

Массивы в Go

Массивы являются одной из самых популярных структур данных по двум причинам. Первая состоит в том, что массивы просты и понятны, а вторая — в том, что они универсальны и позволяют хранить самые разные виды данных.

Ниже показано, как можно объявить массив, в котором хранится четыре целых числа:

```
anArray := [4]int{1, 2, 4, -4}
```

Сначала указывается размер массива, затем его тип, а в конце — список элементов. Узнать длину массива можно с помощью функции `len()`: `len(anArray)`.

Индекс первого элемента любого измерения массива равен нулю, индекс второго элемента — единице и т. д. Это означает, что для одномерного массива с именем `a` допустимы индексы от `0` до `len(a) - 1`.

Возможно, вы знакомы со способами доступа к элементам массива в других языках программирования и использованием цикла `for` с одной или несколькими числовыми переменными, однако в Go есть более характерные для данного языка

способы перебора всех элементов массива. Они предполагают использование ключевого слова `range` и позволяют обойти использование функции `len()` в цикле `for`. Мы уже рассмотрели в качестве такого примера Go-код в файле `loops.go`.

Многомерные массивы

Массивы могут быть многомерными. Однако использование более трех измерений без серьезных причин может затруднить чтение программы и привести к ошибкам.

Массивы позволяют хранить элементы любых типов; мы используем целые числа для простоты, поскольку их легче понять и выводить на экран.

В следующем коде Go показано, как создать двумерный массив `twoD` и трехмерный массив `threeD`:

```
twoD := [4][4]int{{1, 2, 3, 4}, {5, 6, 7, 8}, {9, 10, 11, 12}, {13, 14, 15, 16}}
threeD := [2][2][2]int{{{1, 0}, {-2, 4}}, {{5, -1}, {7, 0}}}
```

Мы можем легко прочитать, изменить или вывести на экран отдельный элемент любого из этих двух массивов. Например, первый элемент массива `twoD` — это `twoD[0][0]`, и его значение равно `1`.

Таким образом, доступ ко всем элементам массива `threeD` можно организовать с помощью нескольких циклов `for` следующим образом:

```
for i := 0; i < len(threeD); i++ {
    for j := 0; j < len(v); j++ {
        for k := 0; k < len(m); k++ {
        }
    }
}
```

Как вы видите, для того чтобы получить доступ ко всем элементам массива, нам нужно столько же циклов `for`, сколько измерений у этого массива. Это же правило применяется к срезам, которые будут рассмотрены в следующем разделе. Использование `x`, `y` и `z` в качестве имен переменных вместо `i`, `j` и `k` может быть хорошей идеей.

Полный пример того, как обращаться с массивами в Go, представлен в коде `usingArrays.go`, который мы рассмотрим, разделив на три части.

Первая часть кода выглядит следующим образом:

```
package main

import (
```

```
    "fmt"
)

func main() {
    anArray := [4]int{1, 2, 4, -4}
    twoD := [4][4]int{{1, 2, 3, 4}, {5, 6, 7, 8}, {9, 10, 11, 12}, {13, 14, 15, 16}}
    threeD := [2][2][2]int{{{1, 0}, {-2, 4}}, {{5, -1}, {7, 0}}}
```

Здесь мы определяем три переменные массива с именами `anArray`, `twoD` и `threeD` соответственно.

Вторая часть `Arrays.go` выглядит так:

```
    fmt.Println("The length of", anArray, "is", len(anArray))
    fmt.Println("The first element of", twoD, "is", twoD[0][0])
    fmt.Println("The length of", threeD, "is", len(threeD))

    for i := 0; i < len(threeD); i++ {
        v := threeD[i]
        for j := 0; j < len(v); j++ {
            m := v[j]
            for k := 0; k < len(m); k++ {
                fmt.Print(m[k], " ")
            }
        }
        fmt.Println()
    }
```

Из первого цикла `for` получаем двумерный массив `threeD[i]`, а из второго цикла `for` — одномерный массив `v[j]`. Последний цикл `for` перебирает элементы этого одномерного массива.

Последняя часть кода содержит следующий код Go:

```
    for _, v := range threeD {
        for _, m := range v {
            for _, s := range m {
                fmt.Print(s, " ")
            }
        }
        fmt.Println()
    }
}
```

Ключевое слово `range` выполняет ту же работу, что и переменные итераций, использованные в циклах `for` предыдущего фрагмента кода, но только более элегантно и понятно. Однако если вы хотите заранее узнать количество итераций, которые будут выполнены, то использовать ключевое слово `range` нельзя.

Ключевое слово range также работает с хеш-таблицами Go, что делает его весьма удобным и предпочтительным способом перебора значений. Как вы увидите в главе 9, ключевое слово range также работает с каналами.

Выполнение `usingArrays.go` приведет к следующим результатам:

```
$ go run usingArrays.go
The length of [1 2 4 -4] is 4
The first element of [[1 2 3 4] [5 6 7 8] [9 10 11 12] [13 14 15 16]] is 1
The length of [[[1 0] [-2 4]] [[5 -1] [7 0]]] is 2
1 0 -2 4
5 -1 7 0
1 0 -2 4
5 -1 7 0
```

Одна из самых больших проблем при работе с массивами — это *ошибки выхода за пределы размерности массива*, что означает попытку доступа к несуществующему элементу. Например, это может быть попытка получить доступ к шестому элементу массива, состоящего всего из пяти элементов. Компилятор Go распознает такие проблемы и сообщает о них как об ошибках компиляции, потому что это помогает в процессе разработки. Таким образом, компилятор Go может обнаруживать ошибки выхода за пределы массива:

```
./a.go:10: invalid array index -1 (index must be non-negative)
./a.go:10: invalid array index 20 (out of bounds for 2-element array)
```

Недостатки массивов Go

У массивов Go есть много недостатков, которые, возможно, заставят вас подумать дважды, прежде чем использовать массивы в своих проектах Go. Прежде всего, после того как массив определен, вы не можете изменить его размер — другими словами, массивы Go не являются динамическими. Таким образом, для того, чтобы добавить элемент в существующий массив, в котором больше нет свободного места, нужно создать массив большего размера и скопировать в него все элементы старого массива. Кроме того, передавая массив функции в качестве параметра, вы на самом деле передаете копию массива, так что любые изменения, внесенные в массив внутри функции, после выхода из этой функции будут потеряны. Наконец, передача большого массива в функцию — довольно медленная операция, главным образом потому, что при этом Go создает копию массива. Решением всех этих проблем является использование срезов Go, которые представлены в следующем разделе.

Из-за своих недостатков массивы используются в Go очень редко!

Срезы в Go

Срезы Go — очень мощные структуры данных. Не будет преувеличением сказать, что срезы способны полностью заменить массивы в Go. Есть лишь несколько случаев, когда приходится использовать массив вместо среза. Наиболее очевидным является случай, когда вы абсолютно уверены в том, что нужно будет хранить фиксированное количество элементов.

Срезы в Go реализованы на основе массивов, следовательно, внутри каждого среза Go лежит массив.

Срезы передаются в функции *по ссылке* — это означает, что фактически в функцию передается адрес памяти переменной среза, и любые изменения, внесенные в срез внутри функции, не будут потеряны после выхода из нее. Кроме того, большие срезы передаются в функцию значительно быстрее, чем массивы с тем же количеством элементов, поскольку в этом случае Go не делает копию среза, а лишь передает адрес памяти переменной среза.

Выполнение основных операций со срезами

Для того чтобы создать *литерал среза*, нужно сделать следующее:

```
aSliceLiteral := []int{1, 2, 3, 4, 5}
```

Таким образом, литералы срезов определяются так же, как и массивы, но без указания количества элементов. Если добавить в это определение количество элементов, то вместо среза получим массив.

Однако, кроме этого, еще есть функция `make()`, которая позволяет создавать пустые срезы желаемой *длины* и *емкости* на основе параметров, передаваемых в `make()`. Параметр емкости можно пропустить, и тогда емкость среза будет равна его длине. Таким образом можно создать пустой срез на 20 элементов, который при необходимости автоматически расширяется:

```
integer := make([]int, 20)
```

Обратите внимание, что Go автоматически инициализирует все элементы нового среза нулевыми значениями заданного типа. Это означает, что значение инициализации зависит от типа объектов, хранимых в срезе. К счастью, Go инициализирует элементы любого среза, создаваемого с помощью `make`.

Получить доступ ко всем элементам среза можно следующим образом:

```
for i := 0; i < len(integer); i++ {
    fmt.Println(integer[i])
}
```

Если необходимо очистить существующий срез, то нулевым значением для переменной среза будет nil:

```
aSliceLiteral = nil
```

Чтобы добавить элемент к срезу, нужно воспользоваться функцией append(), что при необходимости автоматически увеличит размер среза:

```
integer = append(integer, 12345)
```

Чтобы получить доступ к первому элементу среза integer, следует обратиться к нему как integer[0], а к последнему элементу среза integer — как к integer[len(integer)-1].

Наконец, с помощью нотации [:] можно получить доступ к нескольким последовательным элементам среза. Этот оператор выбирает второй и третий элементы среза:

```
integer[1:3]
```

Кроме того, нотация [:] позволяет создать новый срез из уже существующего среза или массива:

```
s2 := integer[1:3]
```

Обратите внимание, что этот процесс, называемый *созданием вторичных срезов*, в некоторых случаях может вызвать проблемы. Рассмотрим следующую программу:

```
package main

import "fmt"

func main() {
    s1 := make([]int, 5)
    reSlice := s1[1:3]
    fmt.Println(s1)
    fmt.Println(reSlice)
    reSlice[0] = -100
    reSlice[1] = 123456
    fmt.Println(s1)
    fmt.Println(reSlice)
}
```

Следует отметить, что для выбора второго и третьего элементов среза с использованием нотации [:] необходимо использовать запись [1: 3], которая означает, что надо выбрать из среза все элементы, начиная с индекса 1 и заканчивая индексом 3, не включая индекс 3.

Для некоторого массива a1 можно создать срез s1, который бы ссылался на этот массив, с помощью оператора s1 := a1[:].

Выполнение предыдущего кода, сохраненного как reslice.go, приведет к следующим результатам:

```
$ go run reslice.go
[0 0 0 0 0]
[0 0]
[0 -100 123456 0 0]
[-100 123456]
```

Таким образом, в конце программы содержимым среза `s1` является `[0 -100 123456 0 0]`, несмотря на то что мы не меняли его напрямую! Это означает, что при изменении элементов вторичного среза изменяются и элементы исходного среза, поскольку оба они указывают на один и тот же базовый массив. Проще говоря, при создании вторичного среза не создается копия исходного среза.

Вторая проблема создания вторичных срезов состоит в том, что базовый массив исходного среза будет сохраняться в памяти до тех пор, пока существует меньший, вторичный срез. Потому что вторичный срез ссылается на исходный срез. Так будет, даже если повторный срез создан для того, чтобы использовать только небольшую часть исходного среза. Для небольших срезов это не очень важно, однако при чтении больших файлов и сохранении данных в виде срезов, из которых вы хотите использовать лишь небольшую часть, это может вызвать проблемы.

Автоматическое расширение срезов

Срезы характеризуются двумя основными свойствами: *емкостью* (capacity) и *длиной* (length). Фокус в том, что обычно эти два свойства имеют разные значения. Длина среза равна длине массива с таким же количеством элементов и может быть получена с помощью функции `len()`. Емкость среза — это то, сколько места в памяти занимает срез в данный момент. Емкость можно узнать с помощью функции `cap()`. Поскольку размер срезов является динамической величиной, то, если длина среза выходит за пределы выделенной для него памяти, Go автоматически удваивает текущую емкость среза, чтобы выделить место для большего количества элементов.

Проще говоря, если длина и емкость среза имеют одинаковые значения, то попытка добавить в срез еще один элемент приводит к тому, что емкость среза будет удвоена, тогда как его длина увеличится только на единицу.

В случае небольших срезов такой прием может быть достаточно эффективным, однако добавление еще одного элемента в действительно огромный срез может потребовать больше памяти, чем вы ожидаете.

Представленный далее код `lenCap.go` более подробно иллюстрирует понятия емкости и длины среза. Мы рассмотрим его, разделив на три части. Первая часть программы выглядит так:

```
package main

import (
    "fmt"
)
```

```
func printSlice(x []int) {
    for _, number := range x {
        fmt.Print(number, " ")
    }
    fmt.Println()
}
```

Функция `printSlice()` помогает вывести на экран одномерный срез без необходимости постоянно повторять один и тот же код Go.

Вторая часть `lenCap.go` содержит следующий код:

```
func main() {
    aSlice := []int{-1, 0, 4}
    fmt.Printf("aSlice: ")
    printSlice(aSlice)
    fmt.Printf("Cap: %d, Length: %d\n", cap(aSlice), len(aSlice))
    aSlice = append(aSlice, -100)
    fmt.Printf("aSlice: ")
    printSlice(aSlice)
    fmt.Printf("Cap: %d, Length: %d\n", cap(aSlice), len(aSlice))
```

В этой, а также в следующей части программы мы добавим в срез `aSlice` новые элементы, чтобы изменить его длину и емкость.

Последняя часть кода Go выглядит так:

```
    aSlice = append(aSlice, -2)
    aSlice = append(aSlice, -3)
    aSlice = append(aSlice, -4)
    printSlice(aSlice)
    fmt.Printf("Cap: %d, Length: %d\n", cap(aSlice), len(aSlice))
}
```

Выполнение `lenCap.go` приведет к следующим результатам:

```
$ go run lenCap.go
aSlice: -1 0 4
Cap: 3, Length: 3
aSlice: -1 0 4 -100
Cap: 6, Length: 4
-1 0 4 -100 -2 -3 -4
Cap: 12, Length: 7
```

Как видно, начальная длина среза была равна 3 и его первоначальная емкость тоже равнялась 3. После добавления в срез нового элемента его длина стала равна 4, а емкость — 6. После добавления в срез еще трех элементов его длина стала равна 7, тогда как емкость еще раз удвоилась[1] и стала равной 12.

[1] Нужно заметить, что емкость среза удваивается, только пока его длина не превысит 1024 элементов. Далее она растет не более чем на 25 %.

Байтовые срезы

Байтовый срез — это срез, типом которого является byte. Чтобы создать байтовый срез с именем s, нужно выполнить следующий код:

```
s := make([]byte, 5)
```

Go знает, что в большинстве случаев байтовые срезы используются для хранения строк, и поэтому позволяет легко переключаться между этим типом и типом string. По способу доступа байтовый срез ничем не отличается от других типов срезов. Просто байтовые срезы используются в операциях ввода и вывода файлов. Как используются байтовые срезы, вы прочитаете в главе 8.

Функция copy()

С помощью функции copy() можно создать срез из элементов существующего массива или как копию существующего среза. Однако, поскольку функция copy() может быть очень запутанной, в этом подразделе разъясняется ее использование с помощью кода Go copySlice.go, который разделим на четыре части.

При использовании функции copy() для срезов следует быть очень осторожным, поскольку количество элементов, копируемых встроенной функцией copy(dst, src), составляет минимум из len(dst) и len(src).

Первая часть программы содержит следующий код Go:

```
package main

import (
    "fmt"
)

func main() {
    a6 := []int{-10, 1, 2, 3, 4, 5}
    a4 := []int{-1, -2, -3, -4}
    fmt.Println("a6:", a6)
    fmt.Println("a4:", a4)

    copy(a6, a4)
    fmt.Println("a6:", a6)
    fmt.Println("a4:", a4)
    fmt.Println()
```

Как видите, в этом коде созданы два среза с именами a6 и a4, их содержимое выводится на экран, а затем предпринимается попытка скопировать a4 в a6.

Поскольку в a6 больше элементов, чем в a4, все элементы a4 будут скопированы в a6. Однако, поскольку у a4 только четыре элемента, а у a6 — шесть элементов, последние два элемента a6 останутся прежними.

Вторая часть copySlice.go выглядит так:

```
    b6 := []int{-10, 1, 2, 3, 4, 5}
    b4 := []int{-1, -2, -3, -4}
    fmt.Println("b6:", b6)
    fmt.Println("b4:", b4)
    copy(b4, b6)
    fmt.Println("b6:", b6)
    fmt.Println("b4:", b4)
```

В этом случае в b4 будут скопированы только первые четыре элемента b6, так как емкость b4 составляет всего четыре элемента.

Третий фрагмент кода copySlice.go содержит следующий код Go:

```
    fmt.Println()
    array4 := [4]int{4, -4, 4, -4}
    s6 := []int{1, 1, -1, -1, 5, -5}
    copy(s6, array4[0:])
    fmt.Println("array4:", array4[0:])
    fmt.Println("s6:", s6)
    fmt.Println()
```

Здесь массив из четырех элементов копируется в срез из шести элементов. Обратите внимание, что массив преобразуется в срез с помощью нотации [:] (array4[0:]).

Последняя часть кода copySlice.go выглядит следующим образом:

```
    array5 := [5]int{5, -5, 5, -5, 5}
    s7 := []int{7, 7, -7, -7, 7, -7, 7}
    copy(array5[0:], s7)
    fmt.Println("array5:", array5)
    fmt.Println("s7:", s7)
}
```

Здесь показано, как скопировать срез в массив, в котором есть место для пяти элементов. Поскольку copy() принимает только аргументы среза, необходимо использовать нотацию [:] для преобразования массива в срез.

Если вы попытаетесь скопировать массив в срез или наоборот, не используя нотацию [:], программа не будет скомпилирована и вы получите одно из следующих сообщений об ошибке:

```
# command-line-arguments
./a.go:42:6: first argument to copy should be slice; have [5]int
./a.go:43:6: second argument to copy should be slice or string; have [5]int
./a.go:44:6: arguments to copy must be slices; have [5]int, [5]int
```

Выполнение `copySlice.go` приведет к следующим результатам:

```
$ go run copySlice.go
a6: [-10 1 2 3 4 5]
a4: [-1 -2 -3 -4]
a6: [-1 -2 -3 -4 4 5]
a4: [-1 -2 -3 -4]
b6: [-10 1 2 3 4 5]
b4: [-1 -2 -3 -4]
b6: [-10 1 2 3 4 5]
b4: [-10 1 2 3]
array4: [4 -4 4 -4]
s6: [4 -4 4 -4 5 -5]
array5: [7 7 -7 -7 7]
s7: [7 7 -7 -7 7 -7 7]
```

Многомерные срезы

Срезы, как массивы, могут быть многомерными. Следующая инструкция создает двумерный срез:

```
s1 := make([][]int, 4)
```

Если вам постоянно приходится использовать многомерные срезы, возможно, имеет смысл пересмотреть подход и выбрать более простую структуру программы, которая бы не требовала многомерных срезов.

В следующем разделе рассмотрим пример кода с использованием многомерного среза.

Еще один пример использования срезов

Будем надеяться, что код Go в программе `slices.go` прояснит многие аспекты использования срезов. Рассмотрим этот код, разделив его на пять частей.

Первая часть программы содержит обычную преамбулу, а также определение двух срезов:

```
package main

import (
    "fmt"
)

func main() {
    aSlice := []int{1, 2, 3, 4, 5}
    fmt.Println(aSlice)
```

```
integers := make([]int, 2)
fmt.Println(integers)
integers = nil
fmt.Println(integers)
```

Во второй части показано, как использовать нотацию `[:]` для создания нового среза, который ссылается на существующий массив. Помните, что мы создаем не копию массива, а лишь ссылку на него, в чем вы убедитесь при выводе данных программы:

```
anArray := [5]int{-1, -2, -3, -4, -5}
refAnArray := anArray[:]

fmt.Println(anArray)
fmt.Println(refAnArray)
anArray[4] = -100
fmt.Println(refAnArray)
```

В третьем фрагменте кода с помощью функции `make()` создаются два среза, одномерный и двумерный:

```
s := make([]byte, 5)
fmt.Println(s)
twoD := make([][]int, 3)
fmt.Println(twoD)
fmt.Println()
```

Поскольку срезы в Go инициализируются автоматически, все элементы этих двух срезов будут иметь нулевое значение, соответствующее типу данного среза, которое для целых чисел равно `0`, а для срезов — `nil`. Напомню, что элементы многомерного среза, в свою очередь, являются срезами.

В четвертой части файла `slices.go`, которая содержит следующий код Go, показано, как вручную инициализировать все элементы двумерного среза:

```
for i := 0; i < len(twoD); i++ {
    for j := 0; j < 2; j++ {
        twoD[i] = append(twoD[i], i*j)
    }
}
```

Как видно из этого кода Go, для того чтобы расширить существующий срез и увеличить его размер, недостаточно просто сослаться на несуществующий индекс — нужно явно использовать функцию `append()`! Обращение к несуществующему индексу приведет к ошибке `panic: runtime error: index out of range`. Обратите внимание, что значения элементов среза выбирались произвольно.

В последней части программы показано, как использовать ключевое слово `range` для перебора и вывода на экран всех элементов двумерного среза:

```
for _, x := range twoD {
    for i, y := range x {
```

```
            fmt.Println("i:", i, "value:", y)
        }
        fmt.Println()
    }
}
```

Если выполнить `slices.go`, то получим следующий результат:

```
$ go run slices.go
[1 2 3 4 5]
[0 0]
[]
[-1 -2 -3 -4 -5]
[-1 -2 -3 -4 -5]
[-1 -2 -3 -4 -100]
[0 0 0 0 0]
[[] [] []]
i: 0 value: 0
i: 1 value: 0
i: 0 value: 0
i: 1 value: 1
i: 0 value: 0
i: 1 value: 2
```

Пусть вас не удивляет, что объекты двумерного среза инициализируются значениями `nil` и поэтому выводятся как пустые; это происходит потому, что нулевое значение для типа среза равно `nil`.

Сортировка срезов с помощью sort.Slice()

В этом подразделе вы увидите, как использовать функцию `sort.Slice()`, которая впервые появилась в Go версии 1.8. Это означает, что представленный здесь код, который хранится в файле `sortSlice.go`, не будет работать в более старых версиях Go. Рассмотрим эту программу, разделив ее на три части. Первая часть выглядит так:

```
package main

import (
    "fmt"
    "sort"
)

type aStructure struct {
    person string
    height int
    weight int
}
```

Помимо ожидаемой преамбулы, в этой книге впервые приведено определение структуры Go. Структуры Go подробно рассматриваются в главе 4. Пока просто запомните, что структуры — это типы данных, в которых хранится несколько переменных разных типов.

Вторая часть `sortSlice.go` содержит следующий код Go:

```
func main() {
    mySlice := make([]aStructure, 0)
    mySlice = append(mySlice, aStructure{"Mihalis", 180, 90})
    mySlice = append(mySlice, aStructure{"Bill", 134, 45})
    mySlice = append(mySlice, aStructure{"Marietta", 155, 45})
    mySlice = append(mySlice, aStructure{"Epifanios", 144, 50})
    mySlice = append(mySlice, aStructure{"Athina", 134, 40})

    fmt.Println("0:", mySlice)
```

Здесь создается новый срез с именем `mySlice`, элементами которого являются элементы созданной ранее структуры `aStructure`.

Заключительная часть программы выглядит так:

```
    sort.Slice(mySlice, func(i, j int) bool {
        return mySlice[i].height < mySlice[j].height
    })
    fmt.Println("<:", mySlice)
    sort.Slice(mySlice, func(i, j int) bool {
        return mySlice[i].height > mySlice[j].height
    })
    fmt.Println(">:", mySlice)
}
```

Здесь дважды выполняется сортировка `mySlice`, для чего используются функция `sort.Slice()` и две анонимные функции. Анонимные функции вызываются последовательно в ходе сортировки, с использованием поля `height` структуры `aStructure`.

Обратите внимание, что sort.Slice() изменяет последовательность элементов в срезе в соответствии с функцией сортировки.

Выполнение `sortSlice.go` приведет к следующим результатам:

```
$ go run sortSlice.go
0: [{Mihalis 180 90} {Bill 134 45} {Marietta 155 45} {Epifanios 144 50}
{Athina 134 40}]
<: [{Bill 134 45} {Athina 134 40} {Epifanios 144 50} {Marietta 155 45}
{Mihalis 180 90}]
>: [{Mihalis 180 90} {Marietta 155 45} {Epifanios 144 50} {Bill 134 45}
{Athina 134 40}]
```

Если вы попробуете выполнить `sortSlice.go` на компьютере UNIX с версией Go ранее 1.8, то получите следующее сообщение об ошибке:

```
$ go version
o version go1.3.3 linux/amd64
$ go run sortSlice.go
# command-line-arguments
./sortSlice.go:24: undefined: sort.Slice
./sortSlice.go:28: undefined: sort.Slice
```

Добавление массива к срезу

В этом подразделе показано, как добавить существующий массив к существующему срезу, на примере способа, приведенного в программе `appendArrayToSlice.go`. Рассмотрим эту программу, разделив ее на две части. Первая часть выглядит так:

```
package main

import (
    "fmt"
)

func main() {
    s := []int{1, 2, 3}
    a := [3]int{4, 5, 6}
```

Пока мы только создали и инициализировали срез с именем `s` и массив с именем `a`.

Вторая часть `appendArrayToSlice.go` выглядит так:

```
    ref := a[:]
    fmt.Println("Existing array:\t", ref)
    t := append(s, ref...)
    fmt.Println("New slice:\t", t)
    s = append(s, ref...)
    fmt.Println("Existing slice:\t", s)
    s = append(s, s...)
    fmt.Println("s+s:\t\t", s)
}
```

Здесь два важных момента. Во-первых, мы создаем новый срез с именем `t`, который содержит элементы `a + s`, — добавляем массив `a` к срезу `s` и сохраняем результат в срезе `s`. Таким образом, появляется выбор: сохранить новый срез в существующей переменной среза или нет. Решение зависит главным образом от того, что вы хотите получить.

Второй важный момент: для того чтобы этот код работал, необходимо создать ссылку на существующий массив (`ref := a[:]`). Обратите внимание, как переменная `ref` используется в двух вызовах `append()`: многоточие (`...`) разбивает массив на элементы, которые добавляются к существующему срезу.

Последние два оператора программы показывают, как можно скопировать срез до конца. Для этого тоже следует использовать многоточие (`...`).

Выполнение `appendArrayToSlice.go` приведет к следующим результатам:

```
$ go run appendArrayToSlice.go
Existing array:    [4 5 6]
New slice:         [1 2 3 4 5 6]
Existing slice:    [1 2 3 4 5 6]
s+s:               [1 2 3 4 5 6 1 2 3 4 5 6]
```

Хеш-таблицы Go

Хеш-таблицы, или карты (map) Go — то же, что известные хеш-таблицы, существующие во многих других языках программирования. Основным их преимуществом является способность использовать любой тип данных в качестве индекса, который в этом случае называется *ключом карты* или просто *ключом*. Несмотря на то что хеш-таблицы Go допускают использование в качестве ключей любых типов данных, тип данных, используемый в качестве ключа, должен быть *сопоставимым*, то есть у компилятора Go должна быть возможность отличать один ключ от другого, или, проще говоря, ключи хеш-таблицы должны поддерживать оператор `==`.

К счастью, почти все типы данных сопоставимы. Однако, как можно догадаться, использование типа данных `bool` в качестве ключа хеш-таблицы определенно ограничило бы ее всего двумя значениями. Кроме того, применение в качестве ключей чисел с плавающей точкой может повлечь проблемы, вызванные точностью, принятой на разных машинах и в разных операционных системах.

Как уже отмечалось, карты Go — то же, что и хеш-таблицы. Хорошо, что Go скрывает реализацию хеш-таблицы и, следовательно, ее сложность. Подробнее о том, как создать собственную хеш-таблицу в Go, вы узнаете из главы 5.

Следующий код с использованием функции `make()` позволяет создать пустую хеш-таблицу с ключами типа `string` и значениями типа `int`:

```
iMap = make(map[string]int)
```

Кроме того, используя *литерал хеш-таблицы* следующего вида, можно создать хеш-таблицу, заполненную данными:

```
anotherMap := map[string]int {
    "k1": 12
    "k2": 13
}
```

Чтобы получить доступ к объектам anotherMap, к ним нужно обратиться как к anotherMap["k1"] и anotherMap["k2"]. Чтобы удалить объект хеш-таблицы, следует воспользоваться функцией delete():

```
delete(anotherMap, "k1")
```

Чтобы перебрать все элементы хеш-таблицы, воспользуйтесь следующим способом:

```
for key, value := range iMap {
    fmt.Println(key, value)
}
```

Более подробно использование хеш-таблиц иллюстрирует код Go из файла usingMaps.go. Рассмотрим эту программу, разделив ее на три части. Первая часть содержит следующий код:

```
package main

import (
    "fmt"
)

func main() {

    iMap := make(map[string]int)
    iMap["k1"] = 12
    iMap["k2"] = 13
    fmt.Println("iMap:", iMap)

    anotherMap := map[string]int{
        "k1": 12,
        "k2": 13,
    }
```

Вторая часть usingMaps.go выглядит так:

```
    fmt.Println("anotherMap:", anotherMap)
    delete(anotherMap, "k1")
    delete(anotherMap, "k1")
    delete(anotherMap, "k1")
    fmt.Println("anotherMap:", anotherMap)

    _, ok := iMap["doesItExist"]
    if ok {
        fmt.Println("Exists!")
    } else {
        fmt.Println("Does NOT exist")
    }
```

Здесь представлен метод, который позволяет определить, есть ли в хеш-таблице данный ключ. Это жизненно важный прием, поскольку без него невозможно узнать, содержит ли хеш-таблица требуемую информацию.

К сожалению, если попытаться получить значение несуществующего ключа хеш-таблицы, в итоге получим ноль, что не позволяет определить, был ли результат нулевым, потому что запрошенный ключ не существует или же потому, что элемент с соответствующим ключом существует и действительно равен нулю. Вот почему мы должны использовать запись _, ok для хеш-таблиц.

Кроме того, здесь продемонстрировано использование функции `delete()`. Многократный вызов одного и того же оператора `delete()` ни на что не влияет и не генерирует предупреждающих сообщений.

Последняя часть программы выглядит так:

```
    for key, value := range iMap {
        fmt.Println(key, value)
    }
}
```

Здесь показано, как удобно использовать ключевое слово `range` для хеш-таблиц.

Если выполнить `usingMaps.go`, то получим следующий результат:

```
$ go run usingMaps.go
iMap: map[k1:12 k2:13]
anotherMap: map[k1:12 k2:13]
anotherMap: map[k2:13]
Does NOT exist
k1 12
k2 13
```

Помните, что вы не можете и не должны делать какие-либо предположения относительно последовательности вывода на экран пар «ключ — значение», поскольку эта последовательность является абсолютно случайной.

Запись в хеш-таблицу со значением nil

Следующий код Go будет работать:

```
aMap := map[string]int{}
aMap["test"] = 1
```

Однако следующий код Go работать не будет, поскольку мы присвоили значение `nil` хеш-таблице, которую затем пытаемся использовать:

```
aMap := map[string]int{}
// var aMap map[string]int
aMap = nil
fmt.Println(aMap)
aMap["test"] = 1
```

Сохранив этот код в `failMap.go` и попытавшись его выполнить, вы получите следующее сообщение об ошибке:

```
$ go run failMap.go
map[]
panic: assignment to entry in nil map
...
```

Это означает, что вставить данные в хеш-таблицу со значением `nil` не получится. Однако поиск, удаление, определение длины и использование циклов `range` для хеш-таблицы со значением `nil` не приведут к краху кода.

Когда использовать хеш-таблицы

Хеш-таблицы более универсальны, чем срезы и массивы. Правда, за такую гибкость приходится платить дополнительными вычислительными затратами на реализацию хеш-таблиц Go. Однако встроенные структуры Go очень быстрые, поэтому не бойтесь использовать хеш-таблицы Go, когда это необходимо. Следует помнить, что хеш-таблицы Go очень удобны и позволяют хранить различные типы данных, при этом просты для понимания и быстро работают.

Константы Go

Go поддерживает *константы* — переменные, значения которых нельзя изменять. Константы в Go определяются с помощью ключевого слова `const`.

В общем случае константы являются глобальными переменными, поэтому, если вы обнаружите, что создаете слишком много констант в локальной области видимости, стоит переосмыслить свой подход к созданию программы.

Основным преимуществом использования констант в программах является гарантия того, что их значение не изменится во время выполнения программы.

Значение констант определяется не во время выполнения программы, а на этапе компиляции.

Внутри Go для хранения констант используются логический, строковый и числовой типы данных. Благодаря этому в Go обеспечивается большая гибкость при работе с константами.

Новые константы определяются так:

```
const HEIGHT = 200
```

Обратите внимание, что в Go имена констант не обязательно писать ПРОПИСНЫМИ БУКВАМИ; это лишь мое личное предпочтение.

Кроме того, если вы хотите объявить сразу несколько констант, например, потому, что они логически связаны между собой, можете использовать следующую запись:

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

Обратите внимание, что результаты всех операций между константами компилятор Go также считает константами. Но если константа является частью другого выражения, это уже будет не так.

Теперь для разнообразия рассмотрим следующие три объявления переменных, которые в Go означают одно и то же:

```
s1 := "My String"
var s2 = "My String"
var s3 string = "My String"
```

Однако, поскольку ни в одном из них не используется ключевое слово `const`, ни одна из этих переменных не является константой. Тем не менее, вы вполне можете определить аналогичным образом две константы:

```
const s1 = "My String"
const s2 string = "My String"
```

И `s1`, и `s2` являются константами, однако для `s2` объявлен тип (`string`), что делает это объявление более ограничивающим, чем объявление `s1`. Это связано с тем, что типизированная константа Go должна соответствовать всем строгим правилам типизированной переменной Go. Константа без типа, напротив, не обязательно должна следовать строгим правилам типизированной переменной — другими словами, ее можно более свободно использовать в смешанных выражениях. Кроме того, даже константы без типа имеют тип по умолчанию, который используется только в тех случаях, когда никакая другая информация о типе не доступна. Основное преимущество такого поведения заключается в том, что, если заранее неизвестно, как именно будет использоваться константа, то можно не использовать жесткие ограничения типизированной переменной Go.

Простым примером является определение числовой константы, такое как `const value = 123`. Поскольку вы можете использовать константу `value` во многих выражениях, то явное объявление типа значительно усложнило бы задачу. Рассмотрим следующий код Go:

```
const s1 = 123
const s2 float64 = 123

var v1 float32 = s1 * 12
var v2 float32 = s2 * 12
```

С определением `v1` у компилятора не возникнет проблем; однако код, используемый для определения `v2`, не будет компилироваться, потому что `s2` и `v2` имеют разные типы:

```
$ go run a.go
# command-line-arguments
./a.go:12:6: cannot use s2 * 12 (type float64) as type float32 in assignment
```

Общая рекомендация такова: если вы используете в программе много констант, хорошо бы объединить их в одном пакете или в структуре Go.

Генератор констант iota

Генератор констант iota применяется для объявления последовательности взаимосвязанных значений, в которых используются увеличивающиеся числа, так что не требуется четко указывать каждое из этих чисел.

Большинство понятий, связанных с ключевым словом `const`, в том числе генератор констант iota, проиллюстрированы в файле `constants.go`, который мы рассмотрим, разделив на четыре части.

Первый фрагмент кода `constants.go` выглядит так:

```
package main

import (
    "fmt"
)

type Digit int
type Power2 int

const PI = 3.1415926

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

В этой части — два новых типа с именами `Digit` и `Power2` и четыре новые константы с именами `PI`, `C1`, `C2` и `C3`.

Ключевое слово type предоставляет способ определить новый именованный тип, который применяет тот же базовый тип, что и существующий тип. Это свойство языка в основном применяется для разделения типов, которые могут использовать один и тот же базовый тип данных.

Вторая часть файла `constants.go` содержит следующий код Go:

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

В этой части мы определяем еще одну константу (`s1`), которая используется в выражении (`v1`).

Третья часть программы выглядит так:

```
    const (
        Zero Digit = iota
        One
        Two
        Three
        Four
    )
    fmt.Println(One)
    fmt.Println(Two)
```

Здесь вы видите определение генератора констант iota на основе типа `Digit`, которое эквивалентно следующему объявлению четырех констант:

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

Последняя часть `constants.go` выглядит так:

```
    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)
}
```

Есть еще один генератор констант iota, который немного отличается от предыдущего. Во-первых, здесь показано использование символа подчеркивания в блоке `const` с генератором констант iota, что позволяет пропускать ненужные значения. Во-вторых, значение `iota` всегда увеличивается и может использоваться в выражениях, что и происходит в данном случае.

Теперь посмотрим, что на самом деле происходит внутри блока `const`. Для `p2_0` значение `iota` равно `0`, а `p2_0` определяется как `1`. Для `p2_2` значение `iota` равно `2`, а `p2_2` определяется как результат выражения `1 << 2`, которое в двоичном представлении равно `00000100`. Десятичное значение `00000100` равно `4`, что и является результатом и значением `p2_2`. Аналогично значение `p2_4` равно `16`, а значение `p2_6` — `32`.

Как видно, использование `iota` экономит время, если оно соответствует вашим потребностям.

Выполнение программы `constants.go` приводит к следующим результатам:

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

Указатели в Go

Go поддерживает *указатели* — адреса памяти, что обеспечивает повышение скорости в обмен на сложный для отладки код и неприятные ошибки. Хотите узнать об этом больше — спросите любого знакомого, кто программирует на C.

Пример использования указателей приведен в главе 2, когда речь шла о небезопасном коде и пакете `unsafe`, а также сборщике мусора Go. В данном разделе эта сложная тема раскрыта глубже. Кроме того, нативные указатели Go безопасны — при условии, что вы знаете, что делаете.

При работе с указателями, если вы хотите получить значение указателя, необходимо использовать операцию `*`, которая называется *разыменованием указателя*, и оператор `&`, чтобы получить адрес памяти переменной.

Разработчикам-любителям следует использовать указатели только тогда, когда этого требуют библиотеки, которыми они пользуются. При небрежном использовании указатели могут стать причиной ужасных и труднообнаруживаемых ошибок.

Чтобы функция принимала указатель в качестве параметра, нужно написать следующий код:

```
func getPointer(n *int) {
}
```

Аналогичным образом функция может возвращать указатель:

```
func returnPointer(n int) *int {
}
```

Использование безопасных указателей Go проиллюстрировано на примере программы `pointers.go`, которую мы рассмотрим, разделив на четыре части. Первый фрагмент кода `pointers.go` выглядит так:

```
package main

import (
    "fmt"
)

func getPointer(n *int) {
    *n = *n * *n
}

func returnPointer(n int) *int {
    v := n * n
    return &v
}
```

С одной стороны, хорошо, что функция `getPointer()` позволяет изменять переданную ей переменную без необходимости возвращать что-либо из функции. Это происходит потому, что указатель, переданный в качестве параметра, содержит адрес памяти этой переменной.

С другой стороны, `returnPointer()` получает целочисленный параметр и возвращает указатель на целое число, что обозначается как `return &v`. Это может показаться не очень полезным, однако вы по-настоящему оцените данную возможность в главе 4, где пойдет речь об указателях на структуры Go, а также в последующих главах, где будут рассматриваться более сложные структуры данных.

Обе функции, `getPointer()` и `returnPointer()`, вычисляют квадрат целого числа, однако используют совершенно разные подходы: `getPointer()` сохраняет результат в переданном параметре, а `returnPointer()` возвращает результат, который должен быть сохранен в другой переменной.

Вторая часть программы содержит следующий код Go:

```
func main() {
    i := -10
    j := 25

    pI := &i
    pJ := &j

    fmt.Println("pI memory:", pI)
    fmt.Println("pI memory:", pJ)
    fmt.Println("pI value:", *pI)
    fmt.Println("pI value:", *pJ)
```

Здесь `i` и `j` — обычные целочисленные переменные, а `pI` и `pJ` — указатели, ссылающиеся на `i` и `j` соответственно. `pI` — это адрес памяти, а `*pI` — значение, сохраненное в памяти по этому адресу.

Третья часть `pointers.go` выглядит так:

```
    *pI = 123456
    *pI--
    fmt.Println("i:", i)
```

Здесь показано, как можно изменить переменную `i` с помощью указателя `pI`, который ссылается на `i`, двумя разными способами: во-первых, путем непосредственного присвоения нового значения и, во-вторых, с помощью оператора `--`.

Последняя часть кода `pointers.go` содержит следующий код Go:

```
    getPointer(pJ)
    fmt.Println("j:", j)
    k := returnPointer(12)
    fmt.Println(*k)
    fmt.Println(k)
}
```

Здесь вызывается функция `getPointer()`, которой в качестве параметра передается `pJ`. Как мы уже говорили, любые изменения, внесенные в переменную, на которую указывает `pJ` внутри `getPointer()`, будут влиять на значение переменной `j`, что и доказывает вывод с помощью оператора `fmt.Println("j:", j)`. Вызов `returnPointer()` возвращает указатель, присвоенный переменной-указателю `k`.

Запуск `pointers.go` приводит к следующим результатам:

```
$ go run pointers.go
pI memory: 0xc0000160b0
pJ memory: 0xc0000160b8
pI value: -10
pJ value: 25
i: 123455
j: 625
144
0xc0000160f0
```

Я не удивлюсь, если у вас возникнут проблемы с пониманием кода Go из `pointers.go`: ведь мы еще не обсуждали функции и определения функций. Не поленитесь заглянуть в главу 6, где более подробно объясняются темы, связанные с функциями.

Обратите внимание, что строки в Go являются переменными, хранящими значение, а не указателями, как в C.

Зачем нужны указатели

Есть две основные причины использовать указатели в ваших программах:

- указатели позволяют обмениваться данными по ссылке, а не по значению, особенно между функциями Go;
- указатели бывают чрезвычайно полезны, если вы хотите отличить нулевое значение от неприсвоенного.

Время и дата

В этом разделе вы узнаете, как анализировать строки времени и даты в Go, как преобразовывать время и дату в различные форматы и как выводить время и дату в желаемом формате. На первый взгляд эта задача может показаться не важной, однако она может оказаться действительно критической, если требуется синхронизировать несколько задач или если приложение должно прочитать дату из одного или нескольких текстовых файлов либо получить ее непосредственно от пользователя.

Центром работы с временем и датами в Go является пакет `time`; в этом разделе будет продемонстрировано использование некоторых из его функций.

Прежде чем учиться анализировать текстовую строку и преобразовывать ее во время или дату, мы рассмотрим простую программу `usingTime.go`, которая даст нам общее представление о пакете `time`. Для изучения этой программы мы разделим ее на три части. Первая часть выглядит так:

```
package main

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

Второй фрагмент кода `usingTime.go` содержит следующий код Go:

```
func main() {
    fmt.Println("Epoch time:", time.Now().Unix())
    t := time.Now()
```

```
    fmt.Println(t, t.Format(time.RFC3339))
    fmt.Println(t.Weekday(), t.Day(), t.Month(), t.Year())

    time.Sleep(time.Second)
    t1 := time.Now()
    fmt.Println("Time difference:", t1.Sub(t))
```

Функция `time.Now().Unix()` возвращает *время эпохи* UNIX — количество секунд, прошедших с 00:00:00 UTC 1 января 1970 года. Функция `Format()` позволяет преобразовывать переменную типа `time` в другой формат, в данном случае в формат `RFC3339`.

В этой книге нам много раз встретится функция `time.Sleep()`, позволяющая эмулировать задержку выполнения какой-либо другой функции. Константа `time.Second` в Go определяет интервал 1 секунда. Для того чтобы задать интервал 10 секунд, нужно умножить `time.Second` на 10. В Go есть еще несколько подобных констант, в том числе `time.Nanosecond`, `time.Microsecond`, `time.Millisecond`, `time.Minute` и `time.Hour`. Таким образом, минимальный временной интервал, который можно определить с помощью пакета `time`, — это наносекунда. Наконец, функция `time.Sub()` позволяет вычислить разность между двумя значениями времени.

Последняя часть программы выглядит так:

```
    formatT := t.Format("01 January 2006")
    fmt.Println(formatT)
    loc, _ := time.LoadLocation("Europe/Paris")
    londonTime := t.In(loc)
    fmt.Println("Paris:", londonTime)
}
```

Здесь с помощью `time.Format()` определяется новый формат даты, который можно будет использовать для вывода переменной типа `time`.

Выполнение программы `usingTime.go` приведет к следующим результатам:

```
$ go run usingTime.go
Epoch time: 1548753515
2019-01-29 11:18:35.01478 +0200 EET m=+0.000339641
2019-01-29T11:18:35+02:00
Tuesday 29 January 2019
Time difference: 1.000374985s
01 January 2019
Paris: 2019-01-29 10:18:35.01478 +0100 CET
```

Теперь, когда мы познакомились с основами пакета `time`, пора углубиться в его функциональность, начиная с работы с временем.

Работа с временем

Если у нас есть переменная типа `time`, ее можно легко преобразовать в любой формат, связанный с временем или датой. Однако основная проблема возникает, когда у вас есть исходная строка и вы хотите проверить, является ли она допустимым значением

времени. Функция, используемая для синтаксического анализа строк, содержащих значения времени и даты, называется `time.Parse()` и принимает два параметра. Первый из них определяет ожидаемый формат анализируемой строки, а второй содержит саму строку, которая должна быть проанализирована. Первый параметр состоит из элементов, принадлежащих списку констант Go, связанных с анализом даты и времени.

Список констант, которые можно использовать для создания формата синтаксического анализа, находится по адресу https://golang.org/src/time/format.go. Go не определяет формат даты или времени в форме наподобие DDYYYYMM или %D %Y %M, как другие языки программирования, а использует собственный подход. Поначалу этот подход может показаться странным, однако впоследствии вы наверняка оцените его по достоинству, так как он предостерегает разработчика от глупых ошибок.

Константы Go для работы со временем равны `15` для часа, `04` для минут и `05` для секунд. Легко догадаться, что все эти числовые значения должны быть уникальными. Также можно использовать строки `PM` и `pm` (в верхнем или нижнем регистре) для указания на время после полудня.

Обратите внимание, что вы не обязаны использовать все доступные константы Go для работы со временем. Основная задача разработчика — разместить эти константы Go в желаемом порядке в соответствии с типом строк, которые будет обрабатывать программа. Окончательную версию строки из этих констант, которая будет передана функции `time.Parse()` в качестве первого параметра, можно рассматривать как *регулярное выражение*.

Синтаксический анализ времени

В этом разделе будет показано, как анализировать строку, которая передается в программу `parseTime.go` в качестве аргумента командной строки, чтобы преобразовать ее в переменную типа `time`. Однако это не всегда возможно, поскольку входная строка может иметь неправильный формат или содержать недопустимые символы. Мы рассмотрим утилиту `parseTime.go`, разделив ее на три части.

Первый фрагмент кода `parseTime.go` выглядит так:

```
package main

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

Вторая часть содержит следующий код:

```
func main() {
    var myTime string
    if len(os.Args) != 2 {
        fmt.Printf("usage: %s string\n", filepath.Base(os.Args[0]))
        os.Exit(1)
    }

    myTime = os.Args[1]
```

Последняя часть `parseTime.go`, в которой и происходит все волшебство, выглядит так:

```
    d, err := time.Parse("15:04", myTime)
    if err == nil {
        fmt.Println("Full:", d)
        fmt.Println("Time:", d.Hour(), d.Minute())
    } else {
        fmt.Println(err)
    }
}
```

Чтобы выполнить синтаксический анализ текстовой строки, содержащей часы и минуты, нужно использовать формат `"15:04"`. Значение переменной `err` говорит о том, был ли анализ успешным.

Выполнение `parseTime.go` приводит к следующим результатам:

```
$ go run parseTime.go
usage: parseTime string
exit status 1
$ go run parseTime.go 12:10
Full: 0000-01-01 12:10:00 +0000 UTC
Time: 12 10
```

Как мы видим, Go выводит строку даты и времени полностью, поскольку именно это хранится в переменной `time`. Если вас интересует только время, без даты, то нужно вывести соответствующие части переменной `time`.

Если при попытке проанализировать строку и преобразовать ее в тип `time` использовать неправильную константу Go, например `22:04`, то получим следующее сообщение об ошибке:

```
$ go run parseTime.go 12:10
parsing time "12:10" as "22:04": cannot parse ":10" as "2"
```

Но если использовать, например, константу Go `11`, предназначенную для анализа, где месяц указан как число, сообщение об ошибке будет немного другим:

```
$ go run parseTime.go 12:10
parsing time "12:10": month out of range
```

Работа с датами

В этом подразделе будет показано, как анализировать строки, обозначающие даты в Go, — для этого нам снова понадобится функция `time.Parse()`.

Константы Go для работы с датами — это `Jan` для анализа трехбуквенной аббревиатуры, используемой для описания месяца, `2006` для анализа года и `02` — для анализа дня месяца. Если вместо `Jan` использовать `January`, то, как и следовало ожидать, вместо трехбуквенного сокращения получим полное название месяца.

Кроме того, можно использовать константу `Monday` для анализа строк, которые содержат полное название дня недели, или `Mon` для сокращенного названия дня недели.

Синтаксический анализ дат

В этом подразделе мы разработаем программу Go под названием `parseDate.go`, которую рассмотрим, разделив на две части.

Первая часть `parseDate.go` содержит следующий код:

```
package main

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

func main() {

    var myDate string
    if len(os.Args) != 2 {
        fmt.Printf("usage: %s string\n", filepath.Base(os.Args[0]))
        return
    }
    myDate = os.Args[1]
```

Вторая часть `parseDate.go` содержит следующий код Go:

```
    d, err := time.Parse("02 January 2006", myDate)
    if err == nil {
        fmt.Println("Full:", d)
        fmt.Println("Time:", d.Day(), d.Month(), d.Year())
    } else {
        fmt.Println(err)
    }
}
```

Если между названием месяца и годом стоит символ `-`, то вместо `"02 January-2006"` в качестве первого параметра для `time.Parse()` можно использовать `"02 January 2006"`.

Выполнение `parseDate.go` приведет к следующим результатам:

```
$ go run parseDate.go
usage: parseDate string
$ go run parseDate.go "20 July 2000"
Full: 2000-07-20 00:00:00 +0000 UTC
Time: 20 July 2000
```

Поскольку `parseDate.go` не ожидает данных о времени, в конец полной строки даты и времени автоматически добавляется строка `00:00:00 +0000 UTC`.

Изменение формата даты и времени

В этом разделе будет показано, как изменить формат строки, содержащей как дату, так и время. Чаще всего такие строки встречаются в файлах журналов веб-серверов, таких как Apache и Nginx. Поскольку мы пока не умеем построчно читать текстовые файлы, текст будет жестко закодирован в программе; однако это не повлияет на ее функциональность.

Мы рассмотрим код Go программы `timeDate.go`, разделив его на четыре части. Первая часть — ожидаемая преамбула:

```
package main

import (
    "fmt"
    "regexp"
    "time"
)
```

Стандартный пакет Go `regexp` необходим для поддержки регулярных выражений.

Вторая часть кода `timeDate.go` выглядит следующим образом:

```
func main() {

    logs := []string{"127.0.0.1 - - [16/Nov/2017:10:49:46 +0200] 325504",
                     "127.0.0.1 - - [16/Nov/2017:10:16:41 +0200]
                     \"GET /CVEN HTTP/1.1\" 200 12531 \"-\"
                     \"Mozilla/5.0 AppleWebKit/537.36",
                     "127.0.0.1 200 9412 - - [12/Nov/2017:06:26:05 +0200]
                     \"GET \"http://www.mtsoukalos.eu/taxonomy/term/47\" 1507",
                     "[12/Nov/2017:16:27:21 +0300]", "[12/Nov/2017:20:88:21 +0200]",
                     "[12/Nov/2017:20:21 +0200]",
}
```

Поскольку мы не можем быть уверенными в данных и их формате, в этой программе представлены различные примеры данных, чтобы охватить множество различных случаев, включая неполные данные, такие как `[12/Nov/2017:20:21 +0200]`, где в разделе времени нет секунд, и ошибочные данные, такие как `[12/Nov/2017:20:88:21 +0200]`, где значение минут равно 88.

Третья часть `timeDate.go` содержит следующий код Go:

```
for _, logEntry := range logs {
    r := regexp.MustCompile(`.*\[(\d\d\/\w+/\d\d\d\d:\d\d:\d\d:\d\d.*)\].*`)
    if r.MatchString(logEntry) {
        match := r.FindStringSubmatch(logEntry)
```

Основное преимущество, которое мы получаем от использования такого трудночитаемого регулярного выражения, в данной программе заключается в том, что оно позволяет выяснить, есть где-то в исходной строке строка даты и времени или же нет.

После того как мы получим эту строку, мы передадим ее в `time.Parse()`, и пускай `time.Parse()` выполнит остальную часть работы.

Последняя часть программы содержит следующий код Go:

```
            dt, err := time.Parse("02/Jan/2006:15:04:05 -0700", match[1])
            if err == nil {
                newFormat := dt.Format(time.RFC850)
                fmt.Println(newFormat)
            } else {
                fmt.Println("Not a valid date time format!")
            }
        } else {
            fmt.Println("Not a match!")
        }
    }
}
```

Как только мы найдем строку, которая соответствует регулярному выражению, мы проанализируем ее, используя `time.Parse()`, чтобы убедиться, что это допустимая строка даты и времени. Если это так, то `timeDate.go` выведет дату и время в *формате RFC850*.

Если выполнить `timeDate.go`, то получим следующий результат:

```
$ go run timeDate.go
Thursday, 16-Nov-17 10:49:46 EET
Thursday, 16-Nov-17 10:16:41 EET
Sunday, 12-Nov-17 06:26:05 EET
Sunday, 12-Nov-17 16:27:21 +0300
Not a valid date time format!
Not a match!
```

Измерение времени выполнения программы

В этом разделе будет показано, как измерить время выполнения одной или нескольких команд в Go. Та же методика может применяться для измерения времени выполнения функции или группы функций. В качестве примера мы рассмотрим программу Go `execTime.go`, которую разделим на три части.

Это простой, но тем не менее очень мощный и удобный прием. Не стоит недооценивать простоту Go.

Первая часть `execTime.go` выглядит следующим образом:

```
package main

import (
    "fmt"
    "time"
)

func main() {
    start := time.Now()
    time.Sleep(time.Second)
    duration := time.Since(start)
    fmt.Println("It took time.Sleep(1)", duration, "to finish.")
```

Чтобы измерить время выполнения команды, нам потребуется функциональность пакета `time`. Вся работа выполняется функцией `time.Since()`. Эта функция принимает один аргумент, который должен означать момент времени в прошлом. В данном случае будет измерено время, необходимое `Go` для выполнения `time.Sleep(time.Second)`, поскольку это единственный оператор, стоящий между `time.Now()` и `time.Since()`.

Вторая часть `execTime.go` имеет следующий вид:

```
    start = time.Now()
    time.Sleep(2 * time.Second)
    duration = time.Since(start)
    fmt.Println("It took time.Sleep(2)", duration, "to finish.")
```

На этот раз мы измеряем время, необходимое Go для выполнения `time.Sleep(2 * time.Second)`. Это может быть очень полезно для определения точности функции `time.Sleep()`, точность которой главным образом определяется точностью внутренних часов Go.

Последняя часть `execTime.go` выглядит следующим образом:

```
    start = time.Now()
    for i := 0; i < 200000000; i++ {
        _ = i
    }
```

```
    duration = time.Since(start)
    fmt.Println("It took the for loop", duration, "to finish.")

    sum := 0
    start = time.Now()
    for i := 0; i < 200000000; i++ {
        sum += i
    }
    duration = time.Since(start)
    fmt.Println("It took the for loop", duration, "to finish.")
```

В этой последней части программы измеряется скорость выполнения двух циклов. Первый цикл не делает ничего, тогда как второй выполняет некоторые вычисления. Как мы увидим в результатах работы программы, второй цикл `for` выполняется быстрее первого.

При выполнении `execTime.go` получим следующие результаты:

```
$ go run execTime.go
It took time.Sleep(1) 1.000768881s to finish.
It took time.Sleep(2) 2.00062487s to finish.
It took the for loop 50.497931ms to finish.
It took the for loop 47.70599ms to finish.
```

Измерение скорости работы сборщика мусора Go

Теперь мы можем переписать программы `sliceGC.go`, `mapNoStar.go`, `mapStar.go` и `mapSplit.go` из предыдущей главы и получить более точные результаты без необходимости использовать утилиту командной строки UNIX `time(1)`. В сущности, единственное, что нужно сделать в каждом из этих файлов, — это вставить строки `time.Now()` и `time.Since()` с `runtime.GC()` между ними и вывести результаты. Обновленные версии `sliceGC.go`, `mapNoStar.go`, `mapStar.go` и `mapSplit.go` будут называться `sliceGCTime.go`, `mapNoStarTime.go`, `mapStarTime.go` и `mapSplitTime.go` соответственно.

Выполнение этих обновленных версий приведет к следующим результатам:

```
$ go run sliceGCTime.go
It took GC() 281.563µs to finish
$ go run mapNoStarTime.go
It took GC() 9.483966ms to finish
$ go run mapStarTime.go
It took GC() 651.704424ms to finish
$ go run mapSplitTime.go
It took GC() 12.743889ms to finish
```

Эти результаты гораздо точнее, чем предыдущие, так как они показывают только время, затраченное `runtime.GC()`, без учета времени, которое потребовалось программе для заполнения срезов и хеш-таблиц, используемых для хранения значений. Тем не менее, результаты по-прежнему подтверждают наши выводы о том, насколько медленно сборщик мусора Go работает с хеш-таблицами, в которых хранится большое количество данных.

Веб-ссылки и упражнения

- Напишите генератор константы iota для дней недели.
- Напишите программу Go, которая бы преобразовывала заданный массив в хеш-таблицу.
- Посетите страницу документации пакета `time` по адресу https://golang.org/pkg/time/.
- Сможете ли вы написать генератор константы iota для степеней числа четыре?

Советую также посетить страницу GitHub, где обсуждаются Go 2 и изменения числовых литералов. Это поможет вам понять, как происходят изменения в Go: https://github.com/golang/proposal/blob/master/design/19308-number-literals.md.

- Напишите собственную версию `parseDate.go`.
- Напишите собственную версию `parseTime.go`. Не забудьте протестировать вашу программу.
- Сможете ли вы создать версию `timeDate.go`, которая бы обрабатывала два формата даты и времени?

Резюме

В этой главе раскрыты многие интересные темы, касающиеся Go, в том числе числовые типы данных, хеш-таблицы, массивы и срезы, а также указатели Go, константы и циклы Go. Кроме того, вы узнали, как Go работает с датами и временем. Теперь вы должны понимать, чем срезы лучше массивов.

Следующая глава посвящена созданию и использованию составных типов в Go — сюда главным образом входят типы, которые создаются с помощью ключевого слова `struct` и называются *структурами*. Затем мы обсудим строковые переменные и *кортежи*.

Помимо этого в следующей главе вы узнаете о *регулярных выражениях* и *сопоставлении с образцом* — это сложные темы не только для Go, но и для других языков программирования. Однако при правильном и аккуратном использовании регулярные выражения и сопоставление с образцом настолько упрощают жизнь разработчика, что имеет смысл узнать о них больше.

JSON — это очень популярный текстовый формат, поэтому в следующей главе мы также обсудим, как создавать, импортировать и экспортировать в Go данные JSON.

Наконец, вы познакомитесь с ключевым словом `switch` и пакетом `strings`, который позволяет манипулировать строками в формате *UTF-8*.

4 Использование составных типов данных

В предыдущей главе рассмотрены многие основные темы Go, включая числовые типы данных, массивы, срезы, хеш-таблицы, указатели, константы, цикл `for`, ключевое слово `range`, а также способы работы с временем и датами.

В этой главе речь пойдет о расширенных свойствах Go, таких как *кортежи* и *строки*, стандартном пакете Go `strings` и операторе `switch`. Также, что наиболее важно, мы рассмотрим широко используемые в Go *структуры*.

Кроме того, в данной главе рассмотрены вопросы работы с текстовыми файлами форматов *JavaScript Object Notation (JSON)* и *Extensible Markup Language (XML)*, реализации простого хранилища данных типа *«ключ — значение»*, создания *регулярных выражений* и выполнения *сопоставления с образцом* в Go.

В этой главе будут раскрыты следующие темы:

- структуры и ключевое слово `struct` в Go;
- кортежи в Go;
- строки, руны и строковые литералы;
- работа с текстовым форматом JSON;
- работа с текстовым форматом XML;
- регулярные выражения в Go;
- сопоставление с образцом в Go;
- оператор `switch`;
- возможности пакета `strings`;
- вычисление числа π с высокой точностью;
- разработка хранилища типа «ключ — значение».

Составные типы данных

Несмотря на то что стандартные типы Go весьма удобны, быстры и гибки, они, как правило, не охватывают все типы данных, которые вам могут потребоваться в программе Go. Go решает эту проблему путем реализации структур, которые представляют собой пользовательские типы данных, создаваемые разработчиком. Кроме того, в Go есть собственный способ поддержки кортежей, главная задача которого — позволить функциям возвращать несколько значений без необходимости группировать их в структуры, как это реализовано в C.

Структуры

Массивы, срезы и хеш-таблицы, безусловно, очень полезны, однако они не позволяют группировать и хранить в одном месте несколько значений. Если нужно создать новый удобный тип, сгруппировав в нем несколько переменных разных типов, можно использовать структуру. Элементы структуры называются *полями структуры* или просто полями.

Рассмотрим для начала простую структуру, которая уже встречалась в файле `sortSlice.go` в предыдущей главе:

```
type aStructure struct {
    person string
    height int
    weight int
}
```

По причинам, которые станут понятны в главе 6, имена полей структуры обычно начинаются с заглавной буквы — это в основном зависит от того, что вы хотите делать с этими полями. Данная структура имеет три поля с именами `person`, `height` и `weight` соответственно. Теперь можно создать новую переменную типа `aStructure`:

```
var s1 aStructure
```

Кроме того, можно получить доступ к определенному полю структуры по его имени. Так, чтобы получить значение поля `person` переменной `s1`, нужно ввести `s1.person`.

Можно также определить *литерал структуры*:

```
p1 := aStructure{"fmt", 12, -2}
```

Но, поскольку запомнить последовательность полей структуры иногда очень сложно, Go позволяет использовать для определения литерала структуры другую форму:

```
p1 := aStructure{weight: 12, height: -2}
```

В этом случае не нужно определять начальное значение для каждого поля структуры.

Теперь, когда вы усвоили основы конструкций, пора рассмотреть более практичный пример. Он хранится в файле `structures.go`, и мы рассмотрим его, разбив на четыре части.

Первая часть `structures.go` содержит следующий код:

```
package main

import (
    "fmt"
)
```

Типы Go и структуры в частности, как правило, определяются вне функции main(), чтобы они относились к глобальной области видимости и были доступны для всего пакета Go, если только вы не хотите специально подчеркнуть, что данный тип полезен лишь в пределах текущей области видимости и его не предполагается использовать в других местах.

Второй фрагмент `structures.go` содержит следующий код Go:

```
func main() {

    type XYZ struct {
        X int
        Y int
        Z int
    }

    var s1 XYZ
    fmt.Println(s1.Y, s1.Z)
```

Как видим, ничто не мешает нам определить новую структуру внутри функции, но должна быть причина для этого.

Третья часть `structures.go` выглядит так:

```
    p1 := XYZ{23, 12, -2}
    p2 := XYZ{Z: 12, Y: 13}
    fmt.Println(p1)
    fmt.Println(p2)
```

Здесь определяются два литерала структур с именами `p1` и `p2`, которые затем выводятся на экран.

Последняя часть `structures.go` содержит следующий код Go:

```
    pSlice := [4]XYZ{}
    pSlice[2] = p1
    pSlice[0] = p2
    fmt.Println(pSlice)
    p2 = XYZ{1, 2, 3}
    fmt.Println(pSlice)
}
```

В этой части мы создали массив структур с именем `pSlice`. Как станет ясно из результатов работы `structures.go`, если присвоить структуру элементу массива, эта структура копируется в массив, поэтому изменение значения исходной структуры не повлияет на объекты массива.

Выполнение `structures.go` приводит к следующим результатам:

```
$ go run structures.go
0 0
{23 12 -2}
{0 13 12}
[{0 13 12} {0 0 0} {23 12 -2} {0 0 0}]
[{0 13 12} {0 0 0} {23 12 -2} {0 0 0}]
```

Обратите внимание, что последовательность, в которой вы добавляли поля в определение типа структуры, имеет значение для идентификации типа этой структуры. Проще говоря, две структуры с одинаковыми полями в Go не будут считаться одинаковыми, если эти поля определены у них в разном порядке.

Как видно из результатов работы `structures.go`, нулевое значение переменной типа `struct` создается путем обнуления всех полей этой переменной в соответствии с их типами.

Указатели на структуры

В главе 3 уже говорилось об указателях. В этом разделе рассмотрим пример, связанный с указателями на структуры. Программа называется `pointerStruct.go`, и мы ее изучим, разделив на четыре части.

Первая часть программы содержит следующий код Go:

```
package main

import (
    "fmt"
)

type myStructure struct {
    Name     string
    Surname  string
    Height   int32
}
```

Второй фрагмент кода `pointerStruct.go` выглядит так:

```
func createStruct(n, s string, h int32) *myStructure {
    if h > 300 {
        h = 0
```

```
    }
    return &myStructure{n, s, h}
}
```

Подход, использованный в createStruct() для создания новой структурной переменной, имеет много преимуществ по сравнению с самостоятельной инициализацией структурных переменных. Одно из этих преимуществ — возможность проверить, является ли предоставленная информация правильной и достоверной. Кроме того, данный подход более понятен — есть центральная точка, в которой инициализируются переменные типа struct. Если со структурными переменными что-то пойдет не так, вы знаете, где искать ошибку и кого винить! Обратите внимание, что некоторые предпочли бы назвать такую функцию не createStruct(), а NewStruct().

Для тех, у кого есть опыт работы на C или C++, вполне естественно, что функция Go возвращает адрес памяти локальной переменной. Ничто не теряется, все счастливы.

Третья часть pointerStruct.go выглядит так:

```
func retStructure(n, s string, h int32) myStructure {
    if h > 300 {
        h = 0
    }
    return myStructure{n, s, h}
}
```

В этой части представлена функция retStructure() — то же самое, что createStruct(), но без указателя. Обе функции работают нормально, поэтому выбор между createStruct() и retStructure() — вопрос личных предпочтений. Более подходящие имена для этих двух функций могли бы быть NewStructurePointer() и NewStructure() соответственно.

Последняя часть pointerStruct.go содержит следующий код Go:

```
func main() {
    s1 := createStruct("Mihalis", "Tsoukalos", 123)
    s2 := retStructure("Mihalis", "Tsoukalos", 123)

    fmt.Println((*s1).Name)
    fmt.Println(s2.Name)
    fmt.Println(s1)
    fmt.Println(s2)
}
```

Если выполнить pointerStruct.go, то получим следующий результат:

```
$ go run pointerStruct.go
Mihalis
Mihalis
```

```
&{Mihalis Tsoukalos 123}
{Mihalis Tsoukalos 123}
```

Как вы видите, главное различие между `createStruct()` и `retStructure()` состоит в том, что первая функция возвращает указатель на структуру, следовательно, вам нужно разыменовать этот указатель, чтобы использовать объект, на который он указывает, из-за чего код может быть несколько некрасивым, тогда как вторая функция возвращает весь объект структуры.

Структуры очень важны в Go и широко используются в реальных программах, так как позволяют группировать произвольное количество значений и обрабатывать эти значения как единое целое.

Ключевое слово new

Go поддерживает ключевое слово `new`, которое позволяет размещать в памяти новые объекты. Однако, используя `new`, нужно помнить очень важную деталь: `new` возвращает адрес памяти выделенного объекта. Проще говоря, `new` возвращает указатель.

Итак, можно создать переменную `aStructure` следующим образом:

```
pS := new(aStructure)
```

После того как выполнится оператор `new`, вы готовы работать с новой переменной, для которой выделена память, и эта память уже обнулена (значения переменных структуры установлены в нулевые значения), но не инициализирована.

Основное различие между new и make состоит в том, что переменные, созданные с помощью make, правильно инициализируются, а не только обнуляется выделенная для них память. Кроме того, make можно применять только к хеш-таблицам, каналам и срезам, и эта функция не возвращает адрес памяти, то есть make не возвращает указатель.

В следующей строке кода с помощью `new` создается срез, указывающий на `nil`[1]:

```
sP := new([]aStructure)
```

---

[1] В этом примере команда new([]aStructure) выделяет в памяти место под поля описания среза (срез в Go — это структура), но так как поля описания среза не инициированы, возвращается указатель на nil. Чтобы в дальнейшем использовать переменную sP, нужно инициализировать срез, выполнив команду make, например, *sP = make([]aStructure, 10).

Кортежи

Строго говоря, *кортеж* — это конечный упорядоченный список, состоящий из нескольких частей. Главное в кортежах то, что Go не поддерживает тип кортежа и, следовательно, официально не обслуживает кортежи, несмотря на то что поддерживает определенные способы использования кортежей.

Интересный момент: в этой книге мы уже использовали кортежи Go, начиная с главы 1, в выражениях наподобие следующего, где функция возвращает два значения в виде одного выражения:

```
min, _ := strconv.ParseFloat(arguments[1], 64)
```

Мы рассмотрим кортежи Go на примере программы, которая называется `tuples.go`, разделив ее для удобства на три фрагмента кода. Обратите внимание, что в представленном коде используется функция, которая возвращает три значения в виде кортежа. Подробнее о функциях вы узнаете из главы 6.

Первая часть `tuples.go` выглядит так:

```
package main

import (
    "fmt"
)

func retThree(x int) (int, int, int) {
    return 2 * x, x * x, -x
}
```

Здесь вы видите реализацию функции `retThree()`, которая возвращает кортеж, содержащий три целочисленных значения. Такая возможность позволяет функции возвращать несколько значений, не требуя группировать различные возвращаемые значения в структуру и возвращать структурную переменную.

В главе 6 вы узнаете, как помещать имена в возвращаемые значения функции Go — очень удобное свойство, способное избавить вас от возможных ошибок.

Вторая часть `tuples.go` выглядит так:

```
func main() {
    fmt.Println(retThree(10))
    n1, n2, n3 := retThree(20)
    fmt.Println(n1, n2, n3)
```

Здесь функция `retThree()` используется дважды: в первый раз без сохранения возвращаемых значений, а во второй — с сохранением трех возвращаемых значений в трех разных переменных, используя один оператор, который в терминологии Go

называется *присваиванием кортежей*. Последнее иногда вводит в заблуждение, заставляя думать, что Go поддерживает кортежи.

Если вас интересуют лишь некоторые из значений, возвращаемых функцией, вы можете поставить вместо не интересующих вас значений *символ подчеркивания* (_). Обратите внимание, что если объявить переменную в Go и затем не использовать ее, то это приведет к ошибке компиляции.

Третья часть программы содержит следующий код Go:

```
    n1, n2 = n2, n1
    fmt.Println(n1, n2, n3)

    x1, x2, x3 := n1*2, n1*n1, -n1
    fmt.Println(x1, x2, x3)
}
```

Как видим, кортежи позволяют делать много умных вещей, таких как обмен значениями без необходимости создавать временную переменную, а также вычисление выражений.

Выполнение `tuples.go` приведет к следующим результатам:

```
$ go run tuples.go
20 100 -10
40 400 -20
400 40 -20
800 160000 -400
```

Регулярные выражения и сопоставление с образцом

Сопоставление с образцом, которое играет ключевую роль в Go, — это метод поиска в строке некоторого набора символов на основе определенного шаблона поиска, основанного на регулярных выражениях и *грамматиках*. Если сопоставление с образцом прошло успешно, то вы сможете извлечь из строки нужные данные, заменить или удалить их.

Пакет Go, отвечающий за определение регулярных выражений и сопоставление с образцом, называется `regexp`. Примеры его использования вы увидите далее в этой главе.

Используя в коде регулярные выражения, следует рассматривать определение регулярного выражения как самую важную часть соответствующего кода, поскольку именно от регулярного выражения зависит функциональность данного кода.

Немного теории

Регулярные выражения компилируются в распознаватель путем построения обобщенной диаграммы переходов, называемой *конечным автоматом*. Конечные автоматы бывают одного из двух видов: детерминированные или недетерминированные. В недетерминированных конечных автоматах у одного и того же набора входных данных может существовать несколько вариантов перехода из этого состояния. *Распознаватель* — это программа, которая принимает на входе строку *x* и определяет, является ли *x* предложением из заданного языка.

Грамматика — это набор порождающих правил, представленных на формальном языке. Порождающие правила описывают, как создавать строки из алфавита языка в соответствии с синтаксисом этого языка. Грамматика не описывает значение строки или того, что можно с ней сделать в каком-либо контексте, — она описывает лишь форму строки. Здесь важно понимать, что грамматики лежат в основе регулярных выражений, потому что без грамматики невозможно определить или использовать регулярное выражение.

Регулярные выражения позволяют решать проблемы, которые иначе было бы слишком трудно решить. Однако не пытайтесь решать с помощью регулярных выражений любые проблемы, с которыми вы сталкиваетесь. Всегда выбирайте для работы правильный инструмент.

В оставшейся части этого раздела будут представлены три примера регулярных выражений и сопоставления с образцом.

Простой пример

В этом подразделе мы узнаем, как выбрать конкретный столбец из строки текста. Чтобы все стало еще интереснее, мы также научимся построчно читать текстовые файлы. Однако о вводе-выводе из файлов вы узнаете из главы 8, поэтому, чтобы получить больше информации о соответствующем коде Go, вам следует прочитать ее.

Наш исходный файл Go называется `selectColumn.go`, и мы его рассмотрим, разделив на пять фрагментов. Для работы утилиты нужны как минимум два аргумента командной строки: первый — требуемый номер столбца, а второй — путь к текстовому файлу, который нужно обработать. Однако мы можем использовать любое количество текстовых файлов — `selectColumn.go` обработает их все по одному.

Первая часть `selectColumn.go` выглядит так:

```
package main

import (
```

```
    "bufio"
    "fmt"
    "io"
    "os"
    "strconv"
    "strings"
)
```

Вторая часть `selectColumn.go` содержит следующий код:

```
func main() {
    arguments := os.Args
    if len(arguments) < 2 {
        fmt.Printf("usage: selectColumn column <file1> [<file2> [... <fileN]]\n")
        os.Exit(1)
    }

    temp, err := strconv.Atoi(arguments[1])
    if err != nil {
        fmt.Println("Column value is not an integer:", temp)
        return
    }

    column := temp
    if column < 0 {
        fmt.Println("Invalid Column number!")
        os.Exit(1)
    }
```

В первую очередь программа выполняет проверку, чтобы убедиться, что она получила достаточное количество аргументов командной строки (`len(arguments) < 2`). Кроме того, нам нужны еще две проверки, чтобы убедиться, что предоставленное значение столбца является числом и что оно больше 0.

Третья часть `selectColumn.go` выглядит так:

```
    for _, filename := range arguments[2:] {
        fmt.Println("\t\t", filename)
        f, err := os.Open(filename)
        if err != nil {
            fmt.Printf("error opening file %s\n", err)
            continue
        }
        defer f.Close()
```

Программа выполняет различные проверки, чтобы убедиться, что текстовый файл существует и его можно прочитать — для открытия текстового файла используется функция `os.Open()`. Помните, что права доступа к текстовому файлу в UNIX не всегда допускают чтение файла.

Четвертый фрагмент кода `selectColumn.go` выглядит так:

```
        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)
            }
```

Как вы узнаете из главы 8, функция `bufio.ReadString()` считывает файл до первого появления переданного этой функции параметра. Так, `bufio.ReadString('\n')` считывает одну строку, потому что `\n` в UNIX является символом новой строки. Функция `bufio.ReadString()` возвращает *байтовый срез*.

Последний фрагмент `selectColumn.go` содержит следующий код:

```
            data := strings.Fields(line)
            if len(data) >= column {
                fmt.Println((data[column-1]))
            }
        }
    }
}
```

Логика программы довольно проста: вы разбиваете текст на строки и выбираете нужный столбец. Однако, поскольку нет уверенности в том, что в текущей строке есть необходимое количество полей, нужно проверять это перед любым выводом на экран. Это самая простая форма сопоставления с образцом, потому что каждая строка разбивается на отдельные слова с использованием пробелов в качестве разделителей.

Если вас заинтересовала тема разбиения строк, то вам будет полезно узнать, что функция `strings.Fields()` разбивает строку на основе пробельных символов, определенных в функции `unicode.IsSpace()`, и возвращает срез строки.

Выполнение `selectColumn.go` приведет к результатам следующего вида:

```
$ go run selectColumn.go 15 /tmp/swtag.log /tmp/adobegc.log   | head
              /tmp/swtag.log
              /tmp/adobegc.log
AdobeGCData
Successfully
Initializing
Stream
***********AdobeGC
Perform
Perform
Trying
```

Утилита `selectColumn.go` выводит имя каждого обработанного файла, даже если из этого файла не были получены данные для вывода.

Важно помнить, что никогда не следует доверять входным данным, особенно если они получены от нетехнических пользователей. Проще говоря, всегда проверяйте, являются ли входные данные тем, что вы ожидали получить.

Более сложный пример

В этом разделе вы узнаете, как сопоставить строку даты и времени, полученную из файла журнала веб-сервера Apache. Чтобы еще больше заинтересовать вас, я покажу, как изменить формат даты и времени файла журнала на другой формат. Напомню: для этого потребуется построчно прочитать файл журнала Apache — файл в простом текстовом формате.

Для этого создадим утилиту командной строки под названием `changeDT.go`. Для ее изучения мы разделим код утилиты на пять частей. Обратите внимание, что `changeDT.go` представляет собой улучшенную версию утилиты `timeDate.go`, о которой речь шла в главе 3, не только потому, что получает данные из внешнего файла, но также по той причине, что `changeDT.go` использует два регулярных выражения и таким образом может сопоставлять строки времени и даты, представленные в двух форматах.

Очень важный момент: не пытайтесь реализовать все возможные функции в первой же версии своих утилит. Лучше создать рабочую версию с меньшим количеством функций и затем постепенно улучшать ее.

Первый фрагмент кода `changeDT.go` выглядит так:

```
package main

import (
    "bufio"
    "fmt"
    "io"
    "os"
    "regexp"
    "strings"
    "time"
)
```

Утилита `changeDT.go` делает много интересного, и поэтому нам нужно достаточное количество пакетов.

Второй фрагмент кода `changeDT.go` выглядит так:

```
func main() {

    arguments := os.Args
    if len(arguments) == 1 {
        fmt.Println("Please provide one text file to process!")
        os.Exit(1)
    }

    filename := arguments[1]
    f, err := os.Open(filename)
```

```
if err != nil {
    fmt.Printf("error opening file %s", err)
    os.Exit(1)
}
defer f.Close()

notAMatch := 0
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)
    }
```

В этой части кода мы просто пытаемся открыть входной файл для чтения, чтобы затем читать его построчно. Переменная `notAMatch` содержит количество строк входного файла, которые не соответствуют ни одному из двух регулярных выражений программы.

Третий фрагмент кода `changeDT.go` содержит следующий код Go:

```
r1 := regexp.MustCompile(`.*\[(\d\d\/\w+/\d\d\d\d:\d\d:\d\d:\d\d.*)\] .*`)
if r1.MatchString(line) {
    match := r1.FindStringSubmatch(line)
    d1, err := time.Parse("02/Jan/2006:15:04:05 -0700", match[1])
    if err == nil {
        newFormat := d1.Format(time.Stamp)
        fmt.Print(strings.Replace(line, match[1], newFormat, 1))
    } else {
        notAMatch++
    }
    continue
}
```

Как видим, если первый формат даты и времени не совпадает, программа продолжает выполнение. Однако когда выполнение доходит до внутренней части блока `if`, будет выполнен оператор `continue`, то есть оставшийся код внешнего цикла `for` будет пропущен. Таким образом, строки, соответствующие первому из поддерживаемых форматов, имеют вид `21/Nov/2017:19:28:09 +0200`.

Функция `regexp.MustCompile()` похожа на функцию `regexp.Compile()`, но вызывает панику, если не может проанализировать выражение. Скобки, в которые заключено регулярное выражение, позволяют использовать найденные впоследствии совпадения. В данном случае у нас может быть только одно совпадение, которое мы получим с помощью функции `regexp.FindStringSubmatch()`.

Четвертая часть `changeDT.go` выглядит так:

```
r2 := regexp.MustCompile(`.*\[(\w+\-\d\d-\d\d:\d\d:\d\d:\d\d.*)\] .*`)
if r2.MatchString(line) {
```

```
        match := r2.FindStringSubmatch(line)
        d1, err := time.Parse("Jan-02-06:15:04:05 -0700", match[1])
        if err == nil {
            newFormat := d1.Format(time.Stamp)
            fmt.Print(strings.Replace(line, match[1], newFormat, 1))
        } else {
            notAMatch++
        }
        continue
    }
```

Второй поддерживаемый формат времени и даты имеет вид `Jun-21-17:19:28:09 +0200`. Как вы понимаете, между этими двумя форматами не так уж много различий. Обратите внимание, что, хотя в программе используется всего два формата даты и времени, вы можете использовать столько форматов, сколько пожелаете.

Последняя часть кода `changeDT.go` содержит следующий код Go:

```
    }
    fmt.Println(notAMatch, "lines did not match!")
}
```

Здесь выводится на экран количество строк, которые не соответствуют ни одному из двух форматов.

Текстовый файл, который используется для тестирования `changeDT.go`, содержит следующие строки:

```
$ cat logEntries.txt
- - [21/Nov/2017:19:28:09 +0200] "GET /AMEv2.tif.zip HTTP/1.1" 200 2188249 "-"
- - [21/Jun/2017:19:28:09 +0200] "GET /AMEv2.tif.zip HTTP/1.1" 200
- - [25/Lun/2017:20:05:34 +0200] "GET /MongoDjango.zip HTTP/1.1" 200 118362
- - [Jun-21-17:19:28:09 +0200] "GET /AMEv2.tif.zip HTTP/1.1" 200
- - [20/Nov/2017:20:05:34 +0200] "GET /MongoDjango.zip HTTP/1.1" 200 118362
- - [35/Nov/2017:20:05:34 +0200] "GET MongoDjango.zip HTTP/1.1" 200 118362
```

Выполнение `changDT.go` приводит к следующим результатам:

```
$ go run changeDT.go logEntries.txt
- - [Nov 21 19:28:09] "GET /AMEv2.tif.zip HTTP/1.1" 200 2188249 "-"
- - [Jun 21 19:28:09] "GET /AMEv2.tif.zip HTTP/1.1" 200
- - [Jun 21 19:28:09] "GET /AMEv2.tif.zip HTTP/1.1" 200
- - [Nov 20 20:05:34] "GET /MongoDjango.zip HTTP/1.1" 200 118362
2 lines did not match!
```

Проверка IPv4-адресов

Адрес IPv4, или просто *IP-адрес*, состоит из четырех частей. Поскольку адрес IPv4 хранится с использованием восьмиразрядных двоичных чисел, каждая часть может иметь значения от 0 (00000000 в двоичном формате) до 255 (11111111 в двоичном формате).

Формат адреса IPv6 намного сложнее, чем формат IPv4, поэтому представленная здесь программа не будет работать с адресами IPv6.

Программа, которую мы рассмотрим, называется `findIPv4.go`. Разделим ее на пять частей. Первая часть `findIPv4.go`:

```
package main

import (
    "bufio"
    "fmt"
    "io"
    "net"
    "os"
    "path/filepath"
    "regexp"
)
```

Поскольку `findIPv4.go` — довольно сложная утилита, ей нужно много стандартных пакетов Go.

Вторая часть содержит следующий код Go:

```
func findIP(input string) string {
    partIP := "(25[0-5]|2[0-4][0-9]|1[0-9][0-9]|[1-9]?[0-9])"
    grammar := partIP + "\\." + partIP + "\\." + partIP + "\\." + partIP
    matchMe := regexp.MustCompile(grammar)
    return matchMe.FindString(input)
}
```

В этом коде внутри функции определено регулярное выражение, которое поможет обнаружить IPv4-адрес. Это самая важная часть программы, потому что если вы неправильно определите регулярное выражение, то никогда не сможете распознавать адреса IPv4.

Прежде чем разъяснить регулярное выражение, что будет немного позже, важно понять, что перед определением одного или нескольких регулярных выражений необходимо знать особенности задачи, которую предстоит решить. Другими словами, если не знать, что десятичные значения адреса IPv4 не могут превышать 255, то никакие регулярные выражения не спасут!

Теперь, когда мы настроились на одну волну, рассмотрим следующие две строки кода:

```
partIP := "(25[0-5]|2[0-4][0-9]|1[0-9][0-9]|[1-9]?[0-9])"
grammar := partIP + "\\." + partIP + "\\." + partIP + "\\." + partIP
```

Регулярное выражение, определенное в `partIP`, соответствует каждой из четырех частей IP-адреса. Допустимый адрес IPv4 может начинаться с 25 и заканчиваться 0, 1, 2, 3, 4 или 5, потому что самое большое восьмибитное двоичное число

(`25[0-5]`), или же оно может начинаться с цифры 2, после которой следует 0, 1, 2, 3 или 4, и заканчиваться 0, 1, 2, 3, 4, 5, 6, 7, 8 или 9 (`2[0-4][0-9]`).

Либо адрес может начинаться с цифры 1, после которой следуют еще две цифры от 0 до 9 (`1[0-9][0-9]`). И последним вариантом будет натуральное число, состоящее из одной или двух цифр. Первой, необязательной цифрой может быть любая цифра от 1 до 9; вторая является обязательной, и это может быть цифра от 0 до 9 (`[1-9]?[0-9]`).

Переменная `grammar` говорит о том, что то, что мы ищем, состоит из четырех отдельных частей, каждая из которых должна соответствовать `partIP`. Переменная `grammar` соответствует полному IPv4-адресу, который мы ищем.

Поскольку findIPv4.go работает с регулярными выражениями для поиска IPv4-адреса в файле, программа может обрабатывать любой текстовый файл, содержащий правильные IPv4-адреса.

Наконец, если есть какие-либо особые требования, например, исключить определенные IPv4-адреса или отслеживать конкретные адреса или сети, то можно легко изменить код Go в `findIPv4.go` и добавить нужные функциональные возможности — удобная гибкость, когда разрабатываешь собственные инструменты.

Третья часть утилиты `findIPv4.go` содержит следующий код Go:

```
func main() {
    arguments := os.Args
    if len(arguments) < 2 {
        fmt.Printf("usage: %s logFile\n", filepath.Base(arguments[0]))
        os.Exit(1)
    }

    for _, filename := range arguments[1:] {
        f, err := os.Open(filename)
        if err != nil {
            fmt.Printf("error opening file %s\n", err)
            os.Exit(-1)
        }
        defer f.Close()
```

Прежде всего убедитесь, что вы получили достаточное количество аргументов командной строки, проверив длину `os.Args`. Затем используйте цикл `for` для перебора всех аргументов командной строки.

Четвертая часть кода выглядит так:

```
        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)
                break
            }
```

Как и в `selectColumn.go`, для построчного чтения входных данных используется функция `bufio.ReadString()`.

Последняя часть `findIPv4.go` содержит следующий код Go:

```
            ip := findIP(line)
            trial := net.ParseIP(ip)
            if trial.To4() == nil {
                continue
            }
            fmt.Println(ip)
        }
    }
}
```

Для каждой строки входного текстового файла нужно вызывать функцию `findIP()`. Функция `net.ParseIP()` проверяет, имеем ли мы дело с корректным адресом IPv4, — перепроверить еще раз никогда не бывает лишним! Если вызов `net.ParseIP()` выполнился успешно, выводим найденный IPv4-адрес на экран, после чего программа переходит к следующей строке входных данных.

Выполнение `findIPv4.go` приводит к следующим результатам:

```
$ go run findIPv4.go /tmp/auth.log
116.168.10.9
192.31.20.9
10.10.16.9
10.31.160.9
192.168.68.194
```

Таким образом, среди результатов `findIPv4.go` могут оказаться одинаковые строки. Помимо этой детали, результат работы утилиты вполне понятен.

Обработка показанных выше результатов с помощью некоторых традиционных утилит командной строки UNIX помогает получить больше информации о данных:

```
$ go run findIPv4.go /tmp/auth.log.1 /tmp/auth.log | sort -rn | uniq -c | sort -rn
    38 xxx.zz.116.9
    33 x.zz.2.190
    25 xx.zzz.1.41
    20 178.132.1.18
    18 x.zzz.63.53
    17 178.zzz.y.9
    15 103.yyy.xxx.179
    10 213.z.yy.194
```

```
 10 yyy.zzz.110.4
  9 yy.xx.65.113
```

Здесь с помощью утилит командной строки UNIX `sort(1)` и `uniq(1)` мы выяснили десять наиболее часто встречающихся адресов IPv4 из обработанных текстовых файлов. В основе этой довольно длинной команды из оболочки `bash(1)` лежит простая логика: результаты утилиты `findIPv4.go` становятся входными данными для первой команды `sort -rn`, которая сортирует их по числам в обратном порядке. Затем команда `uniq -c` удаляет появляющиеся строки, заменяя их одной строкой, перед которой указано, сколько раз данная строка встречается во входных данных. Затем результат сортируется еще раз, чтобы сначала отображались адреса IPv4 с большим числом вхождений.

Заметьте: важно понимать, что в основе функциональности findIPv4.go лежат регулярные выражения. Если регулярное выражение определено неправильно — либо не учитывает всех вариантов (ложноотрицательное), либо пропускает вещи, которые не должны попадать в результат (ложноположительное), то программа не будет работать правильно.

Строки

Строки в Go не являются составным типом, однако функций, которые поддерживают строки, в Go так много, что я решил более подробно описать их в этой главе.

Как отмечено в главе 3, строки в Go являются переменными, хранящими значения, а не указателями, как **строки в С**. Кроме того, Go по умолчанию поддерживает строки формата UTF-8, так что не приходится загружать специальные пакеты или предпринимать другие хитрые маневры для вывода символов Unicode. Однако существуют тонкие различия между символами, рунами и байтами, а также различия между строками и строковыми литералами, которые разъяснены далее.

Строка Go — это байтовый срез, предназначенный только для чтения, который может содержать байты любого типа и иметь произвольную длину.

Строковый литерал можно определить следующим образом:

```
const sLiteral = "\x99\x42\x32\x55\x50\x35\x23\x50\x29\x9c"
```

Возможно, внешний вид строкового литерала вас удивит. Строковую переменную можно определить следующим образом:

```
s2 := "€£³"
```

Чтобы определить длину строковой переменной или строкового литерала, можно воспользоваться функцией `len()`.

Проиллюстрируем всевозможные стандартные операции, связанные со строками, на примере программы `strings.go`, которую рассмотрим, разделив на пять частей. Первая из них содержит следующий код Go:

```
package main

import (
    "fmt"
)
```

Вторая часть кода Go выглядит следующим образом:

```
func main() {
    const sLiteral = "\x99\x42\x32\x55\x50\x35\x23\x50\x29\x9c"
    fmt.Println(sLiteral)
    fmt.Printf("x: %x\n", sLiteral)

    fmt.Printf("sLiteral length: %d\n", len(sLiteral))
```

Каждая последовательность `\xAB` представляет отдельный символ `sLiteral`. В результате вызов функции `len(sLiteral)` вернет количество символов `sLiteral`. Вместо `%x` в `fmt.Printf()` будет подставлена часть `AB` последовательности `\xAB`.

Третий фрагмент кода `strings.go` содержит следующий код Go:

```
    for i := 0; i < len(sLiteral); i++ {
        fmt.Printf("%x ", sLiteral[i])
    }
    fmt.Println()

    fmt.Printf("q: %q\n", sLiteral)
    fmt.Printf("+q: %+q\n", sLiteral)
    fmt.Printf(" x: % x\n", sLiteral)

    fmt.Printf("s: As a string: %s\n", sLiteral)
```

Как видите, здесь получен доступ к строковому литералу как к срезу. Использование `%q` в `fmt.Printf()` со строковым аргументом приведет к выводу строки в двойных кавычках, которая безопасно экранируется синтаксисом Go. Использование `%+q` в `fmt.Printf()` со строковым аргументом гарантирует вывод только символов ASCII.

Наконец, использование `% x` (обратите внимание на пробел между символами `%` и `x`) в `fmt.Printf()` приведет к появлению пробелов между выводимыми байтами. Чтобы вывести строковый литерал в виде строки, нужно вызвать `fmt.Printf()` с `%s`.

Четвертая часть кода `strings.go` выглядит так:

```
    s2 := "€£³"
    for x, y := range s2 {
        fmt.Printf("%#U starts at byte position %d\n", y, x)
    }

    fmt.Printf("s2 length: %d\n", len(s2))
```

Здесь определяем строку с именем `s2`, которая содержит три символа Unicode. Вызов `fmt.Printf()` с `%#U` приведет к выводу на экран символов в формате `U+0058`. Использование ключевого слова `range` в строке, содержащей символы Unicode, позволяет обрабатывать содержащиеся в ней символы Unicode по одному.

Результат `len(s2)` может вас немного удивить. Поскольку переменная `s2` содержит символы Unicode, то ее размер в байтах превышает количество хранящихся в ней символов.

Последняя часть `strings.go` выглядит так:

```
    const s3 = "ab12AB"
    fmt.Println("s3:", s3)
    fmt.Printf("x: % x\n", s3)

    fmt.Printf("s3 length: %d\n", len(s3))

    for i := 0; i < len(s3); i++ {
        fmt.Printf("%x ", s3[i])
    }
    fmt.Println()
}
```

Запуск `strings.go` приведет к следующим результатам:

```
$ go run strings.go
�B2UP5#P)�
x: 9942325550352350299c
sLiteral length: 10
99 42 32 55 50 35 23 50 29 9c
q: "\x99B2UP5#P)\x9c"
+q: "\x99B2UP5#P)\x9c"
x: 99 42 32 55 50 35 23 50 29 9c
s: As a string: �B2UP5#P)�
U+20AC '€' starts at byte position 0
U+00A3 '£' starts at byte position 3
U+00B3 '³' starts at byte position 5
s2 length: 7
s3: ab12AB
x: 61 62 31 32 41 42
s3 length: 6
61 62 31 32 41 42
```

Неудивительно, если информация, представленная в этом разделе, покажется вам сложной, особенно если вы не знакомы с представлением букв и символов в формате Unicode и UTF-8. К счастью, большинство из них в ежедневной работе как разработчика Go вам не понадобится. Едва ли вам придется использовать для вывода данных в своих программах столь примитивные команды, как `fmt.Println()` и `fmt.Printf()`. Но если вы живете за пределами Европы и США, то некоторая информация из этого раздела может оказаться весьма кстати.

Что такое руны

Руны — это значения типа `int32`, следовательно, тип Go, который используется для представления *кодовых пунктов* Unicode. Кодовый пункт, или кодовая позиция Unicode, — это числовое значение, которое обычно используется для представления отдельных символов Unicode; однако он также может иметь и другие значения, например, содержать информацию о форматировании.

Внимание! Строку можно рассматривать как множество рун.

Рунный литерал — это символ, заключенный в одинарные кавычки. Рунный литерал также можно рассматривать как *рунную константу*. Внутреннее представление рунного литерала основано на кодовой позиции Unicode.

Использование рун показано на примере программы `runes.go`, которую мы разделим на три части. Первая часть `runes.go` выглядит так:

```
package main

import (
    "fmt"
)
```

Вторая часть `runes.go` содержит следующий код:

```
func main() {
    const r1 = '€'
    fmt.Println("(int32) r1:", r1)
    fmt.Printf("(HEX) r1: %x\n", r1)
    fmt.Printf("(as a String) r1: %s\n", r1)
    fmt.Printf("(as a character) r1: %c\n", r1)
```

Сначала мы определяем рунный литерал с именем `r1`. (Обратите внимание, что знак евро не относится к таблице символов ASCII.) Затем выводим `r1`, используя различные операторы, а потом выводим его значение в формате `int32` и шестнадцатеричное значение. После этого пытаемся вывести `r1` в виде строки. Наконец, выводим `r1` как символ, что дает тот же результат, что и в определении `r1`.

Третий, последний фрагмент кода `runes.go` содержит следующий код Go:

```
    fmt.Println("A string is a collection of runes:", []byte("Mihalis"))
    aString := []byte("Mihalis")
    for x, y := range aString {
        fmt.Println(x, y)
        fmt.Printf("Char: %c\n", aString[x])
    }
    fmt.Printf("%s\n", aString)
}
```

Здесь видно, что байтовый срез представляет собой набор рун и что вывод байтового среза с помощью `fmt.Println()` может дать не те результаты, которых мы ожидали. Чтобы преобразовать руну в символ, нужно использовать в `fmt.Printf()` символ подстановки `%c`. Чтобы вывести байтовый срез в виде строки, нужно использовать в `fmt.Printf()` символ подстановки `%s`.

Выполнение `runes.go` приведет к следующим результатам:

```
$ go run runes.go
(int32) r1: 8364
(HEX) r1: 20ac
(as a String) r1: %!s(int32=8364)
(as a character) r1: €
A string is a collection of runes: [77 105 104 97 108 105 115]
0 77
Char: M
1 105
Char: i
2 104
Char: h
3 97
Char: a
4 108
Char: l
5 105
Char: i
6 115
Char: s
Mihalis
```

Наконец, самый простой способ получить сообщение об ошибке `illegal rune literal` — использовать при импорте пакета одинарные, а не двойные кавычки:

```
$ cat a.go
package main
import (
    'fmt'
)
func main() {
}
$ go run a.go
package main:
a.go:4:2: illegal rune literal
```

Пакет unicode

Стандартный пакет Go `unicode` содержит множество удобных функций. Одна из них, которая называется `unicode.IsPrint()`, помогает определить, какие части строки можно вывести с использованием рун. Этот прием проиллюстрирован в коде

Go, содержащемся в файле unicode.go, который мы рассмотрим, разделив на две части. Первая часть unicode.go выглядит следующим образом:

```
package main

import (
    "fmt"
    "unicode"
)

func main() {
    const sL = "\x99\x00ab\x50\x00\x23\x50\x29\x9c"
```

Второй фрагмент кода unicode.go содержит следующий код Go:

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

Как уже отмечалось, всю грязную работу выполняет функция unicode.IsPrint(), которая возвращает true, если руну можно напечатать, и false, если нельзя. При желании узнать больше о символах Unicode обязательно загляните на страницу документации пакета unicode. Выполнение unicode.go приведет к следующим результатам:

```
$ go run unicode.go
Not printable!
Not printable!
a
b
P
Not printable!
#
P
)
Not printable!
```

Пакет strings

Стандартный пакет Go strings позволяет манипулировать строками UTF-8 в Go и включает в себя множество мощных функций. Большинство из них будут проиллюстрированы в файле useStrings.go, который рассмотрим, разделив на пять частей. Обратите внимание, что функции пакета strings, связанные с вводом и выводом файлов, будут продемонстрированы в главе 8.

Первая часть `useStrings.go` выглядит так:

```
package main

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

var f = fmt.Printf
```

Пакет strings импортируется особым образом. При использовании такого оператора импорта Go создает псевдоним для пакета. Таким образом, вместо записи strings.FunctionName() теперь можно написать s.FunctionName(), что немного короче. Обратите внимание, что вы больше не сможете вызывать функцию пакета strings как strings.FunctionName().

Еще один полезный прием заключается в том, что если вы постоянно обращаетесь к одной и той же функции и хотите использовать более короткую запись, то можно назначить этой функции имя переменной и использовать это имя вместо функции. В данном примере этот прием применен к функции `fmt.Printf()`. Однако не следует злоупотреблять такой возможностью, потому что иначе будет трудно читать код!

Вторая часть `useStrings.go` содержит следующий код Go:

```
func main() {
    upper := s.ToUpper("Hello there!")
    f("To Upper: %s\n", upper)
    f("To Lower: %s\n", s.ToLower("Hello THERE"))
    f("%s\n", s.Title("tHis wiLL be A title!"))
    f("EqualFold: %v\n", s.EqualFold("Mihalis", "MIHAlis"))
    f("EqualFold: %v\n", s.EqualFold("Mihalis", "MIHAli"))
```

В этом фрагменте кода нам встретилось множество функций, которые позволяют играть с регистром букв в строке. Кроме того, как вы можете видеть, функция `strings.EqualFold()` позволяет определить, являются ли две строки одинаковыми, несмотря на различия в регистре букв.

Третья часть кода `useStrings.go` выглядит так:

```
    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"))

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

```
f("Count: %v\n", s.Count("Mihalis", "I"))
f("Repeat: %s\n", s.Repeat("ab", 5))

f("TrimSpace: %s\n", s.TrimSpace(" \tThis is a line. \n"))
f("TrimLeft: %s", s.TrimLeft(" \tThis is a\t line. \n", "\n\t "))
f("TrimRight: %s\n", s.TrimRight(" \tThis is a\t line. \n", "\n\t "))
```

Функция `strings.Count()` подсчитывает количество случаев, когда второй параметр появляется в строке, заданной в качестве первого параметра. Функция `strings.HasPrefix()` возвращает `true`, если строка, переданная в качестве первого параметра, начинается со строки, переданной в качестве второго параметра, и `false` — в противном случае. Аналогично функция `strings.HasSuffix()` возвращает `true`, когда строка, переданная в качестве первого параметра, заканчивается строкой, переданной в качестве второго параметра, и `false` — в противном случае.

Четвертый фрагмент кода `useStrings.go` содержит следующий код Go:

```
f("Compare: %v\n", s.Compare("Mihalis", "MIHALIS"))
f("Compare: %v\n", s.Compare("Mihalis", "Mihalis"))
f("Compare: %v\n", s.Compare("MIHALIS", "MIHalis"))

f("Fields: %v\n", s.Fields("This is a string!"))
f("Fields: %v\n", s.Fields("Thisis\na\tstring!"))

f("%s\n", s.Split("abcd efg", ""))
```

Эта часть кода содержит несколько более глубоких и оригинальных функций. Первая из них, `strings.Split()`, позволяет разбить заданную строку на части, ограниченные строкой-разделителем, — функция `strings.Split()` возвращает *строковый срез*. Использование `""` в качестве второго параметра `strings.Split()` позволяет разделить строку на отдельные символы и затем обрабатывать их по одному.

Функция `strings.Compare()` сравнивает две строки лексикографически; она возвращает `0`, если строки идентичны, и `-1` или `+1` — если нет.

Наконец, функция `strings.Fields()` разбивает строку, переданную в качестве параметра, на части, используя для разделения пробельные символы. Пробельные символы определяются в функции `unicode.IsSpace()`.

strings.Split() — мощная функция, которую просто необходимо изучить, потому что рано или поздно (скорее, рано) вам придется использовать ее в своих программах.

Последняя часть `useStrings.go` содержит следующий код Go:

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

```
    lines := []string{"Line 1", "Line 2", "Line 3"}
    f("Join: %s\n", s.Join(lines, "+++"))

    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))
}
```

Как и в предыдущей части `useStrings.go`, последний фрагмент кода содержит функции, которые реализуют некоторые очень интеллектуальные функциональные возможности в простой для понимания и использования форме.

Функция `strings.Replace()` принимает четыре параметра. Первый параметр — это строка, которую следует обработать. Второй параметр содержит строку, которая, если будет найдена, заменится на третий параметр функции. Последний параметр — это максимальное количество замен, которые можно выполнить. Если этот параметр меньше нуля, то количество возможных замен не ограничено.

Последние два оператора программы определяют *функцию обрезки*, которая позволяет сохранять интересующие вас руны строки и использовать эту функцию в качестве второго аргумента функции `strings.TrimFunc()`.

Наконец, функция `strings.SplitAfter()` разбивает строку, переданную в качестве первого параметра, на подстроки на основе строки-разделителя, заданной в качестве второго параметра.

Полный список функций, аналогичных unicode.IsLetter(), вы найдете на странице документации стандартного пакета Go unicode.

Выполнение `useStrings.go` приведет к следующим результатам:

```
$ go run useStrings.go
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: [This is a string!]
Fields: [Thisis a string!]
[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
```

Обратите внимание, что здесь представлены далеко не все функции пакета `strings`. Полный перечень доступных функций вы найдете на странице документации пакета `strings` по адресу https://golang.org/pkg/strings/.

Если вы работаете с текстом, то вам определенно необходимо изучить все подробности и функции пакета `strings`. Поэтому обязательно поэкспериментируйте со всеми этими функциями и напишите множество примеров, которые помогут вам прояснить ситуацию.

Оператор switch

Главная причина, по которой оператор `switch` представлен в этой главе, заключается в том, что в `switch` можно использовать регулярные выражения. Однако для начала рассмотрим простой блок `switch`:

```
switch asString {
case "1":
    fmt.Println("One!")
case "0":
    fmt.Println("Zero!")
default:
    fmt.Println("Do not care!")
}
```

Этот блок `switch` выполняет разные действия для строки, равной `"1"`, `"0"`, и для всех остальных вариантов (`default`).

Наличие в блоке switch варианта для всех оставшихся случаев считается очень хорошей практикой. Однако, поскольку последовательность вариантов в блоке switch имеет значение, вариант для всех остальных случаев следует ставить последним. В Go такой вариант носит имя default.

Однако оператор `switch` может быть гораздо более гибким и адаптируемым:

```
switch {
case number < 0:
    fmt.Println("Less than zero!")
case number > 0:
    fmt.Println("Bigger than zero!")
default:
    fmt.Println("Zero!")
}
```

В этом блоке `switch` определяется, имеем ли мы дело с положительным целым числом, отрицательным целым числом или нулем. Как видим, в ветвях оператора `switch` допускаются условия. Вскоре вы узнаете, что ветви оператора `switch` также могут содержать регулярные выражения.

Все эти и некоторые другие примеры вы найдете в файле `switch.go`, который мы рассмотрим, разделив на пять частей.

Первая часть `switch.go` выглядит так:

```
package main

import (
    "fmt"
    "os"
    "regexp"
    "strconv"
)

func main() {
    arguments := os.Args
    if len(arguments) < 2 {
        fmt.Println("usage: switch number")
        os.Exit(1)
    }
```

Пакет `regexp` необходим для поддержки регулярных выражений в `switch`.

Второй фрагмент кода `switch.go` содержит следующий код Go:

```
    number, err := strconv.Atoi(arguments[1])
    if err != nil {
        fmt.Println("This value is not an integer:", number)
    } else {
        switch {
        case number < 0:
            fmt.Println("Less than zero!")
        case number > 0:
            fmt.Println("Bigger than zero!")
        default:
            fmt.Println("Zero!")
        }
    }
```

Третья часть switch.go выглядит так:

```
    asString := arguments[1]
    switch asString {
    case "5":
        fmt.Println("Five!")
    case "0":
        fmt.Println("Zero!")
    default:
        fmt.Println("Do not care!")
}
```

В этом фрагменте кода мы видим, что в вариантах switch могут присутствовать жестко закодированные значения. Как правило, это используется тогда, когда после ключевого слова switch стоит имя переменной.

Четвертая часть кода switch.go содержит следующий код Go:

```
    var negative = regexp.MustCompile(`-`)
    var floatingPoint = regexp.MustCompile(`\d?\.\d`)
    var email = regexp.MustCompile(`^[^@]+@[^@.]+\.[^@.]+`)

    switch {
    case negative.MatchString(asString):
        fmt.Println("Negative number")
    case floatingPoint.MatchString(asString):
        fmt.Println("Floating point!")
    case email.MatchString(asString):
        fmt.Println("It is an email!")
        fallthrough
    default:
        fmt.Println("Something else!")
}
```

Здесь происходит много интересного. Во-первых, определяются три регулярных выражения с именами negative, floatingPoint и email соответственно. Во-вторых, все три выражения используются в блоке switch с помощью функции regexp.MatchString(), которая и выполняет сопоставление.

Наконец, встретив ключевое слово fallthrough, Go выполняет ветвь, которая следует за текущей, — в данном случае это ветвь default. Это означает, что, после того как выполнится код ветви email.MatchString(asString), будет также выполнен код для ветви default.

Последняя часть switch.go выглядит так:

```
    var aType error = nil
    switch aType.(type) {
    case nil:
        fmt.Println("It is nil interface!")
    default:
        fmt.Println("Not nil interface!")
    }
}
```

Как видим, этот `switch` различает типы. Подробнее о взаимодействии `switch` с *интерфейсами* Go вы прочитаете в главе 7.

Выполнение `switch.go` с различными входными аргументами приведет к следующим результатам:

```
$ go run switch.go
usage: switch number.
exit status 1
$ go run switch.go mike@g.com
This value is not an integer: 0
Do not care!
It is an email!
Something else!
It is nil interface!
$ go run switch.go 5
Bigger than zero!
Five!
Something else!
It is nil interface!
$ go run switch.go 0
Zero!
Zero!
Something else!
It is nil interface!
$ go run switch.go 1.2
This value is not an integer: 0
Do not care!
Floating point!
It is nil interface!
$ go run switch.go -1.5
This value is not an integer: 0
Do not care!
Negative number
It is nil interface!
```

Вычисление числа π с высокой точностью

В этом разделе вы узнаете, как с высокой точностью вычислить число π, используя стандартный пакет Go, именуемый `math/big`, и определенные в нем специализированные типы.

В этом разделе содержится самый уродливый код Go, который мне когда-либо встречался; даже код на Java, и тот выглядит лучше!

Программа вычисления числа π по формуле Белларда называется `calculatePi.go`, и мы ее рассмотрим, разделив на четыре части.

Первая часть `convertPi.go` выглядит следующим образом:

```
package main

import (
    "fmt"
    "math"
    "math/big"
    "os"
    "strconv"
)

var precision uint = 0
```

Переменная `precision` определяет желаемую точность вычислений; она сделана глобальной, чтобы быть доступной из любой точки программы.

Второй фрагмент кода `calculatePi.go` содержит следующий код Go:

```
func Pi(accuracy uint) *big.Float {
    k := 0
    pi := new(big.Float).SetPrec(precision).SetFloat64(0)
    k1k2k3 := new(big.Float).SetPrec(precision).SetFloat64(0)
    k4k5k6 := new(big.Float).SetPrec(precision).SetFloat64(0)
    temp := new(big.Float).SetPrec(precision).SetFloat64(0)
    minusOne := new(big.Float).SetPrec(precision).SetFloat64(-1)
    total := new(big.Float).SetPrec(precision).SetFloat64(0)

    two2Six := math.Pow(2, 6)
    two2SixBig := new(big.Float).SetPrec(precision).SetFloat64(two2Six)
```

Вызов функции `new(big.Float)` создает переменную `big.Float` с заданной точностью, которая определяется с помощью функции `SetPrec()`.

Третья часть `convertPi.go` содержит оставшийся код Go функции `Pi()`:

```
    for {
        if k > int(accuracy) {
            break
        }
        t1 := float64(float64(1) / float64(10*k+9))
        k1 := new(big.Float).SetPrec(precision).SetFloat64(t1)
        t2 := float64(float64(64) / float64(10*k+3))
        k2 := new(big.Float).SetPrec(precision).SetFloat64(t2)
        t3 := float64(float64(32) / float64(4*k+1))
        k3 := new(big.Float).SetPrec(precision).SetFloat64(t3)
        k1k2k3.Sub(k1, k2)
        k1k2k3.Sub(k1k2k3, k3)

        t4 := float64(float64(4) / float64(10*k+5))
        k4 := new(big.Float).SetPrec(precision).SetFloat64(t4)
        t5 := float64(float64(4) / float64(10*k+7))
        k5 := new(big.Float).SetPrec(precision).SetFloat64(t5)
```

```
        t6 := float64(float64(1) / float64(4*k+3))
        k6 := new(big.Float).SetPrec(precision).SetFloat64(t6)
        k4k5k6.Add(k4, k5)
        k4k5k6.Add(k4k5k6, k6)
        k4k5k6 = k4k5k6.Mul(k4k5k6, minusOne)
        temp.Add(k1k2k3, k4k5k6)

        k7temp := new(big.Int).Exp(big.NewInt(-1), big.NewInt(int64(k)), nil)
        k8temp := new(big.Int).Exp(big.NewInt(1024), big.NewInt(int64(k)), nil)

        k7 := new(big.Float).SetPrec(precision).SetFloat64(0)
        k7.SetInt(k7temp)
        k8 := new(big.Float).SetPrec(precision).SetFloat64(0)
        k8.SetInt(k8temp)
        t9 := float64(256) / float64(10*k+1)
        k9 := new(big.Float).SetPrec(precision).SetFloat64(t9)
        k9.Add(k9, temp)
        total.Mul(k9, k7)
        total.Quo(total, k8)
        pi.Add(pi, total)

        k = k + 1
    }
    pi.Quo(pi, two2SixBig)
    return pi
}
```

Эта часть программы представляет собой реализацию формулы Белларда на языке Go. Недостаток пакета `math/big` состоит в том, что практически для любого вида вычислений нужна специальная функция этого пакета. По большей части так сделано для того, чтобы эти функции поддерживали точность на заданном уровне. Таким образом, без использования переменных `big.Float` и `big.Int`, а также функций `math/big` невозможно вычислить число π с требуемой точностью.

В последней части `calculatePi.go` содержится реализация функции `main()`:

```
func main() {
    arguments := os.Args
    if len(arguments) == 1 {
        fmt.Println("Please provide one numeric argument!")
        os.Exit(1)
    }

    temp, _ := strconv.ParseUint(arguments[1], 10, 32)
    precision = uint(temp) * 3

    PI := Pi(precision)
    fmt.Println(PI)
}
```

Выполнение `convertPi.go` приведет к следующим результатам:

```
$ go run calculatePi.go
Please provide one numeric argument!
exit status 1
$ go run calculatePi.go 20
3.141592653589793258
$ go run calculatePi.go 200
3.1415926535897932569603993617387624040191831562485732434931792835710464502489134671185117843176153542820179294162928090508139378752834356105863133635486024367680477064898389243819 29
```

Разработка на Go хранилища типа «ключ — значение»

В этом разделе вы узнаете, как разработать на Go несложную версию хранилища типа «ключ — значение», как реализовать основную функциональность такого хранилища без каких-либо дополнительных наворотов. Идея хранилища типа «ключ — значение» скромна: быстро отвечать на запросы и вообще работать максимально быстро. Это означает использование простых алгоритмов и простых структур данных.

Представленная здесь программа реализует четыре основные задачи хранилища «ключ — значение».

1. Добавление нового элемента.
2. Удаление существующего элемента из хранилища «ключ — значение» по ключу этого элемента.
3. Поиск по хранилищу значения конкретного ключа.
4. Изменение значения существующего ключа.

Эти четыре функции позволяют полностью управлять хранилищем «ключ — значение». Команды для них будут называться `ADD`, `DELETE`, `LOOKUP` и `CHANGE` соответственно. Это означает, что программа будет работать, только если получит одну из четырех команд. Кроме того, программа будет останавливаться, если ввести слово `STOP`, и выводить все содержимое хранилища «ключ — значение» при вводе команды `PRINT`.

Программа называется `keyValue.go`; рассмотрим ее, разделив на пять фрагментов кода.

Первый фрагмент кода `keyValue.go` выглядит так:

```
package main

import (
```

```
    "bufio"
    "fmt"
    "os"
    "strings"
)

type myElement struct {
    Name     string
    Surname  string
    Id       string
}

var DATA = make(map[string]myElement)
```

Хранилище «ключ — значение» представляет собой отдельную хеш-таблицу Go, поскольку встроенные структуры данных Go, как правило, работают быстрее. Переменная хеш-таблицы определяется как глобальная; ее ключами являются переменные типа `string`, а значениями — переменные типа `myElement`. Как видим, здесь также определена структура `myElement`.

Второй фрагмент кода `keyValue.go` выглядит так:

```
func ADD(k string, n myElement) bool {
    if k == "" {
        return false
    }

    if LOOKUP(k) == nil {
        DATA[k] = n
        return true
    }
    return false
}

func DELETE(k string) bool {
    if LOOKUP(k) != nil {
        delete(DATA, k)
        return true
    }
    return false
}
```

В этом фрагменте кода реализованы две функции, которые обеспечивают функциональность команд `ADD` и `DELETE`. Обратите внимание: если пользователь попытается добавить в хранилище новый элемент, не предоставив достаточного количества значений для заполнения структуры `myElement`, функция `ADD` завершится с ошибкой. Для этой конкретной программы недостающие поля структуры `myElement` будут заполнены пустыми строками. Однако при попытке добавить в хранилище уже существующий ключ значение этого ключа не изменится, а вы получите сообщение об ошибке.

Третья часть keyValue.go содержит следующий код:

```
func LOOKUP(k string) *myElement {
    _, ok := DATA[k]
    if ok {
        n := DATA[k]
        return &n
    } else {
        return nil
    }
}

func CHANGE(k string, n myElement) bool {
    DATA[k] = n
    return true
}

func PRINT() {
    for k, d := range DATA {
        fmt.Printf("key: %s value: %v\n", k, d)
    }
}
```

В этом фрагменте кода Go реализованы функции, которые поддерживают функциональность команд LOOKUP и CHANGE. При попытке изменить несуществующий ключ программа добавит этот ключ в хранилище, не создавая сообщений об ошибках. В данной части программы также реализована функция PRINT(), которая выводит на экран все содержимое хранилища «ключ — значение».

Имена функций написаны ПРОПИСНЫМИ БУКВАМИ, поскольку эти функции действительно важны для программы.

Четвертая часть keyValue.go выглядит так:

```
func main() {
    scanner := bufio.NewScanner(os.Stdin)
    for scanner.Scan() {
        text := scanner.Text()
        text = strings.TrimSpace(text)
        tokens := strings.Fields(text)

        switch len(tokens) {
        case 0:
            continue
        case 1:
            tokens = append(tokens, "")
            tokens = append(tokens, "")
            tokens = append(tokens, "")
            tokens = append(tokens, "")
        case 2:
            tokens = append(tokens, "")
            tokens = append(tokens, "")
            tokens = append(tokens, "")
```

```
        case 3:
            tokens = append(tokens, "")
            tokens = append(tokens, "")
        case 4:
            tokens = append(tokens, "")
        }
```

В этой части keyValue.go происходит чтение данных, вводимых пользователем. Во-первых, цикл for гарантирует, что программа будет продолжать работать до тех пор, пока пользователь что-то вводит. Во-вторых, программа проверяет, чтобы срез tokens содержал как минимум пять элементов, хотя такое количество элементов требуется только для команды ADD. Таким образом, чтобы операция ADD была завершена и не имела пропущенных значений, потребуется ввести команду вида ADD aKey Field1 Field2 Field3.

Последняя часть keyValue.go содержит следующий код Go:

```
        switch tokens[0] {
        case "PRINT":
            PRINT()
        case "STOP":
            return
        case "DELETE":
            if !DELETE(tokens[1]) {
                fmt.Println("Delete operation failed!")
            }
        case "ADD":
            n := myElement{tokens[2], tokens[3], tokens[4]}
            if !ADD(tokens[1], n) {
                fmt.Println("Add operation failed!")
            }
        case "LOOKUP":
            n := LOOKUP(tokens[1])
            if n != nil {
                fmt.Printf("%v\n", *n)
            }
        case "CHANGE":
            n := myElement{tokens[2], tokens[3], tokens[4]}
            if !CHANGE(tokens[1], n) {
                fmt.Println("Update operation failed!")
            }
        default:
            fmt.Println("Unknown command - please try again!")
        }
    }
}
```

В этой части программы обрабатываются данные, поступающие от пользователя. Благодаря оператору switch программа имеет очень ясную структуру, так что вы избавлены от необходимости использовать несколько операторов if...else.

Выполнение `keyValue.go` приводит к следующим результатам:

```
$ go run keyValue.go
UNKNOWN
Unknown command - please try again!
ADD 123 1 2 3
ADD 234 2 3 4
ADD 345
PRINT
key: 123 value: {1 2 3}
key: 234 value: {2 3 4}
key: 345 value: { }
ADD 345 3 4 5
Add operation failed!
PRINT
key: 123 value: {1 2 3}
key: 234 value: {2 3 4}
key: 345 value: { }
CHANGE 345 3 4 5
PRINT
key: 123 value: {1 2 3}
key: 234 value: {2 3 4}
key: 345 value: {3 4 5}
DELETE 345
PRINT
key: 123 value: {1 2 3}
key: 234 value: {2 3 4}
DELETE 345
Delete operation failed!
PRINT
key: 123 value: {1 2 3}
key: 234 value: {2 3 4}
ADD 345 3 4 5
ADD 567 -5 -6 -7
PRINT
key: 123 value: {1 2 3}
key: 234 value: {2 3 4}
key: 345 value: {3 4 5}
key: 567 value: {-5 -6 -7}
CHANGE 345
PRINT
key: 123 value: {1 2 3}
key: 234 value: {2 3 4}
key: 345 value: { }
key: 567 value: {-5 -6 -7}
STOP
```

Чтобы узнать, как сделать хранилище данных «ключ — значение» постоянным, вам придется подождать до главы 8.

Вы также можете улучшить программу `keyValue.go`, добавив в нее горутины и каналы. Однако добавление горутин и каналов в однопользовательское приложение не имеет практического смысла. Но если сделать так, чтобы программа `keyValue.go` могла работать в сетях с протоколом *Transmission Control Protocol/Internet Protocol (TCP/IP)*, то использование горутин и каналов позволит поддерживать несколько соединений и обслуживать нескольких пользователей.

Подробнее о горутинах и каналах вы узнаете из глав 9 и 10. Затем, в главах 12, 13, вы узнаете о том, как создавать в Go сетевые приложения.

Go и формат JSON

JSON — это очень популярный текстовый формат, обеспечивающий простой и удобный способ передачи информации между приложениями на JavaScript. Однако JSON также используется для создания файлов конфигурации приложений и для хранения данных в структурированном формате.

В пакете `encoding/json` реализованы функции `Encode()` и `Decode()`, которые позволяют преобразовать объект Go в документ JSON и наоборот. Кроме того, в пакет `encoding/json` входят функции `Marshal()` и `Unmarshal()`, работающие аналогично `Encode()` и `Decode()` и основанные на методах `Encode()` и `Decode()`. Основное различие между парой `Marshal()` и `Unmarshal()` и парой `Encode()` и `Decode()` заключается в том, что первая работает с отдельными объектами, а вторая может работать как с несколькими объектами, так и с потоками байтов.

Чтение данных из формата JSON

В этом разделе вы узнаете, как прочитать запись JSON с диска. Воспользуемся программой `readJSON.go`, код которой здесь будет представлен в трех частях.

Первая часть `readJSON.go` содержит следующий код Go:

```
package main

import (
    "encoding/json"
    "fmt"
    "os"
)

type Record struct {
    Name     string
    Surname  string
    Tel      []Telephone
}
```

```
type Telephone struct {
    Mobile    bool
    Number    string
}
```

В этом коде Go определены структурные переменные, в которых будут храниться данные JSON.

Вторая часть `readJSON.go` выглядит так:

```
func loadFromJSON(filename string, key interface{}) error {
    in, err := os.Open(filename)
    if err != nil {
        return err
    }

    decodeJSON := json.NewDecoder(in)
    err = decodeJSON.Decode(key)
    if err != nil {
        return err
    }
    in.Close()
    return nil
}
```

Здесь определяется функция с именем `loadFromJSON()`, которая используется для декодирования данных файла JSON в соответствии со структурой данных, передаваемой функции в качестве второго аргумента. Сначала вызываем функцию `json.NewDecoder()`, чтобы создать новый декодер JSON, связанный с файлом, а затем — функцию `Decode()`, чтобы действительно декодировать содержимое файла и поместить его в нужную структуру данных.

Последняя часть `readJSON.go` выглядит так:

```
func main() {
    arguments := os.Args
    if len(arguments) == 1 {
        fmt.Println("Please provide a filename!")
        return
    }

    filename := arguments[1]

    var myRecord Record
    err := loadFromJSON(filename, &myRecord)
    if err == nil {
        fmt.Println(myRecord)
    } else {
        fmt.Println(err)
    }
}
```

В файле `readMe.json` содержатся следующие данные:

```
$ cat readMe.json
{
    "Name":"Mihalis",
    "Surname":"Tsoukalos",
    "Tel":[
        {"Mobile":true,"Number":"1234-567"},
        {"Mobile":true,"Number":"1234-abcd"},
        {"Mobile":false,"Number":"abcc-567"}
    ]
}
```

Выполнение `readJSON.go` приведет к следующим результатам:

```
$ go run readJSON.go readMe.json
{Mihalis Tsoukalos [{true 1234-567} {true 1234-abcd} {false abcc-567}]}
```

Сохранение данных в формате JSON

В этом подразделе вы узнаете, как записывать данные в формат JSON. Воспользуемся утилитой `writeJSON.go`, которая представлена здесь в трех частях. Она записывает данные в стандартный поток вывода (`os.Stdout`), то есть выводит данные в окно терминала.

Первая часть `writeJSON.go` выглядит следующим образом:

```
package main

import (
    "encoding/json"
    "fmt"
    "os"
)

type Record struct {
    Name     string
    Surname  string
    Tel      []Telephone
}

type Telephone struct {
    Mobile     bool
    Number     string
}
```

Вторая часть `writeJSON.go` выглядит так:

```
func saveToJSON(filename *os.File, key interface{}) {
    encodeJSON := json.NewEncoder(filename)
    err := encodeJSON.Encode(key)
    if err != nil {
```

```
        fmt.Println(err)
        return
    }
}
```

Функция saveToJSON() выполняет за вас всю работу: она создает переменную кодировщика JSON с именем encodeJSON; эта переменная связана с именем файла, в который будут помещаться данные. Вызов функции Encode() кодирует данные и сохраняет их в нужный файл.

Последняя часть writeJSON.go выглядит так:

```
func main() {
    myRecord := Record{
        Name:       "Mihalis",
        Surname:    "Tsoukalos",
        Tel: []Telephone{Telephone{Mobile: true, Number: "1234-567"},
            Telephone{Mobile: true, Number: "1234-abcd"},
            Telephone{Mobile: false, Number: "abcc-567"},
        },
    }
    saveToJSON(os.Stdout, myRecord)
}
```

В этом коде определена структурная переменная, в которой хранятся данные, которые нужно сохранить в формате JSON с помощью функции saveToJSON(). Поскольку используется os.Stdout, то данные выводятся на экран, а не сохраняются в файл.

Выполнение writeJSON.go приведет к следующим результатам:

```
$ go run writeJSON.go
{"Name":"Mihalis","Surname":"Tsoukalos","Tel":[{"Mobile":true,"Number":"1234-
567"},{"Mobile":true,"Number":"1234-abcd"},{"Mobile":false,"Number":"abcc-567"}]}
```

Использование функций Marshal() и Unmarshal()

В этом подразделе вы узнаете, как использовать методы Marshal() и Unmarshal() для реализации тех же функциональных возможностей, что были представлены в программах readJSON.go и writeJSON.go. Код Go, который иллюстрирует функции Marshal() и Unmarshal(), вы найдете в файле mUJSON.go. Рассмотрим его, разделив на три части.

Первая часть mUJSON.go выглядит так:

```
package main

import (
    "encoding/json"
    "fmt"
)

type Record struct {
    Name    string
```

```
    Surname    string
    Tel        []Telephone
}

type Telephone struct {
    Mobile    bool
    Number    string
}
```

В этой части программы определяются две структуры: `Record` и `Telephone`, которые будут использоваться для хранения данных, помещаемых в запись JSON.

Вторая часть `mUJSON.go` выглядит так:

```
func main() {
    myRecord := Record{
    Name:       "Mihalis",
    Surname:    "Tsoukalos",
    Tel: []Telephone{Telephone{Mobile: true, Number: "1234-567"},
        Telephone{Mobile: true, Number: "1234-abcd"},
        Telephone{Mobile: false, Number: "abcc-567"},
    }}

    rec, err := json.Marshal(&myRecord)
    if err != nil {
        fmt.Println(err)
        return
    }
    fmt.Println(string(rec))
```

В этой части мы определяем переменную `myRecord`, которая содержит нужные данные. Здесь также продемонстрировано использование функции `json.Marshal()`, которая принимает ссылку на переменную `myRecord`. Обратите внимание: `json.Marshal()` требует переменную-указатель, которая преобразуется в формат JSON.

Последняя часть `mUJSON.go` содержит следующий код:

```
    var unRec Record
    err1 := json.Unmarshal(rec, &unRec)
    if err1 != nil {
        fmt.Println(err1)
        return
    }
    fmt.Println(unRec)
}
```

Функция `json.Unmarshal()` получает входные данные JSON и преобразует их в структуру Go. Как и в случае с `json.Marshal()`, `json.Unmarshal()` также требует указатель в качестве аргумента.

Выполнение `mUJSON.go` приводит к следующим результатам:

```
$ go run mUJSON.go
{"Name":"Mihalis","Surname":"Tsoukalos","Tel":[{"Mobile":true,"Number":"1234-567"},
```

```
{"Mobile":true,"Number":"1234-abcd"},{"Mobile":false,"Number":"abcc-567"}]}
{Mihalis Tsoukalos [{true 1234-567} {true 1234-abcd} {false abcc-567}]}
```

Пакет Go encoding/json включает в себя два интерфейса: Marshaler и Unmarshaler. Каждый из этих интерфейсов требует реализации своего метода, называемого MarshalJSON() и UnmarshalJSON() соответственно. Если вы хотите выполнить какую-либо нестандартную JSON-маршализацию и демаршализацию, то названные интерфейсы позволят вам это сделать[1].

Синтаксический анализ данных в формате JSON

До сих пор мы изучали обработку структурированных данных JSON в заранее известном формате. Данные такого типа можно сохранять в виде структур Go с использованием методов, описанных в предыдущих подразделах.

В этом подразделе мы покажем, как читать и хранить *неструктурированные данные JSON*. Важно помнить, что неструктурированные данные JSON помещаются не в структуры, а в хеш-таблицы Go — этот процесс будет продемонстрирован на примере программы `parsingJSON.go`, которая представлена здесь в четырех частях.

Первая часть `parsingJSON.go` выглядит так:

```
package main

import (
    "encoding/json"
    "fmt"
    "io/ioutil"
    "os"
)
```

В этой части просто импортируются необходимые пакеты Go.

Вторая часть `parsingJSON.go` выглядит следующим образом:

```
func main() {
    arguments := os.Args
    if len(arguments) == 1 {
        fmt.Println("Please provide a filename!")
        return
    }

    filename := arguments[1]
```

[1] Функция json.Marshal() возвращает срез байт []byte, который после записи обычно становится не нужен и впоследствии удаляется сборщиком мусора. Если ваша программа ориентирована на массовую обработку и запись JSON, то постоянное выделение и освобождение []byte создает огромную нагрузку на сборщик мусора. Для снижения этой нагрузки лучше применять json.NewEncoder().Encode(), который задействует стандартный пакет sync.Pool с целью повторного использования срезов байт []byte для маршализации JSON.

Этот код считывает переданные программе аргументы командной строки и получает первый из них — имя файла JSON, который следует прочитать.

Третья часть parsingJSON.go выглядит так:

```
    fileData, err := ioutil.ReadFile(filename)
    if err != nil {
        fmt.Println(err)
        return
    }

    var parsedData map[string]interface{}
    json.Unmarshal([]byte(fileData), &parsedData)
```

Функция ioutil.ReadFile() позволяет прочитать сразу весь файл, что и нужно.

В этой части программы также определена хеш-таблица с именем parsedData, в которой будет сохранено разобранное содержимое прочитанного файла JSON. Каждый ключ хеш-таблицы является строкой и соответствует отдельному свойству JSON. Значение каждого ключа имеет тип interface{}, который может быть любым, то есть значением ключа хеш-таблицы может быть в том числе другая хеш-таблица.

Функция json.Unmarshal() используется для того, чтобы поместить содержимое файла в хеш-таблицу parsedData.

В главе 7 вы узнаете больше об интерфейсах вообще и interface{} в частности, а также о хеш-таблицах, которые позволяют динамически исследовать тип произвольного объекта и получать информацию о его структуре.

Последняя часть parsingJSON.go содержит следующий код:

```
    for key, value := range parsedData {
        fmt.Println("key:", key, "value:", value)
    }
}
```

Как видно из этого кода, можно перебирать ключи хеш-таблицы, чтобы получить ее содержимое. Однако интерпретация этого содержимого — совсем другое дело, так как это зависит от структуры данных, которая нам пока неизвестна.

Файл JSON с примерами данных, которые используются в этом подразделе, называется noStr.json и имеет такое содержимое:

```
$ cat noStr.json
{
    "Name": "John",
```

```
    "Surname": "Doe",
    "Age": 25,
    "Parents": [
        "Jim",
        "Mary"
    ],
    "Tel":[
        {"Mobile":true,"Number":"1234-567"},
        {"Mobile":true,"Number":"1234-abcd"},
        {"Mobile":false,"Number":"abcc-567"}
    ]
}
```

Выполнение parsingJSON.go приведет к следующим результатам:

```
$ go run parsingJSON.go noStr.json
key: Tel value: [map[Mobile:true Number:1234-567] map[Mobile:true
Number:1234-abcd] map[Mobile:false Number:abcc-567]]
key: Name value: John
key: Surname value: Doe
key: Age value: 25
key: Parents value: [Jim Mary]
```

Обратите внимание: как видно из результатов работы программы, ключи хеш-таблицы выводятся в случайном порядке.

Go и XML

Go поддерживает *XML* — язык разметки, похожий на HTML, но с гораздо более широкими возможностями.

Разработанная нами утилита rwXML.go считывает с диска запись JSON, вносит в нее изменения, преобразует ее в формат XML и выводит на экран. Затем данные XML преобразуются в формат JSON. Рассмотрим код Go этой утилиты, разделив его на четыре части.

Первая часть rwXML.go выглядит так:

```
package main

import (
    "encoding/json"
    "encoding/xml"
    "fmt"
    "os"
)

type Record struct {
    Name    string
```

```
    Surname    string
    Tel        []Telephone
}

type Telephone struct {
    Mobile bool
    Number string
}
```

Вторая часть `rwXML.go` содержит такой код Go:

```
func loadFromJSON(filename string, key interface{}) error {
    in, err := os.Open(filename)
    if err != nil {
        return err
    }

    decodeJSON := json.NewDecoder(in)
    err = decodeJSON.Decode(key)
    if err != nil {
        return err
    }
    in.Close()
    return nil
}
```

Третья часть `rwXML.go` содержит следующий код Go:

```
func main() {
    arguments := os.Args
    if len(arguments) == 1 {
        fmt.Println("Please provide a filename!")
        return
    }

    filename := arguments[1]

    var myRecord Record
    err := loadFromJSON(filename, &myRecord)
    if err == nil {
        fmt.Println("JSON:", myRecord)
    } else {
        fmt.Println(err)
    }

    myRecord.Name = "Dimitris"

    xmlData, _ := xml.MarshalIndent(myRecord, "", "    ")
    xmlData = []byte(xml.Header + string(xmlData))
    fmt.Println("\nxmlData:", string(xmlData))
```

После прочтения входного файла и преобразования его содержимого в формат JSON эти данные помещены в структуру Go. Затем мы внесли изменения

в содержимое этой структуры (`myRecord`). После этого с помощью функции `MarshalIndent()` эти данные преобразованы в формат XML, добавлен заголовок с помощью `xml.Header`.

Функция `MarshalIndent()`, которую, кроме прочего, можно использовать для данных в формате JSON, работает аналогично `Marshal()`, но каждый элемент XML начинается с новой строки и имеет отступ с учетом глубины вложенности. Это связано не со значениями XML, а с представлением данных в формате XML.

Последняя часть `rwXML.go` выглядит так:

```
    data := &Record{}
    err = xml.Unmarshal(xmlData, data)
    if nil != err {
        fmt.Println("Unmarshalling from XML", err)
        return
    }

    result, err := json.Marshal(data)
    if nil != err {
        fmt.Println("Error marshalling to JSON", err)
        return
    }

    _ = json.Unmarshal([]byte(result), &myRecord)
    fmt.Println("\nJSON:", myRecord)
}
```

В этой части программы данные XML преобразуются в формат JSON с помощью функций `Marshal()` и `Unmarshal()`, а затем выводятся на экран.

Выполнение `rwXML.go` приведет к следующим результатам:

```
$ go run rwXML.go readMe.json
JSON: {Mihalis Tsoukalos [{true 1234-567} {true 1234-abcd} {false
abcc-567}]}
xmlData: <?xml version="1.0" encoding="UTF-8"?>
<Record>
    <Name>Dimitris</Name>
    <Surname>Tsoukalos</Surname>
    <Tel>
        <Mobile>true</Mobile>
        <Number>1234-567</Number>
    </Tel>
    <Tel>
        <Mobile>true</Mobile>
        <Number>1234-abcd</Number>
    </Tel>
    <Tel>
        <Mobile>false</Mobile>
        <Number>abcc-567</Number>
    </Tel>
</Record>
JSON: {Dimitris Tsoukalos [{true 1234-567} {true 1234-abcd} {false abcc-567}]}
```

Чтение XML-файла

В этом подразделе вы узнаете, как прочитать XML-файл с диска и сохранить его содержимое в структуре Go. Программа, которая здесь рассмотрена, называется `readXML.go`. Разделим ее на три части.

Первая часть `readXML.go` выглядит так:

```
package main

import (
    "encoding/xml"
    "fmt"
    "os"
)

type Record struct {
    Name    string
    Surname    string
    Tel        []Telephone
}

type Telephone struct {
    Mobile bool
    Number string
}
```

Вторая часть `readXML.go` содержит следующий код:

```
func loadFromXML(filename string, key interface{}) error {
    in, err := os.Open(filename)
    if err != nil {
        return err
    }

    decodeXML := xml.NewDecoder(in)
    err = decodeXML.Decode(key)
    if err != nil {
        return err
    }
    in.Close()
    return nil
}
```

Представленный процесс очень похож на чтение с диска файла JSON.

Последняя часть `readXML.go` выглядит следующим образом:

```
func main() {
    arguments := os.Args
    if len(arguments) == 1 {
```

```
        fmt.Println("Please provide a filename!")
        return
    }

    filename := arguments[1]

    var myRecord Record
    err := loadFromXML(filename, &myRecord)

    if err == nil {
        fmt.Println("XML:", myRecord)
    } else {
        fmt.Println(err)
    }
}
```

Выполнение `readXML.go` приведет к следующим результатам:

```
$ go run readXML.go data.xml
XML: {Dimitris Tsoukalos [{true 1234-567} {true 1234-abcd} {false abcc-567}]}
```

Файл `data.xml` содержит следующие данные:

```
$ cat data.xml
xmlData: <?xml version="1.0" encoding="UTF-8"?>
<Record>
    <Name>Dimitris</Name>
    <Surname>Tsoukalos</Surname>
    <Tel>
        <Mobile>true</Mobile>
        <Number>1234-567</Number>
    </Tel>
    <Tel>
        <Mobile>true</Mobile>
        <Number>1234-abcd</Number>
    </Tel>
    <Tel>
        <Mobile>false</Mobile>
        <Number>abcc-567</Number>
    </Tel>
</Record>
```

Настройка вывода данных в формате XML

В этом подразделе вы узнаете, как изменить и настроить вид данных, выводимых в формате XML. Рассматриваемая утилита называется `modXML.go`, она представлена в трех частях. Обратите внимание, что данные, которые будут преобразованы в XML, жестко закодированы в программе для простоты.

Первая часть `modXML.go` выглядит следующим образом:

```
package main

import (
    "encoding/xml"
    "fmt"
    "os"
)

func main() {
    type Address struct {
        City, Country string
    }
    type Employee struct {
        XMLName    xml.Name  `xml:"employee"`
        Id         int       `xml:"id,attr"`
        FirstName  string    `xml:"name>first"`
        LastName   string    `xml:"name>last"`
        Initials   string    `xml:"name>initials"`
        Height     float32   `xml:"height,omitempty"`
        Address
        Comment    string    `xml:",comment"`
    }
```

Здесь определена структура XML-данных, а также дополнительная информация, касающаяся имени и типа элементов XML. В поле `XMLName` хранится имя записи XML, в данном случае это `employee`.

Поле с тегом `",comment"` является комментарием и форматируется при выводе данных соответствующим образом. Поле с тегом `",attr"` появляется на выходе как атрибут для предоставленного имени поля (в данном случае это `id`). Нотация `"name>first"` указывает Go на то, что нужно встроить тег `first` в тег `name`.

Наконец, поле с параметром `"omitempty"` при выводе пропускается, если оно пустое. Пустое значение — это `0`, `false`, нулевой указатель или интерфейс, а также любой массив, срез, хеш-таблица или строка нулевой длины.

Вторая часть `modXML.go` выглядит так:

```
    r := &Employee{Id: 7, FirstName: "Mihalis", LastName: "Tsoukalos",
                   Initials: "MIT"}
    r.Comment = "Technical Writer + DevOps"
    r.Address = Address{"SomeWhere 12", "12312, Greece"}
```

Здесь определяется и инициализируется структура `employee`.

Последняя часть `modXML.go` содержит следующий код Go:

```
    output, err := xml.MarshalIndent(r, " ", " ")
    if err != nil {
        fmt.Println("Error:", err)
    }
    output = []byte(xml.Header + string(output))
```

```
    os.Stdout.Write(output)
    os.Stdout.Write([]byte("\n"))
}
```

Выполнение modXML.go приведет к следующим результатам:

```
$ go run modXML.go
<?xml version="1.0" encoding="UTF-8"?>
    <employee id="7">
        <name>
            <first>Mihalis</first>
            <last>Tsoukalos</last>
            <initials>MIT</initials>
        </name>
        <City>SomeWhere 12</City>
        <Country>12312, Greece</Country>
        <!--Technical Writer + DevOps-->
    </employee>
```

Go и формат YAML

YAML (Ain't Markup Language) — это еще один очень популярный текстовый формат. Стандартная библиотека Go не поддерживает формат YAML, но есть пакет Go (https://github.com/go-yaml/yaml), который реализует поддержку YAML в Go.

Формат YAML поддерживается пакетом Viper, о котором речь пойдет в главе 8. Если вы хотите больше узнать о том, как Viper выполняет синтаксический анализ YAML-файлов, обратитесь к Go-коду Viper, размещенному по адресу https://github.com/spf13/viper.

Дополнительные ресурсы

Советую вам ознакомиться со следующими ресурсами:

- прочитайте документацию стандартного пакета Go regexp, которую вы найдете по адресу https://golang.org/pkg/regexp/;
- посетите главную страницу утилиты grep(1) и узнайте, как она поддерживает регулярные выражения;
- более подробную информацию о Go-пакете math/big вы найдете по адресу https://golang.org/pkg/math/big/;
- больше информации о YAML вы найдете на сайте https://yaml.org/;
- возможно, вам будет интересно изучить тип sync.Map, описанный по адресу https://golang.org/pkg/sync/;

- советую ознакомиться с документацией стандартного Go-пакета `unicode` по адресу https://golang.org/pkg/unicode/;
- возможно, поначалу это покажется трудным, но начните читать спецификацию языка программирования Go, размещенную по адресу https://golang.org/ref/spec.

Упражнения

- Попробуйте написать программу Go, которая выводит недопустимые части адреса IPv4.
- Можете ли вы сказать, чем отличается `make` от `new`, не заглядывая в текст главы?
- Используя код `findIPv4.go`, напишите программу Go, которая бы выводила самые популярные адреса IPv4, найденные в файле журнала, без обработки вывода какими-либо утилитами UNIX.
- Разработайте программу Go, которая бы находила в файле журнала адреса IPv4, вызвавшие сообщение об ошибке 404 HTML.
- Прочитайте файл JSON, в котором содержится десять целочисленных значений, сохраните его в переменной `struct`, увеличьте каждое целое значение на единицу и запишите измененный JSON на диск. Теперь сделайте то же самое для файла XML.
- Разработайте программу Go, которая бы находила в файле журнала все адреса IPv4, с которых были загружены ZIP-файлы.
- Используя стандартный Go-пакет `math/big`, напишите программу Go, которая бы вычисляла квадратные корни с высокой точностью — алгоритм выберите на свой вкус.
- Напишите утилиту Go, которая бы находила во входных данных дату и время, представленные в заданном формате, и возвращала бы только время.
- Помните ли вы, в чем различие между символом, байтом и руной?
- Разработайте утилиту Go для сопоставления целых чисел в диапазоне от 200 до 400 с использованием регулярного выражения.
- Попробуйте улучшить программу `keyValue.go`, добавив в нее журналирование.

Резюме

В этой главе раскрыты многие удобные функции Go, в том числе создание и использование структур, кортежей, строк и рун, а также функциональности стандартного Go-пакета `unicode`. Кроме того, вы узнали о сопоставлении с образцом, регулярных выражениях, обработке файлов JSON и XML, операторе `switch` и стандартном Go-пакете `strings`.

Наконец, вы разработали на Go хранилище типа «ключ — значение» и узнали, как использовать типы из пакета `math/big` для вычисления числа π с желаемой точностью.

В следующей главе будет показано, как группировать данные и манипулировать ими, используя более сложные механизмы, такие как двоичные деревья, связные списки, двунаправленные связные списки, очереди, стеки и пользовательские хеш-таблицы. Мы также рассмотрим структуры, которые реализованы в стандартном Go-пакете `container`; вы узнаете, как выполнять в Go матричные операции и как проверять на корректность головоломки судоку. Последней темой следующей главы будут случайные числа и генерирование трудноугадываемых строк, которые затем можно использовать в качестве безопасных паролей.

5 Как улучшить код Go с помощью структур данных

В предыдущей главе мы обсуждали составные типы данных, которые создаются с помощью ключевого слова `struct`, обработку в Go данных формата JSON и XML, а также такие темы, как регулярные выражения, сопоставление с образцом, кортежи, руны, строки, стандартные Go-пакеты `unicode` и `strings`. Наконец, мы написали на Go простое хранилище типа «ключ — значение».

Однако бывает так, что структуры, предлагаемые языком программирования, не соответствуют конкретной проблеме. В таких случаях приходится явным, специализированным способом создавать собственные структуры данных для хранения, поиска и получения данных.

Поэтому данная глава посвящена разработке и использованию в Go многих известных структур данных, в том числе *двоичных деревьев, связных списков, пользовательских хеш-таблиц, стеков* и *очередей*, а также исследованию их преимуществ. Поскольку никакое описание структуры данных не сравнится с наглядным изображением, в этой главе вам встретится много поясняющих рисунков.

В последних частях главы мы поговорим о проверке головоломок судоку и выполнении матричных вычислений с использованием срезов.

В этой главе рассмотрены следующие темы:

- *графы* и *узлы*;
- измерение сложности алгоритмов;
- двоичные деревья;
- пользовательские хеш-таблицы;
- связные списки;
- двунаправленные связные списки;
- работа с очередями в Go;
- стеки;
- структуры данных, предлагаемые стандартным Go-пакетом `container`;
- выполнение матричных вычислений;
- работа с головоломками судоку;

- генерация случайных чисел в Go;
- создание случайных строк, которые можно использовать в качестве паролей, трудно поддающихся взлому.

О графах и узлах

Граф (G (V, E)) — это конечное непустое множество вершин (или узлов) V и множество ребер E. Существует два основных типа графов: *циклические* и *ациклические*. Циклический граф — это граф, в котором все или некоторые вершины образуют замкнутый контур. В ациклических графах замкнутых контуров нет.

Направленный граф — это граф, с ребрами которого связано некое направление, а *направленный ациклический граф* — это направленный граф без замкнутых контуров.

Поскольку узел может содержать любую информацию, узлы обычно реализуются на основе структур Go благодаря их универсальности.

Сложность алгоритма

Эффективность алгоритма оценивается по его вычислительной сложности, которая главным образом определяется тем, сколько раз алгоритму для выполнения его работы требуется доступ к входным данным. Для описания сложности алгоритма в информатике используется *нотация «большое “O”»*. Так, алгоритм со сложностью $O(n)$, которому требуется лишь однократный доступ к входным данным, считается лучше, чем алгоритм со сложностью $O(n^2)$. Последний, в свою очередь, лучше, чем алгоритм $O(n^3)$, и т. д. Однако наихудшими являются алгоритмы с временем работы $O(n!)$. Это делает их практически непригодными для случаев, когда объем входных данных насчитывает более чем 300 элементов.

Наконец, большинство операций поиска в Go для встроенных типов, таких как поиск значения по ключу в хеш-таблице или доступ к элементу массива, имеют постоянное время, которое обозначается как $O(1)$. Это означает, что встроенные типы данных работают быстрее, чем созданные пользователем. Таким образом, как правило, лучше использовать именно встроенные типы, если только вы не хотите полностью контролировать то, что происходит внутри программы.

Кроме того, не все структуры данных по своей природе одинаковы. Как правило, операции с массивами выполняются быстрее, чем операции с хеш-таблицами, — такова цена, которую приходится платить за универсальность хеш-таблиц.

Несмотря на то что у каждого алгоритма есть свои недостатки, если только речь не идет о большом количестве данных, сложность алгоритма не особенно важна — лишь бы он точно выполнял свою работу.

Двоичные деревья в Go

Двоичное дерево — это структура данных, в которой под каждым узлом располагается **не более** двух других узлов. Это означает, что узел может быть связан с одним или максимум двумя другими узлами. Первый узел дерева называется *корнем дерева*. *Глубина дерева*, которую также называют *высотой дерева*, определяется как самый длинный путь от корня до узла, тогда как *глубина узла* — это количество ребер, соединяющих узел с корнем дерева. Узел дерева без дочерних узлов называется *листом*.

Дерево является *сбалансированным*, если наибольшая длина от корневого узла до листа не более чем на единицу превышает самую короткую длину. Иначе дерево является *несбалансированным*. Балансировка дерева может быть сложной и медленной операцией, поэтому лучше сохранять сбалансированность дерева с самого начала, а не пытаться сбалансировать уже созданное дерево, особенно если оно состоит из большого числа узлов.

На рис. 5.1 показано несбалансированное двоичное дерево. Его корневой узел — J, а узлы A, G, W и D являются листьями.

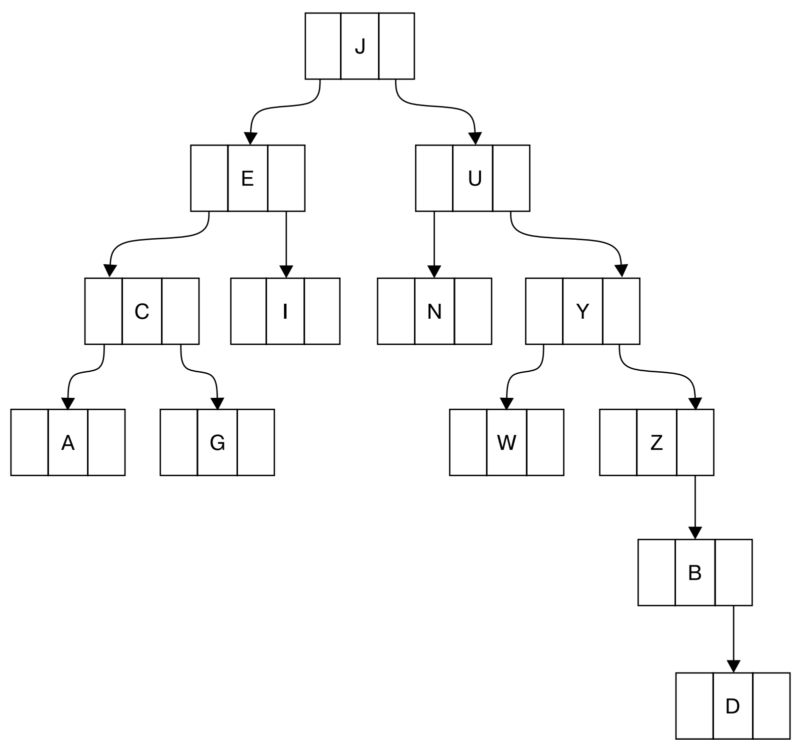


Рис. 5.1. Несбалансированное двоичное дерево

Реализация двоичного дерева в Go

В этом разделе показано, как реализовать двоичное дерево в Go. В качестве примера мы рассмотрим исходный код, который находится в файле `binTree.go`. Разделим `binTree.go` на пять частей. Первая часть программы выглядит так:

```
package main

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

type Tree struct {
    Left     *Tree
    Value    int
    Right    *Tree
}
```

Здесь мы видим определение узла дерева с использованием структуры Go. Поскольку у нас нет реальных данных, мы заполняем дерево случайными числами, для чего используем пакет `math/rand`.

Вторая часть `binTree.go` содержит следующий код Go:

```
func traverse(t *Tree) {
    if t == nil {
        return
    }
    traverse(t.Left)
    fmt.Print(t.Value, " ")
    traverse(t.Right)
}
```

На примере функции `traverse()` вы видите, как можно пройти по всем узлам двоичного дерева, используя рекурсию.

Третий фрагмент кода выглядит следующим образом:

```
func create(n int) *Tree {
    var t *Tree
    rand.Seed(time.Now().Unix())
    for i := 0; i < 2*n; i++ {
        temp := rand.Intn(n * 2)
        t = insert(t, temp)
    }
    return t
}
```

Функция `create()` используется только для заполнения двоичного дерева случайными целыми числами.

Четвертая часть программы выглядит так:

```
func insert(t *Tree, v int) *Tree {
    if t == nil {
        return &Tree{nil, v, nil}
    }
    if v == t.Value {
        return t
    }
    if v < t.Value {
        t.Left = insert(t.Left, v)
        return t
    }
    t.Right = insert(t.Right, v)
    return t
}
```

Функция `insert()` выполняет много важного[1], используя операторы `if`. Первый оператор `if` проверяет, является ли дерево пустым. Если дерево действительно пустое, то новый узел будет корневым и он будет создан как `&Tree{nil, v, nil}`.

Второй оператор `if` определяет, существует ли значение, которое мы пытаемся вставить, в обрабатываемом узле двоичного дерева. Если такое значение в узле дерева уже существует, то функция возвращет указатель на обрабатываемый узел и завершается, ничего не делая.

Третий оператор `if` определяет, будет ли значение, которое мы пытаемся вставить, размещаться слева или справа от проверяемого в данный момент узла, и действует соответственно. Обратите внимание, что в представленной реализации создается несбалансированное двоичное дерево.

Последняя часть `binTree.go` содержит следующий код Go:

```
func main() {
    tree := create(10)
    fmt.Println("The value of the root of the tree is", tree.Value)
    traverse(tree)
    fmt.Println()
    tree = insert(tree, -10)
    tree = insert(tree, -2)
    traverse(tree)
    fmt.Println()
    fmt.Println("The value of the root of the tree is",
    tree.Value)
}
```

[1] Нужно отметить, что функция insert() является рекурсивной, поэтому &Tree{nil, v, nil} создает не только корневой узел дерева, но и все листья, которые потом встраиваются в родительский узел операторами присваивания t.Left = insert(t.Left, v) и t.Right = = insert(t.Right, v).

Выполнение `binTree.go` приведет к следующему результату:

```
$ go run binTree.go
The value of the root of the tree is 18
0 3 4 5 7 8 9 10 11 14 16 17 18 19
-10 -2 0 3 4 5 7 8 9 10 11 14 16 17 18 19
The value of the root of the tree is 18
```

Преимущества двоичных деревьев

Нет ничего лучше дерева, если нужно представить иерархические данные. По этой причине деревья широко используются компиляторами языка программирования для анализа компьютерных программ.

Кроме того, деревья имеют **упорядоченную** структуру, так что не приходится прилагать усилий для их упорядочения; достаточно лишь размещать элементы в нужных местах, и дерево останется упорядоченным. Однако вследствие способа построения деревьев удаление элементов из дерева не всегда является простой операцией.

Если двоичное дерево сбалансировано, то операции поиска, вставки и удаления в нем выполняются за $\log(n)$ шагов, где n — количество элементов, которые содержит дерево. Кроме того, высота сбалансированного двоичного дерева приблизительно равна $\log_2(n)$. Другими словами, высота сбалансированного дерева из 10 000 элементов примерно равна 14, что на удивление мало.

Аналогично высота сбалансированного дерева из 100 000 элементов составляет примерно 17, а сбалансированного дерева из 1 000 000 элементов — около 20. Таким образом, значительное увеличение количества элементов в сбалансированном двоичном дереве не вызывает сильный рост высоты дерева. Иначе говоря, мы можем достичь любого узла дерева из 1 000 000 узлов менее чем за 20 шагов!

Основным недостатком двоичных деревьев является то, что вид дерева зависит от последовательности вставки в него элементов. Если у дерева длинные и сложные ключи, то вставка или поиск элемента могут оказаться медленными из-за большого количества требуемых операций сравнения. Наконец, если дерево не сбалансировано, то скорость выполняемых в нем операций будет непредсказуемой.

Связный список или массив создается быстрее, чем двоичное дерево, однако гибкость, которую обеспечивает двоичное дерево в операциях поиска, возможно, стоит дополнительных затрат и обслуживания.

При поиске элемента в двоичном дереве мы каждый раз проверяем, является ли значение искомого элемента больше или меньше значения текущего узла, и используем это решение, чтобы выбрать, по какой части дерева перейти вниз. Это экономит много времени.

Пользовательские хеш-таблицы в Go

Хеш-таблица — это структура данных, в которой хранится одна или несколько пар «ключ — значение» и используется *хеш-функция* для вычисления индекса в массиве блоков, или слотов памяти, в котором хранится нужное значение. В идеале хеш-функция должна назначать каждому ключу уникальный блок — при наличии необходимого количества блоков (их обычно достаточно).

Хорошая хеш-функция должна обеспечивать равномерное распределение значений хеш-функции, потому что неэффективно иметь неиспользуемые блоки или большие различия в размере блоков. Кроме того, хеш-функция должна работать предсказуемо и выдавать одинаковое хеш-значение для идентичных ключей (рис. 5.2). Иначе было бы невозможно найти нужную информацию.

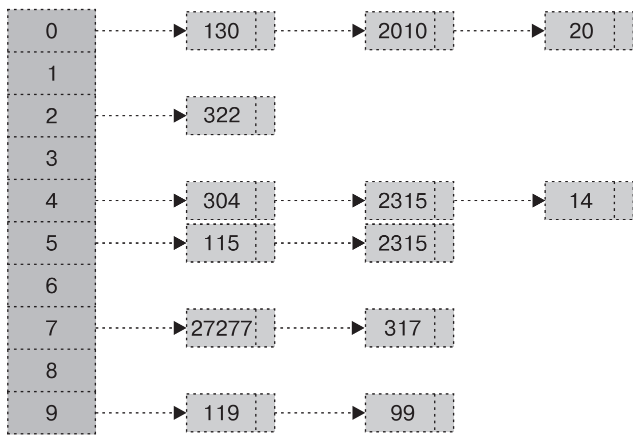


Рис. 5.2. Хеш-таблица из десяти блоков

Реализация пользовательской хеш-таблицы в Go

Go-код `hashTable.go`, который мы разделим на пять частей, поможет понять многое о пользовательских хеш-таблицах.

Первая часть `hashTable.go` выглядит следующим образом:

```
package main

import (
    "fmt"
)

const SIZE = 15
```

```
type Node struct {
    Value int
    Next *Node
}
```

Здесь мы видим определение узла пользовательской хеш-таблицы, который, как и ожидалось, устанавливается с помощью структуры Go. Именованная константа `SIZE` выявляет количество блоков пользовательской хеш-таблицы.

Второй фрагмент `hashTable.go` содержит следующий код Go:

```
type HashTable struct {
    Table map[int]*Node
    Size int
}

func hashFunction(i, size int) int {
    return (i % size)
}
```

В этом фрагменте кода мы видим реализацию хеш-функции, используемой для конкретного хеша. В функции `hashFunction()` используется *оператор деления по модулю*. Основная причина выбора оператора деления по модулю (взятие остатка) заключается в том, что данная пользовательская хеш-таблица должна обрабатывать целочисленные значения. Если бы мы имели дело со строками или числами с плавающей точкой, то мы бы использовали в хеш-функции другую логику.

Физически хеш хранится в структуре `HashTable`, которая состоит из двух полей. Первое поле — это хеш-таблица Go (map), которая связывает целое число со связным списком (`*Node`), а второе — размер пользовательской хеш-таблицы. В результате пользовательская хеш-таблица будет иметь столько связных списков, сколько в ней блоков. Это также означает, что узлы каждого блока такой хеш-таблицы будут храниться в связных списках. Со связными списками вы вскоре познакомитесь ближе.

Третья часть кода `hashTable.go` выглядит так:

```
func insert(hash *HashTable, value int) int {
    index := hashFunction(value, hash.Size)
    element := Node{Value: value, Next: hash.Table[index]}
    hash.Table[index] = &element
    return index
}
```

Функция `insert()` вызывается для вставки элементов в пользовательскую хеш-таблицу. Обратите внимание, что в данной реализации функции `insert()` не проверяется наличие дублирующихся значений.

Четвертая часть `hashTable.go` выглядит так:

```
func traverse(hash *HashTable) {
    for k := range hash.Table {
```

```
        if hash.Table[k] != nil {
            t := hash.Table[k]
            for t != nil {
                fmt.Printf("%d -> ", t.Value)
                t = t.Next
            }
            fmt.Println()
        }
    }
}
```

Функция `traverse()` используется для вывода всех значений пользовательской хеш-таблицы. Эта функция перебирает все связные списки данной хеш-таблицы по одному и выводит хранящиеся в них значения.

Последняя часть кода `hashTable.go` выглядит так:

```
func main() {
    table := make(map[int]*Node, SIZE)
    hash := &HashTable{Table: table, Size: SIZE}
    fmt.Println("Number of spaces:", hash.Size)
    for i := 0; i < 120; i++ {
        insert(hash, i)
    }
    traverse(hash)
}
```

В этой части кода создается новая пользовательская хеш-таблица с именем `hash`, для чего используется переменная `table`. Эта переменная является хеш-таблицей Go, в которой хранятся блоки пользовательской хеш-таблицы. Как вы уже знаете, блоки пользовательской хеш-таблицы реализованы с использованием связных списков.

Основная причина, по которой для хранения связных списков хеш-таблицы использована хеш-таблица Go, а не срез или массив, заключается в том, что ключи среза или массива могут быть только положительными целыми числами, а ключи хеш-таблицы Go могут быть практически любыми.

Выполнение `hashTable.go` приведет к следующим результатам:

```
$ go run hashTable.go
Number of spaces: 15
105 -> 90 -> 75 -> 60 -> 45 -> 30 -> 15 -> 0 ->
110 -> 95 -> 80 -> 65 -> 50 -> 35 -> 20 -> 5 ->
114 -> 99 -> 84 -> 69 -> 54 -> 39 -> 24 -> 9 ->
118 -> 103 -> 88 -> 73 -> 58 -> 43 -> 28 -> 13 ->
119 -> 104 -> 89 -> 74 -> 59 -> 44 -> 29 -> 14 ->
108 -> 93 -> 78 -> 63 -> 48 -> 33 -> 18 -> 3 ->
112 -> 97 -> 82 -> 67 -> 52 -> 37 -> 22 -> 7 ->
113 -> 98 -> 83 -> 68 -> 53 -> 38 -> 23 -> 8 ->
116 -> 101 -> 86 -> 71 -> 56 -> 41 -> 26 -> 11 ->
106 -> 91 -> 76 -> 61 -> 46 -> 31 -> 16 -> 1 ->
```

```
107 -> 92 -> 77 -> 62 -> 47 -> 32 -> 17 -> 2 ->
109 -> 94 -> 79 -> 64 -> 49 -> 34 -> 19 -> 4 ->
117 -> 102 -> 87 -> 72 -> 57 -> 42 -> 27 -> 12 ->
111 -> 96 -> 81 -> 66 -> 51 -> 36 -> 21 -> 6 ->
115 -> 100 -> 85 -> 70 -> 55 -> 40 -> 25 -> 10 ->
```

Данная хеш-таблица идеально сбалансирована, поскольку она имеет дело с непрерывными числами, которые помещаются в блок в соответствии с результатами оператора деления по модулю. В реальных задачах таких удобных результатов может и не быть!

Остаток от евклидова деления натурального числа a на натуральное число b вычисляется по формуле a = bq + r, где q — частное, а r — остаток. Допустимые значения остатка находятся в пределах от 0 до b – 1. Это и является возможным результатом оператора деления по модулю (взятия остатка).

Обратите внимание, что если выполнить `hashTable.go` несколько раз, то, скорее всего, получим результаты, в которых строки будут располагаться в разной последовательности, поскольку способ вывода пар «ключ — значение» для хеш-таблицы Go является преднамеренно абсолютно случайным, поэтому на него нельзя полагаться.

Реализация функции поиска

В этом разделе мы рассмотрим реализацию функции `lookup()`, которая позволяет определить, существует ли данный элемент в пользовательской хеш-таблице. Код функции `lookup()` основан на `traverse()`:

```
func lookup(hash *HashTable, value int) bool {
    index := hashFunction(value, hash.Size)
    if hash.Table[index] != nil {
        t := hash.Table[index]
        for t != nil {
            if t.Value == value {
                return true
            }
            t = t.Next
        }
    }
    return false
}
```

Этот код находится в исходном файле `hashTableLookup.go`. Выполнение `hashTableLookup.go` приведет к следующим результатам:

```
$ go run hashTableLookup.go
120 is not in the hash table!
```

```
121 is not in the hash table!
122 is not in the hash table!
123 is not in the hash table!
124 is not in the hash table!
```

Как видите, функция `lookup()` хорошо справляется со своей работой.

Преимущества пользовательских хеш-таблиц

Если вам показалось, что пользовательские хеш-таблицы не особенно полезны, удобны или интеллектуальны, то подумайте о следующем: если пользовательская хеш-таблица имеет n ключей и k блоков, то, в отличие от линейного поиска со скоростью $O(n)$, скорость поиска для n ключей в такой пользовательской хеш-таблице равна $O(n/k)$. Это улучшение может показаться незначительным, однако для массива хешей, состоящего всего из 20 блоков, время поиска сократится в 20 раз! Это делает пользовательские хеш-таблицы идеальными для таких приложений, как словари, и аналогичных, где приходится искать большие объемы данных.

Связные списки в Go

Связный список — это структура данных, состоящая из конечного множества элементов. Здесь каждый элемент состоит как минимум из двух областей памяти: одна используется для хранения фактических данных, а вторая — для хранения указателя, который связывает текущий элемент со следующим, создавая таким образом последовательность элементов, образующих связный список.

Первый элемент связного списка называется *головным* (head), а последний элемент — *хвостовым* (tail). Первое, что следует сделать при определении связного списка, — это сохранить его головной элемент в отдельной переменной, потому что головной элемент — это единственное, что нужно для доступа ко всему связному списку. Обратите внимание, что если потерять указатель на первый узел односвязного списка, то найти его снова будет невозможно. На рис. 5.3 представлен связный список из пяти узлов.

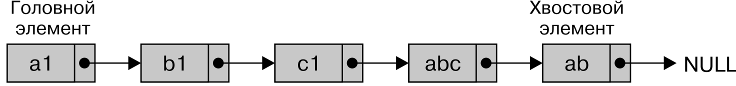


Рис. 5.3. Связный список из пяти узлов

На рис. 5.4 показано, как удалить узел из связного списка, что позволит вам лучше понять этапы данного процесса. Главное, что нужно сделать, — это расположить

указатель на узел, находящийся слева от удаляемого узла, чтобы он ссылался на узел справа от удаляемого узла.

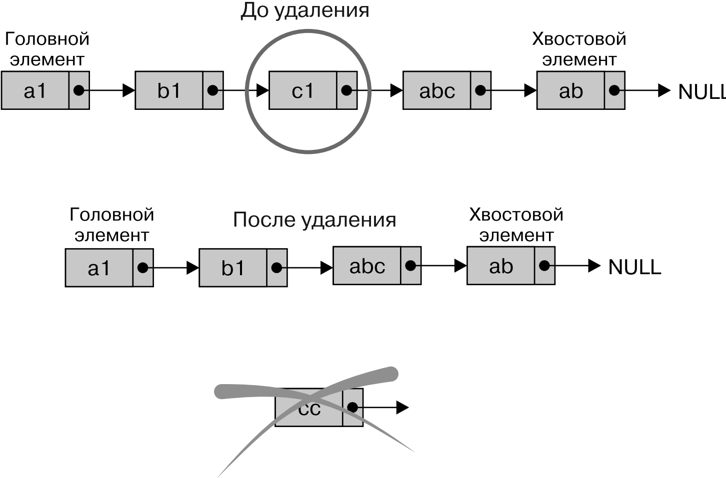


Рис. 5.4. Удаление узла из связного списка

Приведенная далее реализация связного списка является относительно простой и не включает в себя функцию удаления узла — реализация этого функционала оставлена вам в качестве упражнения.

Реализация связного списка в Go

Исходный файл Go, в котором реализован связный список, называется `connectedList.go`. Рассмотрим его, разделив на пять частей.

Первый фрагмент кода файла `LinkList.go` выглядит так:

```
package main

import (
    "fmt"
)

type Node struct {
    Value int
    Next *Node
}

var root = new(Node)
```

В этой части программы определяется структурный тип `Node`, который будет использоваться для узлов связного списка, а также глобальная переменная `root` — она содержит первый элемент связного списка, который будет доступен везде в коде. Учтите, что использование глобальных переменных, вообще говоря, приемлемо разве что для небольших программ и примеров кода — в больших программах это может привести к ошибкам.

Вторая часть файла `linkedList.go` содержит следующий код Go:

```
func addNode(t *Node, v int) int {
    if root == nil {
        t = &Node{v, nil}
        root = t
        return 0
    }

    if v == t.Value {
        fmt.Println("Node already exists:", v)
        return -1
    }

    if t.Next == nil {
        t.Next = &Node{v, nil}
        return -2
    }

    return addNode(t.Next, v)
}
```

Вследствие способа работы связных списков в таких списках обычно не содержатся повторяющиеся записи. Более того, новые узлы обычно добавляются в конец связного списка, если связный список не отсортирован. Именно таким образом функция `addNode()` используется для добавления новых узлов в связный список.

В этой реализации предусмотрены три отдельных варианта, которые рассматриваются с использованием операторов `if`. В первом случае проверяем, имеем ли мы дело с пустым связным списком. Во втором — существует ли уже в списке значение, которое мы хотим добавить. В третьем случае проверяем, достигли ли мы конца связного списка, и тогда добавляем новый узел в конец списка с желаемым значением, используя код `t.Next = &Node{v, nil}`. Если ни одно из этих условий не выполняется, то мы повторяем этот процесс рекурсивным вызовом функции `addNode()` для следующего узла связного списка, используя код `return addNode(t.Next, v)`.

Третий фрагмент кода программы `connectedList.go` содержит реализацию функции `traverse()`:

```
func traverse(t *Node) {
    if t == nil {
```

```
        fmt.Println("-> Empty list!")
        return
    }

    for t != nil {
        fmt.Printf("%d -> ", t.Value)
        t = t.Next
    }
    fmt.Println()
}
```

Четвертая часть `linkedList.go` содержит следующий код:

```
func lookupNode(t *Node, v int) bool {
    if root == nil {
        t = &Node{v, nil}
        root = t
        return false
    }

    if v == t.Value {
        return true
    }

    if t.Next == nil {
        return false
    }

    return lookupNode(t.Next, v)
}

func size(t *Node) int {
    if t == nil {
        fmt.Println("-> Empty list!")
        return 0
    }

    i := 0
    for t != nil {
        i++
        t = t.Next
    }
    return i
}
```

В этой части вы видите реализацию двух очень удобных функций: `lookupNode()` и `size()`. Первая из них проверяет, существует ли данный элемент в связном списке, а вторая возвращает размер связного списка — количество узлов в этом списке.

Логика реализации функции `lookupNode()` проста и понятна: мы перебираем по очереди все элементы односвязного списка, пока не найдем нужное значение. Если мы дойдем до конца связного списка, не найдя этого значения, то поймем, что в данном списке это значение не содержится.

Последняя часть файла `connectedList.go` содержит реализацию функции `main()`:

```
func main() {
    fmt.Println(root)
    root = nil
    traverse(root)
    addNode(root, 1)
    addNode(root, -1)
    traverse(root)
    addNode(root, 10)
    addNode(root, 5)
    addNode(root, 45)
    addNode(root, 5)
    addNode(root, 5)
    traverse(root)
    addNode(root, 100)
    traverse(root)

    if lookupNode(root, 100) {
        fmt.Println("Node exists!")
    } else {
        fmt.Println("Node does not exist!")
    }

    if lookupNode(root, -100) {
        fmt.Println("Node exists!")
    } else {
        fmt.Println("Node does not exist!")
    }
}
```

Выполнение `linkedList.go` приведет к следующим результатам:

```
$ go run linkedList.go
&{0 <nil>}
-> Empty list!
1 -> -1 ->
Node already exists: 5
Node already exists: 5
1 -> -1 -> 10 -> 5 -> 45 ->
1 -> -1 -> 10 -> 5 -> 45 -> 100 ->
Node exists!
Node does not exist!
```

Преимущества связных списков

Самым большим преимуществом связных списков является то, что они просты для понимания и реализации и достаточно универсальны, чтобы их можно было использовать в разных ситуациях. Связные списки применяются для моделирования различных типов данных, начиная с отдельных значений и заканчивая сложными структурами со многими полями. Кроме того, в связных списках очень быстро выполняется последовательный поиск с помощью указателей.

Связные списки помогают не только сортировать данные, но и сохранять порядок данных даже после вставки или удаления элементов. Удаление узла из упорядоченного связного списка выполняется так же, как и в неупорядоченном связном списке; однако вставка нового узла в упорядоченный связный список отличается тем, что новый узел следует поместить в нужное место, чтобы список оставался упорядоченным. На практике это означает, что если у вас есть много данных и вы знаете, что нужно будет все время удалять данные, то лучше использовать связный список, а не пользовательскую хеш-таблицу или двоичное дерево.

Наконец, в *упорядоченных связных списках* можно применять различные методы оптимизации при поиске или вставке узла. Самым распространенным из этих методов является сохранение указателя на центральный узел упорядоченного связного списка, чтобы начинать поиск именно с данного указателя. Эта простая оптимизация позволяет сократить время операции поиска в два раза!

Двусвязные списки в Go

Двусвязный (двунаправленный связный) список — это список, в котором каждый узел хранит указатель на предыдущий и на следующий элементы списка. На рис. 5.5 представлен двунаправленный связный список.

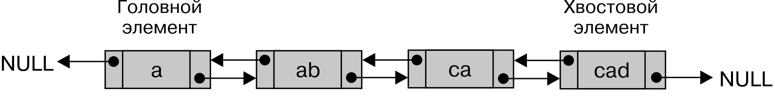


Рис. 5.5. Двунаправленный связный список

Таким образом, в двусвязном списке ссылка первого узла на следующий узел указывает на второй узел, а его ссылка на предыдущий элемент указывает на `nil` (константа, или *NULL*). Аналогично ссылка последнего узла на следующий элемент указывает на `nil`, а его ссылка на предыдущий элемент — на предпоследний узел двусвязного списка.

На рис. 5.6 показано добавление узла в двусвязный список. Как вы понимаете, в данном случае основная задача — это работа с указателями трех узлов: нового узла, узла, расположенного слева от нового узла, и узла, находящегося справа от нового узла.

Таким образом, в действительности основное различие между односвязным и двусвязным списком состоит в том, что последний требует больше операций по обслуживанию. Такова цена, которую приходится платить за возможность доступа к двусвязному списку в обоих направлениях.

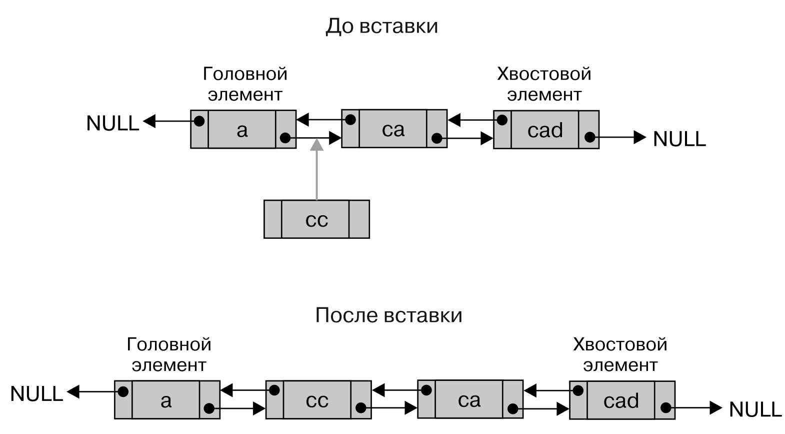


Рис. 5.6. Вставка нового узла в середину двусвязного списка

Реализация двусвязного списка в Go

Программа, реализующая двусвязный список на Go, называется `doublelyLList.go`. Рассмотрим ее, разделив на пять частей. Общая идея двусвязного списка такая же, как и у односвязного, но здесь из-за наличия двух указателей в каждом узле списка придется выполнять больше операций по обслуживанию списка.

Первая часть `dublyLList.go` выглядит так:

```
package main

import (
    "fmt"
)

type Node struct {
    Value     int
    Previous  *Node
    Next      *Node
}
```

В данной части мы видим определение узла двусвязного списка как структуры Go. Однако на этот раз по понятным причинам структура имеет два поля указателей.

Вторая часть файла `duplyLList.go` содержит следующий код Go:

```
func addNode(t *Node, v int) int {
    if root == nil {
        t = &Node{v, nil, nil}
        root = t
        return 0
    }

    if v == t.Value {
        fmt.Println("Node already exists:", v)
        return -1
    }

    if t.Next == nil {
        temp := t
        t.Next = &Node{v, temp, nil}
        return -2
    }

    return addNode(t.Next, v)
}
```

Как и в случае односвязного списка, каждый новый узел помещается в конец текущего двусвязного списка. Однако так поступать не обязательно, если вы хотите построить упорядоченный двусвязный список.

Третья часть `doublyLList.go` выглядит так:

```
func traverse(t *Node) {
    if t == nil {
        fmt.Println("-> Empty list!")
        return
    }

    for t != nil {
        fmt.Printf("%d -> ", t.Value)
        t = t.Next
    }
    fmt.Println()
}

func reverse(t *Node) {
    if t == nil {
        fmt.Println("-> Empty list!")
        return
    }

    temp := t
    for t != nil {
```

```
        temp = t
        t = t.Next
    }

    for temp.Previous != nil {
        fmt.Printf("%d -> ", temp.Value)
        temp = temp.Previous
    }
    fmt.Printf("%d -> ", temp.Value)
    fmt.Println()
}
```

Здесь вы видите код Go для функций `traverse()` и `reverse()`. Реализация функции `traverse()` такая же, как и в программе `connectedList.go`. Однако логика функции `reverse()` очень интересна. Поскольку мы не храним указатель на конец двусвязного списка, то нам нужно перейти к концу такого списка, чтобы получить доступ к его узлам в обратном порядке.

Обратите внимание, что Go позволяет писать такой код, как `a, b = b, a`, чтобы поменять местами значения двух переменных, не создавая временную переменную.

Четвертая часть `dublyLList.go` содержит следующий код Go:

```
func size(t *Node) int {
    if t == nil {
        fmt.Println("-> Empty list!")
        return 0
    }

    n := 0
    for t != nil {
        n++
        t = t.Next
    }
    return n
}

func lookupNode(t *Node, v int) bool {
    if root == nil {
        return false
    }

    if v == t.Value {
        return true
    }

    if t.Next == nil {
        return false
    }
    return lookupNode(t.Next, v)
}
```

Последний фрагмент файла `duplyLList.go` содержит следующий код Go:

```
var root = new(Node)

func main() {
    fmt.Println(root)
    root = nil
    traverse(root)
    addNode(root, 1)
    addNode(root, 1)
    traverse(root)
    addNode(root, 10)
    addNode(root, 5)
    addNode(root, 0)
    addNode(root, 0)
    traverse(root)
    addNode(root, 100)
    fmt.Println("Size:", size(root))
    traverse(root)
    reverse(root)
}
```

Если выполнить `duplyLList.go`, то получим следующий результат:

```
$ go run doublyLList.go
&{0 <nil> <nil>}
-> Empty list!
Node already exists: 1
1 ->
Node already exists: 0
1 -> 10 -> 5 -> 0 ->
Size: 5
1 -> 10 -> 5 -> 0 -> 100 ->
100 -> 0 -> 5 -> 10 -> 1 ->
```

Как видите, функция `reverse()` работает просто отлично!

Преимущества двусвязных списков

Двунаправленные связные списки более универсальны, чем односвязные, поскольку по ним можно перемещаться в любом направлении, в них легче вставлять и удалять элементы. Кроме того, даже если вы потеряете указатель на заголовок двусвязного списка, то все равно сможете найти головной узел такого списка. Однако за эту универсальность приходится платить: поддерживать два указателя для каждого узла. Разработчик должен сам решить, насколько оправданна такая дополнительная сложность. Например, в вашем музыкальном проигрывателе двусвязный список может использоваться для возможности перехода к предыдущей и к следующей композиции.

Очереди в Go

Очередь — это особый вид связного списка, в котором элементы вставляются в начало, а удаляются из хвоста. Нам не понадобится рисунок для описания очереди: представьте себе, что вы идете в банк и ждете, пока те, кто пришел раньше вас, решат свои вопросы, прежде чем вы сможете пообщаться с кассиром.

Основным преимуществом очередей является простота: для доступа к очереди нужны всего две функции. А чем меньше функций, тем меньше вам придется беспокоиться о том, что что-нибудь пойдет не так. Вы можете реализовать очередь любым удобным способом, лишь бы обеспечить поддержку этих двух функций.

Реализация очереди в Go

Программа, на примере которой мы проиллюстрируем реализацию очереди в Go, называется `queue.go`. Разделим ее на пять частей. Обратите внимание, что для реализации очереди используется связный список. Функции `Push()` и `Pop()` предназначены для вставки и удаления узлов из очереди соответственно.

Первая часть кода `queue.go` выглядит так:

```
package main

import (
    "fmt"
)

type Node struct {
    Value int
    Next *Node
}

var size = 0
var queue = new(Node)
```

Наличие переменной `size` для хранения количества узлов в очереди удобно, но не обязательно. В представленной здесь реализации это сделано для того, чтобы упростить работу. Вероятно, вы тоже захотите создать эти поля в своей структуре.

Вторая часть `queue.go` содержит следующий код Go:

```
func Push(t *Node, v int) bool {
    if queue == nil {
        queue = &Node{v, nil}
        size++
        return true
    }

    t = &Node{v, nil}
```

```
    t.Next = queue
    queue = t
    size++

    return true
}
```

Здесь показана реализация функции `Push()`, которая сама по себе очень проста. Если очередь пуста, то новый узел становится очередью. Если очередь не пуста, то создается новый узел, который помещается перед текущей очередью. После этого только что созданный узел становится заголовком очереди.

Третья часть `queue.go` содержит следующий код Go:

```
func Pop(t *Node) (int, bool) {
    if size == 0 {
        return 0, false
    }

    if size == 1 {
        queue = nil
        size--
        return t.Value, true
    }

    temp := t
    for (t.Next) != nil {
        temp = t
        t = t.Next
    }

    v := (temp.Next).Value
    temp.Next = nil

    size--
    return v, true
}
```

В этом коде показана реализация функции `Pop()`, которая удаляет из очереди самый старый элемент. Если очередь пуста (`size == 0`), то из нее ничего не извлекается.

Если в очереди есть только один узел, то извлекается значение этого узла, после чего очередь становится пустой. В противном случае извлекается последний элемент очереди, удаляется последний узел и изменяются необходимые указатели, после чего возвращается нужное значение.

Четвертая часть `queue.go` содержит следующий код Go:

```
func traverse(t *Node) {
    if size == 0 {
        fmt.Println("Empty Queue!")
```

```
        return
    }

    for t != nil {
        fmt.Printf("%d -> ", t.Value)
        t = t.Next
    }
    fmt.Println()
}
```

Функция `traverse()` не обязательна для работы с очередью, но она удобна для просмотра всех узлов очереди.

Последний фрагмент кода `queue.go` содержит следующий код Go:

```
func main() {
    queue = nil
    Push(queue, 10)
    fmt.Println("Size:", size)
    traverse(queue)

    v, b := Pop(queue)
    if b {
        fmt.Println("Pop:", v)
    }
    fmt.Println("Size:", size)

    for i := 0; i < 5; i++ {
        Push(queue, i)
    }
    traverse(queue)
    fmt.Println("Size:", size)

    v, b = Pop(queue)
    if b {
        fmt.Println("Pop:", v)
    }
    fmt.Println("Size:", size)

    v, b = Pop(queue)
    if b {
        fmt.Println("Pop:", v)
    }
    fmt.Println("Size:", size)
    traverse(queue)
}
```

Почти весь код Go в функции `main()` предназначен для проверки работы очереди. Самая важная часть этого кода — два оператора `if`, которые позволяют узнать, вернула ли функция `Pop()` значение, или очередь была пуста и функция не вернула ничего.

Выполнение `queue.go` приведет к следующим результатам:

```
$ go run queue.go
Size: 1
10 ->
Pop: 10
Size: 0
4 -> 3 -> 2 -> 1 -> 0 ->
Size: 5
Pop: 0
Size: 4
Pop: 1
Size: 3
4 -> 3 -> 2 ->
```

Стеки в Go

Стек — это структура данных, которая похожа на стопку тарелок: когда вам понадобится новая тарелка, вы возьмете верхнюю.

Как и в случае с очередью, основным преимуществом стека является его простота, поскольку, чтобы работать со стеком, нужно реализовать всего две функции: добавление в стек нового узла и удаление узла из стека.

Реализация стека в Go

Пришло время рассмотреть реализацию стека в Go. Мы покажем это на примере программы `stack.go`. Как и в случае с очередью, для реализации стека будет использоваться связный список. Как вы знаете, нам понадобятся две функции: одна, с именем `Push()`, для вставки в стек, и вторая, с именем `Pop()`, — для удаления из стека.

Хотя в этом нет необходимости, однако полезно хранить количество элементов в стеке в отдельной переменной, чтобы можно было определить, является ли стек пустым, без необходимости обращаться к самому связному списку.

Мы рассмотрим исходный код `stack.go`, разделив его на четыре части. Первая часть выглядит так:

```
package main

import (
    "fmt"
)

type Node struct {
    Value int
```

```
    Next *Node
}

var size = 0
var stack = new(Node)
```

Во второй части `stack.go` содержится реализация функции `Push()`:

```
func Push(v int) bool {
    if stack == nil {
        stack = &Node{v, nil}
        size = 1
        return true
    }

    temp := &Node{v, nil}
    temp.Next = stack
    stack = temp
    size++
    return true
}
```

Если стек не пустой, то функция создает новый узел (`temp`), который помещается перед текущим стеком. Затем новый узел становится головным узлом стека. Данная версия функции `Push()` всегда возвращает значение `true`, но если занимаемое стеком пространство ограниченно, то можно изменить код так, чтобы функция возвращала `false` при попытке превысить емкость стека.

В третьей части содержится реализация функции `Pop()`:

```
func Pop(t *Node) (int, bool) {
    if size == 0 {
        return 0, false
    }

    if size == 1 {
        size = 0
        stack = nil
        return t.Value, true
    }

    stack = stack.Next
    size--
    return t.Value, true
}
```

Четвертый фрагмент кода `stack.go` выглядит так:

```
func traverse(t *Node) {
    if size == 0 {
        fmt.Println("Empty Stack!")
        return
    }
```

```
    for t != nil {
        fmt.Printf("%d -> ", t.Value)
        t = t.Next
    }
    fmt.Println()
}
```

Поскольку стек реализован с использованием связного списка, для него можно реализовать функцию `traverse()`.

Последняя часть `stack.go` содержит следующий код Go:

```
func main() {
    stack = nil
    v, b := Pop(stack)
    if b {
        fmt.Print(v, " ")
    } else {
        fmt.Println("Pop() failed!")
    }

    Push(100)
    traverse(stack)
    Push(200)
    traverse(stack)

    for i := 0; i < 10; i++ {
        Push(i)
    }

    for i := 0; i < 15; i++ {
        v, b := Pop(stack)
        if b {
            fmt.Print(v, " ")
        } else {
            break
        }
    }
    fmt.Println()
    traverse(stack)
}
```

Как видно, исходный код `stack.go` немного короче, чем код Go для `queue.go`, — прежде всего потому, что принцип стека проще, чем у очереди.

Выполнение `stack.go` приведет к следующим результатам:

```
$ go run stack.go
Pop() failed!
100 ->
200 -> 100 ->
9 8 7 6 5 4 3 2 1 0 200 100
Empty Stack!
```

Внимание! Теперь вы знаете, как можно использовать связный список для реализации пользовательской хеш-таблицы, очереди и стека. Эти примеры должны помочь вам осознать полезность и важность связных списков в программировании в частности и в информатике в целом.

Пакет container

В этом разделе я расскажу об использовании стандартного пакета Go `container`. Пакет `container` поддерживает три структуры данных: *кучу*, *список* и *кольцо*. Эти структуры данных реализованы в пакетах `container/heap`, `container/list` и `container/ring` соответственно.

На случай, если вы не знаете: кольцо представляет собой *циклический список*, так что последний элемент кольца указывает на его первый элемент. По сути, это означает, что все узлы кольца эквивалентны, кольцо не имеет ни начала, ни конца. В результате можно пройти через все кольцо, начиная с любого элемента.

В следующих трех подразделах описаны все пакеты, входящие в состав `container`. Совет: если функциональность стандартного Go-пакета `container` соответствует вашим потребностям, используйте его; в противном случае лучше реализовать и использовать собственные структуры данных.

Использование пакета container/heap

В этом подразделе мы рассмотрим функциональность, которую обеспечивает пакет `container/heap`. Запомните, что пакет `container/heap` реализует кучу — дерево, где значение каждого узла является наименьшим элементом в его поддереве. Обратите внимание, что я использую выражение *«наименьший элемент»* вместо *«минимальное значение»*, чтобы подчеркнуть, что куча поддерживает не только числовые значения.

Однако, как легко догадаться, для того чтобы реализовать дерево кучи в Go, необходимо самостоятельно разработать способ, позволяющий определить, какой из двух элементов меньше. В таких случаях в Go используются *интерфейсы*, которые позволяют определять такое поведение.

Это означает, что пакет `container/heap` сложнее, чем два остальных пакета, входящих в состав `container`, и что вам, прежде чем использовать функциональность пакета `container/heap`, придется кое-что определить. Строго говоря, пакет `container/heap` требует реализации `container/heap.Interface`, который определяется следующим образом:

```
type Interface interface {
    sort.Interface
    Push(x interface{})   // вставляет x как элемент Len()
    Pop() interface{}     // удаляет и возвращает элемент Len() - 1
}
```

Подробнее об интерфейсах вы узнаете в главе 7. Пока просто запомните: для соответствия интерфейсу Go требуется реализация одной или нескольких функций или других интерфейсов, в данном случае это `sort.Interface` и функции `Push()` и `Pop()`.

Для `sort.Interface` необходимо реализовать функции `Len()`, `Less()` и `Swap()`. Это имеет смысл, поскольку вы не сможете выполнять какую-либо сортировку, не имея возможности поменять местами два элемента и вычислить значение того, что хотите сортировать, и не имея возможности определять, какой из двух элементов больше, на основе вычисленных ранее значений. Вам может показаться, что это значительный объем работы. Однако в большинстве случаев реализация этих функций является очень простой.

Поскольку цель данного раздела — продемонстрировать использование пакета `container/heap`, а не усложнить вам жизнь, в примере используются элементы с типом данных `float32`.

Мы рассмотрим код Go, представленный в файле `conHeap.go`. Разделим его на пять частей. Первая часть выглядит следующим образом:

```
package main

import (
    "container/heap"
    "fmt"
)

type heapFloat32 []float32
```

Вторая часть `conHeap.go` содержит следующий код Go:

```
func (n *heapFloat32) Pop() interface{} {
    old := *n
    x := old[len(old)-1]
    new := old[0 : len(old)-1]
    *n = new
    return x
}

func (n *heapFloat32) Push(x interface{}) {
    *n = append(*n, x.(float32))
}
```

Здесь мы определили две функции с именами `Pop()` и `Push()`, которые используются для того, чтобы соответствовать интерфейсу[1]. Чтобы добавлять и удалять элементы в куче, необходимо вызывать функции `heap.Push()` и `heap.Pop()` соответственно.

---

[1] Функция Pop() возвращает тип interface{}, которому удовлетворяет любой тип Go. В приведенном выше примере возвращается тип данных float32, но если нужно хранить в куче указатели, то при извлечении его из кучи нужно не забывать обнулять место его хранения, например, так: old[len(old) – 1] = nil. В противном случае сборщик мусора не сможет очистить место под объект, на который указывает извлеченный из кучи указатель.

Третий фрагмент кода `conHeap.go` содержит следующий код Go:

```
func (n heapFloat32) Len() int {
    return len(n)
}

func (n heapFloat32) Less(a, b int) bool {
    return n[a] < n[b]
}

func (n heapFloat32) Swap(a, b int) {
    n[a], n[b] = n[b], n[a]
}
```

В этой части реализованы три функции, необходимые для совместимости с интерфейсом `sort.Interface`.

Четвертая часть `conHeap.go` выглядит так:

```
func main() {
    myHeap := &heapFloat32{1.2, 2.1, 3.1, -100.1}
    heap.Init(myHeap)
    size := len(*myHeap)
    fmt.Printf("Heap size: %d\n", size)
    fmt.Printf("%v\n", myHeap)
```

И последняя часть кода `conHeap.go` выглядит следующим образом:

```
    myHeap.Push(float32(-100.2))
    myHeap.Push(float32(0.2))
    fmt.Printf("Heap size: %d\n", len(*myHeap))
    fmt.Printf("%v\n", myHeap)
    heap.Init(myHeap)
    fmt.Printf("%v\n", myHeap)
}
```

В этой части `conHeap.go` мы добавляем в кучу `myHeap` два новых элемента, используя функцию `heap.Push()`. Однако, для того чтобы восстановить правильную упорядоченность кучи, нужно снова вызвать `heap.Init()`[1].

Выполнение `conHeap.go` приведет к следующим результатам:

```
$ go run conHeap.go
Heap size: 4
&[-100.1 1.2 3.1 2.1]
Heap size: 6
&[-100.1 1.2 3.1 2.1 -100.2 0.2]
&[-100.2 -100.1 0.2 2.1 1.2 3.1]
```

[1] В приведенном примере для добавления в кучу использовалась функция myHeap.Push(), которая не поддерживает упорядоченность кучи. Это и привело к необходимости повторной инициации кучи функцией heap.Init(). Для автоматического поддержания упорядоченности кучи лучше использовать функцию heap.Push().

Если вам показалось странным, что триплет `2.1 1.2 3.1` в последней строке результатов не отсортирован в соответствии с линейной логикой, то напомню, что куча — это не линейная структура, подобная массиву или срезу, а дерево[1].

Использование пакета container/list

В этом подразделе продемонстрирована работа пакета `container/list` на примере Go-кода из файла `conList.go`, который мы разделим на три части.

Внимание! В пакете container/list реализован двусвязный список.

Первая часть `conList.go` содержит следующий код Go:

```
package main

import (
    "container/list"
    "fmt"
    "strconv"
)

func printList(l *list.List) {
    for t := l.Back(); t != nil; t = t.Prev() {
        fmt.Print(t.Value, " ")
    }
    fmt.Println()

    for t := l.Front(); t != nil; t = t.Next() {
        fmt.Print(t.Value, " ")
    }

    fmt.Println()
}
```

Здесь вы видите функцию `printList()`, которая позволяет выводить на экран содержимое переменной `list.List`, переданной в виде указателя. В коде Go показано, как вывести элементы `list.List`, начиная с первого и заканчивая последним, и в обратном порядке. Обычно в программах нужно использовать только один из

[1] Куча гарантирует, что нулевой элемент (*myHeap)[0] всегда будет минимальным. Для извлечения из кучи элементов в соответствии с приоритетом нужно использовать функцию heap.Pop().

двух методов. Функции `Prev()` и `Next()` позволяют перебирать элементы списка в прямом и обратном порядке.

Второй фрагмент кода `conList.go` выглядит так:

```
func main() {
    values := list.New()

    e1 := values.PushBack("One")
    e2 := values.PushBack("Two")
    values.PushFront("Three")
    values.InsertBefore("Four", e1)
    values.InsertAfter("Five", e2)
    values.Remove(e2)
    values.Remove(e2)
    values.InsertAfter("FiveFive", e2)
    values.PushBackList(values)

    printList(values)

    values.Init()
```

Функция `list.PushBack()` позволяет вставлять объект в конец связного списка, а функция `list.PushFront()` — в начало списка. Обе функции возвращают вставленный в список элемент.

Если вы хотите вставить новый элемент после определенного элемента, то следует использовать функцию `list.InsertAfter()`. Аналогично, для того чтобы вставить элемент перед конкретным элементом, необходимо применить функцию `list.InsertBefore()`. Если такой элемент не существует, то список не изменится. Функция `list.PushBackList()` вставляет копию существующего списка в конец другого списка, а `list.PushFrontList()` помещает копию существующего списка в начало другого списка. Функция `list.Remove()` удаляет из списка заданный элемент.

Обратите внимание на использование функции `values.Init()`, которая либо очищает существующий список, либо инициализирует новый список.

Последняя часть `conList.go` содержит следующий код Go:

```
    fmt.Printf("After Init(): %v\n", values)

    for i := 0; i < 20; i++ {
        values.PushFront(strconv.Itoa(i))
    }

    printList(values)
}
```

Здесь мы создаем новый список с помощью цикла `for`. Функция `strconv.Itoa()` преобразует целочисленное значение в строку.

Таким образом, функции пакета `container/list` достаточно просты и их использование не должно вызывать затруднений.

Выполнение conList.go приведет к следующим результатам:

```
$ go run conList.go
Five One Four Three Five One Four Three
Three Four One Five Three Four One Five
After Init(): &{{0xc420074180 0xc420074180 <nil> <nil>} 0}
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
```

Использование пакета container/ring

В этом разделе продемонстрировано применение пакета container/ring на примере Go-кода из файла conRing.go. Разделим его на четыре части. Обратите внимание, что пакет container/ring намного проще, чем container/list и container/heap, следовательно, содержит меньше функций, чем эти два пакета.

Первый фрагмент кода conRing.go выглядит так:

```
package main

import (
    "container/ring"
    "fmt"
)

var size int = 10
```

Переменная size содержит размер создаваемого кольца.

Вторая часть conRing.go содержит следующий код Go:

```
func main() {
    myRing := ring.New(size + 1)
    fmt.Println("Empty ring:", *myRing)

    for i := 0; i < myRing.Len()-1; i++ {
        myRing.Value = i
        myRing = myRing.Next()
    }

    myRing.Value = 2
```

Таким образом, новое кольцо создается с помощью функции ring.New(), для которой требуется один параметр — размер кольца. Оператор myRing.Value = 2 в конце кода добавляет в кольцо значение 2. Однако такое значение в кольце уже есть, поскольку оно уже было добавлено в цикле for. Наконец, нулевое значение для кольца — это кольцо с одним элементом, значение которого равно nil.

Третья часть conRing.go содержит следующий код Go:

```
    sum := 0
    myRing.Do(func(x interface{}) {
        t := x.(int)
```

```
        sum = sum + t
    })
    fmt.Println("Sum:", sum)
```

Функция `ring.Do()` позволяет вызывать функцию для каждого элемента кольца в хронологическом порядке. Но если эта функция вносит какие-либо изменения в кольцо, то поведение `ring.Do()` становится неопределенным.

Оператор `x.(int)` называется *утверждением типа*. Подробнее об утверждениях типа вы прочитаете в главе 7. Пока просто запомните: этот оператор показывает, что `x` имеет тип `int`.

Последняя часть программы `conRing.go` выглядит так:

```
    for i := 0; i < myRing.Len()+2; i++ {
        myRing = myRing.Next()
        fmt.Print(myRing.Value, " ")
    }
    fmt.Println()
}
```

Единственная проблема с кольцами — это то, что мы можем вызывать `ring.Next()` до бесконечности, поэтому нужно найти способ, как это прекратить. В данном случае это сделано с помощью функции `ring.Len()`. Я предпочитаю использовать для перебора всех элементов кольца функцию `ring.Do()`, поскольку она генерирует более чистый код, но использовать цикл `for` тоже неплохо.

Выполнение `conRing.go` приведет к следующим результатам:

```
$ go run conRing.go
Empty ring: {0xc42000a080 0xc42000a1a0 <nil>}
Sum: 47
0 1 2 3 4 5 6 7 8 9 2 0 1
```

Как видно из результатов, кольцо может содержать повторяющиеся значения, следовательно, если не использовать функцию `ring.Len()`, то у нас не будет безопасного способа узнать размер кольца.

Генерация случайных чисел

Генерация случайных чисел — это искусство, а также предмет исследований в области компьютерных наук. Дело в том, что компьютеры являются чисто логическими машинами и, как оказалось, применять их для генерации случайных чисел очень сложно!

В Go для генерации псевдослучайных чисел применяется пакет `math/rand`. Чтобы начать генерировать числа, нужно *начальное число*. Начальное число используется для инициализации всего процесса и имеет важное значение, поскольку если всегда начинать с одного и того же числа, то мы будем получать одну и ту же последова-

тельность псевдослучайных чисел. Это означает, что данную последовательность сможет восстановить любой желающий и она в итоге не будет случайной.

Утилита, которая поможет нам генерировать псевдослучайные числа, называется `randomNumbers.go`. Рассмотрим ее, разделив на четыре части. Эта утилита принимает различные параметры: нижний и верхний пределы числовой последовательности, которая будет сформирована, а также количество генерируемых чисел. Если указать четвертый параметр команды, то программа будет использовать его в качестве начального числа генератора псевдослучайных чисел, что поможет получить ту же последовательность чисел, — главная причина, для чего это нужно, заключается в возможности тестирования кода.

Первая часть утилиты выглядит так:

```
package main

import (
    "fmt"
    "math/rand"
    "os"
    "strconv"
    "time"
)

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

Всю работу выполняет функция `random()` — она генерирует псевдослучайные числа в заданном диапазоне, вызывая `rand.Intn()`.

Вторая часть утилиты командной строки выглядит так:

```
func main() {
    MIN := 0
    MAX := 100
    TOTAL := 100
    SEED := time.Now().Unix()

    arguments := os.Args
```

В этой части мы инициализируем переменные, которые будут использоваться в программе.

В третьей части `randomNumbers.go` содержится следующий код Go:

```
    switch len(arguments) {
    case 2:
        fmt.Println("Usage: ./randomNumbers MIN MAX TOTAL SEED")
        MIN, _ = strconv.Atoi(arguments[1])
        MAX = MIN + 100
    case 3:
        fmt.Println("Usage: ./randomNumbers MIN MAX TOTAL SEED")
```

```
        MIN, _ = strconv.Atoi(arguments[1])
        MAX, _ = strconv.Atoi(arguments[2])
    case 4:
        fmt.Println("Usage: ./randomNumbers MIN MAX TOTAL SEED")
        MIN, _ = strconv.Atoi(arguments[1])
        MAX, _ = strconv.Atoi(arguments[2])
        TOTAL, _ = strconv.Atoi(arguments[3])
    case 5:
        MIN, _ = strconv.Atoi(arguments[1])
        MAX, _ = strconv.Atoi(arguments[2])
        TOTAL, _ = strconv.Atoi(arguments[3])
        SEED, _ = strconv.ParseInt(arguments[4], 10, 64)
    default:
        fmt.Println("Using default values!")
    }
```

Логика этого блока `switch` относительно проста: в зависимости от количества полученных аргументов командной строки мы используем либо начальные значения отсутствующих аргументов, либо значения, заданные пользователем. Для простоты переменные типа `error` для функций `strconv.Atoi()` и `strconv.ParseInt()` игнорируются (для чего используются символы подчеркивания). Если бы это была коммерческая программа, то нельзя было бы проигнорировать переменные `error` функций `strconv.Atoi()` и `strconv.ParseInt()`.

Наконец, причина использования функции `strconv.ParseInt()` для установки нового значения переменной `SEED` заключается в том, что для функции `rand.Seed()` требуется параметр типа `int64`. Первый параметр `strconv.ParseInt()` — это строка для синтаксического анализа, второй — основание системы счисления для генерируемого числа, а третий — количество битов для этого числа.

Поскольку мы хотим создать десятичное целое число, которое будет занимать 64 бита, то мы указали `10` в качестве основания и `64` в качестве числа битов. Обратите внимание, что для синтаксического анализа целого числа без знака следовало бы использовать функцию `strconv.ParseUint()`.

Последняя часть `randomNumbers.go` содержит следующий код Go:

```
    rand.Seed(SEED)
    for i := 0; i < TOTAL; i++ {
        myrand := random(MIN, MAX)
        fmt.Print(myrand)
        fmt.Print(" ")
    }
    fmt.Println()
}
```

Вместо использования времени эпохи UNIX в качестве начального числа для генератора псевдослучайных чисел можно применять системное устройство /dev/random. Подробнее о чтении из устройства /dev/random вы прочитаете в главе 8.

Выполнение `randomNumbers.go` приведет к следующим результатам:

```
$ go run randomNumbers.go
Using default values!
75 69 15 75 62 67 64 8 73 1 83 92 7 34 8 70 22 58 38 8 54 34 91 65 1 50 76
5 82 61 90 10 38 40 63 6 28 51 54 49 27 52 92 76 35 44 9 66 76 90 10 29 22
20 83 33 92 80 50 62 26 19 45 56 75 40 30 97 23 87 10 43 11 42 65 80 82 25
53 27 51 99 88 53 36 37 73 52 61 4 81 71 57 30 72 51 55 62 63 79
$ go run randomNumbers.go 1 3 2
Usage: ./randomNumbers MIN MAX TOTAL SEED
1 1
$ go run randomNumbers.go 1 3 2
Usage: ./randomNumbers MIN MAX TOTAL SEED
2 2
$ 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
```

Если вас действительно заинтересовала генерация случайных чисел, прочитайте второй том книги «Искусство программирования»[1] Дональда Кнута.

Если эти псевдослучайные числа нужны вам для обеспечения безопасности, важно использовать пакет `crypto/rand`. В этом пакете реализован криптографически безопасный генератор псевдослучайных чисел. Рассмотрим его в этой главе.

Генерация случайных строк

Зная, как в компьютере представляются отдельные символы, нетрудно перейти от псевдослучайных чисел к случайным строкам. В этом разделе описан способ создания трудно угадываемых паролей на основе Go-кода `randomNumbers.go`, представленного в предыдущем разделе. Программа Go для решения этой задачи называется `generatePassword.go`. Рассмотрим ее, разделив на четыре части. Утилита принимает только один параметр командной строки — длину пароля, который требуется сгенерировать.

Первая часть `generatePassword.go` содержит следующий код Go:

```
package main

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

[1] *Кнут Д. Э.* Искусство программирования. Т. 2. Получисленные алгоритмы (The Art of Computer Programming. Volume 2. Seminumerical Algorithms.) — М.: Вильямс, 2001.

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

Вторая часть кода `generatePassword.go` содержит следующий код Go:

```
func main() {
    MIN := 0
    MAX := 94
    var LENGTH int64 = 8

    arguments := os.Args
```

Поскольку мы хотим получать только печатаемые символы ASCII, мы ограничили диапазон генерируемых псевдослучайных чисел: в таблице ASCII содержится 94 печатаемых символа. Это значит, что диапазон псевдослучайных чисел, которые может сгенерировать программа, составляет от 0 до 94, не включая 94.

Третий фрагмент `generatePassword.go` содержит следующий код Go:

```
    switch len(arguments) {
    case 2:
        LENGTH, _ = strconv.ParseInt(os.Args[1], 10, 64)
    default:
        fmt.Println("Using default values!")
    }

    SEED := time.Now().Unix()
    rand.Seed(SEED)
```

Последняя часть `generatePassword.go` выглядит так:

```
    startChar := "!"
    var i int64 = 1
    for {
        myRand := random(MIN, MAX)
        newChar := string(startChar[0] + byte(myRand))
        fmt.Print(newChar)
        if i == LENGTH {
            break
        }
        i++
    }
    fmt.Println()
}
```

В переменной `startChar` хранится первый символ ASCII, который может быть сгенерирован утилитой. В данном случае это восклицательный знак, ASCII-код которого в десятичном представлении равен 33. Учитывая, что программа генерирует псевдослучайные числа от 0 до 94, максимальное значение ASCII, которое можно получить, равно 93 + 33, то есть 126, что соответствует ASCII-коду символа ~.

Далее показана таблица ASCII-кодов с соответствующими десятичными значениями для каждого символа:

```
  0  nul    1  soh    2  st    3  etx     4  eot    5   enq    6  ack    7   bel
  8  bs     9  ht    10  nl   11  vt     12  np    13   cr    14  so    15   si
 16  dle   17  dc1   18  dc   19  dc3    20  dc4   21   nak   22  syn   23   etb
 24  can   25  em    26  su   27  esc    28  fs    29   gs    30  rs    31   us
 32  sp    33   !    34   "   35    #    36   $    37    %    38   &    39    '
 40   (    41   )    42   *   43    +    44   ,    45    -    46   .    47    /
 48   0    49   1    50   2   51    3    52   4    53    5    54   6    55    7
 56   8    57   9    58   :   59    ;    60   <    61    =    62   >    63    ?
 64   @    65   A    66   B   67    C    68   D    69    E    70   F    71    G
 72   H    73   I    74   J   75    K    76   L    77    M    78   N    79    O
 80   P    81   Q    82   R   83    S    84   T    85    U    86   V    87    W
 88   X    89   Y    90   Z   91    [    92   \    93    ]    94   ^    95    _
 96   `    97   a    98   b   99    c   100   d   101    e   102   f   103    g
104   h   105   i   106   j  107    k   108   l   109    m   110   n   111    o
112   p   113   q   114   r  115    s   116   t   117    u   118   v   119    w
120   x   121   y   122   z  123    {   124   |   125    }   126   ~   127   del
```

Чтобы получить таблицу ASCII-кодов в удобочитаемой форме, введите в любой оболочке UNIX команду man ascii.

Выполнение программы `generatePassword.go` с соответствующими параметрами командной строки приведет к следующим результатам:

```
$ go run generatePassword.go
Using default values!
ugs$5mvl
$ go run generatePassword.go
Using default values!
PA/8hA@?
$ go run generatePassword.go 20
HBR+=3\UA'B@ExT4QG|o
$ go run generatePassword.go 20
XLcr|R{*pX/::'t2u^T'
```

Генерация безопасной последовательности случайных чисел

Если вы хотите генерировать на Go более безопасные псевдослучайные числа, следует использовать пакет `crypto/rand`, в котором реализован криптографически безопасный генератор псевдослучайных чисел. Именно об этом пакете речь пойдет в данном разделе.

Мы продемонстрируем применение пакета `crypto/rand` на примере Go-кода `cryptoRand.go`, который разделим на три части.

Первая часть `cryptoRand.go` выглядит так:

```
package main

import (
	"crypto/rand"
	"encoding/base64"
	"fmt"
	"os"
	"strconv"
)

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

	return b, nil
}
```

Вторая часть `cryptoRand.go` содержит следующий код Go:

```
func generatePass(s int64) (string, error) {
	b, err := generateBytes(s)
	return base64.URLEncoding.EncodeToString(b), err
}
```

Последняя часть `cryptoRand.go` выглядит так:

```
func main() {
	var LENGTH int64 = 8
	arguments := os.Args
	switch len(arguments) {
	case 2:
		LENGTH, _ = strconv.ParseInt(os.Args[1], 10, 64)
		if LENGTH <= 0 {
			LENGTH = 8
		}
	default:
		fmt.Println("Using default values!")
	}

	myPass, err := generatePass(LENGTH)
	if err != nil {
		fmt.Println(err)
		return
	}
	fmt.Println(myPass[0:LENGTH])
}
```

Выполнение `cryptoRand.go` приведет к следующему результату:

```
$ go run cryptoRand.go
Using default values!
hIAFYuvW
$ go run cryptoRand.go 120
WTR15SIcjYQmaMKds0lDfFturG27ovH_HZ6iAi_kOnJC88EDLdvNPcv1JjOd9DcF0r0S3q2itXZ
8O1TNaNFpHkT-aMrsjeue6kUyHnx_EaL_vJHy9wL5RTr8
```

Дополнительную информацию о пакете `crypto/rand` вы найдете на странице документации пакета, расположенной по адресу https://golang.org/pkg/crypto/rand/.

Выполнение матричных вычислений

Матрица — это двумерный массив. Самый простой способ представить матрицу в Go — с помощью среза. Но если размеры массива заранее известны, то массив тоже отлично справится с этой задачей. Если оба размера матрицы одинаковы, то такая матрица называется *квадратной*.

Есть несколько правил, которые позволяют определить, можно ли выполнять вычисления между двумя матрицами:

- при сложении или вычитании обе матрицы должны иметь одинаковые размеры;
- при умножении матрицы *A* на матрицу *B* количество столбцов матрицы *A* должно быть равно количеству строк матрицы *B*. В противном случае умножение матриц *A* и *B* невозможно;
- при делении матрицы *A* на матрицу *B* должны быть выполнены два условия. Во-первых, необходимо, чтобы у матрицы *B* существовала обратная матрица, а во-вторых, должна быть возможность умножить матрицу *A* на обратную матрицу *B* в соответствии с предыдущим правилом. Обратные матрицы существуют только у квадратных матриц.

Сложение и вычитание матриц

В этом разделе вы узнаете, как складывать и вычитать матрицы с помощью утилиты `addMat.go`, которую мы рассмотрим, разделив на три части. Для реализации матриц в программе используются срезы.

Первая часть `addMat.go` выглядит так:

```
package main

import (
    "fmt"
    "math/rand"
    "os"
    "strconv"
)
```

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

func negativeMatrix(s [][]int) [][]int {
    for i, x := range s {
        for j, _ := range x {
            s[i][j] = -s[i][j]
        }
    }
    return s
}
```

Функция `negativeMatrix()` получает входные данные среза и возвращает новый срез, в котором каждый целочисленный элемент исходного среза заменяется на это же целое число с противоположным знаком. Как вы вскоре увидите, элементы двух исходных матриц генерируются с использованием псевдослучайных чисел, для чего используется функция `random()`.

Вторая часть `addMat.go` содержит следующий код Go:

```
func addMatrices(m1 [][]int, m2 [][]int) [][]int {
    result := make([][]int, len(m1))
    for i, x := range m1 {
        for j, _ := range x {
            result[i] = append(result[i], m1[i][j]+m2[i][j])
        }
    }
    return result
}
```

Функция `addMatrices()` получает элементы обеих матриц, складывает их и создает матрицу результатов.

Последняя часть `addMat.go` выглядит так:

```
func main() {
    arguments := os.Args
    if len(arguments) != 3 {
        fmt.Println("Wrong number of arguments!")
        return
    }

    var row, col int
    row, err := strconv.Atoi(arguments[1])
    if err != nil {
        fmt.Println("Need an integer: ", arguments[1])
        return
    }

    col, err = strconv.Atoi(arguments[2])
    if err != nil {
```

```
        fmt.Println("Need an integer: ", arguments[2])
        return
    }

    fmt.Printf("Using %dx%d arrays\n", row, col)
    if col <= 0 || row <= 0 {
        fmt.Println("Need positive matrix dimensions!")
        return
    }

    m1 := make([][]int, row)
    m2 := make([][]int, row)

    rand.Seed(time.Now().Unix())
    // Инициализация m1 и m2 случайными числами
    for i := 0; i < row; i++ {
        for j := 0; j < col; j++ {
            m1[i] = append(m1[i], random(-1, i*j+rand.Intn(10)))
            m2[i] = append(m2[i], random(-1, i*j+rand.Intn(10)))
        }
    }
    fmt.Println("m1:", m1)
    fmt.Println("m2:", m2)

    // Сложение
    r1 := addMatrices(m1, m2)
    // Вычитание
    r2 := addMatrices(m1, negativeMatrix(m2))
    fmt.Println("r1:", r1)
    fmt.Println("r2:", r2)
}
```

Функция `main()` представляет собой контроллер программы. Помимо прочего, она проверяет, правильного ли типа данные введены, создает нужные матрицы и заполняет их сгенерированными псевдослучайными числами.

Выполнение `addMat.go` приведет к следующим результатам:

```
$ go run addMat.go 2 3
Using 2x3 arrays
m1: [[0 -1 0] [1 1 1]]
m2: [[2 1 0] [7 4 9]]
r1: [[2 0 0] [8 5 10]]
r2: [[-2 -2 0] [-6 -3 -8]]
$ go run addMat.go 2 3
Using 2x3 arrays
m1: [[0 -1 0] [1 1 1]]
m2: [[2 1 0] [7 4 9]]
r1: [[2 0 0] [8 5 10]]
r2: [[-2 -2 0] [-6 -3 -8]]
$ go run addMat.go 3 2
```

```
Using 3x2 arrays
m1: [[0 -1] [0 0] [0 1]]
m2: [[2 1] [0 3] [1 9]]
r1: [[2 0] [0 3] [1 10]]
r2: [[-2 -2] [0 -3] [-1 -8]]
```

Умножение матриц

Как вы уже знаете, умножение матриц намного сложнее, чем сложение или вычитание. Это заметно по количеству аргументов командной строки, которое требуется утилите `mulMat.go`, описанной в этом разделе. Утилита `mulMat.go`, которую мы рассмотрим, разделив на четыре части, принимает четыре аргумента командной строки, соответствующие размерам первой и второй матриц.

Первая часть `mulMat.go` выглядит так:

```
package main

import (
    "errors"
    "fmt"
    "math/rand"
    "os"
    "strconv"
    "time"
)

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

Вторая часть `mulMat.go` содержит следующий код Go:

```
func multiplyMatrices(m1 [][]int, m2 [][]int) ([][]int, error) {
    if len(m1[0]) != len(m2) {
        return nil, errors.New("Cannot multiply the given matrices!")
    }

    result := make([][]int, len(m1))
    for i := 0; i < len(m1); i++ {
        result[i] = make([]int, len(m2[0]))
        for j := 0; j < len(m2[0]); j++ {
            for k := 0; k < len(m2); k++ {
                result[i][j] += m1[i][k] * m2[k][j]
            }
        }
    }
    return result, nil
}
```

Умножение матриц совершенно не похоже на сложение, и это видно по реализации функции `multiplyMatrices()`. Функция `multiplyMatrices()` также возвращает собственное сообщение об ошибке в случае, если размеры входных матриц не позволят их перемножить.

Третья часть `mulMat.go` выглядит так:

```
func createMatrix(row, col int) [][]int {
    r := make([][]int, row)
    for i := 0; i < row; i++ {
        for j := 0; j < col; j++ {
            r[i] = append(r[i], random(-5, i*j))
        }
    }
    return r
}

func main() {
    rand.Seed(time.Now().Unix())
    arguments := os.Args
    if len(arguments) != 5 {
        fmt.Println("Wrong number of arguments!")
        return
    }
```

Функция `createMatrix()` создает срез нужных размеров и заполняет его целыми числами, которые генерируются случайным образом.

Последняя часть `mulMat.go` выглядит так:

```
    var row, col int
    row, err := strconv.Atoi(arguments[1])
    if err != nil {
        fmt.Println("Need an integer: ", arguments[1])
        return
    }

    col, err = strconv.Atoi(arguments[2])
    if err != nil {
        fmt.Println("Need an integer: ", arguments[2])
        return
    }

    if col <= 0 || row <= 0 {
        fmt.Println("Need positive matrix dimensions!")
        return
    }

    fmt.Printf("m1 is a %dx%d matrix\n", row, col)
    // Инициализация m1 случайными числами
    m1 := createMatrix(row, col)
```

```
    row, err = strconv.Atoi(arguments[3])
    if err != nil {
        fmt.Println("Need an integer: ", arguments[3])
        return
    }

    col, err = strconv.Atoi(arguments[4])
    if err != nil {
        fmt.Println("Need an integer: ", arguments[4])
        return
    }

    if col <= 0 || row <= 0 {
        fmt.Println("Need positive matrix dimensions!")
        return
    }

    fmt.Printf("m2 is a %dx%d matrix\n", row, col)
    // Инициализация m2 случайными числами
    m2 := createMatrix(row, col)
    fmt.Println("m1:", m1)
    fmt.Println("m2:", m2)

    // Умножение
    r1, err := multiplyMatrices(m1, m2)
    if err != nil {
        fmt.Println(err)
        return
    }
    fmt.Println("r1:", r1)
}
```

Функция `main()` является контроллером программы, который определяет способ ее работы и проверяет правильность ввода аргументов командной строки.

Выполнение `mulMat.go` приведет к следующим результатам:

```
$ go run mulMat.go 1 2 2 1
m1 is a 1x2 matrix
m2 is a 2x1 matrix
m1: [[-3 -1]]
m2: [[-2] [-1]]
r1: [[7]]
$ go run mulMat.go 5 2 2 1
m1 is a 5x2 matrix
m2 is a 2x1 matrix
m1: [[-1 -2] [-4 -4] [-4 -1] [-2 2] [-5 -5]]
m2: [[-5] [-3]]
r1: [[11] [32] [23] [4] [40]]
```

```
$ go run mulMat.go 1 2 3 4
m1 is a 1x2 matrix
m2 is a 3x4 matrix
m1: [[-3 -4]]
m2: [[-5 -2 -2 -3] [-1 -4 -3 -5] [-5 -2 3 3]]
Cannot multiply the given matrices!
```

Деление матриц

В этом подразделе вы узнаете, как разделить одну матрицу на другую, используя код Go из файла `divMat.go`, который мы рассмотрим, разделив на пять частей. Основная функция `divMat.go` называется `inverseMatrix()`. Назначение `inverseMatrix()` — вычисление обратной матрицы для исходной матрицы, что является довольно сложной задачей. Существуют готовые пакеты Go, которые позволяют инвертировать матрицу, но я решил реализовать ее с нуля.

Не все матрицы обратимы. Неквадратные матрицы не являются обратимыми. Квадратная матрица, которая необратима, называется сингулярной или вырожденной — так случается, если определитель квадратной матрицы равен нулю. Сингулярные матрицы встречаются очень редко.

Утилита `divMat.go` принимает один аргумент командной строки, который определяет размеры используемых квадратных матриц.

Первая часть `divMat.go` выглядит так:

```
package main

import (
    "errors"
    "fmt"
    "math/rand"
    "os"
    "strconv"
    "time"
)

func random(min, max int) float64 {
    return float64(rand.Intn(max-min) + min)
}
```

На этот раз функция `random()` генерирует числа формата `float64`. Вся утилита `divMat.go` оперирует числами с плавающей точкой, главным образом потому, что элементы матрицы, обратной для матрицы с целочисленными элементами, скорее всего, не будут целочисленными.

Вторая часть `divMat.go` содержит следующий код Go:

```
func getCofactor(A [][]float64, temp [][]float64, p int, q int, n int) {
    i := 0
    j := 0

    for row := 0; row < n; row++ {
        for col := 0; col < n; col++ {
            if row != p && col != q {
                temp[i][j] = A[row][col]
                j++
                if j == n-1 {
                    j = 0
                    i++
                }
            }
        }
    }
}

func determinant(A [][]float64, n int) float64 {
    D := float64(0)
    if n == 1 {
        return A[0][0]
    }

    temp := createMatrix(n, n)
    sign := 1

    for f := 0; f < n; f++ {
        getCofactor(A, temp, 0, f, n)
        D += float64(sign) * A[0][f] * determinant(temp, n-1)
        sign = -sign
    }
    return D
}
```

Функции `getCofactor()` и `determinant()` вычисляют элементы, необходимые для обращения матрицы. Если определитель матрицы равен 0, то матрица является сингулярной.

Третья часть `divMat.go` выглядит так:

```
func adjoint(A [][]float64) ([][]float64, error) {
    N := len(A)
    adj := createMatrix(N, N)
    if N == 1 {
        adj[0][0] = 1
        return adj, nil
    }
    sign := 1
    var temp = createMatrix(N, N)
```

```
    for i := 0; i < N; i++ {
        for j := 0; j < N; j++ {
            getCofactor(A, temp, i, j, N)
            if (i+j)%2 == 0 {
                sign = 1
            } else {
                sign = -1
            }
            adj[j][i] = float64(sign) * (determinant(temp, N-1))
        }
    }
    return adj, nil
}

func inverseMatrix(A [][]float64) ([][]float64, error) {
    N := len(A)
    var inverse = createMatrix(N, N)
    det := determinant(A, N)
    if det == 0 {
        fmt.Println("Singular matrix, cannot find its inverse!")
        return nil, nil
    }

    adj, err := adjoint(A)
    if err != nil {
        fmt.Println(err)
        return nil, nil
    }

    fmt.Println("Determinant:", det)
    for i := 0; i < N; i++ {
        for j := 0; j < N; j++ {
            inverse[i][j] = float64(adj[i][j]) / float64(det)
        }
    }
    return inverse, nil
}
```

Функция `adjoint()` вычисляет сопряженную матрицу для данной матрицы. Функция `inverseMatrix()` вычисляет обратную матрицу для данной матрицы.

Программа divMat.go — отличный пример кода Go, который необходимо тщательно протестировать, прежде чем использовать на практике. Подробнее о тестировании вы прочитаете в главе 11.

Четвертая часть `divMat.go` содержит следующий код Go:

```
func multiplyMatrices(m1 [][]float64, m2 [][]float64) ([][]float64, error)
{
    if len(m1[0]) != len(m2) {
```

```
        return nil, errors.New("Cannot multiply the given matrices!")
    }

    result := make([][]float64, len(m1))
    for i := 0; i < len(m1); i++ {
        result[i] = make([]float64, len(m2[0]))
        for j := 0; j < len(m2[0]); j++ {
            for k := 0; k < len(m2); k++ {
                result[i][j] += m1[i][k] * m2[k][j]
            }
        }
    }
    return result, nil
}

func createMatrix(row, col int) [][]float64 {
    r := make([][]float64, row)
    for i := 0; i < row; i++ {
        for j := 0; j < col; j++ {
            r[i] = append(r[i], random(-5, i*j))
        }
    }
    return r
}
```

Функция `multiplyMatrices()` нужна нам потому, что разделить одну матрицу на другую — это все равно что умножить первую матрицу на матрицу, обратную второй.

Последняя часть `divMat.go` выглядит так:

```
func main() {
    rand.Seed(time.Now().Unix())
    arguments := os.Args
    if len(arguments) != 2 {
        fmt.Println("Wrong number of arguments!")
        return
    }

    var row int
    row, err := strconv.Atoi(arguments[1])
    if err != nil {
        fmt.Println("Need an integer:", arguments[1])
        return
    }

    col := row
    if col <= 0 {
        fmt.Println("Need positive matrix dimensions!")
        return
    }
```

```
    m1 := createMatrix(row, col)
    m2 := createMatrix(row, col)
    fmt.Println("m1:", m1)
    fmt.Println("m2:", m2)

    inverse, err := inverseMatrix(m2)
    if err != nil {
        fmt.Println(err)
        return
    }

    fmt.Println("\t\t\tPrinting inverse matrix!")
    for i := 0; i < len(inverse); i++ {
        for j := 0; j < len(inverse[0]); j++ {
            fmt.Printf("%.2f\t", inverse[i][j])
        }
        fmt.Println()
    }

    fmt.Println("\t\t\tPrinting result!")
    r1, err := multiplyMatrices(m1, inverse)
    if err != nil {
        fmt.Println(err)
        return
    }

    for i := 0; i < len(r1); i++ {
        for j := 0; j < len(r1[0]); j++ {
            fmt.Printf("%.3f\t", r1[i][j])
        }
        fmt.Println()
    }
}
```

Здесь, как и в предыдущих утилитах, функция `main()` управляет выполнением программы и осуществляет необходимые проверки данных, вводимых пользователем, прежде чем продолжать работу.

Выполнение `divMat.go` приведет к следующим результатам:

```
$ go run divMat.go 2
m1: [[-3 -3] [-4 -4]]
m2: [[-3 -5] [-4 -1]]
Determinant: -17
           Printing inverse matrix!
0.06    -0.29
-0.24    0.18
           Printing result!
0.529    0.353
0.706    0.471
$ go run divMat.go 3
```

```
m1: [[-3 -5 -2] [-1 -4 1] [-2 -5 -1]]
m2: [[-2 -4 -5] [-1 0 -2] [-2 -2 1]]
Determinant: -22
           Printing inverse matrix!
0.18    -0.64    -0.36
-0.23    0.55    -0.05
-0.09   -0.18     0.18
           Printing result!
0.773   -0.455    0.955
0.636   -1.727    0.727
0.864   -1.273    0.773
$ go run divMat.go 2
m1: [[-3 -5] [-5 -5]]
m2: [[-5 -3] [-5 -3]]
Singular matrix, cannot find its inverse!
           Printing inverse matrix!
           Printing result!
Cannot multiply the given matrices!
```

Как определить размеры массива

В этом разделе покажу вам способ найти размеры массива, используя Go-программу `dimensions.go`. Этот же метод можно использовать для определения размеров среза.

Go-код программы `dimensions.go` выглядит так:

```
package main

import (
    "fmt"
)

func main() {
    array := [12][4][7][10]float64{}
    x := len(array)
    y := len(array[0])
    z := len(array[0][0])
    w := len(array[0][0][0])
    fmt.Println("x:", x, "y:", y, "z:", z, "w:", w)
}
```

Существует массив `array`, в котором хранятся четыре измерения. Функция `len()`, которой предоставлены соответствующие аргументы, позволяет найти эти измерения. Для того чтобы получить первое измерение, нужно вызвать `len(array)`, чтобы получить второе — `len(array[0])` и т. д.

Выполнение `dimensions.go` приведет к следующим результатам:

```
$ go run dimensions.go
x: 12 y: 4 z: 7 w: 10
```

Разгадывание головоломок судоку

Основная цель этого раздела — помочь вам понять, что всегда следует использовать простейшую структуру данных, способную выполнить необходимую работу. В данном случае эта структура данных будет срезом, который мы используем для представления и проверки головоломки судоку. Мы могли бы использовать массив, потому что головоломки судоку имеют предопределенный размер.

Судоку — это логическая комбинаторная головоломка с числами. Проверка головоломки судоку заключается в том, чтобы убедиться, что она правильно решена, — такие задачи легко выполняются с помощью компьютерных программ.

Чтобы программа была как можно более универсальной, представленная здесь утилита `sudoku.go`, которую мы разделим на четыре части, будет загружать головоломки судоку из внешних файлов.

Первая часть `sudoku.go` выглядит так:

```
package main

import (
    "bufio"
    "errors"
    "fmt"
    "io"
    "os"
    "strconv"
    "strings"
)

func importFile(file string) ([][]int, error) {
    var err error
    var mySlice = make([][]int, 0)

    f, err := os.Open(file)
    if err != nil {
        return nil, err
    }
    defer f.Close()

    r := bufio.NewReader(f)
    for {
        line, err := r.ReadString('\n')
        fields := strings.Fields(line)
        temp := make([]int, 0)
        for _, v := range fields {
            n, err := strconv.Atoi(v)
            if err != nil {
                return nil, err
            }
```

```
            temp = append(temp, n)
        }

        if len(temp) != 0 {
            mySlice = append(mySlice, temp)
        }

        if err == io.EOF {
            break
        } else if err != nil {
            return nil, err
        }

        if len(temp) != len(mySlice[0]) {
            return nil, errors.New("Wrong number of elements!")
        }
    }
    return mySlice, nil
}
```

Функция `importFile()` проверяет, являются ли прочитанные данные допустимыми целыми числами. Проще говоря, `importFile()` принимает отрицательные целые или целые числа больше 9, но не примет значение `a`, которое не является целым числом или числом с плавающей точкой. Функция `importFile()` выполнит еще одну проверку, чтобы убедиться, что все строки входного файла имеют одинаковое количество целых чисел. В первой строке входного текстового файла указывается количество столбцов, которые должны присутствовать во всех остальных входных строках.

Вторая часть `sudoku.go` содержит следующий код Go:

```
func validPuzzle(sl [][]int) bool {
    for i := 0; i <= 2; i++ {
        for j := 0; j <= 2; j++ {
            iEl := i * 3
            jEl := j * 3
            mySlice := []int{0, 0, 0, 0, 0, 0, 0, 0, 0}
            for k := 0; k <= 2; k++ {
                for m := 0; m <= 2; m++ {
                    bigI := iEl + k
                    bigJ := jEl + m
                    val := sl[bigI][bigJ]
                    if val > 0 && val < 10 {
                        if mySlice[val-1] == 1 {
                            fmt.Println("Appeared 2 times:", val)
                            return false
                        } else {
                            mySlice[val-1] = 1
                        }
                    } else {
```

```
                        fmt.Println("Invalid value:", val)
                        return false
                    }
                }
            }
        }
    }
```

Чтобы перебрать все элементы головоломки судоку, в `sudoku.go` используется четыре цикла `for`. Применение четырех циклов `for` для массива 9 × 9 едва ли вызовет проблемы с производительностью, однако такие проблемы наверняка возникли бы, если бы нам приходилось работать с массивами больших размеров.

Третья часть `sudoku.go` выглядит так:

```
    // Тестирование столбцов
    for i := 0; i <= 8; i++ {
        sum := 0
        for j := 0; j <= 8; j++ {
            sum = sum + sl[i][j]
        }
        if sum != 45 {
            return false
        }
        sum = 0
    }

    // Тестирование строк
    for i := 0; i <= 8; i++ {
        sum := 0
        for j := 0; j <= 8; j++ {
            sum = sum + sl[j][i]
        }
        if sum != 45 {
            return false
        }
        sum = 0
    }

    return true
}
```

Представленный код гарантирует, что во всех столбцах и строках головоломки судоку содержатся только числа.

Последняя часть `sudoku.go` выглядит следующим образом:

```
func main() {
    arguments := os.Args
    if len(arguments) != 2 {
        fmt.Printf("usage: loadFile textFile size\n")
        return
    }
```

```
    file := arguments[1]

    mySlice, err := importFile(file)
    if err != nil {
        fmt.Println(err)
        return
    }

    if validPuzzle(mySlice) {
        fmt.Println("Correct Sudoku puzzle!")
    } else {
        fmt.Println("Incorrect Sudoku puzzle!")
    }
}
```

Функция `main()` управляет всей программой.

Выполнение `sudoku.go` для различных входных файлов приведет к результатам следующего вида:

```
$ go run sudoku.go OK.txt
Correct Sudoku puzzle!
$ go run sudoku.go noOK1.txt
Incorrect Sudoku puzzle!
```

Дополнительные ресурсы

Вам будет полезно обратиться к следующим ресурсам:

- изучите сайт, посвященный утилите *Graphviz*. Она позволяет строить графики, используя собственный язык: http://graphviz.org/;
- прочитайте страницу документации подпакетов стандартного Go-пакета `container`: https://golang.org/pkg/container/;
- если вы хотите больше узнать о структурах данных, советую прочитать книгу *The Design and Analysis of Computer Algorithms* Альфреда В. Ахо (Alfred V. Aho), Джона Э. Хопкрофта (John E. Hopcroft) и Джеффри Д. Уллмана (Jeffrey D. Ullman) (Addison-Wesley, 1974). Это отличная книга;
- чтобы больше узнать о хеш-функциях, посетите страницу https://en.wikipedia.org/wiki/Hash_function;
- еще одна действительно интересная книга об алгоритмах и структурах данных — *Programming Pearls* Джона Бентли (Jon Bentley) (Addison-Wesley Professional, 1999), а также *More Programming Pearls: Confessions of a Coder*, написанная тем же Джоном Бентли (Addison-Wesley Professional, 1988). Прочитав обе эти книги, вы сможете улучшить свои навыки программирования.

Упражнения

- Попробуйте изменить логику `generatePassword.go`, выбрав пароль из списка паролей, хранящихся в срезе Go, в сочетании с текущим системным временем или датой.
- Измените код `queue.go` так, чтобы вместо целых чисел хранить числа с плавающей точкой.
- Измените код Go в файле `stack.go` так, чтобы его узлы имели три поля с целочисленными данными — `Value`, `Number` и `Seed`. Какие еще изменения, помимо очевидных изменений в определении `Nodestruct`, необходимо внести в остальную часть программы?
- Можете ли вы изменить код `relatedList.go` так, чтобы узлы связного списка были упорядоченными?
- Аналогичным образом, можете ли вы изменить код Go в файле `duplyLList.go` так, чтобы узлы списка были упорядоченными? Можете ли вы разработать функцию удаления существующих узлов?
- Измените код `hashTableLookup.go` так, чтобы в хеш-таблице не было повторяющихся значений. Используйте для этого функцию `lookup()`.
- Перепишите `sudoku.go` так, чтобы вместо среза использовать хеш-таблицу Go.
- Перепишите `sudoku.go` так, чтобы вместо среза использовать связный список. Почему это сложно?
- Создайте программу Go, которая бы вычисляла мощность матриц. Существуют ли какие-либо условия, которые должны быть выполнены, чтобы можно было вычислить мощность матрицы?
- Реализуйте сложение и вычитание трехмерных массивов.
- Попробуйте реализовать матрицы на основе структур Go. Каковы основные проблемы такой реализации?
- Можете ли вы изменить Go-код `generatePassword.go` так, чтобы генерировать пароли, содержащие только заглавные буквы?
- Попробуйте изменить код `conHeap.go` так, чтобы поддерживать нестандартную и более сложную структуру, а не просто элементы типа `float32`.
- Реализуйте функциональность удаления узла, которая отсутствует в `connectedList.go`.
- Считаете ли вы, что двусвязный список улучшил бы код программы `queue.go`? Попробуйте реализовать очередь, используя двусвязный список вместо односвязного.

Резюме

В этой главе рассмотрено много интересных и практических тем, в том числе реализация связных списков, двусвязных списков, пользовательских хеш-таблиц, очередей и стеков в Go, а также использование функций стандартного Go-пакета `container`, проверка головоломок судоку, а также генерация в Go последовательностей псевдослучайных чисел и трудноугадываемых паролей.

Из этой главы вам следует запомнить, что основой каждой структуры данных является определение и реализация ее узлов. Наконец, вы узнали о реализации матричных вычислений.

Я уверен, что следующая глава станет для вас одной из самых интересных и ценных в книге. Ее основная тема — пакеты Go, а также информация о том, как определять и использовать различные типы функций Go в ваших программах. Кроме того, в следующей главе рассмотрены модули, которые, в сущности, являются пакетами, имеющими версии.

6 Неочевидные знания о пакетах и функциях Go

В предыдущей главе вы познакомились с разработкой и использованием специализированных структур данных, таких как связные списки, двоичные деревья и пользовательские хеш-таблицы, а также генерацией случайных чисел и трудноугадываемых паролей, выполнением операций с матрицами в Go.

В этой главе основное внимание уделено *пакетам* Go, которые позволяют организовывать, доставлять и использовать код Go. Наиболее распространенными компонентами пакета Go являются *функции*, они весьма гибки. Кроме того, из данной главы вы узнаете о *модулях* Go, которые представляют собой пакеты с версиями. В последнем разделе рассмотрены несколько расширенных пакетов — они относятся к стандартной библиотеке Go. Это позволит вам лучше понять, что не все пакеты Go одинаково полезны.

В данной главе рассмотрены следующие темы:

- разработка функций в Go;
- анонимные функции;
- функции, которые возвращают несколько значений;
- присваивание имен возвращаемым значениям функций;
- функции, которые возвращают другие функции;
- функции, которые получают другие функции в качестве параметров;
- функции с переменным числом параметров;
- разработка пакетов в Go;
- разработка модулей Go и работа с ними;
- публичные и приватные объекты пакетов;
- использование функции `init()` в пакетах;
- сложный стандартный Go-пакет `html/template`;

- стандартный пакет `text/template` — еще один по-настоящему сложный Go-пакет, который имеет собственный язык;
- расширенные пакеты `go/scanner`, `go/parser` и `go/token`;
- стандартный Go-пакет `syscall` — пакет низкого уровня, который хоть и редко применяется напрямую, однако широко используется другими Go-пакетами.

Что такое Go-пакеты

В Go все поставляется в виде пакетов. Go-пакет — это файл с исходным кодом Go, который начинается с ключевого слова `package`, после него стоит имя пакета. Некоторые пакеты имеют свою структуру. Например, в пакете `net` есть подкаталоги с именами `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.

Кроме пакетов стандартной библиотеки Go, существуют внешние пакеты, которые можно импортировать по полному адресу. Их следует загружать перед первым использованием. Одним из таких примеров является `github.com/matryer/is`. Он хранится в репозитории GitHub.

Как правило, пакеты используются для объединения взаимосвязанных функций, переменных и констант, чтобы их можно было легко переносить и использовать в программах Go. Заметьте, что, кроме пакета `main`, остальные Go-пакеты не являются автономными программами и не могут быть скомпилированы в исполняемые файлы. Это означает, что данные пакеты необходимо прямо или косвенно вызывать из основного пакета, чтобы их можно было использовать. Поэтому если вы попытаетесь выполнить Go-пакет как автономную программу, то будете разочарованы:

```
$ go run aPackage.go
go run: cannot run non-main package
```

Что такое функции Go

Функции являются важным элементом любого языка программирования: именно они позволяют разбивать большие программы на более мелкие и более управляемые части. Функции, насколько это возможно, должны быть независимы друг от друга. Они должны выполнять только одну задачу, и делать это хорошо. Поэтому, если вы обнаружите, что написали функцию, которая выполняет несколько задач, имеет смысл заменить ее несколькими функциями.

Самая популярная функция Go — `main()`, которая используется в любой независимой программе Go. Как вы уже наверняка знаете, все определения функций начинаются с ключевого слова `func`.

Анонимные функции

Анонимные функции можно определить как встроенные элементы и функции без имени. Обычно такие функции используются для реализации того, что не требует длинного кода. В Go функция может возвращать анонимную функцию или принимать ее в качестве аргумента. Кроме того, анонимная функция может быть присвоена переменной Go в качестве значения. Обратите внимание, что анонимные функции также называются *замыканиями*, особенно это принято в терминологии функционального программирования.

Считается хорошей практикой, чтобы анонимная функция была небольшого размера и имела локальное применение. Если она не локальна, то, возможно, лучше сделать ее обычной.

Если анонимная функция подходит для решения задачи, это очень удобно и значительно облегчает жизнь; но не используйте в своих программах слишком много анонимных функций без веской причины. Немного позже мы рассмотрим применение анонимных функций на практике.

Функции, которые возвращают несколько значений

Как вы уже знаете по примеру `strconv.Atoi()`, функции Go способны возвращать несколько значений, что избавляет от необходимости создавать специальную структуру для получения нескольких значений, возвращаемых из функции. Допустим, можно объявить следующую функцию, которая возвращает четыре значения (из них два — типа `int`, одно — типа `float64` и одно — `string`):

```
func aFunction() (int, int, float64, string) {
}
```

Теперь пора более подробно продемонстрировать работу анонимных функций и функций, которые возвращают несколько значений, используя в качестве примера код Go из файла `functions.go`. Мы рассмотрим этот код, разделив его на пять частей.

Первая часть кода `functions.go` выглядит так:

```
package main

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

Второй фрагмент `functions.go` содержит следующий код Go:

```
func doubleSquare(x int) (int, int) {
    return x * 2, x * x
}
```

Здесь мы видим определение и реализацию функции `doubleSquare()`, которая принимает один параметр типа `int` и возвращает два значения типа `int`.

Третья часть программы `functions.go` выглядит так:

```
func main() {
    arguments := os.Args
    if len(arguments) != 2 {
        fmt.Println("The program needs 1 argument!")
        return
    }

    y, err := strconv.Atoi(arguments[1])
    if err != nil {
        fmt.Println(err)
        return
    }
```

Этот код обрабатывает аргументы командной строки программы.

Четвертая часть программы `functions.go` содержит следующий код Go:

```
    square := func(s int) int {
        return s * s
    }
    fmt.Println("The square of", y, "is", square(y))

    double := func(s int) int {
        return s + s
    }
    fmt.Println("The double of", y, "is", double(y))
```

Обе переменные, `square` и `double`, содержат по анонимной функции. Нужно иметь в виду, что язык позволяет переопределять значение `square`, `double` или любой другой переменной, которой впоследствии будет присвоена анонимная функция, следовательно, функции, присвоенные этим переменным, могут измениться так, что они будут вычислять что-то другое.

В программировании не принято изменять код анонимных функций, которые присвоены переменным, так как это может стать главной причиной ошибок.

Последняя часть `functions.go` выглядит так:

```
    fmt.Println(doubleSquare(y))
    d, s := doubleSquare(y)
```

```
    fmt.Println(d, s)
}
```

Таким образом, мы можем либо вывести на экран значения, возвращаемые функцией, такой как `doubleSquare()`, либо присвоить их другим переменным.

Выполнение `functions.go` приведет к следующим результатам:

```
$ go run functions.go 1 21
The program needs 1 argument!
$ go run functions.go 10.2
strconv.Atoi: parsing "10.2": invalid syntax
$ go run functions.go 10
The square of 10 is 100
The double of 10 is 20
20 100
20 100
```

Функции, возвращающие именованные значения

В отличие от C, Go позволяет присваивать имена значениям, возвращаемым из функций. Кроме того, если в такой функции встречается оператор `return` без аргументов, то функция автоматически возвращает текущее состояние каждого именованного возвращаемого значения в той последовательности, в которой эти значения были объявлены в определении функции.

Файл с исходным кодом, на примере которого продемонстрированы функции Go с именованными возвращаемыми значениями, называется `returnNames.go`. Рассмотрим его, разделив на три части.

Первая часть программы `returnNames.go` выглядит так:

```
package main

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

func namedMinMax(x, y int) (min, max int) {
    if x > y {
        min = y
        max = x
    } else {
        min = x
        max = y
    }
    return
}
```

В этом фрагменте кода вы видите реализацию функции `namedMinMax()` с именованными возвращаемыми параметрами. Однако здесь есть сложный момент: явно

в операторе `return` функция `namedMinMax()` не возвращает никаких переменных или значений. Тем не менее, поскольку возвращаемые значения перечислены в сигнатуре этой функции, параметры `min` и `max` возвращаются автоматически в том порядке, в котором они объявлены в определении функции.

Второй фрагмент кода `returnNames.go` выглядит так:

```
func minMax(x, y int) (min, max int) {
    if x > y {
        min = y
        max = x
    } else {
        min = x
        max = y
    }
    return min, max
}
```

В функции `minMax()` также использованы именованные возвращаемые значения, однако в ее операторе `return` явно определены как возвращаемые переменные, так и последовательность, в которой они будут возвращены.

Последняя часть `returnNames.go` содержит следующий код Go:

```
func main() {
    arguments := os.Args
    if len(arguments) < 3 {
        fmt.Println("The program needs at least 2 arguments!")
        return
    }

    a1, _ := strconv.Atoi(arguments[1])
    a2, _ := strconv.Atoi(arguments[2])

    fmt.Println(minMax(a1, a2))
    min, max := minMax(a1, a2)
    fmt.Println(min, max)

    fmt.Println(namedMinMax(a1, a2))
    min, max = namedMinMax(a1, a2)
    fmt.Println(min, max)
}
```

Цель кода Go в функции `main()` — убедиться, что все методы дают одинаковые результаты. Выполнение `returnNames.go` приведет к следующим результатам:

```
$ go run returnNames.go -20 1
-20 1
-20 1
-20 1
-20 1
```

От научного редактора: использование именованных переменных очень полезно, если вам нужно вернуть значение из функции после обработки паники. Например:

```
func Run() (err error) {
    defer func() {
        if r := recover(); r != nil {
            switch t := r.(type) {
            case string:
                err = error.New(t)
            case error:
                err = t
            }
        }
    }()
    panic("A panic")
}
```

Функции, принимающие указатели

Функция может принимать в качестве параметра указатель при условии, что это допускает ее сигнатура. Мы рассмотрим использование указателей в качестве параметров функции на примере кода Go из файла `ptrFun.go`.

Первая часть `ptrFun.go` выглядит так:

```
package main

import (
    "fmt"
)

func getPtr(v *float64) float64 {
    return *v * *v
}
```

Как видите, функция `getPtr()` принимает в качестве параметра указатель, который ссылается на значение типа `float64`.

Вторая часть программы содержит следующий код Go:

```
func main() {
    x := 12.2
    fmt.Println(getPtr(&x))
    x = 12
    fmt.Println(getPtr(&x))
}
```

Здесь сложность заключается в том, чтобы передать адрес переменной в функцию `getPtr()`, поскольку для этого требуется параметр указателя. Это можно

сделать, поставив перед переменной знак амперсанда (&x). Выполнение ptrFun.go приведет к следующим результатам:

```
$ go run ptrFun.go
148.83999999999997
144
```

Если попытаться передать в getPtr() простое значение, такое как 12.12, и вызвать функцию — например, getPtr(12.12), — то компиляция программы завершится неудачно и будет выведено следующее сообщение об ошибке:

```
$ go run ptrFun.go
# command-line-arguments
./ptrFun.go:15:21: cannot use 12.12 (type float64) as type *float64 in argument
to getPtr
```

Функции, которые возвращают указатели

Как показано в программе pointerStruct.go в главе 4, целесообразно создавать новые структурные переменные с использованием отдельной функции и возвращать указатель на них из этой функции. Поэтому сценарий создания функций, возвращающих указатели, очень распространен. Вообще, такая функция упрощает структуру программы и позволяет разработчику сосредоточиться на более важном, вместо того чтобы раз за разом копировать один и тот же код Go. В этом разделе рассмотрен более простой пример, который вы найдете в файле returnPtr.go.

Первая returnPtr.go содержит следующий код Go:

```
package main

import (
    "fmt"
)

func returnPtr(x int) *int {
    y := x * x
    return &y
}
```

Помимо ожидаемой преамбулы, в этой части кода определена новая функция, которая возвращает указатель на переменную типа int. Единственное, что нужно помнить, — это использовать &y в операторе return, чтобы этот оператор возвращал адрес памяти переменной y.

Вторая часть returnPtr.go выглядит так:

```
func main() {
    sq := returnPtr(10)
    fmt.Println("sq value:", *sq)
```

Символ * разыменовывает переменную указателя, то есть возвращает не сам адрес памяти, а фактическое значение, которое хранится по этому адресу.

Последний фрагмент `returnPtr.go` содержит следующий код Go:

```
    fmt.Println("sq memory address:", sq)
}
```

Этот код возвращает адрес памяти переменной `sq`, а не значение типа `int`, хранящееся в этой переменной.

Если выполнить `returnPtr.go`, то получим следующий результат (у вас адрес памяти будет другим):

```
$ go run returnPtr.go
sq value: 100
sq memory address: 0xc00009a000
```

Функции, которые возвращают другие функции

В этом разделе вы узнаете, как реализовать функцию Go, которая возвращает другую функцию. В качестве примера рассмотрим код Go `returnFunction.go`, который разделим на три части. Первый фрагмент кода `returnFunction.go` выглядит так:

```
package main

import (
    "fmt"
)

func funReturnFun() func() int {
    i := 0
    return func() int {
        i++
        return i * i
    }
}
```

Как видно из реализации функции `funReturnFun()`, ее возвращаемое значение является анонимной функцией (`func() int`).

Второй фрагмент `returnFunction.go` содержит следующий код:

```
func main() {
    i := funReturnFun()
    j := funReturnFun()
```

В этом коде дважды вызывается функция `funReturnFun()`, и возвращаемое ею значение, которое является функцией, присваивается двум переменным с именами `i` и `j`. Как станет ясно из результатов работы программы, эти две переменные совершенно не связаны между собой.

Последний фрагмент кода `returnFunction.go` выглядит так:

```
    fmt.Println("1:", i())
    fmt.Println("2:", i())
    fmt.Println("j1:", j())
    fmt.Println("j2:", j())
    fmt.Println("3:", i())
}
```

В данном коде Go переменная `i` трижды использована как `i()`, а переменная `j` — дважды как `j()`. Здесь важно то, что, хотя `i` и `j` были созданы посредством вызова одной и той же функции `funReturnFun()`, они абсолютно независимы и не имеют между собой ничего общего.

Выполнение `returnFunction.go` приведет к следующим результатам:

```
$ go run returnFunction.go
1: 1
2: 4
j1: 1
j2: 4
3: 9
```

Как видно из результатов работы `returnFunction.go`, значение `i` в функции `funReturnFun()` продолжает увеличиваться и не становится равным `0` после каждого вызова `i()` или `j()`.

Функции, которые принимают другие функции в качестве параметров

Функции Go могут принимать в качестве параметров другие функции Go. Это свойство лишь расширяет спектр того, что мы можем делать с функциями Go. Два самых распространенных способа использования этого свойства — функции сортировки элементов и функция `filepath.Walk()`.

Однако в представленном здесь примере, который называется `funFun.go`, мы рассмотрим более простой случай, который работает с целочисленными значениями. Соответствующий код будет представлен в трех частях.

Первый фрагмент кода `funFun.go` содержит следующий код Go:

```
package main

import "fmt"

func function1(i int) int {
    return i + i
}
```

```
func function2(i int) int {
    return i * i
}
```

Итак, у нас есть две функции, каждая из которых принимает значение типа `int` и возвращает значение типа `int`. Мы будем использовать эти функции в качестве параметров для другой функции.

Второй фрагмент `funFun.go` содержит следующий код:

```
func funFun(f func(int) int, v int) int {
    return f(v)
}
```

Функция `funFun()` принимает два параметра: функцию с именем `f` и значение типа `int`. Параметр `f` должен быть функцией, которая принимает один аргумент типа `int` и возвращает значение типа `int`.

Последний фрагмент кода `funFun.go` выглядит так:

```
func main() {
    fmt.Println("function1:", funFun(function1, 123))
    fmt.Println("function2:", funFun(function2, 123))
    fmt.Println("Inline:", funFun(func(i int) int {return i * i * i}, 123))
}
```

В первом вызове `fmt.Println()` используется `funFun()`, где в качестве начального параметра указано имя функции `function1` без скобок, а во втором вызове `fmt.Println()` используется `funFun()`, где первым параметром является `function2`.

В последнем выражении `fmt.Println()` происходит нечто волшебное: реализация функции-параметра находится внутри вызова `funFun()`. Этот метод отлично работает для простых и коротких функций-параметров, однако в случае функций с большим количеством строк кода Go он может выглядеть гораздо хуже.

Выполнение `funFun.go` приведет к следующим результатам:

```
$ go run funFun.go
function1: 246
function2: 15129
Inline: 1860867
```

Функции с переменным числом параметров

Go также поддерживает функции, которые принимают переменное число аргументов. Наиболее популярные функции с переменным числом аргументов входят в пакет `fmt`. Мы рассмотрим такие функции на примере программы `variadic.go`. Разделим ее на три части.

Первая часть `variadic.go` выглядит так:

```
package main

import (
    "fmt"
    "os"
)

func varFunc(input ...string) {
    fmt.Println(input)
}
```

В этой части кода представлена реализация функции `varFunc()` с переменным числом аргументов строкового типа. Аргумент `input` — срез, который будет обрабатываться внутри функции `varFunc()` именно как срез. Оператор `...`, используемый как `...Type`, называется *оператором упаковки*. Соответствующий *оператор распаковки* начинается со среза и заканчивается знаком `...`. В функции с переменным числом аргументов оператор упаковки может использоваться только один раз.

Вторая часть `variadic.go` содержит следующий код Go:

```
func oneByOne(message string, s ...int) int {
    fmt.Println(message)
    sum := 0
    for i, a := range s {
        fmt.Println(i, a)
        sum = sum + a
    }
    s[0] = -1000
    return sum
}
```

Здесь вы видите еще одну функцию с переменным числом аргументов, которая называется `oneByOne()`. Эта функция принимает одну строку и переменное число целочисленных аргументов. Аргумент функции `s` является срезом.

Последняя часть `variadic.go` выглядит так:

```
func main() {
    arguments := os.Args
    varFunc(arguments...)
    sum := oneByOne("Adding numbers...", 1, 2, 3, 4, 5, -1, 10)
    fmt.Println("Sum:", sum)
    s := []int{1, 2, 3}
    sum = oneByOne("Adding numbers...", s...)
    fmt.Println(s)
}
```

Функция `main()` вызывает и использует две функции с переменным числом аргументов. Поскольку во втором вызове `oneByOne()` используется срез, то любые

изменения, которые вносятся в срез внутри функции с переменным числом параметров, после выхода из функции сохранятся.

Компиляция и выполнение `variadic.go` приведут к следующим результатам:

```
$ ./variadic 1 2
[./variadic 1 2]
Adding numbers...
0 1
1 2
2 3
3 4
4 5
5 -1
6 10
Sum: 24
Adding numbers...
0 1
1 2
2 3
[-1000 2 3]
```

Разработка Go-пакетов

В исходный код Go-пакета может входить несколько файлов и несколько каталогов. Он обычно размещается в одном каталоге, его имя совпадает с именем пакета, за единственным очевидным исключением — пакет `main` может размещаться где угодно.

В этом разделе мы разработаем простой Go-пакет с именем `aPackage`. Файл пакета называется `aPackage.go`, а его исходный код мы рассмотрим, разделив на две части.

Первая часть `aPackage.go` содержит следующий код Go:

```
package aPackage

import (
    "fmt"
)

func A() {
    fmt.Println("This is function A!")
}
```

Обратите внимание, что использование заглавных букв в именах Go-пакетов не принято — имя `aPackage` используется только в качестве примера.

Второй фрагмент кода `aPackage.go` выглядит так:

```
func B() {
    fmt.Println("privateConstant:", privateConstant)
}
```

```
const MyConstant = 123
const privateConstant = 21
```

Как видим, создать Go-пакет с нуля очень просто. Правда, сейчас мы еще не можем использовать этот пакет сам по себе, сначала нужно создать пакет с именем `main` и функцией `main()`, который позволит скомпилировать исполняемый файл. В данном случае программа, которая будет использовать `aPackage`, называется `useAPackage.go`, и в ней содержится следующий код Go:

```
import (
    "aPackage"
    "fmt"
)

func main() {
    fmt.Println("Using aPackage!")
    aPackage.A()
    aPackage.B()
    fmt.Println(aPackage.MyConstant)
}
```

Но если вы сейчас попытаетесь выполнить `useAPackage.go`, то получите сообщение об ошибке, которое говорит о том, что мы еще что-то не доделали:

```
$ go run useAPackage.go
useAPackage.go:4:2: cannot find package "aPackage" in any of:
    /usr/local/Cellar/go/1.9.2/libexec/src/aPackage (from $GOROOT)
    /Users/mtsouk/go/src/aPackage (from $GOPATH)
```

Есть еще кое-что, с чем вам нужно справиться. Как вы уже знаете из главы 1, для установки всех внешних пакетов в Go следует выполнить некоторые команды из оболочки UNIX. Сюда также входят пакеты, разработанные вами. Поэтому нужно поместить предыдущий пакет в соответствующий каталог и сделать его доступным для текущего пользователя UNIX. Чтобы установить разработанный вами пакет, необходимо открыть любую оболочку UNIX и выполнить следующие команды:

```
$ mkdir ~/go/src/aPackage
$ cp aPackage.go ~/go/src/aPackage/
$ go install aPackage
$ cd ~/go/pkg/darwin_amd64/
$ ls -l aPackage.a
-rw-r--r-- 1 mtsouk staff 4980 Dec 22 06:12 aPackage.a
```

Если каталога ~/go еще нет, то его нужно создать с помощью команды mkdir(1). В данном случае вам также нужно будет проделать эту операцию для каталога ~/go/src.

Выполнение `useAPackage.go` приведет к следующим результатам:

```
$ go run useAPackage.go
Using aPackage!
This is function A!
privateConstant: 21
123
```

Компиляция Go-пакета

Несмотря на то что мы не можем выполнить Go-пакет, если он не содержит функцию `main()`, мы можем скомпилировать этот пакет и создать объектный файл:

```
$ go tool compile aPackage.go
$ ls -l aPackage.*
-rw-r--r--@ 1 mtsouk staff 201 Jan 10 22:08 aPackage.go
-rw-r--r-- 1 mtsouk staff 16316 Mar 4 20:01 aPackage.o
```

Закрытые переменные и функции

Приватные переменные и функции отличаются от публичных тем, что первые могут использоваться и вызываться только внутри пакета. Управление тем, какие функции, константы и переменные являются публичными, а какие — приватными, также называется *инкапсуляцией*.

В Go действует простое правило, согласно которому функции, переменные, типы и т. д., имена которых начинаются с заглавной буквы, являются открытыми. А функции, переменные, типы и т. д., имена которых начинаются со строчной буквы, являются закрытыми. Именно поэтому функция `fmt.Println()` называется `Println()`, а не `println()`. Однако это правило не распространяется на имена пакетов, которые могут начинаться как с прописных, так и со строчных букв.

Функция init()

В каждом Go-пакете может присутствовать закрытая функция с именем `init()`, которая автоматически выполняется в начале выполнения пакета.

Функция init() — это закрытая функция, то есть ее нельзя вызвать извне пакета, к которому она принадлежит. Кроме того, поскольку пользователь пакета не имеет контроля над функцией init(), вам следует хорошо подумать, прежде чем использовать ее в общедоступных пакетах или изменять в ней любое глобальное состояние.

Сейчас я приведу пример кода с несколькими функциями `init()` из разных Go-пакетов. Рассмотрим код простейшего Go-пакета, который называется просто `a`:

```
package a

import (
    "fmt"
)

func init() {
    fmt.Println("init() a")
}

func FromA() {
    fmt.Println("fromA()")
}
```

В этом пакете реализована `init()` и публичная функция с именем `FromA()`.

После этого из оболочки UNIX нужно выполнить следующие команды, чтобы пакет стал доступен текущему пользователю UNIX:

```
$ mkdir ~/go/src/a
$ cp a.go ~/go/src/a/
$ go install a
```

Теперь рассмотрим код Go-пакета, который называется `b`:

```
package b

import (
    "a"
    "fmt"
)

func init() {
    fmt.Println("init() b")
}

func FromB() {
    fmt.Println("fromB()")
    a.FromA()
}
```

Что здесь происходит? Пакет `a` использует стандартный Go-пакет `fmt`. Однако пакет `b` должен импортировать пакет `a`, поскольку в нем применяется функция `a.FromA()`. И в `a`, и в `b` есть функция `init()`.

Как и раньше, нам нужно установить этот пакет и сделать его доступным для текущего пользователя UNIX, выполнив из оболочки UNIX следующие команды:

```
$ mkdir ~/go/src/b/
$ cp b.go ~/go/src/b
$ go install b
```

Таким образом, сейчас у нас есть два Go-пакета, в каждом из которых есть функция `init()`. Теперь попытайтесь угадать, что мы получим, если выполним файл `manyInit.go`, который содержит следующий код:

```
package main

import (
    "a"
    "b"
    "fmt"
)

func init() {
    fmt.Println("init() manyInit")
}

func main() {
    a.FromA()
    b.FromB()
}
```

В сущности, главный вопрос — сколько раз будут выполняться функции `init()` этих пакетов? Выполнение `manyInit.go` приведет к следующим результатам, что и будет ответом на наш вопрос:

```
$ go run manyInit.go
init() a
init() b
init() manyInit
fromA()
fromB()
fromA()
```

Как видите, функция `init()` для пакета `a` выполняется только один раз, несмотря на то что пакет импортируется дважды, двумя разными пакетами. Кроме того, поскольку сначала выполняется блок `import` из `manyInit.go`, функции `init()` пакетов `a` и `b` запускаются раньше, чем функция `init()` файла `manyInit.go`, что вполне оправданно. Основная причина этого поведения заключается в том, что функции `init()` файла `manyInit.go` разрешено использовать элемент из `a` или из `b`.

Функция `init()` может быть очень полезной, если нужно присвоить значения некоторым неэкспортированным внутренним переменным. Например, в `init()` можно определить текущий часовой пояс. Наконец, учтите, что в одном файле может быть много функций `init()`, однако в Go эта функция используется редко.

Go-модули

Модули были впервые представлены в версии Go 1.11. На момент написания этой книги последней версией Go была 1.13. Несмотря на то что общая идея модулей Go остается неизменной, некоторые из представленных здесь деталей в будущих версиях Go могут измениться.

Модуль Go похож на Go-пакет с версией. Для версий модулей в Go используется *семантическое управление версиями*. Это означает, что версии начинаются с буквы `v`, за которой следует номер. Поэтому у вас могут быть такие версии, как `v1.0.0`, `v1.0.5` и `v2.0.2`. Здесь `v1`, `v2` или `v3` обозначают основную версию Go-пакета, которая обычно не имеет обратной совместимости. Следовательно, если программа Go работает с версией `v1`, это не гарантирует, что она будет работать с `v2` или `v3` — возможно, программа и будет с ними работать, но рассчитывать на это не следует.

Второе число в версии указывает на свойства модуля. Обычно у `v1.1.0` больше функций, чем у `v1.0.2` или `v1.0.0`, но при этом `v1.1.0` сохраняет совместимость с более ранними версиями.

Наконец, третье число говорит об исправлении ошибок, без внесения каких-либо новых функций. Заметьте, что семантическое управление версиями также используется в именовании версий самого языка Go. Обратите также внимание, что Go-модули позволяют писать код за пределами `GOPATH`.

Создание и использование Go-модулей

В этом подразделе мы создадим первую версию простейшего модуля. Для этого нам понадобится репозиторий GitHub, чтобы хранить код Go. В моем случае хранилище GitHub находится по адресу https://github.com/mactsouk/myModule. Мы начнем с пустого репозитория GitHub, в котором будет только файл `README.md`. Итак, сначала нам нужно выполнить следующую команду, чтобы получить содержимое репозитория GitHub:

```
$ git clone git@github.com:mactsouk/myModule.git
Cloning into 'myModule'...
remote: Enumerating objects: 7, done.
remote: Counting objects: 100% (7/7), done.
remote: Compressing objects: 100% (6/6), done.
remote: Total 7 (delta 1), reused 0 (delta 0), pack-reused 0
Receiving objects: 100% (7/7), done.
Resolving deltas: 100% (1/1), done.
```

Если вы выполните ту же команду на своем компьютере, то получите мой репозиторий GitHub, который к тому моменту, как вы будете читать эти строки, не будет пустым. Если хотите создать собственный Go-модуль с нуля, то вам нужно создать собственный пустой репозиторий GitHub.

Создание версии v1.0.0

Для того чтобы создать версию v1.0.0 простейшего Go-модуля, нужно выполнить следующие команды:

```
$ go mod init
go: creating new go.mod: module github.com/mactsouk/myModule
$ touch myModule.go
$ vi myModule.go
$ git add .
$ git commit -a -m "Initial version 1.0.0"
$ git push
$ git tag v1.0.0
$ git push -q origin v1.0.0
$ go list
github.com/mactsouk/myModule
$ go list -m
github.com/mactsouk/myModule
```

Содержимое файла `myModule.go` выглядит так:

```
package myModule

import (
    "fmt"
)

func Version() {
    fmt.Println("Version 1.0.0")
}
```

Содержимое созданного ранее файла `go.mod` имеет следующий вид:

```
$ cat go.mod
module github.com/mactsouk/myModule
go 1.12
```

Использование версии v1.0.0

Здесь вы узнаете, как использовать версию v1.0.0 созданного нами модуля Go. Чтобы использовать модули Go, нужно создать программу Go, которая в данном случае называется `useModule.go` и содержит следующий код Go:

```
package main

import (
    v1 "github.com/mactsouk/myModule"
)

func main() {
    v1.Version()
}
```

Нам нужно указать путь к модулю Go (`github.com/mactsouk/myModule`) — в данном случае у модуля Go также есть псевдоним (`v1`). Псевдонимы для пакетов в Go используются крайне редко; но в нашем примере это облегчает чтение кода. Тем не менее подобный прием не следует использовать в рабочем коде без уважительной причины.

Если просто попытаться выполнить файл `useModule.go`, который в данном случае будет помещен в каталог `/tmp`, то это не получится, потому что требуемый модуль отсутствует в системе:

```
$ pwd
/tmp
$ go run useModule.go
useModule.go:4:2: cannot find package "github.com/mactsouk/myModule" in any of:
    /usr/local/Cellar/go/1.12/libexec/src/github.com/mactsouk/myModule
(from $GOROOT)
    /Users/mtsouk/go/src/github.com/mactsouk/myModule (from $GOPATH)
```

Поэтому, чтобы получить необходимые модули Go и успешно выполнить `useModule.go`, нужно ввести следующие команды:

```
$ export GO111MODULE=on
$ go run useModule.go
go: finding github.com/mactsouk/myModule v1.0.0
go: downloading github.com/mactsouk/myModule v1.0.0
go: extracting github.com/mactsouk/myModule v1.0.0
Version 1.0.0
```

Таким образом, код `useModule.go` является корректной программой и может быть выполнен. Теперь пора привести все в порядок, присвоив `useModule.go` имя и выполнив сборку:

```
$ go mod init hello
go: creating new go.mod: module hello
$ go build
```

Последняя команда генерирует исполняемый файл и размещает его в каталоге `/tmp`, а также создает еще два дополнительных файла с именами `go.sum` и `go.mod`. Содержимое `go.sum` будет следующим:

```
$ cat go.sum
github.com/mactsouk/myModule v1.0.0
h1:eTCn2Jewnajw0REKONrVhHmeDEJ0Q5TAZ0xsSbh8kFs=
github.com/mactsouk/myModule v1.0.0/go.mod
h1:s3ziarTDDvaXaHWYYOf/ULi97aoBd6JfnvAkM8rSuzg=
```

В файле `go.sum` хранится контрольная сумма всех загруженных модулей.

Содержимое `go.mod` выглядит так:

```
$ cat go.mod
module hello
go 1.12
require github.com/mactsouk/myModule v1.0.0
```

Обратите внимание: если в файле go.mod, принадлежащем нашему проекту, указана версия Go v1.3.0, то Go будет использовать версию v1.3.0, даже если на данный момент доступна более новая версия Go.

Создание версии v1.1.0

В этом пункте мы создадим новую версию `myModule`, используя другой *тег*. Однако на этот раз, в отличие от предыдущего, нет необходимости выполнять команду `go mod init`. Нам нужно лишь выполнить следующие команды:

```
$ vi myModule.go
$ git commit -a -m "v1.1.0"
[master ddd0742] v1.1.0
    1 file changed, 1 insertion(+), 1 deletion(-)
$ git push
$ git tag v1.1.0
$ git push -q origin v1.1.0
```

В этой версии `myModule.go` содержится следующий код:

```
package myModule

import (
    "fmt"
)

func Version() {
    fmt.Println("Version 1.1.0")
}
```

Использование версии v1.1.0

В этом пункте вы узнаете, как использовать версию v1.1.0 созданного нами Go-модуля. Воспользуемся *образом* Docker, чтобы обеспечить максимальную независимость от машины, которую мы использовали для разработки модуля. Чтобы получить образ Docker и перейти в его оболочку UNIX, мы воспользуемся следующей командой:

```
$ docker run --rm -it golang:latest
root@884c0d188694:/go# cd /tmp
root@58c5688e3ee0:/tmp# go version
go version go1.13 linux/amd64
```

Как видим, образ Docker использует последнюю версию Go — на момент написания книги это была версия 1.13. Чтобы использовать один или несколько Go-модулей, нужно создать программу Go, которая называется `useUpdatedModule.go`. Она содержит следующий код Go:

```
package main

import (
```

```
    v1 "github.com/mactsouk/myModule"
)

func main() {
    v1.Version()
}
```

Код Go из файла `useUpdatedModule.go` совпадает с кодом Go в файле `useModule.go`. Его преимущество в том, что вы автоматически получите последнее обновление версии `v1`.

Когда программа в образе Docker будет написана, нужно будет сделать следующее:

```
root@58c5688e3ee0:/tmp# ls -l
total 4
-rw-r--r-- 1 root root 91 Mar 2 19:59 useUpdatedModule.go
root@58c5688e3ee0:/tmp# export GO111MODULE=on
root@58c5688e3ee0:/tmp# go run useUpdatedModule.go
go: finding github.com/mactsouk/myModule v1.1.0
go: downloading github.com/mactsouk/myModule v1.1.0
go: extracting github.com/mactsouk/myModule v1.1.0
Version 1.1.0
```

Как видим, `useUpdatedModule.go` автоматически использует последнюю версию `v1` Go-модуля. Крайне важно выполнить команду `export GO111MODULE=on`, чтобы активизировать поддержку модуля.

Если попытаться выполнить программу `useModule.go`, которая находится в каталоге `/tmp` на вашем локальном компьютере, то получим следующий результат:

```
$ ls -l go.mod go.sum useModule.go
-rw-------  1 mtsouk  wheel   67  Mar  2 21:29 go.mod
-rw-------  1 mtsouk  wheel  175  Mar  2 21:29 go.sum
-rw-r--r--  1 mtsouk  wheel   92  Mar  2 21:12 useModule.go
$ go run useModule.go
Version 1.0.0
```

Это означает, что `useModule.go`, как и раньше, использует более старую версию Go-модуля. Если вы хотите, чтобы в программе `useModule.go` использовалась последняя версия Go-модуля, можете сделать следующее:

```
$ rm go.mod go.sum
$ go run useModule.go
go: finding github.com/mactsouk/myModule v1.1.0
go: downloading github.com/mactsouk/myModule v1.1.0
go: extracting github.com/mactsouk/myModule v1.1.0
Version 1.1.0
```

Если вы захотите вернуться к использованию версии `v1.0.0` модуля, то можете сделать следующее:

```
$ go mod init hello
go: creating new go.mod: module hello
```

```
$ go build
$ go run useModule.go
Version 1.1.0
$ cat go.mod
module hello
go 1.12
require github.com/mactsouk/myModule v1.1.0
$ vi go.mod
$ cat go.mod
module hello
go 1.12
require github.com/mactsouk/myModule v1.0.0
$ go run useModule.go
Version 1.0.0
```

В следующем пункте мы создадим старшую версию Go-модуля. Это означает, что вместо нового тега нам потребуется использовать другую ветку GitHub.

Создание версии v2.0.0

В этом пункте мы создадим вторую старшую версию `myModule`. Обратите внимание, что старшие версии нужно явно указать в операторах `import`.

Таким образом, вместо `github.com/mactsouk/myModule` у нас будет `github.com/mactsouk/myModule/v2` для версии `v2` и `github.com/mactsouk/myModule/v3` для версии `v3`.

Прежде всего мы создадим новую ветку GitHub:

```
$ git checkout -b v2
Switched to a new branch 'v2'
$ git push --set-upstream origin v2
```

Затем сделаем следующее:

```
$ vi go.mod
$ cat go.mod
module github.com/mactsouk/myModule/v2
go 1.12
$ git commit -a -m "Using 2.0.0"
[v2 5af2269] Using 2.0.0
2 files changed, 2 insertions(+), 2 deletions(-)
$ git tag v2.0.0
$ git push --tags origin v2
Counting objects: 4, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (4/4), 441 bytes | 441.00 KiB/s, done.
Total 4 (delta 1), reused 0 (delta 0)
remote: Resolving deltas: 100% (1/1), completed with 1 local object.
To github.com:mactsouk/myModule.git
  * [new branch]     v2 -> v2
  * [new tag]        v2.0.0 -> v2.0.0
```

```
$ git --no-pager branch -a
    master
* v2
  remotes/origin/HEAD -> origin/master
  remotes/origin/master
  remotes/origin/v2
```

В эту старшую версию `myModule.go` мы поместим следующий код:

```
package myModule

import (
    "fmt"
)

func Version() {
    fmt.Println("Version 2.0.0")
}
```

Использование версии v2.0.0

Для того чтобы использовать Go-модули, мы снова создадим программу Go, которая на этот раз будет называться `useV2.go` и содержать следующий код Go:

```
package main

import (
    v "github.com/mactsouk/myModule/v2"
)

func main() {
    v.Version()
}
```

Мы будем использовать образ Docker, как самый удобный способ оперирования с Go-модулями, поскольку начнем с чистой установки Go:

```
$ docker run --rm -it golang:latest
root@191d84fc5571:/go# cd /tmp
root@191d84fc5571:/tmp# cat > useV2.go
package main
import (
    v "github.com/mactsouk/myModule/v2"
)
func main() {
    v.Version()
}
root@191d84fc5571:/tmp# export GO111MODULE=on
root@191d84fc5571:/tmp# go run useV2.go
go: finding github.com/mactsouk/myModule/v2 v2.0.0
go: downloading github.com/mactsouk/myModule/v2 v2.0.0
```

```
go: extracting github.com/mactsouk/myModule/v2 v2.0.0
Version 2.0.0
```

Все работает нормально — образ Docker использует модуль `myModule` версии 2.0.0.

Создание версии v2.1.0

Теперь мы создадим обновленную версию `myModule.go`, которая будет использовать другой тег GitHub. Для этого выполним следующие команды:

```
$ vi myModule.go
$ git commit -a -m "v2.1.0"
$ git push
$ git tag v2.1.0
$ git push -q origin v2.1.0
```

Измененное содержимое `myModule.go` будет выглядеть так:

```
package myModule

import (
    "fmt"
)

func Version() {
    fmt.Println("Version 2.1.0")
}
```

Использование версии v2.1.0

Как вы уже знаете, чтобы использовать Go-модули, нам нужно создать программу Go, которая будет называться `useUpdatedV2.go`. Она содержит следующий код Go:

```
package main

import (
    v "github.com/mactsouk/myModule/v2"
)

func main() {
    v.Version()
}
```

Однако здесь нет необходимости объявлять, что мы хотим использовать последнюю версию `v2` Go-модуля, — об этом позаботится сам Go. Именно поэтому программы `useUpdatedV2.go` и `useV2.go` абсолютно одинаковы.

И снова для простоты мы воспользуемся образом Docker. Причина использования команды `cat(1)` для создания `useUpdatedV2.go` заключается в том, что данный образ Docker поставляется без предустановленной утилиты `vi(1)`.

```
$ docker run --rm -it golang:1.12
root@ccfcd675e333:/go# cd /tmp/
root@ccfcd675e333:/tmp# cat > useUpdatedV2.go
package main
import (
    v "github.com/mactsouk/myModule/v2"
)
func main() {
    v.Version()
}
root@ccfcd675e333:/tmp# ls -l
total 4
-rw-r--r-- 1 root root 92 Mar 2 20:34 useUpdatedV2.go
root@ccfcd675e333:/tmp# go run useUpdatedV2.go
useUpdatedV2.go:4:2: cannot find package "github.com/mactsouk/myModule/v2" in any of:
    /usr/local/go/src/github.com/mactsouk/myModule/v2 (from $GOROOT)
    /go/src/github.com/mactsouk/myModule/v2 (from $GOPATH)
root@ccfcd675e333:/tmp# export GO111MODULE=on
root@ccfcd675e333:/tmp# go run useUpdatedV2.go
go: finding github.com/mactsouk/myModule/v2 v2.1.0
go: downloading github.com/mactsouk/myModule/v2 v2.1.0
go: extracting github.com/mactsouk/myModule/v2 v2.1.0
Version 2.1.0
```

Подробнее о git(1) и GitHub вы прочитаете в главе 7.

Использование двух версий одного и того же Go-модуля

В этом подразделе вы увидите, как использовать две старшие версии одного и того же Go-модуля в одной программе Go. Этот подход применим и в том случае, если вы хотите использовать одновременно более двух старших версий Go-модуля.

Мы создадим следующую программу Go и сохраним ее в файле `useTwo.go`:

```
package main

import (
    v1 "github.com/mactsouk/myModule"
    v2 "github.com/mactsouk/myModule/v2"
)
```

```
func main() {
    v1.Version()
    v2.Version()
}
```

Как видим, нам нужно лишь явно импортировать старшие версии Go-модуля, которые мы хотим использовать, и присвоить им разные псевдонимы.

Выполнение `useTwo.go` приведет к следующим результатам:

```
$ export GO111MODULE=on
$ go run useTwo.go
go: creating new go.mod: module github.com/PacktPublishing/Mastering-Go-Second-Edition
go: finding github.com/mactsouk/myModule/v2 v2.1.0
go: downloading github.com/mactsouk/myModule/v2 v2.1.0
go: extracting github.com/mactsouk/myModule/v2 v2.1.0
Version 1.1.0
Version 2.1.0
```

Где хранятся Go-модули

В этом подразделе мы узнаем, где и как Go хранит код и информацию о Go-модулях, которые мы используем, на примере созданного нами Go-модуля. После использования этого Go-модуля на моем компьютере с MacOS Mojave содержимое каталога `~/go/pkg/mod/github.com/mactsouk` выглядело так:

```
$ ls -lR ~/go/pkg/mod/github.com/mactsouk
total 0
drwxr-xr-x  3 mtsouk  staff     96B Mar  2 22:38 my!module
dr-x------  6 mtsouk  staff    192B Mar  2 21:18 my!module@v1.0.0
dr-x------  6 mtsouk  staff    192B Mar  2 22:07 my!module@v1.1.0
/Users/mtsouk/go/pkg/mod/github.com/mactsouk/my!module:
total 0
dr-x------  6 mtsouk  staff    192B Mar  2 22:38 v2@v2.1.0
/Users/mtsouk/go/pkg/mod/github.com/mactsouk/my!module/v2@v2.1.0:
total 24
-r--r--r--  1 mtsouk  staff     28B Mar  2 22:38 README.md
-r--r--r--  1 mtsouk  staff     48B Mar  2 22:38 go.mod
-r--r--r--  1 mtsouk  staff     86B Mar  2 22:38 myModule.go
/Users/mtsouk/go/pkg/mod/github.com/mactsouk/my!module@v1.0.0:
total 24
-r--r--r--  1 mtsouk  staff     28B Mar  2 21:18 README.md
-r--r--r--  1 mtsouk  staff     45B Mar  2 21:18 go.mod
-r--r--r--  1 mtsouk  staff     86B Mar  2 21:18 myModule.go
/Users/mtsouk/go/pkg/mod/github.com/mactsouk/my!module@v1.1.0:
total 24
-r--r--r--  1 mtsouk  staff     28B Mar  2 22:07 README.md
-r--r--r--  1 mtsouk  staff     45B Mar  2 22:07 go.mod
-r--r--r--  1 mtsouk  staff     86B Mar  2 22:07 myModule.go
```

Лучший способ научиться разрабатывать и использовать Go-модули — экспериментировать. Без Go-модулей вам не обойтись, так что начинайте применять их прямо сейчас.

Команда go mod vendor

Иногда необходимо хранить все зависимости в одном месте, рядом с файлами проекта. Именно это позволяет сделать команда `go mod vendor`:

```
$ cd useTwoVersions
$ go mod init useV1V2
go: creating new go.mod: module useV1V2
$ go mod vendor
$ ls -l
total 24
-rw-------  1 mtsouk  staff    114B Mar  2 22:43   go.mod
-rw-------  1 mtsouk  staff    356B Mar  2 22:43   go.sum
-rw-r--r--@ 1 mtsouk  staff    143B Mar  2 19:36   useTwo.go
drwxr-xr-x  4 mtsouk  staff    128B Mar  2 22:43   vendor
$ ls -l vendor/github.com/mactsouk/myModule
total 24
-rw-r--r--  1 mtsouk  staff     28B Mar  2 22:43   README.md
-rw-r--r--  1 mtsouk  staff     45B Mar  2 22:43   go.mod
-rw-r--r--  1 mtsouk  staff     86B Mar  2 22:43   myModule.go
drwxr-xr-x  6 mtsouk  staff    192B Mar  2 22:43   v2
$ ls -l vendor/github.com/mactsouk/myModule/v2
total 24
-rw-r--r--  1 mtsouk  staff     28B Mar  2 22:43   README.md
-rw-r--r--  1 mtsouk  staff     48B Mar  2 22:43   go.mod
-rw-r--r--  1 mtsouk  staff     86B Mar  2 22:43   myModule.go
```

Главное здесь — выполнение команды `go mod init <имя пакета>` перед командой `go mod vendor`.

Как писать хорошие Go-пакеты

В этом разделе вы найдете полезные советы, которые помогут вам разрабатывать хорошие Go-пакеты. Как вы уже знаете, Go-пакеты хранятся в каталогах, могут содержать публичные и приватные элементы. Публичные элементы могут использоваться как внутри пакета, так и извне, другими пакетами, тогда как приватные могут использоваться только внутри пакета.

Вот несколько хороших правил, которые позволят вам создавать отличные Go-пакеты:

- первое негласное правило хорошего пакета — то, что его элементы должны быть каким-то образом взаимосвязаны. Например, вы можете написать пакет для автомобилей, а вот сделать единый пакет для поддержки и автомобилей, и велосипедов — плохая идея. Проще говоря, лучше лишний раз разделить функциональность между несколькими пакетами, чем вместить слишком много функций в один Go-пакет. Кроме того, пакеты должны быть простыми и стильными, но не слишком простыми и нестабильными;
- второе правило заключается в том, что следует сначала в течение какого-то времени использовать собственные пакеты, прежде чем выставлять их на всеобщее обозрение. Это поможет обнаружить ошибки и убедиться, что ваши пакеты работают должным образом. После этого дайте их другим разработчикам для дополнительного тестирования и только потом сделайте общедоступными;
- попробуйте представить, каким пользователям понравятся ваши пакеты, и убедитесь, что, решив одни проблемы, эти пакеты не создадут новые;
- пакет не должен экспортировать бесчисленные функции без веских на то причин. Пакеты с кратким списком экспортируемых функций понятнее и проще в использовании. После этого постарайтесь дать своим функциям описательные, но не очень длинные имена;
- *интерфейсы* способны повысить полезность ваших функций. Поэтому, если вы считаете это целесообразным, используйте интерфейсы вместо отдельных типов в качестве параметров и возвращаемых значений функций;
- обновляя один из своих пакетов, постарайтесь ничего не поломать, а также не создать несовместимость со старыми версиями, если только в этом нет крайней необходимости;
- при разработке нового Go-пакета старайтесь использовать несколько файлов, чтобы группировать похожие задачи или концепции;
- также старайтесь следовать правилам, принятым для разработки Go-пакетов из стандартной библиотеки. В этом вам поможет чтение кода Go-пакетов, входящих в стандартную библиотеку;
- не создавайте новый пакет, если похожий уже существует. Внесите изменения в существующий пакет или, возможно, создайте собственную версию;
- никому не понравится, если Go-пакет будет выводить на экран информацию о регистрации. Более профессионально создать флаг, который бы позволил активизировать регистрацию, когда это необходимо;
- код Go ваших пакетов должен соответствовать коду Go ваших программ. Посмотрите на программу, в которой используются ваши пакеты: если имена функций плохо выделяются в коде, лучше их изменить. Поскольку имя пакета используется практически везде, постарайтесь давать пакетам краткие и выразительные имена;

- будет удобнее, если вы поместите новые определения типов Go рядом с тем местом, где они будут впервые использоваться: никто, включая вас, не захочет искать в исходных файлах определения новых типов данных;
- постарайтесь создать для каждого пакета тестовые файлы. Пакеты с тестовыми файлами считаются более профессиональными, чем без них; мелкие детали имеют решающее значение, они дают людям понять, что вы серьезный разработчик! Обратите внимание, что написание тестов для пакетов не является обязательным, но следует избегать использования пакетов, которые не включают в себя тесты. Подробнее о тестировании вы прочитаете в главе 11;
- наконец, не пишите Go-пакет только потому, что у вас нет ничего лучше, — поищите что-нибудь более подходящее, не тратьте время зря!

Всегда помните, что готовый код Go в пакете не просто не должен содержать ошибок — следующим по важности свойством успешного пакета является его документация, а также несколько примеров кода, в которых демонстрируется его применение и показаны характерные особенности функционирования пакета.

Пакет syscall

В этом разделе описана лишь небольшая часть функциональности стандартного Go-пакета `syscall`. Обратите внимание, что в состав пакета `syscall` входит множество функций и типов, связанных с низкоуровневыми примитивами операционной системы. Кроме того, пакет `syscall` широко используется другими Go-пакетами, такими как `os`, `net` и `time`, которые предоставляют кросс-платформенный интерфейс для операционной системы. Таким образом, пакет `syscall` не самый переносимый пакет в библиотеке Go — это и не входит в его задачу.

Несмотря на то что у систем UNIX есть много общего, в них также есть всевозможные различия, особенно если говорить о внутренних системах. Задача пакета `syscall` — как можно осторожнее сгладить все эти несовместимости. Это вовсе не секрет, пакет хорошо задокументирован, и именно поэтому `syscall` так успешно применяется.

Системный вызов (system call) — это способ, позволяющий приложению программно получить что-то от ядра операционной системы. Вследствие этого системные вызовы отвечают за доступ и работу с большинством низкоуровневых элементов UNIX, таких как процессы, устройства хранения данных, печать данных, сетевые интерфейсы и всевозможные файлы. Проще говоря, вы не можете работать в системе UNIX, не используя системные вызовы. Чтобы проверить системные

вызовы процесса UNIX, можно воспользоваться такими утилитами, как strace(1) и dtrace(1) из главы 2.

Мы рассмотрим использование пакета syscall на примере программы useSyscall.go, которую разделим на четыре части.

Вам может и не понадобиться использовать пакет syscall напрямую, если только вы не работаете над низкоуровневыми программами. Есть Go-пакеты «не для всех», и syscall — один из них!

Первая часть кода useSyscall.go выглядит так:

```
package main

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

Это простая часть программы, где мы просто импортируем нужные Go-пакеты.

Вторая часть useSyscall.go содержит следующий код Go:

```
func main() {
    pid, _, _ := syscall.Syscall(39, 0, 0, 0)
    fmt.Println("My pid is", pid)
    uid, _, _ := syscall.Syscall(24, 0, 0, 0)
    fmt.Println("User ID:", uid)
```

В этой части мы с помощью двух вызовов функции syscall.Syscall() получаем информацию об идентификаторах процесса и пользователя. Первый параметр в вызове syscall.Syscall() определяет запрашиваемую информацию.

Третий фрагмент useSyscall.go содержит следующий код Go:

```
    message := []byte{'H', 'e', 'l', 'l', 'o', '!', '\n'}
    fd := 1
    syscall.Write(fd, message)
```

В этой части мы выводим на экран сообщение с помощью syscall.Write(). Первый параметр этой функции — дескриптор файла, в который будут записываться данные, а второй — байтовый срез, в котором, собственно, и хранится выводимое сообщение. Функция syscall.Write() является переносимой.

Последняя часть программы useSyscall.go содержит такой код:

```
    fmt.Println("Using syscall.Exec()")
    command := "/bin/ls"
    env := os.Environ()
    syscall.Exec(command, []string{"ls", "-a", "-x"}, env)
}
```

В последней части программы показано, как можно использовать функцию `syscall.Exec()` для выполнения внешней команды. Однако мы не можем контролировать данные, выводимые командой, — они выводятся на экран автоматически.

Выполнение `useSyscall.go` в операционной системе macOS Mojave приводит к следующим результатам:

```
$ go run useSyscall.go
My pid is 14602
User ID: 501
Hello!
Using syscall.Exec()
.                ..                  a.go
funFun.go        functions.go        html.gohtml
htmlT.db         htmlT.go            manyInit.go
ptrFun.go        returnFunction.go   returnNames.go
returnPtr.go     text.gotext         textT.go
useAPackage.go   useSyscall.go
```

Выполнение этой же программы на компьютере с Debian Linux привело к следующим результатам:

```
$ go run useSyscall.go
My pid is 20853
User ID: 0
Hello!
Using syscall.Exec()
.                ..                  a.go
funFun.go        functions.go        html.gohtml
htmlT.db         htmlT.go            manyInit.go
ptrFun.go        returnFunction.go   returnNames.go
returnPtr.go     text.gotext         textT.go
useAPackage.go   useSyscall.go
```

Таким образом, несмотря на то что большая часть выходных данных не изменилась, вызов `syscall.Syscall(39, 0, 0, 0)` в Linux не работает, поскольку идентификатор пользователя в Linux не может быть равен `0`. Это означает, что данная команда не является переносимой.

Чтобы узнать, какие стандартные Go-пакеты используют пакет `syscall`, можно выполнить из оболочки UNIX следующую команду:

```
$ grep \"syscall\" `find /usr/local/Cellar/go/1.12/libexec/src -name "*.go"`
```

Только не забудьте заменить в ней `/usr/local/Cellar/go/1.12/libexec/src` на правильный путь к каталогу в вашей системе.

Как на самом деле работает fmt.Println()

Если вы хотите понять, в чем заключается настоящая польза от пакета `syscall`, для начала прочитайте этот подраздел. Реализация функции `fmt.Println()`, которая находится в файле https://golang.org/src/fmt/print.go, выглядит следующим образом:

```
func Println(a ...interface{}) (n int, err error) {
    return Fprintln(os.Stdout, a...)
}
```

Таким образом, для выполнения своей работы функция `fmt.Println()` вызывает `fmt.Fprintln()`. Реализация `fmt.Fprintln()`, которая находится в том же файле, выглядит так:

```
func Fprintln(w io.Writer, a ...interface{}) (n int, err error) {
    p := newPrinter()
    p.doPrintln(a)
    n, err = w.Write(p.buf)
    p.free()
    return
}
```

Как видим, на самом деле операцию записи в `fmt.Fprintln()` выполняет функция `Write()` из интерфейса `io.Writer`. В данном случае интерфейс `io.Writer` — это `os.Stdout`, который описан в файле https://golang.org/src/os/file.go следующим образом:

```
var (
    Stdin = NewFile(uintptr(syscall.Stdin), "/dev/stdin")
    Stdout = NewFile(uintptr(syscall.Stdout), "/dev/stdout")
    Stderr = NewFile(uintptr(syscall.Stderr), "/dev/stderr")
)
```

Теперь рассмотрим реализацию функции `NewFile()`, которая находится в файле https://golang.org/src/os/file_plan9.go:

```
func NewFile(fd uintptr, name string) *File {
    fdi := int(fd)
    if fdi < 0 {
        return nil
    }
    f := &File{&file{fd: fdi, name: name}}
    runtime.SetFinalizer(f.file, (*file).close)
    return f
}
```

Увидев исходный файл Go с именем `file_plan9.go`, мы можем предположить, что в нем содержатся команды, специфичные для разных вариантов UNIX, следовательно, в этом файле хранится код, который не является переносимым.

На самом деле здесь показан тип структуры `file`, который встроен в тип `File` и который, судя по его имени, экспортируется. Итак, наши поиски функций, которые применяются к структуре `File` или к указателю на структуру `File` и позволяют записывать данные, мы начнем с файла https://golang.org/src/os/file_plan9.go. Поскольку функция, которую мы ищем, называется `Write()` — посмотрите еще раз на реализацию `Fprintln()`, — чтобы найти ее, нам понадобится пересмотреть все исходные файлы пакета `os`:

```
$ grep "func (f \*File) Write(" *.go
file.go:func (f *File) Write(b []byte) (n int, err error) {
```

Реализация функции `Write()` находится в файле https://golang.org/src/os/file.go и выглядит следующим образом:

```
func (f *File) Write(b []byte) (n int, err error) {
    if err := f.checkValid("write"); err != nil {
        return 0, err
    }
    n, e := f.write(b)
    if n < 0 {
        n = 0
    }
    if n != len(b) {
        err = io.ErrShortWrite
    }

    epipecheck(f, e)

    if e != nil {
        err = f.wrapErr("write", e)
    }

    return n, err
}
```

Таким образом, теперь нам нужно найти функцию `write()`. Результаты поиска строки `write` в https://golang.org/src/os/file_plan9.go сообщают нам, что в файле https://golang.org/src/os/file_plan9.go есть следующая функция:

```
func (f *File) write(b []byte) (n int, err error) {
    if len(b) == 0 {
        return 0, nil
    }
    return fixCount(syscall.Write(f.fd, b))
}
```

Таким образом, вызов функции `fmt.Println()` реализован с использованием вызова `syscall.Write()`. Этот пример наглядно демонстрирует, насколько полезен и важен пакет `syscall`.

Пакеты go/scanner, go/parser и go/token

В этом разделе вы познакомитесь с пакетами `go/scanner`, `go/parser` и `go/token`, а также `go/ast`. Вы узнаете, как в Go на низком уровне выполняется сканирование и синтаксический анализ кода, а это поможет вам понять, как работает Go. Впрочем,

вы можете пропустить этот раздел, если считаете, что низкоуровневые операции не для вас.

Синтаксический анализ языка выполняется в два этапа. Первый из них заключается в разбиении входных данных на лексемы *(лексический анализ)*, а второй — в передаче этих лексем синтаксическому анализатору, чтобы убедиться, что эти лексемы имеют смысл и находятся в правильном порядке *(семантический анализ)*, ведь, если произвольно сочетать английские слова, не всегда получится правильное предложение.

Пакет go/ast

Абстрактное синтаксическое дерево (Abstract Syntax Tree, AST) — структурированное представление исходного кода программы Go. Это дерево строится в соответствии с правилами, указанными в спецификации языка. Пакет `go/ast` используется для объявления типов данных, необходимых для представления AST в Go. Если вы хотите больше узнать о типе `ast.*`, то пакет `go/ast` будет наилучшим источником такой информации.

Пакет go/scanner

Сканер — это то, что читает программу, написанную на языке программирования, и генерирует лексемы. В данном случае это код Go, который читает программу, созданную на языке Go.

Пакет `go/scanner` используется для чтения программ Go и генерации последовательности лексем. Продемонстрируем использование пакета `go/scanner` на примере программы `goScanner.go`. Разделим ее на три части.

Первая часть `goScanner.go` выглядит так:

```
package main

import (
    "fmt"
    "go/scanner"
    "go/token"
    "io/ioutil"
    "os"
)

func main() {
    if len(os.Args) == 1 {
        fmt.Println("Not enough arguments!")
        return
    }
```

В пакете go/token определены константы, которые представляют лексемы языка программирования Go.

Во второй части goScanner.go содержится следующий код Go:

```
    for _, file := range os.Args[1:] {
        fmt.Println("Processing:", file)
        f, err := ioutil.ReadFile(file)
        if err != nil {
            fmt.Println(err)
            return
        }
        One := token.NewFileSet()
        files := one.AddFile(file, one.Base(), len(f))
```

Исходный файл, который разбивается на лексемы, хранится в переменной file, а его содержимое — в переменной f.

Последняя часть goScanner.go выглядит так:

```
        var myScanner scanner.Scanner
        myScanner.Init(files, f, nil, scanner.ScanComments)

        for {
            pos, tok, lit := myScanner.Scan()
            if tok == token.EOF {
                break
            }
            fmt.Printf("%s\t%s\t%q\n", one.Position(pos), tok, lit)
        }
    }
}
```

Цикл for используется здесь для прохода по входному файлу. Конец файла с исходным кодом представлен значением token.EOF, которое является признаком завершения цикла for. Метод scanner.Scan() возвращает текущую позицию в файле, лексему и литерал. Использование константы scanner.ScanComments в scanner.Init() указывает сканеру возвращать комментарии как лексемы COMMENT. Вместо scanner.ScanComments можно поставить 1 или 0, если вы хотите исключить все лексемы COMMENT из выходных данных.

Сборка и выполнение goScanner.go даст следующие результаты:

```
$ ./goScanner a.go
Processing: a.go
a.go:1:1    package  "package"
a.go:1:9    IDENT    "a"
a.go:1:10   ;        "\n"
a.go:3:1    import   "import"
a.go:3:8    (        ""
a.go:4:2    STRING   "\"fmt\""
```

```
a.go:4:7    ;        "\n"
a.go:5:1    )        ""
a.go:5:2    ;        "\n"
a.go:7:1    func     "func"
a.go:7:6    IDENT    "init"
a.go:7:10   (        ""
a.go:7:11   )        ""
a.go:7:13   {        ""
a.go:8:2    IDENT    "fmt"
a.go:8:5    .        ""
a.go:8:6    IDENT    "Println"
a.go:8:13   (        ""
a.go:8:14   STRING   "\"init() a\""
a.go:8:24   )        ""
a.go:8:25   ;        "\n"
a.go:9:1    }        ""
a.go:9:2    ;        "\n"
a.go:11:1   func     "func"
a.go:11:6   IDENT    "FromA"
a.go:11:11  (        ""
a.go:11:12  )        ""
a.go:11:14  {        ""
a.go:12:2   IDENT    "fmt"
a.go:12:5   .        ""
a.go:12:6   IDENT    "Println"
a.go:12:13  (        ""
a.go:12:14  STRING   "\"fromA()\""
a.go:12:23  )        ""
a.go:12:24  ;        "\n"
a.go:13:1   }        ""
a.go:13:2   ;        "\n"
```

Результат `goScanner.go` предельно прост. Обратите внимание, что `goScanner.go` может сканировать файлы любого типа, даже двоичные. Однако при сканировании двоичного файла можно получить менее читаемый результат. Как видно из результатов работы `goScanner.go`, сканер Go добавляет точки с запятой автоматически. Обратите внимание, что `IDENT` уведомляет идентификатор о том, какой тип лексем является наиболее распространенным.

Следующий подраздел посвящен процессу синтаксического анализа.

Пакет go/parser

Синтаксический анализатор получает результаты работы сканера (лексемы) и синтезирует из них структуру. Используя грамматику, которая описывает язык программирования, синтаксический анализатор проверяет, образуют ли полученные лексемы допустимую программу. Созданная структура имеет вид дерева AST.

Применение пакета go/parser, который обрабатывает выходные данные, полученные от go/token, продемонстрировано на примере программы goParser.go. Разделим ее на четыре части.

Первая часть goParser.go выглядит так:

```
package main

import (
    "fmt"
    "go/ast"
    "go/parser"
    "go/token"
    "os"
    "strings"
)

type visitor int
```

Вторая часть goParser.go содержит следующий код Go:

```
func (v visitor) Visit(n ast.Node) ast.Visitor {
    if n == nil {
        return nil
    }
    fmt.Printf("%s%T\n", strings.Repeat("\t", int(v)), n)
    return v + 1
}
```

Метод Visit() будет вызываться для каждого узла AST.

Третья часть goParser.go выглядит так:

```
func main() {
    if len(os.Args) == 1 {
        fmt.Println("Not enough arguments!")
        return
    }
```

Последняя часть goParser.go выглядит следующим образом:

```
    for _, file := range os.Args[1:] {
        fmt.Println("Processing:", file)
        one := token.NewFileSet()
        var v visitor
        f, err := parser.ParseFile(one, file, nil, parser.AllErrors)
        if err != nil {
            fmt.Println(err)
            return
        }
        ast.Walk(v, f)
    }
}
```

Функция `Walk()`, вызываемая рекурсивно, перебирает все узлы дерева AST в порядке поиска в глубину.

Сборка и выполнение `goParser.go` с целью построения AST для простого небольшого Go-модуля приводят к следующим результатам:

```
$ ./goParser a.go
Processing: a.go
*ast.File
    *ast.Ident
    *ast.GenDecl
        *ast.ImportSpec
            *ast.BasicLit
    *ast.FuncDecl
        *ast.Ident
        *ast.FuncType
            *ast.FieldList
        *ast.BlockStmt
            *ast.ExprStmt
                *ast.CallExpr
                    *ast.SelectorExpr
                        *ast.Ident
                        *ast.Ident
                    *ast.BasicLit
    *ast.FuncDecl
        *ast.Ident
        *ast.FuncType
            *ast.FieldList
        *ast.BlockStmt
            *ast.ExprStmt
                *ast.CallExpr
                    *ast.SelectorExpr
                        *ast.Ident
                        *ast.Ident
                    *ast.BasicLit
```

Результат работы `goParser.go` предельно прост. Однако он совершенно не похож на то, что дает на выходе `goScanner.go`.

Теперь, когда вы знаете, как выглядят результаты работы сканера и синтаксического анализатора Go, вы готовы к тому, чтобы рассмотреть несколько практических примеров.

Практический пример

В этом подразделе мы напишем программу Go, которая подсчитывает, сколько раз ключевое слово встречается во входных файлах. В данном случае будет подсчитано количество вхождений ключевого слова `var`. Используемая для этого утилита

называется varTimes.go. Рассмотрим ее, разделив на четыре части. Первая часть varTimes.go выглядит так:

```
package main

import (
    "fmt"
    "go/scanner"
    "go/token"
    "io/ioutil"
    "os"
)

var KEYWORD = "var"
var COUNT = 0
```

Мы можем искать любое ключевое слово Go, какое захотим, и даже можем изменять значение глобальной переменной KEYWORD во время выполнения программы, внеся соответствующие изменения в код varTimes.go.

Вторая часть varTimes.go содержит следующий код Go:

```
func main() {
    if len(os.Args) == 1 {
        fmt.Println("Not enough arguments!")
        return
    }

    for _, file := range os.Args[1:] {
        fmt.Println("Processing:", file)
        f, err := ioutil.ReadFile(file)
        if err != nil {
            fmt.Println(err)
            return
        }
        one := token.NewFileSet()
        files := one.AddFile(file, one.Base(), len(f))
```

Третья часть varTimes.go выглядит так:

```
        var myScanner scanner.Scanner
        myScanner.Init(files, f, nil, scanner.ScanComments)

        localCount := 0
        for {
            _, tok, lit := myScanner.Scan()
            if tok == token.EOF {
                break
            }
```

В данном случае позиция, в которой была обнаружена лексема, игнорируется, поскольку это не имеет значения. Однако переменная tok все равно необходима, чтобы определить конец файла.

Последняя часть varTimes.go выглядит так:

```
            if lit == KEYWORD {
                COUNT++
                localCount++
            }
        }
        fmt.Printf("Found _%s_ %d times\n", KEYWORD, localCount)
    }
    fmt.Printf("Found _%s_ %d times in total\n", KEYWORD, COUNT)
}
```

Компиляция и выполнение varTimes.go приведут к следующим результатам:

```
$ go build varTimes.go
$ ./varTimes varTimes.go variadic.go a.go
Processing: varTimes.go
Found _var_ 3 times
Processing: variadic.go
Found _var_ 0 times
Processing: a.go
Found _var_ 0 times
Found _var_ 3 times in total
```

Поиск имен переменных заданной длины

В этом подразделе рассмотрим еще один практический пример, более сложный, чем тот, что был представлен в программе varTimes.go. Вы узнаете, как найти имена переменных заданной длины, — для этого можно задать любую длину строки, какую пожелаете. Кроме того, программа будет различать *глобальные* и *локальные переменные*.

Локальная переменная определена внутри функции, тогда как глобальная — вне функции. Глобальные переменные также называются переменными пакета.

Утилита, которую мы рассмотрим, называется varSize.go. Разделим ее на четыре части. Первая часть varSize.go выглядит так:

```
package main

import (
    "fmt"
    "go/ast"
    "go/parser"
    "go/token"
    "os"
    "strconv"
)
```

```
var SIZE = 2
var GLOBAL = 0
var LOCAL = 0

type visitor struct {
    Package map[*ast.GenDecl]bool
}

func makeVisitor(f *ast.File) visitor {
    k1 := make(map[*ast.GenDecl]bool)
    for _, aa := range f.Decls {
        v, ok := aa.(*ast.GenDecl)
        if ok {
            k1[v] = true
        }
    }

    return visitor{k1}
}
```

Поскольку мы хотим считать локальные и глобальные переменные отдельно, определим две глобальные переменные с именами `GLOBAL` и `LOCAL`, чтобы хранить эти значения. Структура `visitor` поможет нам различать локальные и глобальные переменные, поэтому создадим в структуре `visitor` поле `map`. Метод `makeVisitor()` используется для инициализации активной структуры `visitor` в соответствии со значениями ее параметра, который является узлом `File`, представляющим весь файл целиком.

Вторая часть `varSize.go` содержит реализацию метода `Visit()`:

```
func (v visitor) Visit(n ast.Node) ast.Visitor {
    if n == nil {
        return nil
    }

    switch d := n.(type) {
    case *ast.AssignStmt:
        if d.Tok != token.DEFINE {
            return v
        }

        for _, name := range d.Lhs {
            v.isItLocal(name)
        }
    case *ast.RangeStmt:
        v.isItLocal(d.Key)
        v.isItLocal(d.Value)
    case *ast.FuncDecl:
        if d.Recv != nil {
            v.CheckAll(d.Recv.List)
        }
```

```
        v.CheckAll(d.Type.Params.List)
        if d.Type.Results != nil {
            v.CheckAll(d.Type.Results.List)
        }
    case *ast.GenDecl:
        if d.Tok != token.VAR {
            return v
        }
        for _, spec := range d.Specs {
            value, ok := spec.(*ast.ValueSpec)
            if ok {
                for _, name := range value.Names {
                    if name.Name == "_" {
                        continue
                    }
                    if v.Package[d] {
                        if len(name.Name) == SIZE {
                            fmt.Printf("** %s\n", name.Name)
                            GLOBAL++
                        }
                    } else {
                        if len(name.Name) == SIZE {
                            fmt.Printf("* %s\n", name.Name)
                            LOCAL++
                        }
                    }
                }
            }
        }
    }
    return v
}
```

Главная задача функции `Visit()` — определить тип узла, с которым она работает, и действовать соответственно. Это выполняется с помощью оператора `switch`.

Узел `ast.AssignStmt` — присвоения или короткие объявления переменных. Узел `ast.RangeStmt` является структурным типом для представления оператора `for` с условием `range` — еще одно место, где могут быть созданы локальные переменные.

Узел `ast.FuncDecl` является структурным типом для представления объявлений функций — переменные, определенные внутри функций, являются локальными. Наконец, узел `ast.GenDecl` — структурный тип для представления объявлений импорта, констант, типа и переменных. Однако нас интересуют только лексемы `token.VAR`.

Третья часть `varSize.go` выглядит так:

```
func (v visitor) isItLocal(n ast.Node) {
    identifier, ok := n.(*ast.Ident)
    if ok == false {
```

```
        return
    }

    if identifier.Name == "_" || identifier.Name == "" {
        return
    }

    if identifier.Obj != nil && identifier.Obj.Pos() == identifier.Pos() {
        if len(identifier.Name) == SIZE {
            fmt.Printf("* %s\n", identifier.Name)
            LOCAL++
        }
    }
}

func (v visitor) CheckAll(fs []*ast.Field) {
    for _, f := range fs {
        for _, name := range f.Names {
            v.isItLocal(name)
        }
    }
}
```

Эти две функции являются вспомогательными методами. Первый из них решает, является ли узел идентификатора локальной переменной, а второй посещает узел `ast.Field` и проверяет его содержимое на наличие локальных переменных.

Последняя часть `varSize.go` выглядит таким образом:

```
func main() {
    if len(os.Args) <= 2 {
        fmt.Println("Not enough arguments!")
        return
    }

    temp, err := strconv.Atoi(os.Args[1])
    if err != nil {
        SIZE = 2
        fmt.Println("Using default SIZE:", SIZE)
    } else {
        SIZE = temp
    }

    var v visitor
    all := token.NewFileSet()
    for _, file := range os.Args[2:] {
        fmt.Println("Processing:", file)
        f, err := parser.ParseFile(all, file, nil, parser.AllErrors)
        if err != nil {
            fmt.Println(err)
```

```
            continue
        }

        v = makeVisitor(f)
        ast.Walk(v, f)
    }
    fmt.Printf("Local: %d, Global:%d with a length of %d.\n", LOCAL, GLOBAL, SIZE)
}
```

Программа генерирует дерево AST на основе полученных входных данных и затем обрабатывает это дерево, извлекая оттуда нужную информацию. Кроме метода `Visit()`, который является частью интерфейса, остальное выполняется в функции `main()` с помощью метода `ast.Walk()`, который автоматически обходит все узлы дерева AST каждого обрабатываемого файла.

Сборка и выполнение `varSize.go` приведут к результатам следующего вида:

```
$ go build varSize.go
$ ./varSize
Not enough arguments!
$ ./varSize 2 varSize.go variadic.go
Processing: varSize.go
* k1
* aa
* ok
* ok
* ok
* fs
Processing: variadic.go
Local: 6, Global:0 with a length of 2.
$ ./varSize 3 varSize.go variadic.go
Processing: varSize.go
* err
* all
* err
Processing: variadic.go
* sum
* sum
Local: 5, Global:0 with a length of 3.
$ ./varSize 7 varSize.go variadic.go
Processing: varSize.go
Processing: variadic.go
* message
Local: 1, Global:0 with a length of 7.
```

Чтобы получить более упорядоченный результат, можно удалить из `varSize.go` многочисленные вызовы `fmt.Println()`.

Go позволяет создавать уникальные решения: если уметь анализировать программы, при желании можно даже написать свой анализатор для собственного языка программирования! Если вы действительно хотите разобраться в том, как

работают синтаксические анализаторы, очень советую заглянуть на страницу документации пакета `go/ast`, которую вы найдете по адресу https://golang.org/pkg/go/ast/, а также почитать его исходный код, расположенный по адресу https://github.com/golang/go/tree/master/src/go/ast.

Шаблоны для текста и HTML

Тема этого раздела, вероятно, вас приятно удивит: оба представленных здесь пакета обеспечивают потрясающую гибкость. Я уверен, вы найдете множество творческих способов ее применения. *Шаблоны* — это то, что обычно позволяет отделить форматирование от выводимых данных. Обратите внимание, что шаблон Go может быть как файлом, так и строкой. Общая идея такова: следует использовать встроенные строки для небольших шаблонов и внешние файлы для шаблонов крупнее.

Нельзя импортировать в одну и ту же программу Go одновременно и text/template, и html/template, поскольку данные пакеты имеют одно и то же имя (template). Если это крайне необходимо, то следует присвоить одному из них псевдоним. Как это сделать, показано в коде useStrings.go из главы 4.

Текст обычно выводится прямо на экран, тогда как HTML-страницы можно увидеть с помощью браузера. Но текст обычно предпочтительнее, чем HTML, если вы планируете обрабатывать его с помощью утилит Go, а также других утилит командной строки UNIX. Поэтому лучше использовать пакет `text/template`, а не `html/template`.

Заметьте, что пакеты `text/template` и `html/template` являются хорошими примерами того, насколько сложным может быть пакет Go. Как вы вскоре узнаете, каждый из них поддерживает собственный язык программирования — благодаря хорошему программному обеспечению сложное выглядит простым и элегантным.

Вывод простого текста

Если вам нужно реализовать вывод простого текста, то для этого отлично подойдет пакет `text/template`. Мы продемонстрируем использование пакета `text/template` на примере программы `textT.go`, которую разделим на пять частей.

Поскольку шаблоны обычно хранятся во внешних файлах, в этом примере используем файл шаблона `text.gotext`. Рассмотрим его, также разделив на три части. Обычно данные считываются из текстовых файлов или из Интернета. Однако в этом примере мы для простоты жестко закодируем данные в программе `textT.go`, представив их в виде среза.

Сначала рассмотрим код `textT.go`. Первая часть кода `textT.go` выглядит так:

```
package main

import (
    "fmt"
    "os"
    "text/template"
)
```

Второй фрагмент кода `textT.go`:

```
type Entry struct {
    Number int
    Square int
}
```

Вам нужно определить новый тип для хранения данных, если только вы не имеете дело с очень простыми данными.

В третьей части `textT.go` содержится следующий код Go:

```
func main() {
    arguments := os.Args
    if len(arguments) != 2 {
        fmt.Println("Need the template file!")
        return
    }

    tFile := arguments[1]
    DATA := [][]int{{-1, 1}, {-2, 4}, {-3, 9}, {-4, 16}}
```

Исходная версия данных хранится в переменной `DATA`, которая представляет собой двумерный срез.

В четвертой части `textT.go` содержится следующий код Go:

```
    var Entries []Entry

    for _, i := range DATA {
        if len(i) == 2 {
            temp := Entry{Number: i[0], Square: i[1]}
            Entries = append(Entries, temp)
        }
    }
```

Этот код создает *срез структур* на основе переменной `DATA`.

Последний фрагмент кода `textT.go` выглядит так:

```
    t := template.Must(template.ParseGlob(tFile))
    t.Execute(os.Stdout, Entries)
}
```

Для выполнения всех необходимых инициализаций используется функция `template.Must()`. Она возвращает данные типа `Template` — эта структура представля-

ет собой структуру, которая содержит в себе шаблон, обработанный синтаксическим анализатором. Функция `template.ParseGlob()` читает внешний файл шаблона. Заметьте, что я предпочитаю использовать для файлов шаблонов расширение `gohtml`, но вы можете выбрать любое другое расширение — только будьте последовательны.

Наконец, функция `template.Execute()` выполняет всю остальную работу, которая включает в себя обработку данных и вывод результатов в нужный файл — в данном случае это `os.Stdout`.

Теперь рассмотрим код в файле шаблона. Первая часть файла текстового шаблона выглядит так:

```
Calculating the squares of some integers
```

Обратите внимание, что пустые строки в файле текстового шаблона очень важны: они будут отображаться как пустые строки при итоговом выводе данных.

Вторая часть шаблона выглядит так:

```
{{ range . }} The square of {{ printf "%d" .Number}} is {{ printf "%d" .Square}}
```

Здесь много интересного. Ключевое слово `range` позволяет перебрать строки входных данных, которые представлены в виде среза структур. Обычный текст выводится как текст, а переменные и динамический текст должны заключаться в двойные фигурные скобки: `{{ … }}`. Поля структуры доступны под именами `.Number` и `.Square`. Обратите внимание на знак точки перед именем поля типа данных `Entry`. Наконец, для форматирования окончательных результатов используется команда `printf`.

Третья часть файла `text.gotext` выглядит так:

```
{{ end }}
```

Выполнение команды `{{ range }}` заканчивается командой `{{ end }}`. Если случайно поставить `{{ end }}` не в том месте, это изменит весь вывод данных. Подчеркну: пустые строки в файлах текстовых шаблонов имеют большое значение и будут отражаться при итоговом выводе.

Выполнение `textT.go` приведет к результатам следующего вида:

```
$ go run textT.go text.gotext
Calculating the squares of some integers
 The square of -1 is 1
 The square of -2 is 4
 The square of -3 is 9
 The square of -4 is 16
```

Вывод текста в формате HTML

В этом разделе продемонстрировано использование пакета `html/template` на примере программы `htmlT.go`. Мы разделим код этой программы на шесть частей.

Принцип пакета `html/template` такой же, как и у `text/template`. Главное различие между ними состоит в том, что `html/template` генерирует текст в формате HTML, защищенный от атак типа «внедрение кода».

Текст в формате HTML возможно создать и с помощью пакета text/template, в конце концов, HTML — это лишь текст. Однако если вы хотите создать именно HTML, то лучше вместо этого пакета использовать html/template.

Для простоты мы будем считывать представленные ниже данные из базы данных SQLite. Однако вы можете использовать любую другую базу данных по своему выбору, если у вас есть или если вы сможете написать соответствующие драйверы для Go. Для упрощения будем заполнять таблицу базы данных в нашем примере, перед тем как выполнить чтение из нее.

Первая часть кода `htmlT.go` выглядит так:

```
package main

import (
    "database/sql"
    "fmt"
    _ "github.com/mattn/go-sqlite3"
    "html/template"
    "net/http"
    "os"
)

type Entry struct {
    Number int
    Double int
    Square int
}

var DATA []Entry
var tFile string
```

Как видим, в блоке `import` появился новый пакет с именем `net/http` — этот пакет используется для создания в Go серверов и клиентов HTTP. Подробнее о сетевом программировании на Go и об использовании стандартных Go-пакетов `net` и `net/http` вы узнаете из глав 12 и 13.

Кроме пакета `net/http`, мы видим здесь определение типа данных `Entry`, в котором представлены записи, прочитанные из таблицы SQLite3, а также две глобальные переменные с именами `DATA` и `tFile`, в которых хранятся данные, предназначенные для передачи в файл шаблона, и имя файла шаблона соответственно.

Наконец, мы видим здесь использование пакета https://github.com/mattn/go-sqlite3 для связи с базой данных SQLite3 посредством интерфейса `database/sql`.

Вторая часть `htmlT.go` выглядит так:

```
func myHandler(w http.ResponseWriter, r *http.Request) {
    fmt.Printf("Host: %s Path: %s\n", r.Host, r.URL.Path)
    myT := template.Must(template.ParseGlob(tFile))
    myT.ExecuteTemplate(w, tFile, DATA)
}
```

Функция `myHandler()` феноменально проста и эффективна, особенно если учесть ее размер! Функция `template.ExecuteTemplate()` выполняет всю работу. Ее первый параметр — это переменная, в которой хранится соединение с HTTP-клиентом, второй параметр — файл шаблона, который будет использоваться для форматирования данных, а третий параметр — срез структур с данными.

Третий фрагмент `htmlT.go` содержит следующий код Go:

```
func main() {
    arguments := os.Args
    if len(arguments) != 3 {
        fmt.Println("Need Database File + Template File!")
        return
    }

    database := arguments[1]
    tFile = arguments[2]
```

В четвертой части кода `htmlT.go` мы начинаем работать с базой данных:

```
    db, err := sql.Open("sqlite3", database)
    if err != nil {
        fmt.Println(nil)
        return
    }

    fmt.Println("Emptying database table.")
    _, err = db.Exec("DELETE FROM data")
    if err != nil {
        fmt.Println(nil)
        return
    }

    fmt.Println("Populating", database)
    stmt, _ := db.Prepare("INSERT INTO data(number, double, square) values(?,?,?)")
    for i := 20; i < 50; i++ {
        _, _ = stmt.Exec(i, 2*i, i*i)
    }
```

Функция `sql.Open()` открывает соединение с базой данных. С помощью функции `db.Exec()` мы можем выполнять команды базы данных, не ожидая от нее об-

ратной связи. Наконец, db.Prepare() позволяет многократно выполнять команду базы данных, изменяя только ее параметры и вызывая Exec().

В пятой части htmlT.go содержится следующий код Go:

```
    rows, err := db.Query("SELECT * FROM data")
    if err != nil {
        fmt.Println(nil)
        return
    }

    var n int
    var d int
    var s int
    for rows.Next() {
        err = rows.Scan(&n, &d, &s)
        temp := Entry{Number: n, Double: d, Square: s}
        DATA = append(DATA, temp)
    }
```

В этой части программы мы считываем данные из заданной таблицы, используя функцию db.Query(), многократные вызовы Next() и Scan(). Считывая данные, мы помещаем их в срез структур и на этом заканчиваем работу с базой данных.

Последняя часть программы посвящена настройке веб-сервера и содержит следующий код Go:

```
    http.HandleFunc("/", myHandler)
    err = http.ListenAndServe(":8080", nil)
    if err != nil {
        fmt.Println(err)
        return
    }
}
```

Здесь функция http.HandleFunc() сообщает веб-серверу, какие URL-адреса и какой функцией-обработчиком (myHandler()) будут поддерживаться. Текущий обработчик поддерживает адрес /URL, что в языке Go соответствует всем URL. Это избавляет нас от необходимости создавать какие-либо дополнительные статические или динамические страницы.

Код программы htmlT.go делится на две виртуальные части. Первая из них получает данные из базы и помещает их в срез структур, а вторая, похожая на textT.go, предназначена для отображения данных в браузере.

У SQLite есть два больших преимущества: вам не приходится запускать серверный процесс для сервера баз данных, и базы данных SQLite хранятся в автономных файлах. Последнее означает, что вся база данных SQLite может храниться в одном файле.

Обратите внимание: чтобы сократить код Go и обеспечить возможность многократного запуска программы `htmlT.go`, придется вручную создать таблицу базы данных и базу данных SQLite3. Для этого нужно лишь выполнить следующие команды:

```
$ sqlite3 htmlT.db
SQLite version 3.19.3 2017-06-27 16:48:08
Enter ".help" for usage hints.
sqlite> CREATE TABLE data (
    ...> number INTEGER PRIMARY KEY,
    ...> double INTEGER,
    ...> square INTEGER );
sqlite> ^D
$ ls -l htmlT.db
-rw-r--r-- 1 mtsouk staff 8192 Dec 26 22:46 htmlT.db
```

Первая команда выполняется из оболочки UNIX, она создает файл базы данных. Вторая команда выполняется из оболочки SQLite3 и создает таблицу базы данных с именем `data`. У этой таблицы три поля с именами `number`, `double` и `square`.

Кроме того, нам понадобится внешний файл шаблона, который называется `html.gohtml`. Мы будем использовать этот файл для генерации выходных данных программы.

Первая часть `html.gohtml` выглядит так:

```
<!doctype html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Doing Maths in Go!</title>
    <style>
        html {
            font-size: 14px;
        }
        table, th, td {
            border: 2px solid blue;
        }
    </style>
</head>
<body>
```

В основе HTML-кода, который получит браузер, лежит содержимое файла `html.gohtml`. Это значит, что вам нужно сформировать правильную итоговую HTML-страницу, для чего и нужен показанный выше код HTML со встроенными в него стилями CSS для форматирования создаваемой таблицы HTML.

Во второй части `html.gohtml` содержится следующий код:

```
<table>
    <thead>
```

```
        <tr>
            <th>Number</th>
            <th>Double</th>
            <th>Square</th>
        </tr>
    </thead>
    <tbody>
        {{ range . }}
        <tr>
            <td> {{ .Number }} </td>
            <td> {{ .Double }} </td>
            <td> {{ .Square }} </td>
        </tr>
        {{ end }}
    </tbody>
</table>
```

Как видно из этого кода, нам все равно приходится использовать `{{ range }}` и `{{ end }}`, чтобы перебрать все элементы среза структур, который был передан в `template.ExecuteTemplate()`. Однако на этот раз в файле шаблона `html.gohtml` содержится много HTML-кода для представления данных из среза структур в заданном формате.

Последняя часть файла шаблона HTML выглядит так:

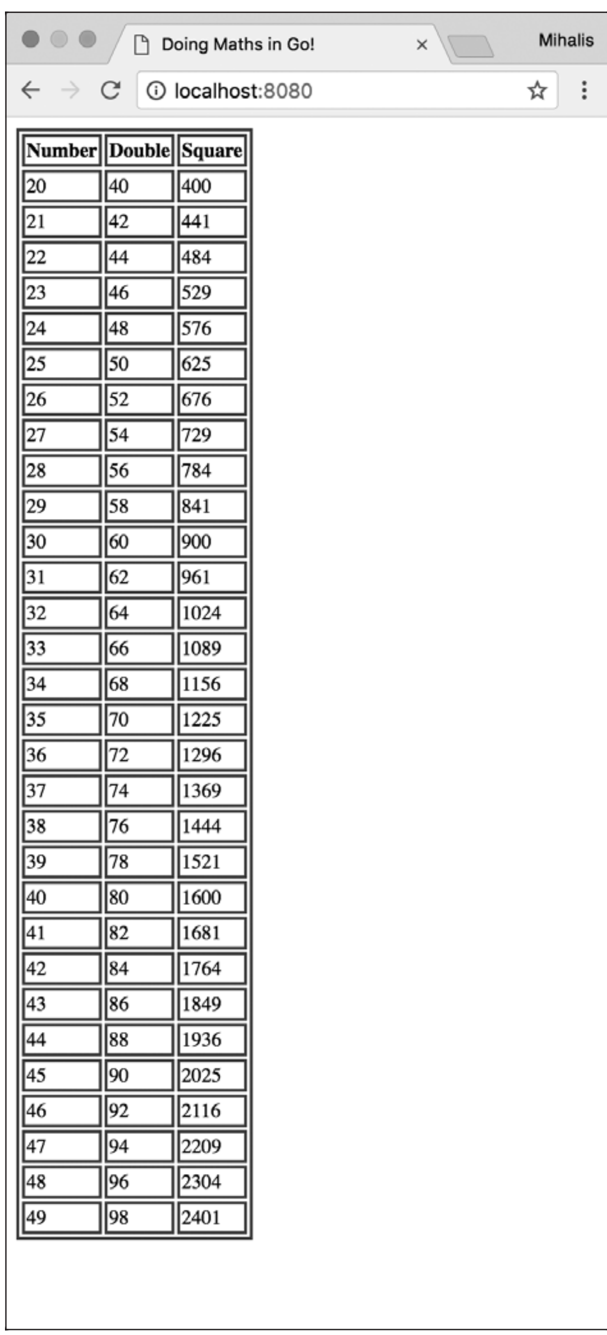
```
</body>
</html>
```

Эта часть файла `html.gohtml` предназначена главным образом для правильного завершения сформированного HTML-кода в соответствии со стандартами HTML. Прежде чем скомпилировать и выполнить `htmlT.go`, нам необходимо загрузить пакет, который обеспечит взаимодействие языка программирования Go и базы данных SQLite3. Для этого нужно выполнить следующую команду:

```
$ go get github.com/mattn/go-sqlite3
```

Как вы уже знаете, на компьютере с macOS исходный код загруженного пакета находится в каталоге `~/go/src`, а его скомпилированная версия — в каталоге `~/go/pkg/darwin_amd64`. Если у вас другая ОС, просмотрите содержимое каталога `~/go/pkg`, у которого может быть иная структура. Обратите внимание, что символом `~` обозначается домашний каталог текущего пользователя.

Следует учитывать, что есть и дополнительные пакеты Go, которые обеспечивают обмен данными с базой SQLite3. Однако тот пакет, который использован здесь, является единственным в настоящее время поддерживающим интерфейс `database/sql`. Выполнение `htmlT.go` приведет к тому, что в окне браузера появится таблица, представленная на рис. 6.1.



Number	Double	Square
20	40	400
21	42	441
22	44	484
23	46	529
24	48	576
25	50	625
26	52	676
27	54	729
28	56	784
29	58	841
30	60	900
31	62	961
32	64	1024
33	66	1089
34	68	1156
35	70	1225
36	72	1296
37	74	1369
38	76	1444
39	78	1521
40	80	1600
41	82	1681
42	84	1764
43	86	1849
44	88	1936
45	90	2025
46	92	2116
47	94	2209
48	96	2304
49	98	2401

Рис. 6.1. Результат работы программы htmlT.go

Программа `htmlT.go` также выведет в оболочке UNIX результаты следующего вида — в основном это отладочная информация:

```
$ go run htmlT.go htmlT.db  html.gohtml
Emptying database table.
Populating htmlT.db
Host: localhost:8080 Path:  /
Host: localhost:8080 Path:  /favicon.ico
Host: localhost:8080 Path:  /123
```

Для того чтобы просмотреть сформированную программой HTML-страницу из оболочки UNIX, можно воспользоваться утилитой `wget(1)`:

```
$ wget -qO- http://localhost:8080
<!doctype html>
<html lang="en">
    <head>
        <meta charset="UTF-8">
        <title>Doing Maths in Go!</title>
        <style>
            html {
                font-size: 14px;
            }
            table, th, td {
                border: 2px solid blue;
            }
        </style>
    </head>
    <body>
    <table>
        <thead>
            <tr>
```

И `text/template`, и `html/template` являются мощными пакетами, которые способны сэкономить вам много времени. Поэтому я советую использовать их в тех случаях, когда они соответствуют требованиям ваших приложений.

Дополнительные ресурсы

Следующие ресурсы будут для вас очень полезны:

- посетите страницу документации стандартного Go-пакета `syscall` по адресу https://golang.org/pkg/syscall/. Это одна из самых больших страниц документации Go, какие мне когда-либо встречались;
- посетите страницу документации пакета `text/template` по адресу https://golang.org/pkg/text/template/;
- также зайдите по ссылке на страницу https://golang.org/pkg/html/template/, где вы найдете документацию к пакету `html/template`;

- подробную информацию о пакете `go/token` вы узнаете по адресу https://golang.org/pkg/go/token/;
- чтобы больше узнать о пакете `go/parser`, зайдите по ссылке на страницу https://golang.org/pkg/go/parser/;
- чтобы больше узнать о пакете `go/scanner`, зайдите на страницу https://golang.org/pkg/go/scanner/;
- подробнее о пакете `go/ast` читайте по адресу https://golang.org/pkg/go/ast/;
- посетите домашнюю страницу SQLite3 по адресу https://www.sqlite.org/;
- посмотрите видео Мэта Райера (Mat Ryer) о том, как писать красивые пакеты на Go: https://www.youtube.com/watch?v=cAWlv2SeQus;
- если вы хотите узнать, что такое *Plan 9*, зайдите сюда: https://plan9.io/plan9/;
- не пожалейте времени на то, чтобы познакомиться с инструментом командной строки `find(1)`, посетив страницу его документации (`man 1 find`).

Упражнения

- Найдите дополнительную информацию о том, как в действительности реализована функция `fmt.Printf()`.
- Сможете ли вы написать функцию, которая сортирует три значения типа `int`? Попробуйте написать две версии такой функции: с именованными возвращаемыми значениями и без таких значений. Какая из них, по-вашему, лучше?
- Сможете ли вы изменить код Go программы `htmlT.go` так, чтобы использовать в ней пакет `text/template` вместо `html/template`?
- Сможете ли вы изменить код Go программы `htmlT.go` так, чтобы использовать для связи с базой данных SQLite3 пакет https://github.com/feyeleanor/gosqlite3 или https://github.com/phf/go-sqlite3?
- Создайте свой Go-модуль и разработайте для него три старшие версии.
- Напишите программу Go, подобную `htmlT.go`, которая бы получала данные из базы данных MySQL. Отметьте, какие изменения вы внесли в код.

Резюме

В этой главе вы познакомились с такими тремя основными темами, как функции Go, Go-пакеты и Go-модули. Главным преимуществом Go-модулей является то, что в них записаны точные требования к зависимостям, что позволяет легко создавать воспроизводимые сборки.

Здесь также приведены исчерпывающие советы по разработке хороших Go-пакетов. Затем вы изучили пакеты `text/template` и `html/template`, которые позволяют выводить на экран простой текст и текст в формате HTML на основе шаблонов, а также пакеты `go/token`, `go/parser` и `go/scanner`. Наконец, вы узнали о стандартном Go-пакете `syscall` и его дополнительных возможностях.

В следующей главе мы обсудим два важных свойства Go: интерфейсы и рефлексию. Кроме того, речь пойдет об объектно-ориентированном программировании в методах Go, методах отладки и типах Go. Это углубленные темы, и поначалу они могут показаться вам сложными. Однако, узнав о них больше, вы, несомненно, улучшите свои навыки программирования на Go.

Наконец, в следующей главе содержится краткое описание основных возможностей утилиты `git`, которая используется для создания Go-модулей.

7 Рефлексия и интерфейсы на все случаи жизни

Из предыдущей главы вы узнали, как разрабатывать пакеты, модули и функции в Go, а также как работать с текстовыми и HTML-шаблонами с помощью пакетов `text/template` и `html/template`. Кроме того, вы научились использовать пакет `syscall`.

В этой главе рассмотрены три очень интересные, удобные и более глубокие концепции Go: *рефлексия*, *интерфейсы* и методы *типов*. Интерфейсы в Go используются повсеместно, в отличие от рефлексии, так как обычно в этом нет необходимости. Вы также познакомитесь с операциями утверждения типа, отладчиком Delve и *объектно-ориентированным программированием* на Go. Наконец, в этой главе описаны основы работы с *git* и *GitHub*.

Таким образом, в данной главе рассмотрены следующие темы:

- методы типов;
- интерфейсы Go;
- операции утверждения типа;
- разработка и использование собственных интерфейсов;
- GitHub и git;
- основы работы с отладчиком Delve;
- объектно-ориентированное программирование на Go;
- рефлексия и стандартный Go-пакет `reflect`;
- рефлексия и библиотека `reflectwalk`.

Методы типов

Метод типа (type method) в Go — это функция со специальным аргументом-приемником. Такой метод объявляется как обычная функция с дополнительным параметром, который ставится перед именем функции. Данный параметр связывает функцию с типом этого дополнительного параметра. Именно этот параметр называется *приемником метода*.

Следующий код Go представляет собой реализацию функции `Close()`, которая находится в файле https://golang.org/src/os/file_plan9.go:

```
func (f *File) Close() error {
    if err := f.checkValid("close"); err != nil {
        return err
    }
    return f.file.close()
}
```

Функция `Close()` является методом типа `File`, поскольку перед ее именем, после ключевого слова `func`, стоит параметр `(f *File)`. Параметр `f` называется приемником метода. В терминологии объектно-ориентированного программирования этот процесс можно описать как передачу сообщения *объекту*. В Go приемнику метода присваивается обычное имя переменной, без специального ключевого слова, такого как `this` или `self`.

Теперь позвольте представить вам полный пример — код Go из файла `method.go`, который мы рассмотрим, разделив на четыре части.

Первая часть `method.go` содержит следующий код Go:

```
package main

import (
    "fmt"
)

type twoInts struct {
    X int64
    Y int64
}
```

В этом коде Go мы видим определение новой структуры `twoInts`, имеющей два поля.

Второй фрагмент кода `methods.go` выглядит так:

```
func regularFunction(a, b twoInts) twoInts {
    temp := twoInts{X: a.X + b.X, Y: a.Y + b.Y}
    return temp
}
```

В этой части мы определяем новую функцию с именем `normalFunction()`. Она принимает два параметра типа `twoInts` и возвращает одно значение типа `twoInts`.

В третьей части программы содержится следующий код Go:

```
func (a twoInts) method(b twoInts) twoInts {
    temp := twoInts{X: a.X + b.X, Y: a.Y + b.Y}
    return temp
}
```

Функция `method()` эквивалентна функции `normalFunction()`, определенной в предыдущей части `method.go`. Однако `method()` — это метод типа, и вскоре вы узнаете, как его вызывать.

Самое интересное, что у method() точно такая же реализация, как и у regularFunction()!

Последний фрагмент кода `method.go` выглядит так:

```
func main() {
    i := twoInts{X: 1, Y: 2}
    j := twoInts{X: -5, Y: -2}
    fmt.Println(regularFunction(i, j))
    fmt.Println(i.method(j))
}
```

Как видите, способ вызова метода типа (`i.method(j)`) отличается от способа вызова обычной функции (`regularFunction(i, j)`).

Выполнение `method.go` приведет к следующим результатам:

```
$ go run methods.go
{-4 0}
{-4 0}
```

Обратите внимание, что методы типа связаны с интерфейсами, о которых пойдет речь в следующем разделе. Скоро вы больше узнаете о методах типа.

Интерфейсы в Go

Интерфейсный тип в Go определяет поведение других типов путем предоставления списка методов, которые необходимо реализовать. Чтобы тип соответствовал интерфейсу, в нем необходимо реализовать все методы, предусмотренные этим интерфейсом, — обычно этих методов не очень много.

Проще говоря, интерфейс — это *абстрактный тип*, определяющий множество функций, которые необходимо реализовать для типа, чтобы его можно было считать экземпляром интерфейса. Если это сделано, можно сказать, что тип удовлетворяет данному интерфейсу. Таким образом, интерфейс — это две составляющие: набор методов и тип. Интерфейс используется для определения поведения других типов.

Главное преимущество, которое мы получаем благодаря наличию и использованию интерфейса, состоит в том, что мы можем передавать переменную типа, реализующего этот интерфейс, любой функции, которая ожидает параметр данного интерфейса. Без такой удивительной возможности интерфейсы были бы лишь формальностью без сколь-нибудь значительной практической пользы.

Обратите внимание: если определение и реализация интерфейса находятся в одном и том же Go-пакете, то, возможно, стоит пересмотреть концепцию. Дело не в том, что это технически некорректно, а в том, что это, скорее всего, неверно с логической точки зрения.

Из самых распространенных интерфейсов Go отметим два: `io.Reader` и `io.Writer`. Они используются в операциях ввода и вывода файлов. Точнее, `io.Reader` применяется для чтения из файла, тогда как `io.Writer` — для записи в файл любого типа.

Определение `io.Reader` содержится в файле https://golang.org/src/io/io.go и выглядит так:

```
type Reader interface {
    Read(p []byte) (n int, err error)
}
```

Следовательно, чтобы тип удовлетворял интерфейсу `io.Reader`, необходимо реализовать метод `Read()` согласно определению интерфейса.

Аналогично определение `io.Writer` находится в том же файле https://golang.org/src/io/io.go и выглядит так:

```
type Writer interface {
    Write(p []byte) (n int, err error)
}
```

Чтобы удовлетворить требованиям интерфейса `io.Writer`, достаточно реализовать всего один метод с именем `Write()`.

Оба интерфейса, и `io.Reader`, и `io.Writer`, требуют реализации лишь одного метода. Тем не менее оба эти интерфейса очень мощные — очевидно, их сила заключается в их простоте. Вообще говоря, большинство интерфейсов довольно просты.

В следующих подразделах показано, как определить интерфейс самостоятельно и как использовать его в других Go-пакетах. Обратите внимание, что интерфейс не обязательно должен быть необычным или впечатляющим — главное, чтобы он выполнял свое предназначение.

Проще говоря, интерфейсы следует использовать в тех случаях, когда необходимо гарантировать выполнение определенных условий и обеспечить определенное поведение, ожидаемое от элемента Go.

Операции утверждения типа

Операции утверждения типа — это записи вида `x.(T)`, где `x` — интерфейсный тип, а `T` — тип. Кроме того, реальное значение, хранящееся в `x`, должно иметь тип `T`, и `T` должен соответствовать интерфейсному типу `x`. Прочитав следующие несколько

абзацев, а также рассмотрев пример кода, вы лучше поймете это немного странное определение операции утверждения типа.

Операции утверждения типа помогают выполнить **две задачи**. Во-первых, они проверяют, соответствует ли значение интерфейса определенному типу. При таком использовании операция утверждения типа возвращает два значения: основное и логическое значение. Скорее всего, вам нужно именно основное значение, однако логическое значение указывает на то, была ли операция утверждения типа успешной.

Во-вторых, операции утверждения типа позволяют использовать конкретное значение, хранящееся в интерфейсе, или присвоить его новой переменной. Например, если в интерфейсе есть переменная типа `int`, то можно получить это значение, используя операцию утверждения типа.

Но если операция утверждения типа завершилась неудачно и этот сбой не был обработан, то программа запаникует. Рассмотрим код Go программы `assertion.go`, который мы разделим на две части. Первая часть выглядит так:

```
package main

import (
    "fmt"
)

func main() {
    var myInt interface{} = 123

    k, ok := myInt.(int)
    if ok {
        fmt.Println("Success:", k)
    }

    v, ok := myInt.(float64)
    if ok {
        fmt.Println(v)
    } else {
        fmt.Println("Failed without panicking!")
    }
```

Сначала мы объявили переменную `myInt`, которая имеет динамический тип `int` и значение `123`. Затем мы дважды использовали операцию утверждения типа для проверки интерфейса переменной `myInt`: один раз для типа `int` и еще один раз для типа `float64`.

Поскольку переменная `myInt` не содержит значения типа `float64`, операция утверждения типа `myInt.(float64)` завершится сбоем, если не будет обработана

должным образом. К счастью, в данном случае правильное использование переменной ok не даст программе уйти в панику.

Вторая часть содержит следующий код Go:

```
    i := myInt.(int)
    fmt.Println("No checking:", i)

    j := myInt.(bool)
    fmt.Println(j)
}
```

Здесь присутствуют две операции утверждения типа. Первая из них успешна, поэтому проблем с ней не возникает. Расскажу подробнее об этой операции утверждения типа. Здесь типом переменной i будет int, а ее значением — число 123, которое хранится в myInt. Таким образом, поскольку тип int соответствует интерфейсу myInt, — в данном случае из-за того, что интерфейс myInt не требует реализации никаких функций, — значение myInt.(int) имеет тип int.

Однако вторая операция утверждения типа — myInt.(bool) — вызовет панику, поскольку основное значение myInt не является логическим (bool).

Поэтому выполнение assertion.go приведет к следующим результатам:

```
$ go run assertion.go
Success: 123
Failed without panicking!
No cheking: 123
panic: interface conversion: interface {} is int, not bool
goroutine 1 [running]:
main.main()
    /Users/mtsouk/Desktop/mGo2nd/Mastering-Go-Second-
Edition/ch07/assertion.go:25 +0x1c1
exit status 2
```

Go четко указывает на причину паники: interface {} is int, not bool.

Вообще, где интерфейсы — там и операции утверждения типа. Еще больше операций утверждений типа встретится в программе useInterface.go, которую мы вскоре рассмотрим.

Как писать свои интерфейсы

В этом разделе вы узнаете, как разрабатывать собственные интерфейсы. Это сравнительно просто, главное — знать, что именно вы хотите создать.

Методика создания интерфейсов продемонстрирована на примере кода Go из файла myInterface.go, который мы вскоре рассмотрим. Интерфейс, который мы создадим, поможет нам работать с геометрическими фигурами на плоскости.

В myInterface.go содержится следующий код:

```
package myInterface

type Shape interface {
    Area() float64
    Perimeter() float64
}
```

Определение интерфейса shape действительно очень простое. Оно требует реализации всего двух функций — Area() и Perimeter(), которые возвращают значение типа float64. Первая из этих функций будет использоваться для расчета площади плоской фигуры, а вторая — для расчета периметра.

После этого нужно установить пакет myInterface.go и сделать его доступным для текущего пользователя. Как вы уже знаете, процесс установки включает в себя выполнение следующих команд UNIX:

```
$ mkdir ~/go/src/myInterface
$ cp myInterface.go ~/go/src/myInterface
$ go install myInterface
```

Использование интерфейса Go

В этом подразделе вы научитесь использовать интерфейс, созданный в myInterface.go. Для этого мы рассмотрим Go-программу useInterface.go, разделив ее на пять частей.

В первой части useInterface.go содержится следующий код Go:

```
package main

import (
    "fmt"
    "math"
    "myInterface"
)

type square struct {
    X float64
}

type circle struct {
    R float64
}
```

Поскольку нужный нам интерфейс определен в отдельном пакете, неудивительно, что мы импортируем этот пакет myInterface.

Вторая часть `useInterface.go` содержит следующий код:

```
func (s square) Area() float64 {
    return s.X * s.X
}

func (s square) Perimeter() float64 {
    return 4 * s.X
}
```

В этой части программы реализован интерфейс `shape` для типа `square`.

В третьей части программы содержится следующий код Go:

```
func (s circle) Area() float64 {
    return s.R * s.R * math.Pi
}

func (s circle) Perimeter() float64 {
    return 2 * s.R * math.Pi
}
```

Здесь реализован интерфейс `shape` для типа `circle`.

В четвертой части `useInterface.go` содержится такой код Go:

```
func Calculate(x myInterface.Shape) {
    _, ok := x.(circle)
    if ok {
        fmt.Println("Is a circle!")
    }

    v, ok := x.(square)
    if ok {
        fmt.Println("Is a square:", v)
    }

    fmt.Println(x.Area())
    fmt.Println(x.Perimeter())
}
```

Как видим, в этом коде реализована функция, для которой требуется один параметр типа `shape`, — `myInterface.Shape`. Эта магия станет более очевидной, когда вы поймете, что здесь подойдет любой параметр типа `shape`, то есть любой параметр, тип которого реализует интерфейс `shape`.

Код, представленный в начале функции, позволяет различать типы данных, которые реализуют желаемый интерфейс. Во втором блоке показано, как получить значения, хранящиеся в параметре `square`, — этот способ можно использовать для любого типа, который реализует интерфейс `myInterface.Shape`.

Последний фрагмент кода содержит следующий код:

```
func main() {
    x := square{X: 10}
    fmt.Println("Perimeter:", x.Perimeter())
    Calculate(x)
    y := circle{R: 5}
    Calculate(y)
}
```

В этой части программы показано, что параметрами реализованной ранее функции `Calculate()` могут служить как переменные типа `circle`, так и переменные типа `square`.

Если выполнить `useInterface.go`, то получим следующие результаты:

```
$ go run useInterface.go
Perimeter: 40
Is a square: {10}
100
40
Is a circle!
78.53981633974483
31.41592653589793
```

Использование переключателей для интерфейсов и типов данных

В этом подразделе показано, как использовать оператор `switch` для переключения в зависимости от типов данных. Мы рассмотрим это на примере кода Go из файла `switch.go`, который разделим на четыре части.

Код программы `switch.go` в некоторой степени повторяет `useInterface.go`, однако в нем добавлен еще один тип, `rectangle`, и здесь нет необходимости в реализации методов какого-либо интерфейса.

Первая часть программы выглядит так:

```
package main

import (
    "fmt"
)
```

Поскольку код `switch.go` не работает с интерфейсом, определенным в `myInterface.go`, то нет необходимости импортировать пакет `myInterface`.

Во второй части определены три новых типа данных, которые будут использоваться в программе:

```
type square struct {
    X float64
}
```

```
type circle struct {
    R float64
}

type rectangle struct {
    X float64
    Y float64
}
```

Все эти три типа данных довольно просты.

Третий фрагмент `switch.go` содержит следующий код на Go:

```
func tellInterface(x interface{}) {
    switch v := x.(type) {
    case square:
        fmt.Println("This is a square!")
    case circle:
        fmt.Printf("%v is a circle!\n", v)
    case rectangle:
        fmt.Println("This is a rectangle!")
    default:
        fmt.Printf("Unknown type %T!\n", v)
    }
}
```

Здесь вы видите реализацию функции `tellInterface()` с одним параметром `x` и типом `interface{}`.

Этот прием поможет нам различать разные типы передаваемых функции параметров `x`. Вся магия происходит благодаря оператору `x.(type)`, который возвращает тип элемента `x`. Чтобы получить значение типа, в `fmt.Printf()` используется операция `%v`.

В последней части `switch.go` содержится реализация функции `main()`:

```
func main() {
    x := circle{R: 10}
    tellInterface(x)
    y := rectangle{X: 4, Y: 1}
    tellInterface(y)
    z := square{X: 4}
    tellInterface(z)
    tellInterface(10)
}
```

Выполнение `switch.go` приведет к результатам такого вида:

```
$ go run switch.go
{10} is a circle!
This is a rectangle!
This is a square!
Unknown type int!
```

Рефлексия

Рефлексия — это дополнительная функциональность Go, которая позволяет динамически, в процессе выполнения программы, узнавать тип произвольного объекта, а также получать информацию о его структуре. Для работы с рефлексией в Go создан пакет `reflect`. Советую запомнить одну вещь: едва ли вам придется часто использовать рефлексию при разработке программ на Go. Отсюда сразу следуют два вопроса: зачем нужна рефлексия и когда ее стоит использовать?

Рефлексия необходима для реализации таких пакетов, как `fmt`, `text/template` и `html/template`. В пакете `fmt` рефлексия избавляет нас от необходимости обрабатывать каждый из существующих типов данных по отдельности. Но даже если нам хватило терпения написать код для каждого из известных нам типов данных, мы все равно не смогли бы учесть все возможные типы! В этом случае благодаря рефлексии методы пакета `fmt` могут получить структуру любого типа данных и работать с ним.

Таким образом, рефлексия будет полезной в тех случаях, когда вы хотите обеспечить максимально возможную универсальность или когда хотите гарантировать возможность работы с типами данных, которые не существуют на момент написания кода, но могут появиться в будущем. Кроме того, рефлексия удобна при работе со значениями типов, которые не соответствуют общему интерфейсу.

Как видим, рефлексия помогает работать с неизвестными типами данных и неизвестными значениями типов. Однако за такую гибкость приходится дорого платить.

Звездами пакета `reflect` являются два типа данных — `reflect.Value` и `reflect.Type`. Первый из них используется для хранения значений любого типа, а второй — для представления типов Go.

Простой пример рефлексии

В этом подразделе приводится сравнительно простой пример рефлексии, чтобы вы могли уверенно себя чувствовать с расширенным функционалом Go.

Файл с исходным кодом Go для этого примера называется `reflection.go`. Рассмотрим его, разделив на четыре части. Цель `reflection.go` — исследовать «неизвестную» структурную переменную и получить дополнительную информацию о ней во время выполнения программы. Чтобы было еще интереснее, в программе определены два новых типа `struct`. На основании этих типов мы также добавим две новые переменные, однако исследуем только одну из них.

Если не передать программе аргументы командной строки, то она будет исследовать первую из этих переменных, в противном случае — вторую. На практике это означает, что программа не знает заранее (в процессе выполнения) тип структурной переменной, которую ей следует обработать.

В первой части `reflection.go` содержится следующий код Go:

```
package main

import (
    "fmt"
    "os"
    "reflect"
)

type a struct {
    X int
    Y float64
    Z string
}

type b struct {
    F int
    G int
    H string
    I float64
}
```

Здесь мы видим определение типов данных `struct`, которые будут использоваться в программе.

Второй фрагмент кода `mirror.go` выглядит так:

```
func main() {
    x := 100
    xRefl := reflect.ValueOf(&x).Elem()
    xType := xRefl.Type()
    fmt.Printf("The type of x is %s.\n", xType)
```

Этот код Go — небольшой, наивный пример отражения. Сначала мы объявляем переменную `x` и вызываем функцию `reflect.ValueOf(&x).Elem()`. Затем мы вызываем `xRefl.Type()`, чтобы получить тип переменной, который хранится в `xType`. В этих трех строках кода показано, как можно получить тип переменной, используя рефлексию. Но если вам нужно узнать только тип переменной, можно просто вызвать `reflect.TypeOf(x)`.

Третья часть `mirror.go` содержит следующий код Go:

```
    A := a{100, 200.12, "Struct a"}
    B := b{1, 2, "Struct b", -1.2}
    var r reflect.Value
```

```
    arguments := os.Args
    if len(arguments) == 1 {
        r = reflect.ValueOf(&A).Elem()
    } else {
        r = reflect.ValueOf(&B).Elem()
    }
```

Здесь мы объявляем две переменные — `A` и `B`. Переменная `A` относится к типу `a`, а переменная `B` — к типу `b`. Типом переменной `r` должен быть `reflect.Value`, поскольку функция `reflect.ValueOf()` возвращает значение именно этого типа. Метод `Elem()` возвращает значение, содержащееся в интерфейсе рефлексии (`reflect.Value`).

Последняя часть `reflection.go` выглядит так:

```
    iType := r.Type()
    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++ {
        fmt.Printf("Field name: %s ", iType.Field(i).Name)
        fmt.Printf("with type: %s ", r.Field(i).Type())
        fmt.Printf("and value %v\n", r.Field(i).Interface())
    }
}
```

В этой части программы для получения необходимой информации используются соответствующие функции пакета `reflect`. Метод `NumField()` возвращает количество полей в структуре `reflect.Value`, а функция `Field()` возвращает поле структуры, переданной ей в качестве параметра. Функция `Interface()` возвращает значение поля структуры `reflect.Value` в качестве интерфейса.

Если дважды запустить `reflection.go`, то получим следующий результат:

```
$ go run reflection.go 1
The type of x is int.
i Type: main.b
The 4 fields of main.b are:
Field name: F with type: int and value 1
Field name: G with type: int and value 2
Field name: H with type: string and value Struct b
Field name: I with type: float64 and value -1.2
$ go run reflection.go
The type of x is int.
i Type: main.a
The 3 fields of main.a are:
Field name: X with type: int and value 100
Field name: Y with type: float64 and value 200.12
Field name: Z with type: string and value Struct a
```

Важно отметить, что в Go используется свое, внутреннее представление для вывода типов данных переменных `A` и `B` — `main.a` и `main.b` соответственно. Однако к переменной `x`, типом которой является `int`, это не относится.

Более сложный пример рефлексии

В этом подразделе мы рассмотрим более сложные способы использования рефлексии, которые продемонстрируем на сравнительно небольших примерах с использованием кода Go из файла `advRefl.go`.

Мы разделим программу `advRefl.go` на пять частей. Первая часть выглядит так:

```
package main

import (
    "fmt"
    "os"
    "reflect"
)

type t1 int
type t2 int
```

Обратите внимание, что, хотя оба типа, `t1` и `t2`, основаны на типе `int` и, следовательно, по сути, относятся к одному и тому же типу `int`, для Go это совершенно разные типы. Их внутреннее представление после того, как Go проанализирует код программы, будет `main.t1` и `main.t2` соответственно.

Вторая часть кода `advRefl.go` выглядит так:

```
type a struct {
    X    int
    Y    float64
    Text string
}

func (a1 a) compareStruct(a2 a) bool {
    r1 := reflect.ValueOf(&a1).Elem()
    r2 := reflect.ValueOf(&a2).Elem()
    for i := 0; i < r1.NumField(); i++ {
        if r1.Field(i).Interface() != r2.Field(i).Interface() {
            return false
        }
    }
    return true
}
```

В этом коде определяется тип структуры Go с именем `a` и реализуется функция Go с именем `compareStruct()`. Цель этой функции — выяснить, являются ли две переменные типа `a` абсолютно одинаковыми. Как видите, для выполнения этой задачи в `compareStruct()` используется код Go из программы `mirror.go`.

Третий фрагмент `advRefl.go` содержит следующий код Go:

```
func printMethods(i interface{}) {
    r := reflect.ValueOf(i)
```

```
    t := r.Type()
    fmt.Printf("Type to examine: %s\n", t)

    for j := 0; j < r.NumMethod(); j++ {
        m := r.Method(j).Type()
        fmt.Println(t.Method(j).Name, "-->", m)
    }
}
```

Функция `printMethods()` выводит список методов переменной. Для демонстрации применения `printMethods()` в `advRefl.go` будет использоваться переменная типа `os.File`.

Четвертый фрагмент `advRefl.go` содержит следующий код Go:

```
func main() {
    x1 := t1(100)
    x2 := t2(100)
    fmt.Printf("The type of x1 is %s\n", reflect.TypeOf(x1))
    fmt.Printf("The type of x2 is %s\n", reflect.TypeOf(x2))

    var p struct{}
    r := reflect.New(reflect.ValueOf(&p).Type()).Elem()
    fmt.Printf("The type of r is %s\n", reflect.TypeOf(r))
```

Последняя часть кода из файла `advRefl.go` выглядит так:

```
    a1 := a{1, 2.1, "A1"}
    a2 := a{1, -2, "A2"}

    if a1.compareStruct(a1) {
        fmt.Println("Equal!")
    }

    if !a1.compareStruct(a2) {
        fmt.Println("Not Equal!")
    }

    var f *os.File
    printMethods(f)
}
```

Как вы вскоре узнаете, вызов `a1.compareStruct(a1)` возвращает `true`, поскольку мы сравниваем переменную `a1` с самой собой, тогда как вызов `a1.compareStruct(a2)` возвращает `false`, так как переменные `a1` и `a2` имеют разные значения.

Выполнение `advRefl.go` приведет к следующим результатам:

```
$ go run advRefl.go
The type of x1 is main.t1
The type of x2 is main.t2
The type of r is reflect.Value
Equal!
Not Equal!
```

```
Type to examine: *os.File
Chdir --> func() error
Chmod --> func(os.FileMode) error
Chown --> func(int, int) error
Close --> func() error
Fd --> func() uintptr
Name --> func() string
Read --> func([]uint8) (int, error)
ReadAt --> func([]uint8, int64) (int, error)
Readdir --> func(int) ([]os.FileInfo, error)
Readdirnames --> func(int) ([]string, error)
Seek --> func(int64, int) (int64, error)
Stat --> func() (os.FileInfo, error)
Sync --> func() error
Truncate --> func(int64) error
Write --> func([]uint8) (int, error)
WriteAt --> func([]uint8, int64) (int, error)
WriteString --> func(string) (int, error)
```

Как видим, типом переменной `r`, которая возвращается функцией `reflect.New()`, является `reflect.Value`. Кроме того, результат метода `printMethods()` говорит о том, что тип `*os.File` поддерживает множество методов, в том числе `Chdir()` и `Chmod()`.

Три недостатка рефлексии

Без сомнения, рефлексия в Go — мощная функциональность. Однако, как и любой другой инструмент, рефлексию следует использовать рационально, и на то есть три основные причины. Первая из них заключается в том, что из-за чересчур интенсивного использования рефлексии код становится трудно читать и поддерживать. Иногда эта проблема решается посредством хорошей документации, но, как известно, разработчикам обычно не хватает времени на написание необходимой документации.

Вторая причина состоит в том, что код Go, в котором используется рефлексия, замедляет выполнение программы. Вообще, код Go, предназначенный для работы с определенным типом данных, всегда будет быстрее, чем код Go, в котором используется рефлексия для динамической работы с любым типом данных. Кроме того, такой динамический код затрудняет работу для инструментов рефакторинга или анализа кода.

Последняя причина — ошибки рефлексии не распознаются на этапе сборки и обнаруживаются только в процессе выполнения, вызывая панику, так что ошибки рефлексии способны привести к сбою программ. Это может случиться через несколько месяцев или даже лет после разработки программы Go! Одно из решений данной проблемы — провести глубокое тестирование, прежде чем использовать вызов опасной функции. Однако это увеличивает количество кода Go в программах, что делает их еще медленнее.

Библиотека reflectwalk

Библиотека `reflewalk` позволяет перебирать сложные значения в Go, используя рефлексию, — подобно тому как мы перебираем файлы в файловой системе. В следующем примере программы `walkRef.go` показано, как перебрать все элементы структуры. Мы рассмотрим этот код, разделив его на пять частей.

Первая часть `walkRef.go` выглядит так:

```
package main

import (
    "fmt"
    "github.com/mitchellh/reflectwalk"
    "reflect"
)

type Values struct {
    Extra map[string]string
}
```

Поскольку `reflewalk` не является стандартным пакетом Go, нам нужно вызвать его, указав полный адрес.

Вторая часть `walkRef.go` выглядит так:

```
type WalkMap struct {
    MapVal  reflect.Value
    Keys    map[string]bool
    Values  map[string]bool
}

func (t *WalkMap) Map(m reflect.Value) error {
    t.MapVal = m
    return nil
}
```

Интерфейс, определеннный в `reflewalk`, требует реализации функции `Map()`, которая используется для поиска в хеш-таблицах.

Третья часть `walkRef.go` выглядит так:

```
func (t *WalkMap) MapElem(m, k, v reflect.Value) error {
    if t.Keys == nil {
        t.Keys = make(map[string]bool)
        t.Values = make(map[string]bool)
    }

    t.Keys[k.Interface().(string)] = true
    t.Values[v.Interface().(string)] = true
    return nil
}
```

Четвертая часть walkRef.go выглядит следующим образом:

```
func main() {
    w := new(WalkMap)

    type S struct {
        Map map[string]string
    }

    data := &S{
    Map: map[string]string{
            "V1": "v1v",
            "V2": "v2v",
            "V3": "v3v",
            "V4": "v4v",
        },
    }

    err := reflectwalk.Walk(data, w)
    if err != nil {
        fmt.Println(err)
    return
}
```

Здесь мы определяем новую переменную с именем data, которая содержит хеш-таблицу, и вызываем функцию reflewalk.Walk(), чтобы ее исследовать.

Последняя часть walkRef.go выглядит так:

```
    r := w.MapVal
    fmt.Println("MapVal:", r)
    rType := r.Type()
    fmt.Printf("Type of r: %s\n", rType)

    for _, key := range r.MapKeys() {
        fmt.Println("key:", key, "value:", r.MapIndex(key))
    }
}
```

В последней части walkRef.go показано, как использовать рефлексию для вывода на экран содержимого поля MapVal структуры WalkMap. Метод MapKeys() возвращает срез reflect.Values, каждое значение которого содержит один из ключей хеш-таблицы. Метод MapIndex() позволяет вывести на экран значение ключа. Методы MapKeys() и MapIndex() работают только с типом reflect.Map и позволяют перебирать все значения хеш-таблицы, однако последовательность возвращаемых элементов хеш-таблицы будет случайной.

Прежде чем в первый раз использовать библиотеку reflectwalk, необходимо скачать ее, что можно сделать следующим образом:

```
$ go get github.com/mitchellh/reflectwalk
```

Если вы решите использовать Go-модули, то процесс загрузки библиотеки reflectwalk будет автоматическим, что гораздо проще.

Выполнение `walkRef.go` приведет к следующим результатам:

```
$ go run walkRef.go
MapVal: map[V1:v1v V2:v2v V3:v3v V4:v4v]
Type of r: map[string]string
key: V2 value: v2v
key: V3 value: v3v
key: V4 value: v4v
key: V1 value: v1v
```

Объектно-ориентированное программирование на Go

Как вы уже знаете, в Go не используется наследование — вместо этого язык поддерживает композицию. Интерфейсы Go обеспечивают своего рода полиморфизм. Таким образом, хотя Go и не является объектно-ориентированным языком, некоторые его возможности позволяют имитировать объектно-ориентированное программирование.

Если вы действительно хотите разрабатывать приложения на основе объектно-ориентированной методологии, то, возможно, Go не лучший вариант для этого. Мне не очень нравится Java, поэтому я бы предложил рассмотреть варианты с C++ или Python.

Прежде всего позвольте показать вам два приема, которые будут применяться в программе Go из этого раздела. В первом из них используются методы, позволяющие ассоциировать функцию с типом. Другими словами, в некотором смысле функция и этот тип образуют объект. Второй прием позволяет встраивать тип в новый структурный тип, чтобы создать своего рода иерархию.

Есть еще третий прием, в котором с помощью интерфейса Go можно создать несколько элементов — *объектов одного и того же класса*. Этот прием не рассмотрен в данном разделе, поскольку он был показан ранее в главе.

Здесь главное, что интерфейс Go позволяет определять общее поведение для нескольких разных элементов, так что все эти элементы имеют общие характеристики объекта. Это позволяет утверждать, что все разные элементы являются объектами одного и того же класса; однако в реальном объектно-ориентированном языке программирования объекты и классы имеют гораздо больше возможностей.

Мы рассмотрим первые два приема на примере программы `ooo.go`, которую разделим на четыре части.

Первый фрагмент `ooo.go` содержит следующий код Go:

```
package main

import (
    "fmt"
)

type a struct {
    XX int
    YY int
}

type b struct {
    AA string
    XX int
}
```

Вторая часть программы выглядит так:

```
type c struct {
    A a
    B b
}
```

Таким образом, композиция позволяет создать структуру из элементов Go, используя несколько типов `struct`. В данном случае в типе данных `C` объединены переменные `a` и `b`.

Третья часть `ooo.go` содержит следующий код Go:

```
func (A a) A() {
    fmt.Println("Function A() for A")
}

func (B b) A() {
    fmt.Println("Function A() for B")
}
```

Здесь определены два метода, которые могут иметь одно и то же имя (`A()`), поскольку у них разные заголовки функций (они определены для разных типов) — первый работает с переменными `a`, второй — с переменными `b`. Этот прием позволяет использовать одно и то же имя функции для нескольких типов.

Последняя часть `ooo.go` выглядит так:

```
func main() {
    var i c
    i.A.A()
    i.B.A()
}
```

Код Go в ooo.go очень прост по сравнению с кодом объектно-ориентированного языка программирования, в котором реализованы абстрактные классы и наследование. Однако этого более чем достаточно для создания типов и структурных элементов, а также для использования разных типов данных, имеющих методы с одинаковыми именами.

Выполнение ooo.go приведет к следующим результатам:

```
$ go run ooo.go
Function A() for A
Function A() for B
```

Однако, как показывает следующий код, композиция не то же самое, что наследование, и тип first ничего не знает об изменениях, добавленных в функцию shared() типом second:

```
package main

import (
    "fmt"
)

type first struct{}

func (a first) F() {
    a.shared()
}

func (a first) shared() {
    fmt.Println("This is shared() from first!")
}

type second struct {
    first
}

func (a second) shared() {
    fmt.Println("This is shared() from second!")
}

func main() {
    first{}.F()
    second{}.shared()
    i := second{}
    j := i.first
    j.F()
}
```

Обратите внимание, что тип first встроен в тип second, и эти два типа совместно используют функцию с именем shared().

Если сохранить этот код Go в файле goCoIn.go и выполнить его, то получим следующий результат:

```
$ go run goCoIn.go
This is shared() from first!
This is shared() from second!
This is shared() from first!
```

Вызовы `first{}.F()` и `second{}.shared()` приводят к ожидаемым результатам, однако функция `j.F()` по-прежнему вызывает `first.shared()`, а не `second.shared()`. И это несмотря на то, что для типа `second` реализация функции `shared()` была изменена. В объектно-ориентированной терминологии это называется *переопределением метода*.

Обратите внимание, что вызов `j.F()` может быть записан как `(i.first).F()` или как `(second{}.first).F()` без необходимости определять слишком много переменных. Но, чтобы этот код был понятнее, лучше разделить его на три строки.

Основы git и GitHub

GitHub — это веб-сайт и сервис для хранения и сборки программного обеспечения. Для работы с GitHub можно использовать как его графический интерфейс, так и утилиты командной строки. С другой стороны, у нас есть `git(1)` — утилита командной строки, которая позволяет выполнять много задач, включая работу с репозиториями GitHub.

Альтернативой GitHub является GitLab. Большинство, если не все представленные здесь команды и опции git(1) будут работать и для связи с GitLab без каких-либо изменений.

В этом разделе представлено краткое описание утилиты `git(1)` и ее основных наиболее часто используемых команд.

Использование git

Обратите внимание, что у `git(1)` огромное количество команд и параметров, но далеко не все из них вам придется использовать ежедневно. В этом подразделе я приведу самые полезные и распространенные команды `git(1)`, исходя из моего опыта и стиля работы.

Для того чтобы перенести существующий репозиторий GitHub на локальный компьютер, нужно использовать команду `git clone`, в которой указать URL-адрес репозитория:

```
$ git clone git@github.com:mactsouk/go-kafka.git
Cloning into 'go-kafka'...
remote: Enumerating objects: 13, done.
remote: Counting objects: 100% (13/13), done.
remote: Compressing objects: 100% (8/8), done.
```

```
remote: Total 13 (delta 4), reused 10 (delta 4), pack-reused 0
Receiving objects: 100% (13/13), done.
Resolving deltas: 100% (4/4), done.
```

Команда git status

Команда `git status` показывает состояние рабочего дерева. Если оно полностью синхронизировано, то `git status` возвращает примерно следующее:

```
$ git status
On branch master
Your branch is up to date with 'origin/master'.
nothing to commit, working tree clean
```

Если же есть несинхронизированные изменения, то результат `git status` будет выглядеть примерно так:

```
On branch master
Your branch is up to date with 'origin/master'.
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)
   modified:
main.go
Untracked files:
  (use "git add <file>..." to include in what will be committed)
   newFile.go
no changes added to commit (use "git add" and/or "git commit -a")
```

Команда git pull

Команда `git pull` используется для получения обновлений из удаленного репозитория. Она особенно полезна, когда с одним репозиторием работают несколько человек или если вы работаете с нескольких компьютеров.

Команда git commit

Команда `git commit` предназначена для записи изменений в репозиторий. После выполнения команды `git commit` обычно нужно выполнить `git push`, чтобы передать изменения в удаленный репозиторий. Типичный способ выполнения команды `git commit` выглядит так:

```
$ git commit -a -m "Commit message"
```

Параметр `-m` указывает на то, что вместе с данными на сервер будет передано сообщение, в параметр `-a` заставляет `git commit` автоматически учесть все измененные файлы. Заметьте, что сюда не входят новые файлы, которые нужно сначала добавить в репозиторий с помощью команды `git add`.

Команда git push

Чтобы перенести локальные изменения в репозиторий GitHub, нужно выполнить команду `git push`. Результат работы выглядит примерно так:

```
$ touch a_file.go
$ git add a_file.go
$ git commit -a -m "Adding a new file"
[master 782c4da] Adding a new file
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 ch07/a_file.go
$ git push
Enumerating objects: 5, done.
Counting objects: 100% (5/5), done.
Delta compression using up to 8 threads
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 337 bytes | 337.00 KiB/s, done.
Total 3 (delta 2), reused 0 (delta 0)
remote: Resolving deltas: 100% (2/2), completed with 2 local objects.
To github.com:PacktPublishing/Mastering-Go-Second-Edition.git
   98f8a77..782c4da master -> master
```

Работа с ветками

Ветка — это способ управления рабочим процессом, позволяющий отделить изменения от основной ветки. В каждом репозитории есть главная ветка, используемая по умолчанию, — обычно она называется `master` — и, возможно, несколько других веток.

Чтобы создать на локальном компьютере новую ветку с именем `new_branch` и перейти к ней, нужно использовать следующую команду:

```
$ git checkout -b new_branch
Switched to a new branch 'new_branch'
```

Если вы хотите подключить эту ветку к GitHub, следует выполнить такую команду:

```
$ git push --set-upstream origin new_branch
Total 0 (delta 0), reused 0 (delta 0)
remote:
remote: Create a pull request for 'new_branch' on GitHub by visiting:
remote:
https://github.com/PacktPublishing/Mastering-Go-Second-Edition/pull/new/new_branch
remote:
To github.com:PacktPublishing/Mastering-Go-Second-Edition.git
 * [new branch]
new_branch -> new_branch
Branch 'new_branch' set up to track remote branch 'new_branch' from 'origin'.
```

Если вы хотите перейти с текущей ветки на основную ветку, то нужно выполнить следующую команду:

```
$ git checkout master
Switched to branch 'master'
Your branch is up to date with 'origin/master'.
```

Для того чтобы удалить локальную ветку — в данном случае new_branch, — нужно выполнить команду git branch -D:

```
$ git --no-pager branch -a
* master
  new_branch
  remotes/origin/HEAD -> origin/master
  remotes/origin/master
  remotes/origin/new_branch
$ git branch -D new_branch
Deleted branch new_branch (was 98f8a77).
$ git --no-pager branch -a
* master
  remotes/origin/HEAD -> origin/master
  remotes/origin/master
  remotes/origin/new_branch
```

Работа с файлами

Когда вы создаете в репозитории или удаляете оттуда один или несколько файлов, необходимо сообщить git(1) об этом. Для того чтобы удалить файл a_file.go, нужно сделать следующее:

```
$ rm a_file.go
$ git rm a_file.go
rm 'ch07/a_file.go'
```

Если после этого выполнить git status, то получим такой результат:

```
$ git status
On branch master
Your branch is up to date with 'origin/master'.
Changes to be committed:
 (use "git reset HEAD <file>..." to unstage)
   deleted:    a_file.go
```

Чтобы изменения вступили в силу, нужно сначала выполнить git commit, а затем git push:

```
$ git commit -a -m "Deleting a_file.go"
[master 1b06700] Deleting a_file.go
```

```
 1 file changed, 0 insertions(+), 0 deletions(-)
 delete mode 100644 ch07/a_file.go
$ git push
Enumerating objects: 5, done.
Counting objects: 100% (5/5), done.
Delta compression using up to 8 threads
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 296 bytes | 296.00 KiB/s, done.
Total 3 (delta 2), reused 0 (delta 0)
remote: Resolving deltas: 100% (2/2), completed with 2 local objects.
To github.com:PacktPublishing/Mastering-Go-Second-Edition.git
   782c4da..1b06700 master -> master
```

Файл .gitignore

В `.gitignore` содержится список файлов и каталогов, которые следует игнорировать при записи изменений на GitHub. Содержимое файла `.gitignore` может выглядеть, например, так:

```
$ cat .gitignore
public/
.DS_Store
*.swp
```

Обратите внимание: после того как файл `.gitignore` создан, его следует добавить в текущую ветку с помощью команды `git add`.

Команда git diff

Команда `git diff` показывает различия между зафиксированными изменениями и рабочим репозиторием, веткой и т. п.

Следующая команда покажет изменения между вашими файлами и файлами, которые находятся на GitHub (до последней команды `git push`). Именно эти изменения будут добавлены в версию, хранящуюся на GitHub, при выполнении команды `git push`:

```
$ git diff
diff --git a/content/blog/Stats.md b/content/blog/Stats.md
index 0f36b60..af64ec3 100644
--- a/content/blog/Stats.md
+++ b/content/blog/Stats.md
@@ -16,6 +16,8 @@ title: Statistical analysis of random numbers
## Developing a Kafka producer in Go
+Please note that the format of the first record that is written to Kafka
+specifies the format of the subsequent records
### Viewing the data in Lenses
```

Работа с тегами

Тег — это способ пометить определенную версию кода. Тег можно представить как ветку, которая никогда не меняется.

Чтобы создать *легкий тег*, нужно выполнить такую команду:

```
$ git tag c7.0
```

Чтобы получить информацию о конкретном теге, нужно сделать следующее:

```
$ git --no-pager show v1.0.0
commit f415872e62bd71a004b680d50fa089c139359533 (tag: v1.0.0)
Author: Mihalis Tsoukalos <mihalistsoukalos@gmail.com>
Date:   Sat Mar 2 20:33:58 2019 +0200
    Initial version 1.0.0
diff --git a/go.mod b/go.mod
new file mode 100644
index 0000000..c4928c5
--- /dev/null
+++ b/go.mod
@@ -0,0 +1,3 @@
+module github.com/mactsouk/myModule
+
+go 1.12
diff --git a/myModule.go b/myModule.go
index e69de29..fa6b0fe 100644
--- a/myModule.go
+++ b/myModule.go
@@ -0,0 +1,9 @@
+package myModule
+
+import (
+    "fmt"
+)
+
+func Version() {
+    fmt.Println("Version 1.0.0")
+}
```

Чтобы получить список всех доступных тегов, можно воспользоваться командой git tag:

```
$ git --no-pager tag
c7.0
```

Чтобы внести тег в GitHub, нужно сделать следующее:

```
$ git push origin c7.0
Total 0 (delta 0), reused 0 (delta 0)
To github.com:PacktPublishing/Mastering-Go-Second-Edition.git
* [new tag]      c7.0 -> c7.0
```

Чтобы удалить существующий тег с локального компьютера, нужно выполнить следующую команду:

```
$ git tag -d c7.0
Deleted tag 'c7.0' (was 1b06700)
```

Можно также удалить тег с сервера GitHub:

```
$ git push origin :refs/tags/c7.0
To github.com:PacktPublishing/Mastering-Go-Second-Edition.git
- [deleted]      c7.0
```

Команда git cherry-pick

Команда `git cherry-pick` — это сложная команда, которую нужно использовать осторожно, поскольку она вносит в текущую ветку изменения, зафиксированные некоторыми из последних команд `git commit`. Следующая команда вносит в текущую ветвь фиксацию номер `4226f2c4`:

```
$ git cherry-pick 4226f2c4
```

Эта команда вносит в текущую ветку все фиксации с `4226f2c4` по `0d820a87`, исключая фиксацию номер `4226f2c4`:

```
$ git cherry-pick 4226f2c4..0d820a87
```

Данная команда вносит в текущую ветку все фиксации с `4226f2c4` по `0d820a87`, включая фиксацию `4226f2c4`:

```
$ git cherry-pick 4226f2c4^..0d820a87
```

Представленный список команд и параметров утилиты git(1) далеко не полный, однако этого достаточно, чтобы вы смогли работать с git(1) и GitHub. Эти знания пригодятся вам при создании модулей Go.

Отладка с помощью Delve

Delve — это текстовый *отладчик* программ Go, также написанный на Go. Чтобы скачать Delve для macOS Mojave, нужно сделать следующее:

```
$ go get -u github.com/go-delve/delve/cmd/dlv
$ ls -l ~/go/bin/dlv
-rwxr-xr-x 1 mtsouk staff    16M Mar 7 09:04 /Users/mtsouk/go/bin/dlv
```

Поскольку Delve зависит от множества модулей и пакетов Go, процесс установки займет некоторое время. Двоичный файл Delve размещается в каталоге `~/go/bin`.

Чтобы узнать информацию о версии отладчика, нужно выполнить команду `dlv version`:

```
$ ~/go/bin/dlv version
Delve Debugger
Version: 1.2.0
Build: $Id: 068e2451004e95d0b042e5257e34f0f08ce01466 $
```

Обратите внимание, что Delve также работает на компьютерах с Linux и Microsoft Windows. Учтите, что Delve — это внешняя программа, то есть для работы Delve не нужно подключать какие-либо пакеты к вашим программам Go.

Поскольку этот раздел представляет собой краткое описание основ Delve, далее мы рассмотрим небольшой пример, который поможет вам начать работу с отладчиком Delve. Приведенная здесь информация раскрывает общие принципы не только Delve, но и почти любого другого отладчика.

Пример отладки

Если путь `~/go/bin` хранится в системной переменной `PATH`, то мы можем вызывать Delve откуда захотим, просто как `dlv`. В противном случае нужно указывать полный путь. В этом разделе я буду использовать полный путь к утилите.

Первая команда Delve, которую следует усвоить, — `debug`. По этой команде Delve компилирует пакет `main` в текущем рабочем каталоге и начинает его отладку. Если в текущем рабочем каталоге нет пакета `main`, то выводится следующее сообщение об ошибке:

```
$ ~/go/bin/dlv debug
go: cannot find main module; see 'go help modules'
exit status 1
```

Перейдем в каталог `./ch07/debug` и отладим настоящую программу, которая хранится в файле `main.go`. Код в `main.go` выглядит так:

```
package main

import (
    "fmt"
    "os"
)

func function(i int) int {
    return i * i
}

func main() {
    if len(os.Args) == 1 {
        fmt.Println("Need at least one argument.")
```

```
        return
    }

    i := 5
    fmt.Println("Debugging with Delve")
    fmt.Println(i)
    fmt.Println(function(i))

    for arg, _ := range os.Args[1:] {
        fmt.Println(arg)
    }
}
```

Чтобы передать в программу аргументы командной строки, следует запустить Delve так:

```
$ ~/go/bin/dlv debug -- arg1 arg2 arg3
```

Мы выполним Delve следующим образом:

```
$ ~/go/bin/dlv debug -- 1 2 3
Type 'help' for list of commands.
(dlv)
```

Экранное приглашение Delve, `(dlv)`, открывает строку, в которой можно вводить команды Delve. Если ввести здесь просто `c` (от `continue` — «продолжить»), то программа Go будет выполняться так, как если бы мы ввели команду `go run` из оболочки UNIX:

```
(dlv) c
Debugging with Delve
5
25
0
1
2
Process 57252 has exited with status 0
(dlv)
```

Если снова ввести `c` или `continue`, то получим следующее:

```
(dlv) c
Process 57252 has exited with status 0
```

Так происходит потому, что программа закончила работу и в данный момент ее выполнение не может быть продолжено. Чтобы иметь возможность ее дальнейшей отладки, придется запустить программу заново. Для перезапуска программы нужно ввести `r` или `restart`, что эквивалентно выполнению Delve из оболочки UNIX:

```
(dlv) r
Process restarted with PID 57257
(dlv)
```

Теперь попробуем что-нибудь еще. Мы создадим две точки останова, одну для функции `main()` и вторую — для `function()`:

```
$ ~/go/bin/dlv debug -- 1 2 3
Type 'help' for list of commands.
(dlv)
(dlv) break main.main
Breakpoint 1 set at 0x10b501b for main.main() ./main.go:12
(dlv) break function
Breakpoint 2 set at 0x10b4fe0 for main.function() ./main.go:8
(dlv)
```

Если теперь ввести `continue`, то отладчик остановит программу на строке `main()` или `function()`, в зависимости от того, какая из них ему встретится раньше. Поскольку это исполняемая программа, то функция `main()` будет первой:

```
(dlv) c
> main.main() ./main.go:12 (hits goroutine(1):1 total:1) (PC: 0x10b501b)
     7:
     8:    func function(i int) int {
     9:        return i * i
    10:    }
    11:
=>  12:    func main() {
    13:        if len(os.Args) == 1 {
    14:            fmt.Println("Need at least one argument.")
    15:            return
    16:        }
    17:
(dlv)
```

Двойная стрелка `=>` указывает на строку исходного кода, в которой произошел останов. Если ввести `next`, то отладчик перейдет к следующему оператору Go — команду `next` можно вводить сколько угодно раз, пока мы не достигнем конца программы:

```
(dlv) next
> main.main() ./main.go:13 (PC: 0x10b5032)
     8:    func function(i int) int {
     9:        return i * i
    10:    }
    11:
    12:    func main() {
=>  13:    if len(os.Args) == 1 {
    14:        fmt.Println("Need at least one argument.")
    15:        return
    16:    }
    17:
    18:    i := 5
(dlv) next
> main.main() ./main.go:18 (PC: 0x10b50d0)
```

```
    13:     if len(os.Args) == 1 {
    14:         fmt.Println("Need at least one argument.")
    15:         return
    16:     }
    17:
=>  18:     i := 5
    19:     fmt.Println("Debugging with Delve")
    20:     fmt.Println(i)
    21:     fmt.Println(function(i))
    22:
    23:     for arg, _ := range os.Args[1:] {
(dlv) next
> main.main() ./main.go:19 (PC: 0x10b50db)
    14:             fmt.Println("Need at least one argument.")
    15:             return
    16:         }
    17:
    18:         i := 5
=>  19:         fmt.Println("Debugging with Delve")
    20:         fmt.Println(i)
    21:         fmt.Println(function(i))
    22:
    23:         for arg, _ := range os.Args[1:] {
    24:             fmt.Println(arg)
(dlv) print i
5
```

Последняя команда (`print i`) выводит на экран значение переменной `i`. Если после этого ввести `continue`, то мы перейдем к следующей точке останова, если она есть, или к концу программы:

```
(dlv) c
Debugging with Delve
5
> main.function() ./main.go:8 (hits goroutine(1):1 total:1) (PC: 0x10b4fe0)
     3:     import (
     4:         "fmt"
     5:         "os"
     6:     )
     7:
=>   8:     func function(i int) int {
     9:         return i * i
    10:     }
    11:
    12:     func main() {
    13:         if len(os.Args) == 1 {
```

В данном случае следующей точкой останова является функция `function()`, как было определено ранее.

Обратите внимание, что для отладки Go-тестов следует использовать команду `dlv test`, и отладчик Delve сам позаботится об остальном.

Дополнительные ресурсы

Следующие ресурсы будут для вас очень полезны:

- посетите страницу документации стандартного Go-пакета `reflect`, расположенную по адресу https://golang.org/pkg/reflect/. У этого пакета гораздо больше возможностей, чем описано в данной главе;
- GitHub: https://github.com/;
- GitLab: https://gitlab.com/;
- подробнее о Delve читайте здесь: https://github.com/go-delve/delve;
- больше о библиотеке `reflectwalk`, созданной Митчеллом Хашимото (Mitchell Hashimoto), вы узнаете на странице https://github.com/mitchellh/reflectwalk. Изучая его код, вы лучше поймете, что такое рефлексия.

Упражнения

- Напишите собственный интерфейс и используйте его в отдельной программе Go. В чем польза вашего интерфейса?
- Напишите интерфейс для вычисления объема трехмерных фигур, таких как кубы и сферы.
- Напишите интерфейс для вычисления длины отрезков и расстояния между двумя точками на плоскости.
- Исследуйте рефлексию, используя собственный пример.
- Как работает рефлексия с хеш-таблицами Go?
- Если вы хорошо разбираетесь в математике, попробуйте написать интерфейс, который бы реализовал четыре основные математические операции для действительных и комплексных чисел. Не используйте стандартные типы Go `complex64` и `complex128`, а определите собственную структуру для поддержки комплексных чисел.

Резюме

В этой главе вы познакомились с процессом отладки, `git(1)`, GitHub и интерфейсами, которые похожи на контракты. Вы также узнали о методах типа, операциях утверждения типа и рефлексии в Go. Несмотря на то что рефлексия — очень мощное свойство Go, она способна замедлить работу ваших программ Go, поскольку вводит новый уровень сложности на этапе выполнения программы. Кроме того, если неосторожно использовать рефлексию, программа Go может прерваться с ошибкой.

Вы также узнали о том, как писать код Go, который следует принципам объектно-ориентированного программирования. Если вы собираетесь запомнить только что-то одно из этой главы, то следует учитывать, что Go не является объектно-ориентированным языком программирования, но может имитировать некоторые функциональные возможности, предлагаемые объектно-ориентированными языками, такими как Java и C++. Другими словами, если вы планируете постоянно разрабатывать программное обеспечение на основе объектно-ориентированной парадигмы, лучше выбрать вместо Go другой язык программирования. Впрочем, объектно-ориентированное программирование не является панацеей и можно создать лучшую, более ясную и надежную структуру программы, выбрав такой язык программирования, как Go!

Возможно, в этой главе было больше теории, чем вы ожидали, зато следующая глава вознаградит ваше терпение; она посвящена системному программированию на Go. В ней рассмотрены вопросы файлового ввода/вывода, работа с системными файлами UNIX, обработка сигналов UNIX и поддержка каналов в UNIX.

В следующей главе также рассказано об использовании пакетов `flag` и `viper` для поддержки инструментов командной строки с несколькими аргументами и параметрами, а также о пакете `cobra`, разрешениях на доступ к файлам UNIX и некоторых расширенных возможностях использования функций, предлагаемых стандартным Go-пакетом `syscall`. Если вы действительно интересуетесь системным программированием на Go, обратитесь к моей книге *Go Systems Programming*, в которой этот вопрос обсуждается более подробно.

8 Как объяснить UNIX-системе, что она должна делать

В предыдущей главе рассмотрены две актуальные, но несколько теоретические темы Go: интерфейсы и рефлексия. Код Go, с которым вы познакомитесь в этой главе, едва ли можно назвать теоретическим!

Тема данной главы — *системное программирование*. Ведь, в конце концов, Go — это полноценный язык системного программирования, появившийся в результате разочарования. Его создатели были недовольны выбором языков программирования для создания системного программного обеспечения, который существовал на тот момент, и поэтому решили создать новый язык программирования.

В этой главе содержится ряд интересных и в некоторой степени узких тем, которые не были включены в книгу Go Systems Programming (Packt Publishing, 2017).

В этой главе рассмотрены следующие темы:

- процессы UNIX;
- пакет `flag`;
- пакет `viper`;
- пакет `cobra`;
- использование интерфейсов `io.Reader` и `io.Writer`;
- обработка сигналов UNIX средствами Go с помощью пакета `os/signal`;
- поддержка каналов UNIX в системных утилитах UNIX;
- разработка клиентов Go для Docker;
- чтение текстовых файлов;
- чтение файлов в формате CSV;

- запись файлов;
- пакет `bytes`;
- расширенные возможности использования пакета `syscall`;
- полномочия доступа к файлам UNIX.

О процессах в UNIX

Процесс — это, с одной стороны, среда выполнения, в которой содержатся инструкции, пользовательские данные и части системных данных, а также другие типы ресурсов, получаемые во время выполнения. С другой стороны, *программа* — это двоичный файл, в котором содержатся инструкции и данные, используемые для инициализации инструкций и частей пользовательских данных процесса. Каждый действующий процесс UNIX имеет уникальный идентификатор — целое число без знака, которое называется *идентификатором процесса*.

Существует три категории процессов: *пользовательские процессы*, *процессы-демоны* и *процессы ядра*. Пользовательские процессы выполняются в пространстве пользователя и обычно не имеют специальных прав доступа. Процессы-демоны — это программы, которые размещаются в пользовательском пространстве и работают в фоновом режиме, без необходимости использования терминала. Процессы ядра выполняются только в пространстве ядра и имеют полный доступ ко всем структурам данных ядра.

С-стиль создания новых процессов заключается в системном вызове функции fork(). Значение, возвращаемое fork(), позволяет программисту различать родительский и дочерний процессы. В Go, наоборот, такая функциональность не поддерживается, вместо нее используются горутины.

Пакет flag

Флаги — это строки, отформатированные специальным образом, которые передаются в программу при запуске для управления ее поведением. Работать с флагами напрямую может оказаться очень сложно, особенно если вы хотите поддерживать несколько флагов. Поэтому, если вы разрабатываете утилиты командной строки UNIX, пакет `flag` будет вам очень интересен и полезен.

Пакет `flag` не делает никаких предположений о последовательности аргументов и параметров командной строки; он выводит полезные сообщения в случае возникновения ошибки при выполнении утилиты командной строки.

Главное преимущество пакета flag заключается в том, что он является частью стандартной библиотеки Go, следовательно, он тщательно протестирован и отлажен.

Мы рассмотрим две программы Go, в которых используется пакет `flag`: одну проще, а вторую — сложнее. Первая программа называется `simpleFlag.go`. Разделим ее на четыре части. Программа `simpleFlag.go` распознает два аргумента командной строки — логический и целочисленный.

В первой части `simpleFlag.go` содержится следующий код Go:

```
package main

import (
    "flag"
    "fmt"
)
```

Вторая часть кода `simpleFlag.go` выглядит так:

```
func main() {
    minusK := flag.Bool("k", true, "k flag")
    minusO := flag.Int("O", 1, "O")
    flag.Parse()
```

Оператор `flag.Bool("k", true, "k flag")` определяет логический аргумент командной строки с именем `k`, значение которого по умолчанию равно `true`. Последний параметр оператора — это строка использования, которая будет отображаться вместе с другой информацией об использовании программы. Аналогично функция `flag.Int()` добавляет в программу поддержку целочисленного аргумента командной строки.

После определения аргументов командной строки, которые поддерживает программа, всегда нужно вызывать flag.Parse().

В третьей части программы `simpleFlag.go` содержится следующий код Go:

```
    valueK := *minusK
    valueO := *minusO
    valueO++
```

В этом коде Go показано, как можно получить значения аргументов командной строки. К счастью, пакет `flag` автоматически преобразует входные данные, связанные с флагом `flag.Int()`, в целочисленное значение и нам не нужно делать это самостоятельно. Кроме того, пакет `flag` следит, чтобы аргументу было присвоено допустимое целочисленное значение.

Остальной код Go в программе `simpleFlag.go` выглядит так:

```
    fmt.Println("-k:", valueK)
    fmt.Println("-O:", valueO)
}
```

Теперь, получив значения нужных параметров, мы готовы их использовать.

Взаимодействие с `simpleFlag.go` приводит к результатам такого вида:

```
$ go run simpleFlag.go -O 100
-k: true
-O: 101
$ go run simpleFlag.go -O=100
-k: true
-O: 101
$ go run simpleFlag.go -O=100 -k
-k: true
-O: 101
$ go run simpleFlag.go -O=100 -k false
-k: true
-O: 101
$ go run simpleFlag.go -O=100 -k=false
-k: false
-O: 101
```

Если при выполнении файла `simpleFlag.go` возникает ошибка, то на экран выводится сообщение следующего вида от пакета `flag`:

```
$ go run simpleFlag.go -O=notAnInteger
invalid value "notAnInteger" for flag -O: parse error
Usage of /var/folders/sk/ltk8cnw50lzdtr2hxcj5sv2m0000gn/T/go-
build593534621/b001/exe/simpleFlag:
    -O int
          O (default 1)
    -k    flag (default true)
exit status 2
```

Обратите внимание на удобное сообщение, которое выводится автоматически, если пользователь ошибочно задаст аргументы командной строки, определенные для данной программы.

Теперь пора представить более реалистичную, расширенную программу, использующую пакет `flag`. Эта программа называется `funWithFlag.go`. Рассмотрим ее, разделив на пять частей. Утилита `funWithFlag.go` распознает разные типы аргументов, в том числе такой, который принимает несколько значений, разделенных запятыми. Кроме того, на примере этой программы вы увидите, как можно получить доступ к аргументам командной строки, которые расположены в конце строки запуска исполняемого файла и не соответствуют ни одному из возможных вариантов.

Функция `flag.Var()`, используемая в `funWithFlag.go`, создает флаг любого типа, который соответствует интерфейсу `flag.Value`. Этот интерфейс определяется следующим образом:

```
type Value interface {
    String() string
    Set(string) error
}
```

В первой части `funWithFlag.go` содержится следующий код Go:

```
package main

import (
    "flag"
    "fmt"
    "strings"
)

type NamesFlag struct {
    Names []string
}
```

Структуру `NamesFlag` мы вскоре используем для интерфейса `flag.Value`.

Вторая часть `funWithFlag.go` выглядит так:

```
func (s *NamesFlag) GetNames() []string {
    return s.Names
}

func (s *NamesFlag) String() string {
    return fmt.Sprint(s.Names)
}
```

Третья часть `funWithFlag.go` содержит следующий код:

```
func (s *NamesFlag) Set(v string) error {
    if len(s.Names) > 0 {
        return fmt.Errorf("Cannot use names flag more than once!")
    }

    names := strings.Split(v, ",")
    for _, item := range names {
        s.Names = append(s.Names, item)
    }
    return nil
}
```

Прежде всего здесь метод `Set()` гарантирует, что соответствующему аргументу командной строки еще не присвоено значение. Затем этот метод получает входные

данные и разделяет их на отдельные аргументы с помощью функции `strings.Split()`. После этого аргументы сохраняются в поле `Names` структуры `NamesFlag`.

Четвертый фрагмент `funWithFlag.go` содержит следующий код Go:

```
func main() {
    var manyNames NamesFlag
    minusK := flag.Int("k", 0, "An int")
    minusO := flag.String("o", "Mihalis", "The name")
    flag.Var(&manyNames, "names", "Comma-separated list")

    flag.Parse()
    fmt.Println("-k:", *minusK)
    fmt.Println("-o:", *minusO)
```

Последняя часть утилиты `funWithFlag.go` выглядит так:

```
    for i, item := range manyNames.GetNames() {
        fmt.Println(i, item)
    }

    fmt.Println("Remaining command line arguments:")
    for index, val := range flag.Args() {
        fmt.Println(index, ":", val)
    }
}
```

Срез `flag.Args()` содержит оставшиеся аргументы командной строки, а переменная `manyNames` — значения аргументов командной строки из `flag.Var()`.

Выполнение кода из `funWithFlag.go` приводит к результатам такого вида:

```
$ go run funWithFlag.go -names=Mihalis,Jim,Athina 1 two Three
-k: 0
-o: Mihalis
0 Mihalis
1 Jim
2 Athina
Remaining command line arguments:
0 : 1
1 : two
2 : Three
$ go run funWithFlag.go -Invalid=Marietta 1 two Three
flag provided but not defined: -Invalid
Usage of funWithFlag:
  -k int
        An int
  -names value
        Comma-separated list
  -o string
        The name (default "Mihalis")
```

```
exit status 2
$ go run funWithFlag.go -names=Marietta -names=Mihalis
invalid value "Mihalis" for flag -names: Cannot use names flag more than
once!
Usage of funWithFlag:
  -k int
        An int
  -names value
        Comma-separated list
  -o string
        The name (default "Mihalis")
exit status 2
```

Вам, скорее всего, не избежать использования пакета Go для обработки аргументов командной строки вашей программы, если только вы не разрабатываете простейшую утилиту командной строки, которая не требует аргументов.

Пакет viper

Пакет `viper` — мощный Go-пакет, который поддерживает большое количество опций. Все проекты `viper` соответствуют единому шаблону. Сначала нужно инициализировать `viper`, а затем определить интересующие вас элементы. После этого вы получаете доступ к этим элементам, можете читать и использовать их значения. Обратите внимание, что пакет `viper` способен полностью заменить пакет `flag`.

Нужные значения можно получить либо напрямую, как при использовании пакета `flag` стандартной библиотеки Go, так и косвенно, с помощью файлов конфигурации. При использовании файлов конфигурации, в которых свойства представлены в формате JSON, YAML, TOML, HCL или Java, `viper` автоматически выполняет весь синтаксический анализ, что избавляет вас от необходимости писать и отлаживать большой объем кода Go. Пакет `viper` также позволяет извлекать значения из структур Go и сохранять их в этих структурах. Однако для этого необходимо, чтобы поля структуры Go 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 утилитах — применяйте только те из них, которые вам действительно нужны. Общее правило заключается в том, чтобы использовать те функции `viper`, которые упрощают код. Иначе говоря, если вашей утилите требуется слишком много аргументов и флагов командной строки, то лучше использовать файл конфигурации.

Простой пример использования viper

Прежде чем перейти к более сложным примерам, мы рассмотрим код Go, в котором используется `viper`. Программа называется `usingViper.go`. Разделим ее на три части.

Первая часть `useViper.go` выглядит так:

```
package main

import (
    "fmt"
    "github.com/spf13/viper"
)
```

Вторая часть `useViper.go` выглядит следующим образом:

```
func main() {
    viper.BindEnv("GOMAXPROCS")
    val := viper.Get("GOMAXPROCS")
    fmt.Println("GOMAXPROCS:", val)
    viper.Set("GOMAXPROCS", 10)
    val = viper.Get("GOMAXPROCS")
    fmt.Println("GOMAXPROCS:", val)
```

Последняя часть `useViper.go` содержит такой код Go:

```
    viper.BindEnv("NEW_VARIABLE")
    val = viper.Get("NEW_VARIABLE")
    if val == nil {
        fmt.Println("NEW_VARIABLE not defined.")
        return
    }
    fmt.Println(val)
}
```

Цель этой программы — показать, как читать и изменять переменные среды, используя `viper`. Пакет `flag` не предоставляет такой функциональности — это позволяет делать стандартный Go-пакет `os`, однако не так просто, как `viper`.

Для начала необходимо загрузить пакет `viper`. Если вы не используете Go-модули, это можно сделать следующим образом:

```
$ go get -u github.com/spf13/viper
```

Если вы используете Go-модули, то Go автоматически загрузит `viper` при первой попытке запустить программу Go, в которой применяется этот пакет.

Выполнение `useViper.go` приведет к результатам такого вида:

```
$ go run useViper.go
GOMAXPROCS: <nil>
GOMAXPROCS: 10
NEW_VARIABLE not defined.
```

От flag к viper

Возможно, у вас уже есть программа Go, в которой используется пакет `flag` и которую вы хотели бы перевести на пакет `viper`. Например, рассмотрим следующий код Go, где используется пакет `flag`:

```
package main

import (
    "flag"
    "fmt"
)

func main() {
    minusI := flag.Int("i", 100, "i parameter")
    flag.Parse()
    i := *minusI
    fmt.Println(i)
}
```

В новой версии этой программы, которую мы сохраним в файле `flagToViper.go`, будет использоваться пакет `viper`. Мы рассмотрим эту программу, разделив ее на три части. Первая часть `flagToViper.go` выглядит так:

```
package main

import (
    "flag"
    "fmt"
    "github.com/spf13/pflag"
    "github.com/spf13/viper"
)
```

Для того чтобы `viper` мог работать с аргументами командной строки, нам нужно импортировать пакет `pflag`.

Вторая часть `flagToViper.go` содержит следующий код Go:

```
func main() {
    flag.Int("i", 100, "i parameter")
    pflag.CommandLine.AddGoFlagSet(flag.CommandLine)
    pflag.Parse()
```

Как видим, здесь по-прежнему используется `flag.Int()`, чтобы изменить как можно меньше кода, но для синтаксического анализа вызывается `pflag.Parse()`. Однако магия заключается в вызове `pflag.CommandLine.AddGoFlagSet(flag.CommandLine)`: именно этот вызов импортирует данные из пакета `flag` в пакет `pflag`.

Последняя часть `flagToViper.go` выглядит так:

```
    viper.BindPFlags(pflag.CommandLine)
    i := viper.GetInt("i")
    fmt.Println(i)
}
```

Осталось вызвать последнюю функцию — `viper.BindPFlags()`, и теперь можно получить значение целочисленного аргумента командной строки с помощью функции `viper.GetInt()`. Для иных типов данных придется вызывать другие функции `viper`.

Здесь может понадобиться загрузить в программу `flagToViper.go` Go-пакет `pflag`. Это можно сделать так:

```
$ go get -u github.com/spf13/pflag
```

Выполнение `flagToViper.go` приведет к результатам следующего вида:

```
$ go run flagToViper.go
100
$ go build flagToViper.go
$ ./flagToViper -i 0
0
$ ./flagToViper -i abcd
invalid argument "abcd" for "-i, --i" flag: parse error
Usage of ./flagToViper:
    -i, --i int i parameter
invalid argument "abcd" for "-i, --i" flag: parse error
```

Если по какой-либо причине программе `flagToViper.go` был передан неизвестный аргумент командной строки, `viper` выдаст в ответ сообщение об ошибке:

```
$ ./flagToViper -j 200
unknown shorthand flag: 'j' in -j
Usage of ./flagToViper:
    -i, --i int i parameter (default 100)
unknown shorthand flag: 'j' in -j
exit status 2
```

Чтение конфигурационных файлов в формате JSON

В этом подразделе показано, как с помощью пакета `viper` можно читать конфигурационные файлы в формате JSON. Для этого мы рассмотрим утилиту `readJSON.go`, которую разделим на три части. Первая часть `readJSON.go` выглядит так:

```
package main

import (
    "fmt"
    "github.com/spf13/viper"
)
```

Во второй части `readJSON.go` содержится следующий код:

```
func main() {
    viper.SetConfigType("json")
    viper.SetConfigFile("./myJSONConfig.json")
    fmt.Printf("Using config: %s\n", viper.ConfigFileUsed())
    viper.ReadInConfig()
```

Здесь выполняется синтаксический анализ конфигурационного файла. Обратите внимание, что имя конфигурационного JSON-файла в `readJSON.go` жестко закодировано с помощью вызова функции `viper.SetConfigFile("./myJSONConfig.json")`.

Последняя часть `readJSON.go` выглядит так:

```
    if viper.IsSet("item1.key1") {
        fmt.Println("item1.key1:", viper.Get("item1.key1"))
    } else {
        fmt.Println("item1.key1 not set!")
    }

    if viper.IsSet("item2.key3") {
        fmt.Println("item2.key3:", viper.Get("item2.key3"))
    } else {
        fmt.Println("item2.key3 is not set!")
    }

    if !viper.IsSet("item3.key1") {
        fmt.Println("item3.key1 is not set!")
    }
}
```

Именно здесь программа проверяет значения конфигурационного JSON-файла, чтобы выяснить, есть ли там нужные ключи.

Содержимое файла `myJSONConfig.json` выглядит так:

```
{
    "item1": {
        "key1": "val1",
        "key2": false,
        "key3": "val3",
    },

    "item2": {
        "key1": "val1",
        "key2": true,
        "key3": "val3",
    }
}
```

Выполнение кода из `readJSON.go` приведет к результатам следующего вида:

```
$ go run readJSON.go
Using config: ./myJSONConfig.json
item1.key1: val1
item2.key3: val3
item3.key1 is not set!
```

Если программа не находит `myJSONConfig.json`, то она не сообщит об ошибке, а будет действовать так, как будто прочитала пустой конфигурационный файл:

```
$ mv myJSONConfig.json ..
$ go run readJSON.go
```

```
Using config: ./myJSONConfig.json
item1.key1 not set!
item2.key3 is not set!
item3.key1 is not set!
```

Чтение конфигурационных файлов в формате YAML

YAML — это еще один популярный текстовый формат, который используется для конфигурационных файлов. В этом подразделе показано, как читать конфигурационные файлы формата YAML с помощью пакета `viper`. На этот раз имя конфигурационного YAML-файла будет передаваться утилите в виде аргумента командной строки. Кроме того, в утилите будет использоваться функция `viper.AddConfigPath()` для добавления трех путей поиска, по которым `viper` будет автоматически искать конфигурационные файлы. Утилита, которую мы рассмотрим, называется `readYAML.go`. Разделим ее на четыре части.

Первая часть `readYAML.go` выглядит так:

```
package main

import (
    "fmt"
    flag "github.com/spf13/pflag"
    "github.com/spf13/viper"
    "os"
)
```

В преамбуле программы определяется псевдоним (`flag`) для Go-пакета `pflag`.

Вторая часть `readYAML.go` содержит следующий код Go:

```
func main() {
    var configFile *string = flag.String("c", "myConfig",
         "Setting the configuration file")
    flag.Parse()

    _, err := os.Stat(*configFile)

    if err == nil {
        fmt.Println("Using User Specified Configuration file!")
        viper.SetConfigFile(*configFile)
    } else {
        viper.SetConfigName(*configFile)
        viper.AddConfigPath("/tmp")
        viper.AddConfigPath("$HOME")
        viper.AddConfigPath(".")
    }
```

Этот код проверяет, существует ли файл, указанный в значении флага конфигурации (`--c`), используя вызов `os.Stat()`. Если указанный файл существует,

то будет использован предоставленный файл; в противном случае — имя файла конфигурации, предлагаемое по умолчанию (`myConfig`). Обратите внимание, что мы не указываем точно, что хотим использовать конфигурационный файл в формате YAML, — программа будет искать все поддерживаемые форматы файлов при условии, что имя файла без расширения — `myConfig`. Именно так работает `viper`. Для поиска конфигурационных файлов будут использоваться три пути в следующем порядке: `/tmp`, домашний каталог текущего пользователя и текущий рабочий каталог.

Вообще говоря, использовать каталог `/tmp` для хранения конфигурационных файлов не рекомендуется — прежде всего потому, что содержимое каталога `/tmp` автоматически удаляется после каждой перезагрузки системы; здесь этот каталог используется только для простоты.

Использование `viper.ConfigFileUsed()` совершенно оправданно, поскольку конфигурационный файл не задан жестко, следовательно, нам придется определять его самостоятельно.

Третья часть `readYAML.go` выглядит так:

```
    err = viper.ReadInConfig()
    if err != nil {
        fmt.Printf("%v\n", err)
        return
    }
    fmt.Printf("Using config: %s\n", viper.ConfigFileUsed())
```

Чтение и синтаксический анализ YAML-файла выполняются с помощью вызова функции `viper.ReadInConfig()`.

Последняя часть `readYAML.go` выглядит следующим образом:

```
    if viper.IsSet("item1.k1") {
        fmt.Println("item1.val1:", viper.Get("item1.k1"))
    } else {
        fmt.Println("item1.k1 not set!")
    }

    if viper.IsSet("item1.k2") {
        fmt.Println("item1.val2:", viper.Get("item1.k2"))
    } else {
        fmt.Println("item1.k2 not set!")
    }

    if !viper.IsSet("item3.k1") {
        fmt.Println("item3.k1 is not set!")
    }
}
```

Именно здесь программа проверяет содержимое проанализированного конфигурационного файла, чтобы выяснить, существуют ли там нужные ключи.

Содержимое `myConfig.yaml` выглядит так:

```
item1:
    k1:
        - true
    k2:
        - myValue
```

Выполнение `readYAML.go` приведет к результатам следующего вида:

```
$ go build readYAML.go
$ ./readYAML
Using config: /Users/mtsouk/Desktop/mGo2nd/Mastering-Go-Second-
Edition/ch08/viper/myConfig.yaml
item1.val1: [true]
item1.val2: [myValue]
item3.k1 is not set!
$ mv myConfig.yaml /tmp
$ ./readYAML
Using config: /tmp/myConfig.yaml
item1.val1: [true]
item1.val2: [myValue]
item3.k1 is not set!
```

Пакет cobra

Пакет `cobra` — очень удобный и популярный Go-пакет, позволяет разрабатывать утилиты командной строки, имеющие команды, подкоманды и псевдонимы. Если вам когда-либо приходилось использовать `hugo`, `docker` или `kubectl`, то вы сразу поймете, что делает `cobra`, поскольку все эти инструменты разработаны с использованием данного пакета.

Как вы скоро увидите, команды в `cobra` могут иметь один или несколько псевдонимов. Это очень удобно, если вы хотите угодить как любителям, так и опытным пользователям. Пакет `cobra` также поддерживает постоянные и локальные флаги — соответственно флаги, доступные для всех команд, и флаги, доступные только для одной команды. Для синтаксического анализа аргументов командной строки в `cobra` по умолчанию используется пакет `viper`.

Все проекты с использованием `cobra` создаются по одному шаблону. Сначала используется инструмент `cobra`, затем создаются команды, а потом вносятся необходимые изменения в сгенерированные файлы исходного кода Go, чтобы реализовать желаемую функциональность. В зависимости от сложности создаваемой утилиты таких изменений может понадобиться много. Несмотря на то что `cobra` экономит много времени, вам все равно придется писать код, который реализует желаемую функциональность.

Если хотите установить двоичную версию утилиты, вы всегда можете выполнить команду `go install` из любого каталога проекта `cobra`. По умолчанию двоичный исполняемый файл будет размещен в каталоге `~/go/bin`.

Домашняя страница пакета `cobra` находится на GitHub (https://github.com/spf13/cobra).

Простой пример cobra

В этом разделе реализуем простую утилиту командной строки, использующую `cobra` и инструмент `~/go/bin/cobra`, который поставляется в комплекте с пакетом. Если выполнить команду `~/go/bin/cobra` без аргументов командной строки, то получим следующий результат:

```
$ ~/go/bin/cobra
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.
Usage:
  cobra [command]
Available Commands:
  add         Add a command to a Cobra Application
  help        Help about any command
  init        Initialize a Cobra Application
Flags:
  -a, --author string     author name for copyright attribution
                          (default "YOUR NAME")
      --config string     config file (default is $HOME/.cobra.yaml)
  -h, --help              help for cobra
  -l, --license string    name of license for the project
      --viper             use Viper for configuration (default true)
Use "cobra [command] --help" for more information about a command.
```

Теперь, получив эту информацию, мы создадим новый проект `cobra`:

```
$ ~/go/bin/cobra init cli
Your Cobra application is ready at
/Users/mtsouk/go/src/cli
Give it a try by going there and running `go run main.go`.
Add commands to it by running `cobra add [cmdname]`.
$ cd ~/go/src/cli
$ ls -l
total 32
-rw-r--r--  1 mtsouk  staff    11358 Mar 13 09:51 LICENSE
drwxr-xr-x  3 mtsouk  staff       96 Mar 13 09:51 cmd
-rw-r--r--  1 mtsouk  staff      669 Mar 13 09:51 main.go
$ ~/go/bin/cobra add cmdOne
cmdOne created at /Users/mtsouk/go/src/cli/cmd/cmdOne.go
$ ~/go/bin/cobra add cmdTwo
cmdTwo created at /Users/mtsouk/go/src/cli/cmd/cmdTwo.go
```

По команде `cobra init` в каталоге `~/go/src` создается новый проект `cobra`, название которого соответствует последнему аргументу команды (`cli`). По команде `cobra add` в инструмент командной строки добавляется новая команда и создаются все необходимые файлы и функции Go для выполнения этой команды.

Таким образом, при каждом запуске команды `cobra add` Cobra выполняет за нас большую часть грязной работы. Однако вам все равно придется реализовать функциональность добавленных команд — в данном случае это команды `cmdOne` и `cmdTwo`. Команда `cmdOne` принимает локальный флаг командной строки с именем `number` — чтобы эта функция работала, потребуется написать дополнительный код.

Теперь, если выполнить команду `go run main.go`, то получим следующий результат:

```
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.
Usage:
  cli [command]
Available Commands:
  cmdOne    A brief description of your command
  cmdTwo    A brief description of your command
  help        Help about any command
Flags:
      --config string   config file (default is $HOME/.cli.yaml)
  -h, --help            help for cli
  -t, --toggle          Help message for toggle
Use "cli [command] --help" for more information about a command.
```

Код Go для реализации команды `cmdOne` находится в файле `./cmd/cmdOne.go`, а для команды `cmdTwo` — в файле `./cmd/cmdTwo.go`.

Окончательная версия `./cmd/cmdOne.go` выглядит так:

```
package cmd

import (
    "fmt"
    "github.com/spf13/cobra"
)

// cmdOneCmd represents the cmdOne command
var cmdOneCmd = &cobra.Command{
    Use: "cmdOne",
    Short: "A brief description of your command",
    Long: `A longer description that spans multiple lines and likely
contains examples
and usage of using your command. 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.`,
    Run: func(cmd *cobra.Command, args []string) {
        fmt.Println("cmdOne called!")
        number, _ := cmd.Flags().GetInt("number")
        fmt.Println("Going to use number", number)
        fmt.Printf("Square: %d\n", number*number)
    },
}

func init() {
    rootCmd.AddCommand(cmdOneCmd)
    cmdOneCmd.Flags().Int("number", 0, "A help for number")
}
```

Данный код Go не содержит комментариев, которые автоматически генерируются пакетом `cobra`. В следующей строке кода из функции `init()` определяется новый флаг командной строки:

```
cmdOneCmd.Flags().Int("number", 0, "A help for number")
```

Этот флаг называется `number` и используется в блоке `cobra.Command` следующим образом:

```
number, _ := cmd.Flags().GetInt("number")
```

Теперь мы можем использовать переменную `number` любым удобным способом.

Окончательная версия кода Go из файла `./cmd/cmdTwo.go`, которая включает в себя комментарии и информацию о лицензии, выглядит так:

```
// Copyright © 2019 NAME HERE <EMAIL ADDRESS>
//
// Licensed under the Apache License, Version 2.0 (the "License");
// you may not use this file except in compliance with the License.
// You may obtain a copy of the License at
//
//    http://www.apache.org/licenses/LICENSE-2.0
//
// Unless required by applicable law or agreed to in writing, software
// distributed under the License is distributed on an "AS IS" BASIS,
// WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
// See the License for the specific language governing permissions and
// limitations under the License.

package cmd

import (
    "fmt"
    "github.com/spf13/cobra"
)
```

```
// cmdTwoCmd represents the cmdTwo command
var cmdTwoCmd = &cobra.Command{
    Use: "cmdTwo",
    Short: "A brief description of your command",
    Long: `A longer description that spans multiple lines and likely
contains examples
and usage of using your command. 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.`,
    Run: func(cmd *cobra.Command, args []string) {
        fmt.Println("cmdTwo called!")
    },
}

func init() {
    rootCmd.AddCommand(cmdTwoCmd)

    // Here you will define your flags and configuration settings.

    // Cobra supports Persistent Flags which will work for this command
    // and all subcommands, e.g.:
    // cmdTwoCmd.PersistentFlags().String("foo", "", "A help for foo")

    // Cobra supports local flags which will only run when this command
    // is called directly, e.g.:
    // cmdTwoCmd.Flags().BoolP("toggle", "t", false, "Help message for toggle")
}
```

Это реализация по умолчанию команды `cmdTwo`, сгенерированной пакетом `cobra`.

При выполнении команды `cli` получим результат следующего вида:

```
$ go run main.go cmdOne
cmdOne called!
Going to use number 0
Square: 0
$ go run main.go cmdOne --number -20
cmdOne called!
Going to use number -20
Square: 400
$ go run main.go cmdTwo
cmdTwo called!
```

Если предоставить инструменту неверные входные данные, то он выдаст такие сообщения об ошибках:

```
$ go run main.go cmdThree
Error: unknown command "cmdThree" for "cli"
Run 'cli --help' for usage.
```

```
unknown command "cmdThree" for "cli"
exit status 1
$ go run main.go cmdOne --n -20
Error: unknown flag: --n
Usage:
  cli cmdOne [flags]
Flags:
  -h, --help           help for cmdOne
      --number int       A help for number
Global Flags:
      --config string    config file (default is $HOME/.cli.yaml)
unknown flag: --n
exit status 1
```

Автоматически генерируемое справочное сообщение для команды `cmdOne` выглядит так:

```
$ go run main.go cmdOne --help
A longer description that spans multiple lines and likely contains examples
and usage of using your command. 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.
Usage:
  cli cmdOne [flags]
Flags:
  -h, --help            help for cmdOne
      --number int      A help for number
Global Flags:
      --config string    config file (default is $HOME/.cli.yaml)
```

Структуру каталогов и файлов окончательной версии утилиты можно представить следующим образом с помощью команды `tree(1)`:

```
$ tree
.
├── LICENSE
├── cmd
│    ├── cmdOne.go
│    ├── cmdTwo.go
│    └── root.go
└── main.go
1 directory, 5 files
```

Создание псевдонимов команд

В этом подразделе показано, как с помощью `cobra` создавать псевдонимы для уже существующих команд.

Как и прежде, сначала нам нужно создать новый проект `cobra`, который в данном случае будет называться `aliases`, а также желаемые команды. Это можно сделать так:

```
$ ~/go/bin/cobra init aliases
$ cd ~/go/src/aliases
$ ~/go/bin/cobra add initialization
initialization created at
/Users/mtsouk/go/src/aliases/cmd/initialization.go
$ ~/go/bin/cobra add preferences
preferences created at /Users/mtsouk/go/src/aliases/cmd/preferences.go
```

Итак, теперь у нас есть утилита командной строки, которая поддерживает две команды: `initialization` и `preferences`.

Все псевдонимы существующих команд `cobra` должны быть четко указаны в коде Go. Для команды `initialization` нам понадобится следующая строка кода из файла `./cmd/initialization.go`:

```
Aliases: []string{"initialize", "init"},
```

Этот оператор создает два псевдонима для команды `initialization`: `initialize` и `init`. Содержимое окончательной версии `./cmd/initialization.go` без учета комментариев будет следующим:

```
package cmd

import (
    "fmt"
    "github.com/spf13/cobra"
)

var initializationCmd = &cobra.Command{
    Use:      "initialization",
    Aliases:  []string{"initialize", "init"},
    Short:    "A brief description of your command",
    Long:     `A longer description of your command`,
    Run:      func(cmd *cobra.Command, args []string) {
        fmt.Println("initialization called")
    },
}

func init() {
    rootCmd.AddCommand(initializationCmd)
}
```

Аналогично для команды `preferences` нужно добавить в файл `./cmd/preferences.go` следующую строку кода Go:

```
Aliases: []string{"prefer", "pref", "prf"},
```

Это утверждение создает три псевдонима для команды `preferences`: `prefer`, `pref` и `prf`.

В итоге окончательная версия `./cmd/preferences.go` без учета комментариев выглядит так:

```
package cmd

import (
    "fmt"
    "github.com/spf13/cobra"
)

var preferencesCmd = &cobra.Command{
    Use:      "preferences",
    Aliases:  []string{"prefer", "pref", "prf"},
    Short:    "A brief description of your command",
    Long:     `A longer description of your command`,
    Run:      func(cmd *cobra.Command, args []string) {
        fmt.Println("preferences called")
    },
}

func init() {
    rootCmd.AddCommand(preferencesCmd)
}
```

Выполнение утилиты командной строки `aliases` приведет к результатам такого вида:

```
$ go run main.go prefer
preferences called
$ go run main.go prf
preferences called
$ go run main.go init
initialization called
```

Если вы дадите псевдониму ошибочную команду, то он выдаст сообщение об ошибке следующего типа:

```
$ go run main.go inits
Error: unknown command "inits" for "aliases"
Run 'aliases --help' for usage.
unknown command "inits" for "aliases"
exit status 1
$ go run main.go prefeR
Error: unknown command "prefeR" for "aliases"
Did you mean this?
    preferences
Run 'aliases --help' for usage.
unknown command "prefeR" for "aliases"
Did you mean this?
    preferences
exit status 1
```

Утилита `tree(1)` не входит в стандартную комплектацию большинства систем UNIX, поэтому ее приходится устанавливать отдельно. Она поможет нам получить представление о структуре каталогов и файлов созданной утилиты `cobra`:

```
$ tree
.
├── LICENSE
├── cmd
│   ├── initialization.go
│   ├── preferences.go
│   └── root.go
└── main.go
1 directory, 5 files
```

В этой главе представлены далеко не все возможности viper и cobra.

Интерфейсы io.Reader и io.Writer

Как отмечено в предыдущей главе, для соответствия интерфейсу `io.Reader` тип данных требует реализации метода `Read()`, а для соответствия интерфейсу `io.Writer` — метода `Write()`. Оба эти интерфейса очень популярны в Go, и мы скоро их применим.

Буферизованный и небуферизованный ввод и вывод в файл

Буферизованный ввод и вывод в файл происходит в том случае, когда существует буфер для временного хранения данных перед чтением данных или записью. Таким образом, вместо того чтобы читать файл побайтно, мы одновременно считываем несколько байтов. Мы помещаем эти данные в буфер и ждем, пока кто-нибудь их прочтет желаемым способом. Небуферизованный ввод и вывод в файл происходит тогда, когда у нас нет буфера для временного хранения данных перед их фактическим чтением или записью.

Возможно, вы спросите: как решить, когда использовать буферизованный, а когда небуферизованный ввод и вывод в файл? При работе с критически важными данными, как правило, лучше выбрать небуферизованный ввод и вывод, поскольку буферизованное чтение данных может привести к устареванию данных, а буферизованная запись — к потере данных в случае отключения питания компьютера. Однако в большинстве случаев на данный вопрос нет однозначного ответа. Это означает, что мы можем использовать любой вариант, если он облегчает выполнение нашей задачи.

Пакет bufio

Как следует из названия, пакет `bufio` предназначен для буферизованного ввода и вывода данных (buffered input and output). Однако внутри пакета `bufio` все равно используются объекты типа `io.Reader` и `io.Writer`, для которых в пакете создается обертка, так что получаются объекты типа `bufio.Reader` и `bufio.Writer` соответственно. Как станет ясно из следующих подразделов, пакет `bufio` очень часто применяется для чтения текстовых файлов.

Чтение текстовых файлов

Текстовый файл — самый распространенный тип файлов, встречающихся в системах UNIX. В этом разделе вы познакомитесь с тремя способами чтения текстовых файлов: построчно, по словам и посимвольно. Вы увидите, что самым простым способом доступа к текстовому файлу является построчное чтение, в то время как чтение текстового файла по словам является самым сложным из всех методов.

Если вы внимательно посмотрите на программы `byLine.go`, `byWord.go` и `byCharacter.go`, то увидите в их коде Go много общего. Во-первых, все три утилиты читают входной файл построчно. Во-вторых, эти утилиты имеют одинаковую функцию `main()`, различающуюся только функцией, которая вызывается в цикле `for`. Наконец, три функции, которые обрабатывают входные текстовые файлы, практически идентичны, за исключением части, которая реализует фактическую функциональность.

Построчное чтение текстового файла

Проходить файл строка за строкой — самый распространенный метод чтения текстового файла. Именно поэтому мы рассмотрим его в первую очередь. Понять этот способ вам поможет код Go программы `byLine.go`, который мы разделим на три части.

Первый фрагмент `byLine.go` содержит следующий код Go:

```
package main

import (
    "bufio"
    "flag"
    "fmt"
    "io"
    "os"
)
```

Как видим, здесь включен пакет bufio, следовательно, мы будем использовать буферизованный ввод.

Вторая часть byLine.go содержит следующий код Go:

```
func lineByLine(file string) error {
    var err 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)
            break
        }
        fmt.Print(line)
    }
    return nil
}
```

Главная магия происходит в функции lineByLine(). Убедившись, что файл с таким именем открывается для чтения, мы создаем нового читателя, используя функцию bufio.NewReader(). Затем мы используем этого читателя для построчного чтения входного файла с помощью функции bufio.ReadString(). Этот прием выполняется с помощью аргумента функции bufio.ReadString(). Данный аргумент представляет собой символ, который дает bufio.ReadString() команду продолжать чтение, пока не будет найден заданный символ. Если постоянно вызывать bufio.ReadString(), задавая в качестве этого аргумента символ новой строки, то получим построчное чтение входного файла. Используя функцию fmt.Print() вместо fmt.Println() для вывода на экран считанной строки, мы можем увидеть, что в каждую считанную строку включен символ новой строки.

Третья часть byLine.go выглядит так:

```
func main() {
    flag.Parse()
    if len(flag.Args()) == 0 {
        fmt.Printf("usage: byLine <file1> [<file2> ...]\n")
        return
    }
```

```
    for _, file := range flag.Args() {
        err := lineByLine(file)
        if err != nil {
            fmt.Println(err)
        }
    }
}
```

Выполнение программы `byLine.go` и обработка ее результатов с помощью `wc(1)` приведет к результатам следующего вида:

```
$ go run byLine.go /tmp/swtag.log /tmp/adobegc.log | wc
    4761    88521    568402
```

Следующая команда позволяет убедиться в точности предыдущих результатов:

```
$ wc /tmp/swtag.log /tmp/adobegc.log
     131       693      8440 /tmp/swtag.log
    4630     87828    559962 /tmp/adobegc.log
    4761     88521    568402 total
```

Чтение текстового файла по словам

Представленный в этом подразделе способ продемонстрирован на примере программы `byWord.go`, которую мы разделим на четыре части. Как вы увидите в коде Go, разделение строки на слова — иногда сложная задача.

Первая часть этой утилиты выглядит так:

```
package main

import (
    "bufio"
    "flag"
    "fmt"
    "io"
    "os"
    "regexp"
)
```

Вторая часть `byWord.go` содержит следующий код Go:

```
func wordByWord(file string) error {
    var err 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
        }
```

Эта часть функции `wordByWord()` аналогична функции `lineByLine()` утилиты `byLine.go`.

Третья часть `byWord.go`:

```
        r := regexp.MustCompile("[^\\s]+")
        words := r.FindAllString(line, -1)
        for i := 0; i < len(words); i++ {
            fmt.Println(words[i])
        }
    }
    return nil
}
```

Последняя часть кода функции `wordByWord()` выглядит совершенно незнакомо. Здесь с помощью регулярных выражений каждая строка входного файла разделяется на отдельные слова. Регулярное выражение, определенное в операторе `regexp.MustCompile("[^\\s]+")`, предполагает, что слова отделяются одно от другого посредством пробельных символов.

Последний фрагмент кода `byWord.go` выглядит так:

```
func main() {
    flag.Parse()
    if len(flag.Args()) == 0 {
        fmt.Printf("usage: byWord <file1> [<file2> ...]\n")
        return
    }

    for _, file := range flag.Args() {
        err := wordByWord(file)
        if err != nil {
            fmt.Println(err)
        }
    }
}
```

Выполнение `byWord.go` приведет к результатам следующего вида:

```
$ go run byWord.go /tmp/adobegc.log
01/08/18
20:25:09:669
|
[INFO]
```

Чтобы проверить правильность работы byWord.go, можно воспользоваться утилитой wc(1):

```
$ go run byWord.go /tmp/adobegc.log | wc
  91591    91591    559005
$ wc /tmp/adobegc.log
   4831    91591    583454 /tmp/adobegc.log
```

Как видим, количество слов, вычисленных с помощью wc(1), равно количеству строк и слов, которые мы получили, выполнив byWord.go.

Посимвольное чтение текстового файла

В этом разделе показано, как прочитать текстовый файл посимвольно, что на практике приходится делать довольно редко, если только вы не разрабатываете текстовый редактор. Соответствующий код Go находится в файле byCharacter.go. Разделим его на четыре части.

Первая часть byCharacter.go содержит следующий код Go:

```
package main

import (
    "bufio"
    "flag"
    "fmt"
    "io"
    "os"
)
```

Как видим, для этой задачи нам не придется использовать регулярные выражения.

Второй фрагмент кода byCharacter.go выглядит так:

```
func charByChar(file string) error {
    var err 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
        }
```

В третьей части byCharacter.go заключается логика программы:

```
        for _, x := range line {
            fmt.Println(string(x))
        }
    }
    return nil
}
```

Здесь мы перебираем все прочитанные строки и разделяем каждую строку, используя range, который возвращает два значения. Первое из них, которое является местоположением текущего символа в строковой переменной, мы отбрасываем и используем второе. Однако это значение не является символом, поэтому нам необходимо преобразовать его в символ с помощью функции string().

Обратите внимание, что оператор fmt.Println(string(x)) выводит каждый символ в отдельной строке и поэтому вывод программы будет длинным. Для того чтобы сделать его более компактным, нужно использовать функцию fmt.Print().

Последняя часть byCharacter.go содержит следующий код Go:

```
func main() {
    flag.Parse()
    if len(flag.Args()) == 0 {
        fmt.Printf("usage: byChar <file1> [<file2> ...]\n")
        return
    }

    for _, file := range flag.Args() {
        err := charByChar(file)
        if err != nil {
            fmt.Println(err)
        }
    }
}
```

Выполнение byCharacter.go приведет к результатам следующего вида:

```
$ go run byCharacter.go /tmp/adobegc.log
0
1
/
0
8
/
1
8
```

Обратите внимание, что представленный здесь код Go можно использовать для подсчета количества символов, найденных во входном файле, что можно использовать для реализации Go-версии удобной утилиты командной строки wc(1).

Чтение из /dev/random

В этом подразделе показано, как читать данные с устройства `/dev/random`. Назначение системного устройства `/dev/random` — генерировать случайные данные, которые можно использовать для тестирования программ или, как в данной ситуации, в качестве начального числа для генератора случайных чисел. Получить данные из `/dev/random` иногда немного сложнее, и именно поэтому данная тема выделена в отдельный раздел.

На компьютере с macOS Mojave файл `/dev/random` имеет следующие полномочия доступа:

```
$ ls -l /dev/random
crw-rw-rw- 1 root wheel 14, 0 Mar 12 20:24 /dev/random
```

Аналогично на компьютере с Debian Linux для системного устройства `/dev/random` установлены следующие полномочия доступа:

```
$ ls -l /dev/random
crw-rw-rw- 1 root root 1, 8 Oct 13 12:19 /dev/random
```

Это означает, что в обоих вариантах UNIX файл `/dev/random` имеет примерно одинаковые полномочия доступа. Единственное различие между этими вариантами UNIX — это группа UNIX, к которой принадлежит файл: `wheel` в macOS и `root` в Debian Linux.

Программа, которая рассмотрена далее, называется `devRandom.go`. Разделим ее на три части. Первая часть программы выглядит так:

```
package main

import (
    "encoding/binary"
    "fmt"
    "os"
)
```

Для чтения данных из устройства `/dev/random` нужно импортировать стандартный Go-пакет `encoding/binary`, поскольку `/dev/random` предоставляет двоичные данные, которые необходимо декодировать. Вторая часть кода `devRandom.go` будет такой:

```
func main() {
    f, err := os.Open("/dev/random")
    defer f.Close()

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

Мы открываем `/dev/random` как обычный файл, поскольку все в UNIX — это файлы.

Последний фрагмент `devRandom.go` содержит следующий код Go:

```
    var seed int64
    binary.Read(f, binary.LittleEndian, &seed)
    fmt.Println("Seed:", seed)
}
```

Для чтения данных с системного устройства `/dev/random` нам нужна функция `binary.Read()`, которая принимает три аргумента. Значение второго из них (`binary.LittleEndian`) указывает, что мы хотим использовать *прямой порядок* байтов. Второй из возможных вариантов, `binary.BigEndian`, выбирается в тех случаях, когда на компьютере используется *обратный порядок* байтов.

Выполнение `devRandom.go` приведет к результатам следующего вида:

```
$ go run devRandom.go
Seed: -2044736418491485077
$ go run devRandom.go
Seed: -5174854372517490328
$ go run devRandom.go
Seed: 7702177874251412774
```

Чтение заданного количества данных

В этом разделе показано, как прочитать именно столько данных, сколько нам нужно. Такой метод особенно полезен при чтении двоичных файлов, когда требуется декодировать считанные данные определенным образом. Однако это работает и для текстовых файлов.

Логика метода заключается в следующем: мы создаем *байтовый срез* нужного размера и используем его для чтения данных. Чтобы было еще интереснее, мы реализуем этот функционал как функцию с двумя параметрами: один параметр будет указывать количество данных, которые мы хотим прочитать, а второй (с типом `*os.File`) будет использоваться для доступа к нужному файлу. Функция будет возвращать прочитанные данные.

Программа Go, которую мы рассмотрим, называется `readSize.go`. Разделим ее на четыре части. Эта утилита принимает единственный аргумент — размер байтового среза.

Программа, реализующая метод, описанный в этом разделе, поможет вам скопировать любой файл, используя соответствующий размер буфера.

Первая часть readSize.go содержит ожидаемую преамбулу:

```
package main

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

Вторая часть readSize.go содержит следующий код Go:

```
func readSize(f *os.File, size int) []byte {
    buffer := make([]byte, size)

    n, err := f.Read(buffer)
    if err == io.EOF {
        return nil
    }

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

    return buffer[0:n]
}
```

Эта функция нам уже встречалась. Ее код прост, однако есть один момент, который необходимо пояснить. Метод `io.Reader.Read()` возвращает два значения: количество прочитанных байтов и переменную `error`. Функция `readSize()`, как и раньше, возвращает то же значение, которое возвращает `io.Read()`, чтобы вернуть байтовый срез заданного размера. Эта информация важна только тогда, когда достигнут конец файла. Однако она гарантирует, что данные, выводимые утилитой, будут такими же, как и входные данные, и в них не будет содержаться никаких дополнительных символов.

Наконец, остается код, который проверяет наличие `io.EOF`, что является ошибкой и означает, что мы достигли конца файла. Когда возникает такая ошибка, функция завершает работу. Дейв Чейни (Dave Cheney), участник проекта Go и разработчик программного обеспечения с открытым кодом, называет такие ошибки «ошибками дозорного», поскольку они означают, что на самом деле ошибки не произошло.

Третья часть кода этой утилиты выглядит так:

```
func main() {
    arguments := os.Args
    if len(arguments) != 3 {
        fmt.Println("<buffer size> <filename>")
```

```
        return
    }

    bufferSize, err := strconv.Atoi(os.Args[1])
    if err != nil {
        fmt.Println(err)
        return
    }

    file := os.Args[2]
    f, err := os.Open(file)
    if err != nil {
        fmt.Println(err)
        return
    }
    defer f.Close()
```

Последний фрагмент кода `readSize.go`:

```
    for {
        readData := readSize(f, bufferSize)
        if readData != nil {
            fmt.Print(string(readData))
        } else {
            break
        }
    }
}
```

Здесь мы продолжаем чтение входного файла, пока функция `readSize()` не вернет ошибку или `nil`.

Выполнение `readSize.go` с аргументом, указывающим на обработку двоичного файла, и последующая обработка результата с помощью `wc(1)` позволяют проверить правильность работы программы:

```
$ go run readSize.go 1000 /bin/ls | wc
      80    1032      38688
$ wc /bin/ls
      80    1032      38688 /bin/ls
```

Преимущества двоичных форматов

В предыдущем разделе на примере утилиты `readSize.go` показано, как читать файл побайтно — такой способ обычно применяется к двоичным файлам. Возможно, вы спросите: зачем читать данные в двоичном формате, когда текстовые форматы намного понятнее? Основная причина — экономия места. Представьте, что вы хотите сохранить в файле число 20 в виде строки. Нетрудно понять, что для хранения числа 20 в формате символов ASCII потребуется два байта: один для хранения символа 2, а другой — для хранения символа 0.

Для хранения числа 20 в двоичном формате достаточно всего одного байта, поскольку 20 можно представить как 00010100 в двоичном формате или как 0x14 в шестнадцатеричном формате.

Это различие может показаться незначительным, пока речь идет о небольших объемах данных, но оно существенно, когда речь заходит о данных, используемых в таких приложениях, как серверы баз данных.

Чтение CSV-файлов

CSV-файлы — это простые форматированные текстовые файлы. В этом разделе показано, как прочитать текстовый файл, в котором находятся координаты точек на плоскости, — другими словами, в каждой строке содержится пара координат. Кроме того, мы также воспользуемся внешней библиотекой Go, которая называется *Glot*, — она поможет нам построить график по точкам, координаты которых будут прочитаны из CSV-файла. Обратите внимание, что в Glot используется *Gnuplot*, поэтому, чтобы использовать Glot, придется установить библиотеку Gnuplot на ваш компьютер UNIX.

Файл с программой, которую мы рассмотрим в этом разделе, называется `CSVplot.go`. Разделим его на пять частей. Первая из них выглядит так:

```
package main

import (
    "encoding/csv"
    "fmt"
    "github.com/Arafatk/glot"
    "os"
    "strconv"
)
```

Вторая часть `CSVplot.go` содержит следующий код Go:

```
func main() {
    if len(os.Args) != 2 {
        fmt.Println("Need a data file!")
        return
    }

    file := os.Args[1]
    _, err := os.Stat(file)
    if err != nil {
        fmt.Println("Cannot stat", file)
        return
    }
```

В этом коде показано, как с помощью мощной функции `os.Stat()` проверить, существует ли данный файл.

Третья часть `CSVplot.go` выглядит так:

```
    f, err := os.Open(file)
    if err != nil {
        fmt.Println("Cannot open", file)
        fmt.Println(err)
        return
    }

    defer f.Close()
    reader := csv.NewReader(f)
    reader.FieldsPerRecord = -1
    allRecords, err := reader.ReadAll()
    if err != nil {
        fmt.Println(err)
        return
    }
```

Четвертый фрагмент `CSVplot.go` содержит следующий код Go:

```
    xP := []float64{}
    yP := []float64{}
    for _, rec := range allRecords {
        x, _ := strconv.ParseFloat(rec[0], 64)
        y, _ := strconv.ParseFloat(rec[1], 64)
        xP = append(xP, x)
        yP = append(yP, y)
    }

    points := [][]float64{}
    points = append(points, xP)
    points = append(points, yP)
    fmt.Println(points)
```

Здесь мы преобразуем прочитанные строковые значения в числа, которые затем помещаем в двумерный срез с именем `points`.

Последняя часть `CSVplot.go` содержит следующий код Go:

```
    dimensions := 2
    persist := true
    debug := false
    plot, _ := glot.NewPlot(dimensions, persist, debug)

    plot.SetTitle("Using Glot with CSV data")
    plot.SetXLabel("X-Axis")
    plot.SetYLabel("Y-Axis")
    style := "circle"
    plot.AddPointGroup("Circle:", style, points)
    plot.SavePlot("output.png")
}
```

В этом коде Go показано, как создать PNG-файл с помощью библиотеки Glot и ее функции `glot.SavePlot()`.

Нетрудно догадаться, что, прежде чем вы сможете скомпилировать и выполнить код `CSVplot.go`, вам нужно загрузить библиотеку Glot, для чего необходимо выполнить из оболочки UNIX следующую команду:

```
$ go get github.com/Arafatk/glot
```

Исходный CSV-файл, где содержатся координаты точек, по которым будет построен график, имеет следующий вид:

```
$ cat /tmp/dataFile
1,2
2,3
3,3
4,4
5,8
6,5
-1,12
-2,10
-3,10
-4,10
```

Выполнение `CSVplot.go` приведет к следующим результатам:

```
$ go run CSVplot.go /tmp/doesNoExist
Cannot stat /tmp/doesNoExist
$ go run CSVplot.go /tmp/dataFile
[[1 2 3 4 5 6 -1 -2 -3 -4] [2 3 3 4 8 5 12 10 10 10]]
```

Результаты работы `CSVplot.go` лучше видны на рис. 8.1.

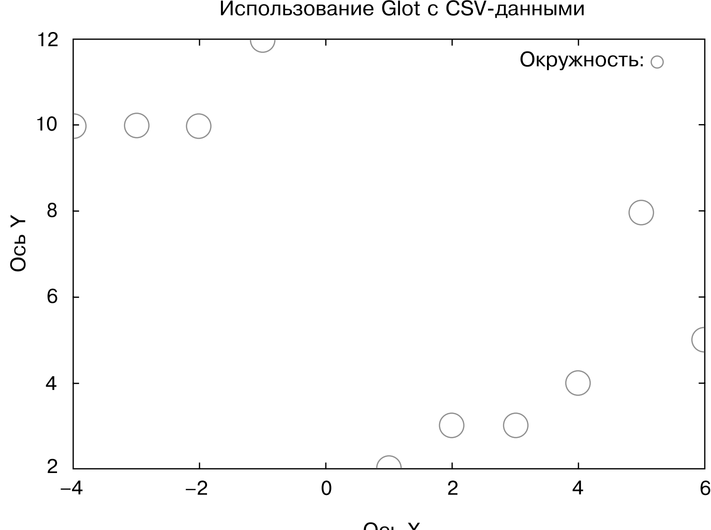


Рис. 8.1. График, который можно получить с помощью Glot

Запись в файл

Для записи данных в файлы на диске можно использовать функциональность интерфейса `io.Writer`. Тем не менее, на примере кода Go из программы `save.go` рассмотрим пять разных способов записи данных в файл. Для этого мы разделим программу `save.go` на шесть частей.

Первая часть `save.go` выглядит так:

```
package main

import (
    "bufio"
    "fmt"
    "io"
    "io/ioutil"
    "os"
)
```

Вторая часть `save.go` содержит следующий код Go:

```
func main() {
    s := []byte("Data to write\n")

    f1, err := os.Create("f1.txt")
    if err != nil {
        fmt.Println("Cannot create file", err)
        return
    }
    defer f1.Close()
    fmt.Fprintf(f1, string(s))
```

Обратите внимание, что мы будем использовать байтовый срез `s` в каждой строке этой программы Go, где производится запись. Кроме того, применяемая здесь функция `fmt.Fprintf()` позволяет записывать данные в ваши собственные файлы журнала, используя нужный формат. В нашем случае `fmt.Fprintf()` записывает данные в файл, обозначенный как `f1`.

В третьей части `save.go` содержится следующий код Go:

```
    f2, err := os.Create("f2.txt")
    if err != nil {
        fmt.Println("Cannot create file", err)
        return
    }
    defer f2.Close()
    n, err := f2.WriteString(string(s))
    fmt.Printf("wrote %d bytes\n", n)
```

В данном случае для записи данных в файл используется функция `f2.WriteString()`.

Четвертый фрагмент кода `save.go` выглядит так:

```
    f3, err := os.Create("f3.txt")
    if err != nil {
        fmt.Println(err)
        return
    }
    w := bufio.NewWriter(f3)
    n, err = w.WriteString(string(s))
    fmt.Printf("wrote %d bytes\n", n)
    w.Flush()
```

В данном случае функция `bufio.NewWriter()` открывает файл для записи, а `bufio.WriteString()` записывает данные.

В пятой части `save.go` показан другой метод записи в файл:

```
    f4 := "f4.txt"
    err = ioutil.WriteFile(f4, s, 0644)
    if err != nil {
        fmt.Println(err)
        return
    }
```

Этот метод требует только одного вызова функции `ioutil.WriteFile()` для записи данных, и не требует использования `os.Create()`.

Последний фрагмент кода `save.go` выглядит так:

```
    f5, err := os.Create("f5.txt")
    if err != nil {
        fmt.Println(err)
        return
    }
    n, err = io.WriteString(f5, string(s))
    if err != nil {
        fmt.Println(err)
        return
    }
    fmt.Printf("wrote %d bytes\n", n)
}
```

В этом методе для записи данных в файл использована функция `io.WriteString()`.

Выполнение `save.go` приведет к следующим результатам:

```
$ go run save.go
wrote 14 bytes
wrote 14 bytes
wrote 14 bytes
$ ls -l f?.txt
-rw-r--r-- 1 mtsouk staff 14 Jan 23 20:30 f1.txt
```

```
-rw-r--r-- 1 mtsouk staff 14 Jan 23 20:30 f2.txt
-rw-r--r-- 1 mtsouk staff 14 Jan 23 20:30 f3.txt
-rw-r--r-- 1 mtsouk staff 14 Jan 23 20:30 f4.txt
-rw-r--r-- 1 mtsouk staff 14 Jan 23 20:30 f5.txt
$ cat f?.txt
Data to write
Data to write
Data to write
Data to write
Data to write
```

В следующем разделе показано, как сохранить данные в файле с помощью специализированной функции пакета, который входит в состав стандартной библиотеки Go.

Загрузка и сохранение данных на диске

Помните ли вы приложение `keyValue.go` из главы 4? Оно, правда, было далеко от совершенства. В этом разделе вы узнаете, как записать данные из *хранилища «ключ — значение»* на диск и как загрузить их обратно в память при следующем запуске приложения.

Мы создадим две новые функции: одну, с именем `save()`, для сохранения данных на диск, и вторую, с именем `load()`, для получения данных с диска. Таким образом, мы представим здесь только различия между кодом `keyValue.go` и `kvSaveLoad.go` с помощью UNIX-утилиты командной строки `diff(1)`.

UNIX-утилита командной строки diff(1) бывает очень удобной, если нужно найти различия между двумя текстовыми файлами. Чтобы узнать о ней больше, введите man 1 diff в командной строке оболочки UNIX.

Если вы зададитесь вопросом, как реализовать эту задачу, то поймете, что для ее решения вам, в сущности, нужен простой способ сохранить содержимое отображения Go на диске, а также способ загрузить данные из файла и поместить их в хеш-таблицу Go.

Процесс преобразования данных в поток байтов называется *сериализацией*. Процесс чтения файла данных и преобразования этих данных в объект называется *десериализацией*. Для сериализации и десериализации в программе `kvSaveLoad.go` используется стандартный Go-пакет `encoding/gob`. Пакет `encoding/gob` сохраняет данные в формате `gob`. Официально такие форматы называются *потоковым кодированием*. Преимущество формата `gob` заключается в том, что Go выполняет всю грязную работу, и вам не приходится беспокоиться об этапах кодирования и декодирования.

В Go для сериализации и десериализации данных также используются такие пакеты, как `encoding/xml`, который преобразует данные в *формат XML*, и `encoding/json`, который сохраняет данные в *формате JSON*.

Далее показаны различия в коде `kvSaveLoad.go` и `keyValue.go`, без учета реализаций функций `save()` и `load()`, которые мы внимательно рассмотрим позже:

```
$ diff keyValue.go kvSaveLoad.go
4a5
>    "encoding/gob"
16a18,55
> var DATAFILE = "/tmp/dataFile.gob"
> func save() error {
>
>        return nil
> }
>
> func load() error {
>
> }
59a99,104
>
>        err := load()
>        if err != nil {
>            fmt.Println(err)
>        }
>
88a134,137
>            err = save()
>            if err != nil {
>                fmt.Println(err)
>            }
```

Важной частью результатов, полученных благодаря `diff(1)`, является определение глобальной переменной `DATAFILE`. Эта переменная содержит путь к файлу, который используется для хранилища данных «ключ — значение». Помимо этого, здесь видно, где вызывается функция `load()`, а также в какой точке вызывается `save()`. Сначала в функции `main()` используется `load()`, а затем, когда пользователь запускает команду `STOP`, выполняется `save()`.

При каждом запуске `kvSaveLoad.go` программа проверяет, есть ли данные для чтения, пытаясь прочитать файл данных, предлагаемый по умолчанию. Если файла данных для чтения нет, то программа начнет работу с пустого хранилища «ключ — значение». Перед завершением работы программа записывает все данные на диск с помощью функции `save()`.

Реализация `save()` выглядит так:

```
func save() error {
    fmt.Println("Saving", DATAFILE)
```

```
    err := os.Remove(DATAFILE)
    if err != nil {
        fmt.Println(err)
    }

    saveTo, err := os.Create(DATAFILE)
    if err != nil {
        fmt.Println("Cannot create", DATAFILE)
        return err
    }
    defer saveTo.Close()

    encoder := gob.NewEncoder(saveTo)
    err = encoder.Encode(DATA)
    if err != nil {
        fmt.Println("Cannot save to", DATAFILE)
        return err
    }
    return nil
}
```

Обратите внимание: функция `save()` прежде всего удаляет существующий файл данных с помощью функции `os.Remove()`, чтобы позже создать новый.

Одна из самых важных задач, которые выполняет функция `save()`, — убедиться, что мы действительно можем создать и записать нужный файл. Есть много способов сделать это, и в `save()` используется самый простой из них: проверка значения `error`, возвращаемого функцией `os.Create()`. Если это значение не равно `nil`, это указывает на проблему и функция `save()` завершается без сохранения каких-либо данных.

Функция `load()` реализована следующим образом:

```
func load() error {
    fmt.Println("Loading", DATAFILE)
    loadFrom, err := os.Open(DATAFILE)
    defer loadFrom.Close()
    if err != nil {
        fmt.Println("Empty key/value store!")
        return err
    }

    decoder := gob.NewDecoder(loadFrom)
    decoder.Decode(&DATA)
    return nil
}
```

Одна из задач функции `load()` — убедиться, что файл, который мы пытаемся прочитать, действительно существует и его без проблем можно прочитать.

В функции `load()` также используется самый простой подход, который заключается в просмотре значения, возвращаемого функцией `os.Open()`. Если возвращаемое значение `error` равно `nil`, то все в порядке.

Важно также закрыть файл после чтения данных, поскольку впоследствии он будет перезаписан функцией `save()`. Освобождение файла выполняется с помощью инструкции `defer loadFrom.Close()`.

При выполнении программы `kvSaveLoad.go` будут получены результаты такого вида:

```
$ go run kvSaveLoad.go
Loading /tmp/dataFile.gob
Empty key/value store!
open /tmp/dataFile.gob: no such file or directory
ADD 1 2 3
ADD 4 5 6
STOP
Saving /tmp/dataFile.gob
remove /tmp/dataFile.gob: no such file or directory
$ go run kvSaveLoad.go
Loading /tmp/dataFile.gob
PRINT
key: 1 value: {2 3 }
key: 4 value: {5 6 }
DELETE 1
PRINT
key: 4 value: {5 6 }
STOP
Saving /tmp/dataFile.gob
rMacBook:code mtsouk$ go run kvSaveLoad.go
Loading /tmp/dataFile.gob
PRINT
key: 4 value: {5 6 }
STOP
Saving /tmp/dataFile.gob
$ ls -l /tmp/dataFile.gob
-rw-r--r-- 1 mtsouk wheel 80 Jan 22 11:22 /tmp/dataFile.gob
$ file /tmp/dataFile.gob
/tmp/dataFile.gob: data
```

В главе 13 представлена окончательная версия хранилища «ключ — значение», которое способно работать через соединение TCP/IP и обслуживать несколько сетевых клиентов посредством *горутин*.

И снова пакет strings

Удобный пакет `strings` вам впервые встретился в главе 4. В этом разделе мы рассмотрим функции пакета `strings`, связанные с вводом и выводом данных в файл.

Первая часть программы `str.go` содержит следующий код Go:

```
package main

import (
    "fmt"
    "io"
    "os"
    "strings"
)
```

Второй фрагмент кода `str.go` выглядит так:

```
func main() {
    r := strings.NewReader("test")
    fmt.Println("r length:", r.Len())
```

Функция `strings.NewReader()` создает на основе строки объект `Reader`, предназначенный только для чтения. Объект `strings.Reader` реализует интерфейсы `io.Reader`, `io.ReaderAt`, `io.Seeker`, `io.WriterTo`, `io.ByteScanner` и `io.RuneScanner`.

Третья часть `str.go` выглядит таким образом:

```
    b := make([]byte, 1)
    for {
        n, err := r.Read(b)
            if err == io.EOF {
            break
        }

        if err != nil {
            fmt.Println(err)
            continue
        }
        fmt.Printf("Read %s Bytes: %d\n", b, n)
    }
```

Здесь показано, как можно использовать `strings.Reader` в качестве `io.Reader` для побайтного чтения строки с помощью функции `Read()`.

Последняя часть `str.go` содержит следующий код Go:

```
    s := strings.NewReader("This is an error!\n")
    fmt.Println("r length:", s.Len())
    n, err := s.WriteTo(os.Stderr)

    if err != nil {
        fmt.Println(err)
        return
    }
    fmt.Printf("Wrote %d bytes to os.Stderr\n", n)
}
```

В этом фрагменте кода показано, как записать сообщение в стандартный канал вывода ошибки с помощью пакета `strings`.

Выполнение `str.go` приведет к следующему результату:

```
$ go run str.go
r length: 4
Read t Bytes: 1
Read e Bytes: 1
Read s Bytes: 1
Read t Bytes: 1
r length: 18
This is an error!
Wrote 18 bytes to os.Stderr
$ go run str.go 2>/dev/null
r length: 4
Read t Bytes: 1
Read e Bytes: 1
Read s Bytes: 1
Read t Bytes: 1
r length: 18
Wrote 18 bytes to os.Stderr
```

Пакет bytes

Подобно тому как стандартный Go-пакет `strings` помогает работать со *строками*, стандартный Go-пакет `bytes` содержит функции для работы с *байтовыми срезами*. В этом разделе мы рассмотрим программу `bytes.go`, код которой разделим на три части.

Первая часть `bytes.go` выглядит так:

```
package main

import (
    "bytes"
    "fmt"
    "io"
    "os"
)
```

Вторая часть `bytes.go` содержит следующий код Go:

```
func main() {
    var buffer bytes.Buffer
    buffer.Write([]byte("This is"))
    fmt.Fprintf(&buffer, " a string!\n")
    buffer.WriteTo(os.Stdout)
    buffer.WriteTo(os.Stdout)
```

Сначала мы создаем переменную `bytes.Buffer` и помещаем в нее данные с помощью функций `buffer.Write()` и `fmt.Fprintf()`, а затем дважды вызываем функцию `buffer.WriteTo()`.

При первом вызове `buffer.WriteTo()` будет выведено содержимое переменной `buffer`. Однако при втором вызове `buffer.WriteTo()` вывести нечего, поскольку после первого вызова `buffer.WriteTo()` переменная `buffer` станет пустой.

Последняя часть `bytes.go` выглядит так:

```
    buffer.Reset()
    buffer.Write([]byte("Mastering Go!"))
    r := bytes.NewReader([]byte(buffer.String()))
    fmt.Println(buffer.String())
    for {
        b := make([]byte, 3)
        n, err := r.Read(b)
        if err == io.EOF {
            break
        }

        if err != nil {
            fmt.Println(err)
            continue
        }
        fmt.Printf("Read %s Bytes: %d\n", b, n)
    }
}
```

Метод `Reset()` обнуляет переменную `buffer`, а метод `Write()` снова помещает в нее данные. Затем мы создаем новый объект `Reader` с помощью `bytes.NewReader()`, а после этого используем метод `Read()` интерфейса `io.Reader` для чтения данных из переменной `buffer`.

Выполнение `bytes.go` приведет к следующим результатам:

```
$ go run bytes.go
This is a string!
Mastering Go!
Read Mas Bytes: 3
Read ter Bytes: 3
Read ing Bytes: 3
Read Go Bytes: 3
Read ! Bytes: 1
```

Полномочия доступа к файлам

Популярная тема в программировании систем UNIX — полномочия доступа к файлам UNIX. В этом разделе показано, как вывести на экран полномочия для любого файла при условии, что вы обладаете необходимыми полномочиями в среде UNIX. Программа, которую мы рассмотрим, называется `permissions.go`. Разделим ее на три части.

Первая часть permissions.go содержит следующий код Go:

```
package main

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

Второй фрагмент permissions.go содержит такой код Go:

```
func main() {
    arguments := os.Args
    if len(arguments) == 1 {
        fmt.Printf("usage: permissions filename\n")
        return
    }
```

Последняя часть этой утилиты выглядит так:

```
    filename := arguments[1]
    info, _ := os.Stat(filename)
    mode := info.Mode()
    fmt.Println(filename, "mode is", mode.String()[1:10])
}
```

Вызов os.Stat(filename) возвращает структуру с большим количеством данных. Поскольку нас интересуют только права доступа к файлу, мы вызываем метод Mode() и выводим на экран его результаты. В сущности, мы выводим только их часть, обозначенную как mode.String()[1:10], поскольку именно здесь находятся интересующие нас данные.

Выполнение permissions.go приведет к результатам такого вида:

```
$ go run permissions.go /tmp/adobegc.log
/tmp/adobegc.log mode is rw-rw-rw-
$ go run permissions.go /dev/random
/dev/random mode is crw-rw-rw
```

Запустив утилиту ls(1), мы можем проверить правильность работы permissions.go:

```
$ ls -l /dev/random /tmp/adobegc.log
crw-rw-rw-  1 root  wheel    14,  0   Jan  8 20:24 /dev/random
-rw-rw-rw-  1 root  wheel    583454   Jan 16 19:12 /tmp/adobegc.log
```

Обработка сигналов в UNIX

Чтобы помочь разработчикам в обработке сигналов, Go предоставляет им пакет os/signal. В этом разделе показано, как использовать этот пакет для обработки сигналов UNIX.

Прежде всего позвольте сообщить вам некоторую полезную информацию о сигналах UNIX. Вы когда-нибудь нажимали Ctrl+C, чтобы остановить работающую программу? Если да, то вы уже знакомы с сигналами UNIX, поскольку нажатие Ctrl+C отправляет в программу *сигнал SIGINT*. Строго говоря, *сигналы* UNIX — это программные прерывания, к которым можно обращаться по имени или по номеру и которые являются способом обработки асинхронных событий в системе UNIX. Вообще, безопаснее называть сигнал по имени, потому что тогда меньше вероятность случайно отправить неправильный сигнал.

Программа не способна обработать все существующие сигналы. Некоторые из них не перехватываются, однако и проигнорировать их также нельзя. Сигналы `SIGKILL` и `SIGSTOP` нельзя перехватить, заблокировать или проигнорировать, так как они предоставляют ядру и пользователю `root` возможность остановить любой процесс. Сигнал `SIGKILL`, который имеет номер 9, обычно вызывается в экстремальных ситуациях, когда нужно действовать быстро. Это единственный сигнал, который обычно вызывается по номеру — просто потому, что так быстрее.

Сигнал signal.SIGINFO на компьютерах с Linux недоступен. Поэтому, если вы обнаружите его в программе Go, которую хотите запустить на компьютере с Linux, вам придется заменить его на другой сигнал, иначе программа Go не будет скомпилирована и выполнена.

Самый распространенный способ передачи сигнала процессу — использование утилиты `kill(1)`. По умолчанию `kill(1)` передает сигнал `SIGTERM`. Если вы хотите узнать, какие сигналы поддерживаются на вашем компьютере с UNIX, используйте команду `kill -l`.

Если вы попытаетесь отправить сигнал процессу, не имея на то необходимых полномочий, `kill(1)` не выполнит эту операцию и вы получите такое сообщение об ошибке:

```
$ kill 1210
-bash: kill: (1210) - Operation not permitted
```

Обработка двух сигналов

В этом подразделе показано, как обрабатывать два сигнала в программе Go. Для этого мы рассмотрим код, который находится в файле `handleTwo.go`. Разделим его на четыре части. Программа `handleTwo.go` обрабатывает сигналы `SIGINFO` и `SIGINT`, которые в Go называются `syscall.SIGINFO` и `os.Interrupt` соответственно.

Если вы посмотрите в документацию пакета os, то увидите, что есть только два сигнала, которые гарантированно присутствуют во всех системах. Это syscall.SIGKILL и syscall.SIGINT, которые в Go также определены как os.Kill и os.Interrupt соответственно.

Первая часть `handleTwo.go` содержит следующий код Go:

```
package main

import (
    "fmt"
    "os"
    "os/signal"
    "syscall"
    "time"
)
```

Вторая часть программы `handleTwo.go` выглядит так:

```
func handleSignal(signal os.Signal) {
    fmt.Println("handleSignal() Caught:", signal)
}
```

Функция `handleSignal()` используется для обработки сигнала `syscall.SIGINFO`, а сигнал `os.Interrupt` будет обрабатываться непосредственно.

Третий фрагмент `handleTwo.go` содержит следующий код Go:

```
func main() {
    sigs := make(chan os.Signal, 1)
    signal.Notify(sigs, os.Interrupt, syscall.SIGINFO)
    go func() {
        for {
            sig := <-sigs
            switch sig {
            case os.Interrupt:
                fmt.Println("Caught:", sig)
            case syscall.SIGINFO:
                handleSignal(sig)
                return
            }
        }
    }()
```

Этот метод работает так: сначала мы определяем канал с именем `sigs`, по которому будут передаваться данные. Затем мы вызываем функцию `signal.Notify()`, чтобы указать на интересующие нас сигналы. После этого мы реализуем *анонимную функцию*, которая выполняется как горутина и включается тогда, когда поступает любой из интересующих нас сигналов. В главе 9 вы узнаете больше о горутинах и каналах.

Последняя часть программы `handleTwo.go` выглядит так:

```
    for {
        fmt.Printf(".")
        time.Sleep(20 * time.Second)
    }
}
```

Вызов time.Sleep() не дает программе завершиться, поскольку она не выполняет никакой полезной функции. В реальном приложении не было бы необходимости использовать подобный код.

Поскольку для передачи программе сигналов с помощью утилиты kill(1) нам нужен идентификатор процесса программы, сначала мы скомпилируем handleTwo.go и вместо выполнения команды go runhandleTwo.go запустим исполняемый файл. Результатом работы handleTwo будет примерно следующее:

```
$ go build handleTwo.go
$ ls -l handleTwo
-rwxr-xr-x 1 mtsouk staff 2005200 Jan 18 07:49 handleTwo
$ ./handleTwo
.^CCaught: interrupt
.Caught: interrupt
handleSignal() Caught: information request
.Killed: 9
```

Обратите внимание, что для взаимодействия с handleTwo.go и получения показанных выше результатов вам потребуется дополнительное окно терминала, в котором нужно выполнить следующие команды:

```
$ ps ax | grep ./handleTwo | grep -v grep
47988 s003 S+ 0:00.00 ./handleTwo
$ kill -s INT 47988
$ kill -s INFO 47988
$ kill -s USR1 47988
$ kill -9 47988
```

Первая команда позволяет получить идентификатор процесса исполняемого файла handleTwo, а остальные команды используются для передачи этому процессу требуемых сигналов. Сигнал SIGUSR1 игнорируется и не появляется на выходе.

У handleTwo.go есть проблема: если программа получает сигнал, который не запрограммирован для обработки, то она его игнорирует. Поэтому в следующем подразделе представлен другой подход для более эффективной обработки сигналов.

Обработка всех сигналов

В этом подразделе показано, как обрабатывать все сигналы, но реагировать только на те, которые действительно вас интересуют. Это гораздо лучший и более безопасный метод, чем тот, что был описан в предыдущем разделе. Этот метод будет продемонстрирован на примере Go-кода handleAll.go, который мы разделим на четыре части.

Первая часть handleAll.go содержит следующий код Go:

```
package main

import (
```

```
    "fmt"
    "os"
    "os/signal"
    "syscall"
    "time"
)

func handle(signal os.Signal) {
    fmt.Println("Received:", signal)
}
```

Второй фрагмент кода `handleAll.go` выглядит так:

```
func main() {
    sigs := make(chan os.Signal, 1)
    signal.Notify(sigs)
```

Таким образом, главная магия выполняется оператором `signal.Notify(sigs)`. Поскольку никакие конкретные сигналы не указаны, будут обработаны все входящие сигналы.

Мы можем вызывать signal.Notify() несколько раз в одной и той же программе для одних и тех же сигналов, используя разные каналы. В таком случае каждый канал получит копию сигналов, на обработку которых он был запрограммирован.

Третья часть кода утилиты `handleAll.go` выглядит следующим образом:

```
    go func() {
        for {
            sig := <-sigs
            switch sig {
            case os.Interrupt:
                handle(sig)
            case syscall.SIGTERM:
                handle(sig)
                os.Exit(0)
            case syscall.SIGUSR2:
                fmt.Println("Handling syscall.SIGUSR2!")
            default:
                fmt.Println("Ignoring:", sig)
            }
        }
    }()
```

Очень удобно использовать один из сигналов для выхода из программы. Это дает возможность при необходимости убирать в программе лишнее. В данном

случае для этой цели используется сигнал `syscall.SIGTERM`, что не мешает нам использовать `SIGKILL` для остановки программы.

Остальной код Go в `handleAll.go` выглядит так:

```
    for {
        fmt.Printf(".")
        time.Sleep(30 * time.Second)
    }
}
```

Нам по-прежнему нужно вызывать `time.Sleep()`, чтобы предотвратить немедленное завершение программы.

В этом случае, как и в прошлый раз, лучше создать исполняемый файл для `handleAll.go`, используя инструмент `go build`. Запуск программы `handleAll` и взаимодействие с ней из другого окна терминала приведет к следующим результатам:

```
$ go build handleAll.go
$ ls -l handleAll
-rwxr-xr-x 1 mtsouk staff 2005216 Jan 18 08:25 handleAll
$ ./handleAll
.Ignoring: hangup
Handling syscall.SIGUSR2!
Ignoring: user defined signal 1
Received: interrupt
^CReceived: interrupt
Received: terminated
```

Во втором окне терминала нужно ввести следующие команды:

```
$ ps ax | grep ./handleAll | grep -v grep
49902 s003 S+    0:00.00  ./handleAll
$ kill -s HUP 49902
$ kill -s USR2 49902
$ kill -s USR1 49902
$ kill -s INT 49902
$ kill -s TERM 49902
```

Программирование UNIX-каналов на Go

Согласно философии UNIX, каждая утилита командной строки UNIX должна выполнять только одну задачу, и должна делать это хорошо. На практике это означает, что вместо огромных утилит, выполняющих множество задач, следует разрабатывать небольшие программы, которые в совместном применении будут выполнять желаемую работу. Самый распространенный способ взаимодействия нескольких утилит командной строки UNIX — использование каналов. В *канале UNIX* результаты работы одной утилиты командной строки становятся входными

данными для другой утилиты. Этот процесс может включать в себя более двух программ. Для обозначения каналов UNIX используется символ |.

У каналов есть два серьезных ограничения: во-первых, они обычно работают только в одном направлении, а во-вторых, их можно использовать только для процессов, имеющих общего предка. Идея, лежащая в основе реализации каналов UNIX, заключается в том, что если у вас нет файла для обработки, то следует подождать поступления данных из стандартного потока ввода.

Аналогично если нет указания сохранять результаты в файле, то их следует вывести в стандартный поток вывода — либо для того, чтобы их увидел пользователь, либо для другой программы, которая будет их обрабатывать.

В главе 1 вы узнали, как читать данные из стандартного потока ввода и как записывать их в стандартный поток вывода. Если у вас остаются сомнения по поводу этих двух операций, самое время вернуться к Go-коду `stdOUT.go` и `stdIN.go`.

Реализация утилиты cat(1) на Go

В этом подразделе продемонстрирована Go-версия утилиты `cat(1)`. Скорее всего, вас удивит размер этой программы. Мы рассмотрим код `cat.go`, разделив его на три части. Первая часть `cat.go` выглядит так:

```
package main

import (
    "bufio"
    "fmt"
    "io"
    "os"
)
```

Второй фрагмент `cat.go` содержит следующий код Go:

```
func printFile(filename string) error {
    f, err := os.Open(filename)
    if err != nil {
        return err
    }
    defer f.Close()
    scanner := bufio.NewScanner(f)
    for scanner.Scan() {
        io.WriteString(os.Stdout, scanner.Text())
        io.WriteString(os.Stdout, "\n")
    }
    return nil
}
```

В этой части программы реализована функция, цель которой — вывод содержимого файла в стандартный поток вывода.

Последняя часть `cat.go` выглядит так:

```
func main() {
    filename := ""
    arguments := os.Args
    if len(arguments) == 1 {
        io.Copy(os.Stdout, os.Stdin)
        return
    }

    for i := 1; i < len(arguments); i++ {
        filename = arguments[i]
        err := printFile(filename)
        if err != nil {
            fmt.Println(err)
        }
    }
}
```

В данном коде содержится вся магия `cat.go`: именно здесь определяется, как будет вести себя программа. Прежде всего, если выполнить `cat.go` без аргументов командной строки, то программа просто скопирует стандартный поток ввода в стандартный поток вывода, что реализовано с помощью оператора `io.Copy(os.Stdout, os.Stdin)`. Но если передать программе аргументы командной строки, то программа обработает их в том порядке, в котором они были заданы.

Выполнение `cat.go` приведет к следующим результатам:

```
$ go run cat.go
Mastering Go!
Mastering Go!
1 2 3 4
1 2 3 4
```

Но если выполнить `cat.go`, задействовав каналы UNIX, все становится по-настоящему интересным:

```
$ go run cat.go /tmp/1.log /tmp/2.log | wc
    2367        44464    279292
$ go run cat.go /tmp/1.log /tmp/2.log | go run cat.go | wc
    2367        44464    279292
```

С помощью `cat.go` также можно вывести на экран содержимое нескольких файлов:

```
$ go run cat.go 1.txt 1 1.txt
    2367        44464    279292
    2367        44464    279292
open 1: no such file or directory
    2367        44464    279292
    2367        44464    279292
```

Обратите внимание, что если попытаться выполнить `cat.go` как `go run cat.go cat.go`, то вместо ожидаемого вывода на экран содержимого файла `cat.go` процесс завершится ошибкой и вы получите следующее сообщение:

```
package main: case-insensitive file name collision: "cat.go" and "cat.go"
```

Это происходит из-за того, что Go не понимает, что второй файл `cat.go` следует использовать как аргумент командной строки для команды `go run cat.go`. Вместо этого `go run` пытается дважды скомпилировать `cat.go`, что вызывает сообщение об ошибке. Решение этой проблемы состоит в том, чтобы сначала выполнить `go build cat.go`, а затем использовать `cat.go` или любой другой исходный файл Go в качестве аргумента для сгенерированного двоичного исполняемого файла.

Структура syscall.PtraceRegs

Если вы решили, что мы закончили изучение стандартного Go-пакета `syscall`, то вы ошибаетесь! В этом разделе мы поработаем со структурой `syscall.PtraceRegs`, в которой содержится информация о состоянии реестров.

Вы научитесь выводить на экран значения следующих реестров, используя Go-код из файла `ptraceRegs.go`, который мы разделим на четыре части. Звездой утилиты `ptraceRegs.go` является функция `syscall.PtraceGetRegs()`. Для работы с реестрами существуют также функции `syscall.PtraceSetRegs()`, `syscall.PtraceAttach()`, `syscall.PtracePeekData()` и `syscall.PtracePokeData()`, однако в `ptraceRegs.go` эти функции использоваться не будут.

Первая часть утилиты `ptraceRegs.go` выглядит так:

```
package main

import (
    "fmt"
    "os"
    "os/exec"
    "syscall"
    "time"
)
```

Во второй части `ptraceRegs.go` содержится следующий код Go:

```
func main() {
    var r syscall.PtraceRegs
    cmd := exec.Command(os.Args[1], os.Args[2:]...)

    cmd.Stdout = os.Stdout
    cmd.Stderr = os.Stderr
```

Последние два оператора перенаправляют стандартный поток вывода и стандартный поток ошибок выполняемой команды в стандартный поток вывода UNIX и стандартный поток ошибок соответственно.

В третьей части `ptraceRegs.go` содержится следующий код Go:

```
    cmd.SysProcAttr = &syscall.SysProcAttr{Ptrace: true}
    err := cmd.Start()
    if err != nil {
        fmt.Println("Start:", err)
        return
    }

    err = cmd.Wait()
    fmt.Printf("State: %v\n", err)
    wpid := cmd.Process.Pid
```

В этом коде Go мы вызываем внешнюю команду, которая указана в аргументах командной строки программы, и находим идентификатор процесса, который будет использоваться в вызове `syscall.PtraceGetRegs()`. Оператор `&syscall.SysProcAttr{Ptrace: true}` указывает на то, что мы хотим использовать `ptrace` в дочернем процессе.

Последний фрагмент кода `ptraceRegs.go` выглядит так:

```
    err = syscall.PtraceGetRegs(wpid, &r)
    if err != nil {
        fmt.Println("PtraceGetRegs:", err)
        return
    }
    fmt.Printf("Registers: %#v\n", r)
    fmt.Printf("R15=%d, Gs=%d\n", r.R15, r.Gs)

    time.Sleep(2 * time.Second)
}
```

Здесь мы вызываем `syscall.PtraceGetRegs()` и выводим на экран результаты, хранящиеся в переменной `r`, которая должна быть передана как указатель.

Выполнение `ptraceRegs.go` на компьютере с macOS Mojave приведет к следующим результатам:

```
$ go run ptraceRegs.go
# command-line-arguments
./ptraceRegs.go:11:8: undefined: syscall.PtraceRegs
./ptraceRegs.go:14:9: undefined: syscall.PtraceGetRegs
```

Это значит, что данная программа не работает на компьютерах под управлением macOS и Mac OS X.

Выполнение `ptraceRegs.go` на компьютере с Debian Linux даст следующий результат:

```
$ go version
go version go1.7.4 linux/amd64
$ go run ptraceRegs.go echo "Mastering Go!"
State: stop signal: trace/breakpoint trap
Registers: syscall.PtraceRegs{R15:0x0, R14:0x0, R13:0x0, R12:0x0, Rbp:0x0,
Rbx:0x0, R11:0x0, R10:0x0, R9:0x0, R8:0x0, Rax:0x0, Rcx:0x0, Rdx:0x0,
Rsi:0x0, Rdi:0x0, Orig_rax:0x3b, Rip:0x7f4045f81c20, Cs:0x33, Eflags:0x200,
Rsp:0x7ffe1905b070, Ss:0x2b, Fs_base:0x0, Gs_base:0x0, Ds:0x0, Es:0x0,
Fs:0x0, Gs:0x0}
R15=0, Gs=0
Mastering Go!
```

Вы также найдете полный список реестров на странице документации пакета `syscall`.

Отслеживание системных вызовов

В этом разделе представлена усовершенствованная технология, использующая пакет `syscall` и позволяющая отслеживать системные вызовы, выполняемые в программе Go.

Утилита Go, которую мы рассмотрим, называется `traceSyscall.go`. Разделим ее на пять фрагментов кода. Первая часть `traceSyscall.go` выглядит так:

```
package main

import (
    "bufio"
    "fmt"
    "os"
    "os/exec"
    "strings"
    "syscall"
)

var maxSyscalls = 0

const SYSCALLFILE = "SYSCALLS"
```

О назначении переменной `SYSCALLFILE` мы поговорим чуть позже.

Второй фрагмент кода `traceSyscall.go` выглядит так:

```
func main() {
    var SYSTEMCALLS []string
    f, err := os.Open(SYSCALLFILE)
    defer f.Close()
```

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

    scanner := bufio.NewScanner(f)
    for scanner.Scan() {
        line := scanner.Text()
        line = strings.Replace(line, " ", "", -1)
        line = strings.Replace(line, "SYS_", "", -1)
        temp := strings.ToLower(strings.Split(line, "=")[0])
        SYSTEMCALLS = append(SYSTEMCALLS, temp)
        maxSyscalls++
    }
```

Обратите внимание, что содержимое текстового файла `SYSCALLS` взято из документации пакета `syscall`. В этом файле каждому системному вызову поставлен в соответствие номер, который является внутренним представлением Go для данного системного вызова. Этот файл нужен главным образом для вывода на экран имен системных вызовов, которые используются в отслеживаемой программе.

Содержимое текстового файла `SYSCALLS` имеет следующий формат:

```
SYS_READ = 0
SYS_WRITE = 1
SYS_OPEN = 2
SYS_CLOSE = 3
SYS_STAT = 4
```

Прочитав текстовый файл, программа создает для хранения этой информации срез `SYSTEMCALLS`.

Третья часть `traceSyscall.go` выглядит так:

```
    COUNTER := make([]int, maxSyscalls)
    var regs syscall.PtraceRegs
    cmd := exec.Command(os.Args[1], os.Args[2:]...)

    cmd.Stdin = os.Stdin
    cmd.Stdout = os.Stdout
    cmd.Stderr = os.Stderr
    cmd.SysProcAttr = &syscall.SysProcAttr{Ptrace: true}

    err = cmd.Start()
    err = cmd.Wait()
    if err != nil {
        fmt.Println("Wait:", err)
    }

    pid := cmd.Process.Pid
    fmt.Println("Process ID:", pid)
```

В срезе COUNTER хранятся данные о том, сколько раз каждый системный вызов был обнаружен в отслеживаемой программе.

Четвертый фрагмент traceSyscall.go содержит следующий код Go:

```
before := true
forCount := 0
for {
    if before {
        err := syscall.PtraceGetRegs(pid, &regs)
        if err != nil {
            break
        }
        if regs.Orig_rax > uint64(maxSyscalls) {
            fmt.Println("Unknown:", regs.Orig_rax)
            return
        }

        COUNTER[regs.Orig_rax]++
        forCount++
    }

    err = syscall.PtraceSyscall(pid, 0)
    if err != nil {
        fmt.Println("PtraceSyscall:", err)
        return
    }

    _, err = syscall.Wait4(pid, nil, 0, nil)
    if err != nil {
        fmt.Println("Wait4:", err)
        return
    }
    before = !before
}
```

Функция syscall.PtraceSyscall() заставляет Go продолжать выполнение отслеживаемой программы, но останавливаться, когда эта программа входит или выходит из системного вызова, — именно это нам и нужно! Каждый системный вызов отслеживается перед обработкой, сразу после того, как он завершил свою работу. Поэтому мы используем переменную before, чтобы учитывать каждый системный вызов только один раз.

Последняя часть traceSyscall.go выглядит так:

```
for i, x := range COUNTER {
    if x != 0 {
        fmt.Println(SYSTEMCALLS[i], "->", x)
    }
```

```
    }
    fmt.Println("Total System Calls:", forCount)
}
```

Здесь мы выводим на экран содержимое среза COUNTER. Срез SYSTEMCALLS в данном случае используется для определения имени системного вызова по известному числовому представлению этого вызова в Go.

Выполнение traceSyscall.go на компьютере с macOS Mojave приведет к следующим результатам:

```
$ go run traceSyscall.go
# command-line-arguments
./traceSyscall.go:36:11: undefined: syscall.PtraceRegs
./traceSyscall.go:57:11: undefined: syscall.PtraceGetRegs
./traceSyscall.go:70:9: undefined: syscall.PtraceSyscall
```

Как видим, утилита traceSyscall.go тоже не работает на macOS и Mac OS X.

Выполнение этой же программы на компьютере с Debian Linux приведет к следующему результату:

```
$ go run traceSyscall.go ls /tmp/
Wait: stop signal: trace/breakpoint trap
Process ID: 5657
go-build084836422 test.go upload_progress_cache
read -> 11
write -> 1
open -> 37
close -> 27
stat -> 1
fstat -> 25
mmap -> 39
mprotect -> 16
munmap -> 4
brk -> 3
rt_sigaction -> 2
rt_sigprocmask -> 1
ioctl -> 2
access -> 9
execve -> 1
getdents -> 2
getrlimit -> 1
statfs -> 2
arch_prctl -> 1
futex -> 1
set_tid_address -> 1
openat -> 1
set_robust_list -> 1
Total System Calls: 189
```

В завершение работы программа `traceSyscall.go` выводит количество появлений каждого системного вызова в программе. Правильность работы `traceSyscall.go` подтверждается результатами утилиты `strace -c`:

```
$ strace -c ls /tmp
test.go upload_progress_cache
% time     seconds  usecs/call     calls    errors syscall
------ ----------- ----------- --------- --------- ----------------
  0.00    0.000000           0        11           read
  0.00    0.000000           0         1           write
  0.00    0.000000           0        37        13 open
  0.00    0.000000           0        27           close
  0.00    0.000000           0         1           stat
  0.00    0.000000           0        25           fstat
  0.00    0.000000           0        39           mmap
  0.00    0.000000           0        16           mprotect
  0.00    0.000000           0         4           munmap
  0.00    0.000000           0         3           brk
  0.00    0.000000           0         2           rt_sigaction
  0.00    0.000000           0         1           rt_sigprocmask
  0.00    0.000000           0         2           ioctl
  0.00    0.000000           0         9         9 access
  0.00    0.000000           0         1           execve
  0.00    0.000000           0         2           getdents
  0.00    0.000000           0         1           getrlimit
  0.00    0.000000           0         2         2 statfs
  0.00    0.000000           0         1           arch_prctl
  0.00    0.000000           0         1           futex
  0.00    0.000000           0         1           set_tid_address
  0.00    0.000000           0         1           openat
  0.00    0.000000           0         1           set_robust_list
------ ----------- ----------- --------- --------- -------------
100.00    0.000000                   189        24 total
```

Идентификаторы пользователя и группы

В этом разделе вы узнаете, как найти идентификатор текущего пользователя, а также идентификаторы групп, к которым принадлежит этот пользователь. Идентификаторы пользователя и группы являются положительными целыми числами, которые хранятся в системных файлах UNIX.

Утилита, которую мы рассмотрим, называется `ids.go`. Разделим ее на две части. Первая часть этой утилиты выглядит так:

```
package main

import (
    "fmt"
```

```
    "os"
    "os/user"
)

func main() {
    fmt.Println("User id:", os.Getuid())
```

Для того чтобы узнать идентификатор текущего пользователя, нужно просто вызвать функцию `os.Getuid()`.

Вторая часть `ids.go` выглядит так:

```
    var u *user.User
    u, _ = user.Current()
    fmt.Print("Group ids: ")
    groupIDs, _ := u.GroupIds()
    for _, i := range groupIDs {
        fmt.Print(i, " ")
    }
    fmt.Println()
}
```

В то же время поиск идентификаторов группы, к которой принадлежит данный пользователь, является более сложной задачей.

Выполнение `ids.go` приведет к результатам следующего вида:

```
$ go run ids.go
User id: 501
Group ids: 20 701 12 61 79 80 81 98 33 100 204 250 395 398 399
```

Docker API и Go

Если вы работаете с Docker, то этот раздел будет для вас особенно полезен, так как вы научитесь общаться с Docker с помощью Go и Docker API.

Мы рассмотрим утилиту `dockerAPI.go`, которую разделим на четыре части. Эта утилита реализует команды `docker ps` и `docker image ls`. Первая команда перечисляет все запущенные контейнеры, а вторая — все доступные образы на локальном компьютере.

Первая часть `dockerAPI.go` выглядит так:

```
package main

import (
    "fmt"
    "github.com/docker/docker/api/types"
    "github.com/docker/docker/client"
    "golang.org/x/net/context"
)
```

Поскольку dockerAPI.go требует много внешних пакетов, было бы неплохо выполнить эту программу с использованием Go-модулей; поэтому перед первым запуском dockerAPI.go выполните команду export GO111MODULE = on. Это также избавит вас от необходимости загружать все необходимые пакеты вручную.

Вторая часть dockerAPI.go содержит следующий код Go:

```
func listContainers() error {
    cli, err := client.NewEnvClient()
    if err != nil {
        return (err)
    }

    containers, err := cli.ContainerList(context.Background(),
         types.ContainerListOptions{})
    if err != nil {
        return (err)
    }

    for _, container := range containers {
        fmt.Println("Images:", container.Image, "with ID:", container.ID)
    }
    return nil
}
```

Определение структуры types.Container (https://godoc.org/github.com/docker/docker/api/types#Container), возвращаемой функцией ContainerList(), выглядит так:

```
type Container struct {
    ID         string `json:"Id"`
    Names      []string
    Image      string
    ImageID    string
    Command    string
    Created    int64
    Ports      []Port
    SizeRw     int64 `json:",omitempty"`
    SizeRootFs int64 `json:",omitempty"`
    Labels     map[string]string
    State      string
    Status     string
    HostConfig struct {
        NetworkMode string `json:",omitempty"`
    }
    NetworkSettings *SummaryNetworkSettings
    Mounts          []MountPoint
}
```

Чтобы найти любую другую информацию о списке запущенных образов (контейнеров) Docker, следует просто использовать другие поля структуры types.Container.

Третья часть dockerAPI.go выглядит так:

```
func listImages() error {
    cli, err := client.NewEnvClient()
    if err != nil {
        return (err)
    }

    images, err := cli.ImageList(context.Background(), types.ImageListOptions{})
    if err != nil {
        return (err)
    }

    for _, image := range images {
        fmt.Printf("Images %s with size %d\n", image.RepoTags, image.Size)
    }
    return nil
}
```

Определение структуры types.ImageSummary (https://godoc.org/github.com/docker/docker/api/types#ImageSummary), которая является срезом, возвращаемым ImageList(), имеет следующий вид:

```
type ImageSummary struct {
    Containers int64 `json:"Containers"`
    Created int64 `json:"Created"`
    ID string `json:"Id"`
    Labels map[string]string `json:"Labels"`
    ParentID string `json:"ParentId"`
    RepoDigests []string `json:"RepoDigests"`
    RepoTags []string `json:"RepoTags"`
    SharedSize int64 `json:"SharedSize"`
    Size int64 `json:"Size"`
    VirtualSize int64 `json:"VirtualSize"`
}
```

Последняя часть dockerAPI.go выглядит так:

```
func main() {
    fmt.Println("The available images are:")
    err := listImages()
    if err != nil {
        fmt.Println(err)
    }

    fmt.Println("The running Containers are:")
    err = listContainers()
    if err != nil {
        fmt.Println(err)
    }
}
```

Выполнение `dockerAPI.go` на моей машине с macOS Mojave привело к следующим результатам:

```
$ go run dockerAPI.go
The available images are:
Images [golang:1.12] with size 772436547
Images [landoop/kafka-lenses-dev:latest] with size 1379088343
Images [confluentinc/cp-kafka:latest] with size 568503378
Images [landoop/fast-data-dev:latest] with size 1052076831
The running Containers are:
Images: landoop/kafka-lenses-dev with ID:
90e1caaab43297810341290137186425878ef5891c787f6707c03be617862db5
```

Если Docker недоступен или не работает, то вы получите следующее сообщение об ошибке:

```
$ go run dockerAPI.go
The available images are:
Cannot connect to the Docker daemon at unix:///var/run/docker.sock. Is the
docker daemon running?
The running Containers are:
Cannot connect to the Docker daemon at unix:///var/run/docker.sock. Is the
docker daemon running?
```

Дополнительные ресурсы

Следующие веб-ссылки будут вам очень полезны:

- почитайте страницу документации пакета `io`, которую вы найдете по адресу https://golang.org/pkg/io/;
- узнайте у Дейва Чейни, как обрабатывать ошибки: https://dave.cheney.net/2016/04/27/dont-just-check-errors-handle-them-graceful;
- чтобы больше узнать о библиотеке печати Glot, посетите официальную веб-страницу по адресу https://github.com/Arafatk/glot;
- другие примеры использования Docker API в Go, Python и HTTP вы найдете по адресу https://docs.docker.com/develop/sdk/examples/;
- вы узнаете больше о стандартном пакете `encoding/binary`, если посетите страницу https://golang.org/pkg/encoding/binary/;
- загляните на страницу документации пакета `encoding/gob`, которую вы найдете по адресу https://golang.org/pkg/encoding/gob/;
- советую также посмотреть видео https://www.youtube.com/watch?v=JRFNIKUROPE и https://www.youtube.com/watch?v=w8nFRoFJ6EQ;

- об *Endianness* пишут на многих ресурсах, в том числе здесь: https://en.wikipedia.org/wiki/Endianness;
- посетите страницу документации пакета `flag`, расположенную по адресу https://golang.org/pkg/flag/.

Упражнения

- Напишите программу Go, которая принимает три аргумента: имя текстового файла и две строки. Затем эта утилита должна заменить каждое вхождение первой строки в файле второй строкой. В целях безопасности окончательный результат выведите на экран, чтобы исходный текстовый файл остался без изменений.
- Используя пакет `encoding/gob`, напишите программу для сериализации и десериализации хеш-таблицы Go, а также среза структур.
- Создайте программу Go, которая бы обрабатывала три любых выбранных вами сигнала.
- Напишите на Go утилиту, которая бы заменяла все символы табуляции, найденные в текстовом файле, заданным количеством пробелов, указанным в качестве аргумента командной строки программы. Результат, как и в прошлый раз, выводите на экран.
- Разработайте утилиту, которая бы построчно читала текстовый файл и удаляла пробелы из каждой строки, используя функцию `strings.TrimSpace()`.
- Измените программу `kvSaveLoad.go` так, чтобы она поддерживала аргумент командной строки — имя файла, которое будет использоваться для загрузки и сохранения данных.
- Можете ли вы создать Go-версию утилиты `wc(1)`? Чтобы узнать, какие аргументы командной строки она поддерживает, загляните на справочную страницу `wc(1)`.
- Можете ли вы написать программу, которая бы использовала *Glot* для построения функции?
- Измените программу `traceSyscall.go` таким образом, чтобы она отображала каждый системный вызов во время его отслеживания.
- Измените программу `cat.go` таким образом, чтобы для копирования содержимого файла она просто выполняла `io.Copy(os.Stdout, f)`, вместо того чтобы сканировать его целиком.
- Используйте Docker API для написания утилиты, которая завершает все контейнеры, начинающиеся с заданной строки.

- Пакет `cobra` также поддерживает подкоманды — команды, связанные с определенными командами, такими как список команд `go run main.go`. Попробуйте реализовать утилиту с подкомандами.
- Для чтения строки по словам также можно использовать `bufio.NewScanner()` и `bufio.ScanWords`. Узнайте, как создать, и создайте новую версию утилиты `byWord.go`.

Резюме

В этой важной главе рассмотрено много интересных тем, в том числе чтение файлов, запись в файлы, использование Docker API, а также пакетов `flag`, `cobra` и `viper`. Тем не менее, остается много других тем, связанных с системным программированием, не упомянутых в этой главе, таких как работа с каталогами; копирование, удаление и переименование файлов; работа с пользователями, группами и процессами UNIX; изменение полномочий доступа к файлам UNIX; создание *неполных файлов*; блокировка и создание файлов; использование и замена ваших файлов журнала, а также информация из структуры, возвращаемой при вызове `os.Stat()`.

В конце главы я представил две расширенные утилиты, написанные на Go. Первая из них позволяет проверять состояние реестров, а во второй реализован метод, позволяющий отслеживать системные вызовы любой программы.

В следующей главе речь пойдет о программах, каналах и конвейерах — уникальных и мощных функциях Go.

9 Конкурентность в Go: горутины, каналы и конвейеры

В предыдущей главе рассмотрено системное программирование на Go, в том числе функции и методы Go, которые позволяют взаимодействовать с операционной системой. Но есть еще две области системного программирования, которые не были затронуты в предыдущей главе, — конкурентное программирование, а также способы создания и управления несколькими потоками. Эти темы разберем в настоящей и следующей главах.

В Go реализован собственный уникальный, инновационный способ обеспечения конкурентности в виде *горутин* и *каналов*. Горутины — это самые маленькие сущности Go, которые могут быть выполнены самостоятельно в программе Go. Каналы могут конкурентно и эффективно получать данные из горутин. Благодаря этому у горутин есть некая точка опоры, что позволяет им обмениваться данными между собой. В Go все выполняется посредством горутин. Это имеет смысл, поскольку Go — конкурентный язык программирования. Поэтому, когда начинается выполнение программы Go, ее единственная горутина вызывает функцию `main()`, которая и инициирует фактическое выполнение программы.

Содержание этой главы и представленные здесь фрагменты кода довольно просты, так что у вас не должно возникнуть проблем с их пониманием. Более сложные вопросы, касающиеся горутин и каналов, я оставил для главы 10.

В этой главе рассмотрены следующие темы:

- в чем различия между процессами, потоками и программами;
- планировщик Go;
- конкурентность и параллелизм;
- модели конкурентности в языках Erlang и Rust;
- создание горутин;
- создание каналов;
- чтение или получение данных из канала;

- запись или отправка данных в канал;
- создание конвейеров;
- и напоследок: ждем ваших горутин!

О процессах, потоках и горутинах

Процесс — это среда выполнения, в которой содержатся инструкции, пользовательские данные и части системных данных, а также другие типы ресурсов, полученные во время выполнения программы, тогда как сама *программа* — это файл, в котором содержатся инструкции и данные, используемые для инициализации инструкций и пользовательских данных процесса.

Поток — это меньшая и более легкая сущность, чем процесс или программа. Потоки создаются процессами, имеют собственный процесс управления и стек. Быстрый и простой способ отличить поток от процесса — рассматривать процесс как исполняемый двоичный файл, а поток — как подмножество процесса.

Горутина — это минимальная сущность Go, которая может быть выполнена конкурентно. Использование слова «минимальная» здесь очень важно, поскольку горутины не являются автономными сущностями, такими как процессы UNIX, — горутины живут в потоках UNIX, которые, в свою очередь, живут в процессах UNIX. Основным преимуществом горутин является то, что они чрезвычайно легкие, так что запуск тысяч или сотен тысяч горутин на одной машине не является проблемой.

Горутины легче потоков, которые, в свою очередь, легче процессов. На практике это означает, что в процессе может быть несколько потоков и множество горутин, в то время как горутина может существовать только в среде процесса. Таким образом, для создания горутины необходим процесс хотя бы с одним потоком — о процессах и управлении потоками позаботится UNIX, а Go и разработчик должны подумать о горутинах.

Теперь, когда вы познакомились с основными свойствами процессов, программ, потоков и горутин, поговорим немного о *планировщике Go*.

Планировщик Go

Планировщик ядра UNIX отвечает за выполнение потоков программы. Однако в среде выполнения Go есть собственный планировщик, который отвечает за выполнение горутин, используя технологию, называемую *планированием m:n*, при которой *m* горутин выполняется в *n* потоках операционной системы посредством мультиплексирования. Планировщик Go — это компонент Go, отвечающий за способ и последовательность выполнения горутин, из которых состоит программа Go.

Это делает планировщик Go очень важной частью языка программирования Go, поскольку все, что делает программа Go, выполняется в виде горутин.

Следует учитывать, что, поскольку планировщик Go оперирует горутинами только одной программы, он намного проще, дешевле и быстрее, чем планировщик ядра операционной системы.

Подробнее о том, как работает планировщик Go, вы узнаете из главы 10.

Конкурентность и параллелизм

Бытует очень распространенное заблуждение о том, что *конкурентность* и *параллелизм* — одно и то же. Это просто неправда! Параллелизм — это одновременное выполнение нескольких сущностей определенного вида, тогда как конкурентность — способ структурирования компонентов, позволяющий им выполняться независимо, когда это возможно.

Разработка конкурентных программных компонентов — необходимое условие того, что их можно будет безопасно выполнять параллельно, когда и если операционная система и оборудование это позволят. Это давным-давно было реализовано в языке программирования Erlang — задолго до появления многоядерных процессоров и компьютеров с большим объемом оперативной памяти.

В правильной конкурентной архитектуре добавление конкурентных сущностей приводит к тому, что вся система начинает работать быстрее, потому что в ней больше задач может выполняться параллельно. Таким образом, чтобы достичь желаемой степени параллелизма, необходимо обеспечить наилучшее конкурентное описание и реализацию задачи. Разработчик должен учитывать конкурентность на этапе проектирования системы — только тогда он получит выгоду от потенциального параллельного выполнения системных компонентов. Таким образом, разработчик должен думать не о параллелизме, а о том, как разбить систему на независимые компоненты, которые, будучи объединенными, решали бы исходную задачу.

Даже если на вашем компьютере с UNIX разработанные вами функции не могут выполняться параллельно, правильное конкурентное построение программы все равно улучшит ее структуру и удобство сопровождения. Другими словами, конкурентность лучше параллелизма!

Горутины

Чтобы определить, создать и выполнить новую горутину, нужно воспользоваться ключевым словом go, после которого поставить имя функции или полное определение *анонимной функции*. Ключевое слово go приводит к немедленному возврату (в основной поток) после вызова функции, в то время как сама функция выполняется в фоновом режиме как горутина, а остальная программа тем временем продолжает выполнение.

Однако, как вы скоро узнаете, мы не можем контролировать или делать какие-либо предположения о последовательности выполнения горутин, поскольку это зависит от планировщика операционной системы, планировщика Go и степени загрузки операционной системы.

Создание горутины

В этом подразделе показаны два способа создания горутин. Первый из них заключается в использовании обычных функций, а второй — в использовании анонимных функций. Эти способы эквивалентны.

В данном разделе мы рассмотрим программу `simple.go`. Разделим ее на три части.

Первая часть `simple.go` содержит следующий код Go:

```
package main

import (
    "fmt"
    "time"
)

func function() {
    for i := 0; i < 10; i++ {
        fmt.Print(i)
    }
}
```

Кроме блока `import`, в этом коде содержится определение функции с именем `function()`, которой мы вскоре воспользуемся.

Имя функции `function()` не является чем-то особенным — вы можете дать ей любое допустимое имя.

Вторая часть `simple.go` выглядит так:

```
func main() {
    go function()
```

Этот код начинается с выполнения `function()` в качестве горутины. После этого программа продолжает выполнение, в то время как `function()` начинает работать в фоновом режиме.

Последняя часть кода `simple.go` содержит следующий код Go:

```
    go func() {
        for i := 10; i < 20; i++ {
            fmt.Print(i, " ")
        }
    }()
```

```
    time.Sleep(1 * time.Second)
    fmt.Println()
}
```

С помощью этого кода мы создаем горутину, используя анонимную функцию. Этот метод лучше подходит для сравнительно небольших функций. Если же кода много, то рекомендуется создать обычную функцию и выполнить ее с помощью ключевого слова `go`.

Как вы увидите в следующем разделе, мы можем создавать сколько угодно горутин, в том числе используя цикл `for`.

Выполнение `simple.go` три раза подряд приведет к результатам такого вида:

```
$ go run simple.go
10 11 12 13 14 15 16 17 18 19 0123456789
$ go run simple.go
10 11 12 13 14 15 16 0117 2345678918 19
$ go run simple.go
10 11 12 012345678913 14 15 16 17 18 19
```

Обычно важно, чтобы программа генерировала одинаковые выходные данные для одних и тех же входных данных. Однако результаты, которые мы получаем от `simple.go`, не всегда одинаковы. Показанный выше вывод программы подтверждает тот факт, что мы не можем контролировать последовательность выполнения горутин, если специально не проследим за этим, то есть не напишем специальный дополнительный код. О том, как управлять последовательностью выполнения горутин, а также выводить результаты одной подпрограммы перед результатами следующей, вы узнаете из главы 10.

Создание нескольких горутин

В этом подразделе продемонстрировано, как создать переменное число горутин. Программа, которую мы рассмотрим, позволяет разрабатывать динамическое число горутин. Она называется `create.go`. Разделим ее на четыре части. Количество горутин задается как аргумент командной строки программы, а для обработки аргумента командной строки используется пакет `flag`.

Первая часть кода `create.go` выглядит так:

```
package main

import (
    "flag"
    "fmt"
    "time"
)
```

Второй фрагмент `create.go` содержит следующий код Go:

```
func main() {
    n := flag.Int("n", 10, "Number of goroutines")
    flag.Parse()

    count := *n
    fmt.Printf("Going to create %d goroutines.\n", count)
```

В этом коде считывается значение аргумента командной строки `n`, который определяет количество создаваемых горутин. Если аргумент командной строки `n` не задан, то значение переменной `n` будет равно 10.

Третья часть кода `create.go` выглядит так:

```
    for i := 0; i < count; i++ {
        go func(x int) {
            fmt.Printf("%d ", x)
        }(i)
    }
```

Цикл `for` используется для создания желаемого количества горутин. Еще раз повторю: следует помнить, что вы не можете делать какие-либо предположения о порядке, в котором они будут созданы и выполнены.

Последняя часть кода Go из `create.go` выглядит так:

```
    time.Sleep(time.Second)
    fmt.Println("\nExiting...")
}
```

Цель оператора `time.Sleep()` — дать горутинам достаточно времени, чтобы они успели завершить свою работу и их результаты можно было увидеть на экране. В реальной программе оператор `time.Sleep()` не нужен, так как вы захотите завершить программу как можно скорее. Кроме того, вы вскоре познакомитесь с более эффективным методом, позволяющим сделать так, чтобы программа ожидала завершения различных горутин, прежде чем завершить работу и вернуть результат функции `main()`.

Выполнение `create.go` несколько раз подряд приводит к результатам такого вида:

```
$ go run create.go -n 100
Going to create 100 goroutines.
5 3 2 4 19 9 0 1 7 11 10 12 13 14 15 31 16 20 17 22 8 18 28 29 21 52 30 45
25 24 49 38 41 46 6 56 57 54 23 26 53 27 59 47 69 66 51 44 71 48 74 33 35
73 39 37 58 40 50 78 85 86 90 67 72 91 32 64 65 95 75 97 99 93 36 60 34 77
94 61 88 89 83 84 43 80 82 87 81 68 92 62 55 98 96 63 76 79 42 70
Exiting...
$ go run create.go -n 100
Going to create 100 goroutines.
2 5 3 16 6 7 8 9 1 22 10 12 13 17 11 18 15 14 19 20 31 23 26 21 29 24 30 25
37 32 36 38 35 33 45 41 43 42 40 39 34 44 48 46 47 56 53 50 0 49 55 59 58
28 54 27 60 4 57 51 52 64 61 65 72 62 63 67 69 66 74 73 71 75 89 70 76 84
85 68 79 80 93 97 83 82 99 78 88 91 92 77 81 95 94 98 87 90 96 86
Exiting...
```

Здесь вы снова видите, что выходные данные программы являются недетерминированными и неупорядоченными в том смысле, что необходимо просмотреть все выходные данные, чтобы найти то, что вам нужно. Кроме того, если не подобрать подходящую задержку при вызове `time.Sleep()`, то мы вообще не увидим результат выполнения горутин. На данный момент нам подошел вариант с `time.Second`, но в будущем такой код может привести к неприятным и непредсказуемым ошибкам.

В следующем разделе показано, как предоставить горутинам достаточно времени, чтобы они успели выполнить свою работу до завершения программы, без необходимости вызывать `time.Sleep()`.

Как дождаться завершения горутин, прежде чем закончить программу

В этом разделе показано, как предотвратить завершение функции `main()`, пока она ожидает завершения своих горутин, с помощью пакета `sync`. Логика программы `syncGo.go` основана на коде из `create.go`, который рассмотрен в предыдущем разделе.

Первая часть `syncGo.go` выглядит так:

```
package main

import (
    "flag"
    "fmt"
    "sync"
)
```

Как видим, нам не требуется импортировать и применять пакет `time`, поскольку мы будем использовать функциональность пакета `sync` и ожидать столько времени, сколько необходимо для завершения всех горутин.

В главе 10 вы познакомитесь с двумя технологиями, позволяющими отменить выполнение горутин, если они занимают больше времени, чем рассчитывалось.

Второй фрагмент `syncGo.go` содержит следующий код Go:

```
func main() {
    n := flag.Int("n", 20, "Number of goroutines")
    flag.Parse()
    count := *n
    fmt.Printf("Going to create %d goroutines.\n", count)

    var waitGroup sync.WaitGroup
```

В этом коде Go мы определяем переменную `sync.WaitGroup`. Если вы заглянете в исходный код Go-пакета `sync`, а именно в файл `waitgroup.go`, который находится в каталоге `sync`, то увидите, что тип `sync.WaitGroup` — это не что иное, как структура с тремя полями:

```
type WaitGroup struct {
    noCopy
    state1  [12]byte
    sema    uint32
}
```

Как работают переменные `sync.WaitGroup`, станет более понятно, когда мы увидим результат работы `syncGo.go`. Количество горутин, принадлежащих группе `sync.WaitGroup`, определяется одним или несколькими вызовами функции `sync.Add()`.

Третья часть `syncGo.go` содержит следующий код Go:

```
fmt.Printf("%#v\n", waitGroup)
for i := 0; i < count; i++ {
    waitGroup.Add(1)
    go func(x int) {
        defer waitGroup.Done()
        fmt.Printf("%d ", x)
    }(i)
}
```

Здесь мы создаем желаемое количество горутин, используя цикл `for`. (Вместо этого можно применять несколько последовательных операторов Go.)

Каждый вызов `sync.Add()` увеличивает счетчик в переменной `sync.WaitGroup` на единицу. Обратите внимание: очень важно вызвать `sync.Add(1)` перед оператором `go`, чтобы предотвратить возможное *состояние гонки*. Когда горутина завершает работу, выполняется функция `sync.Done()`, которая уменьшает тот же счетчик на единицу.

Последняя часть кода `syncGo.go` выглядит так:

```
    fmt.Printf("%#v\n", waitGroup)
    waitGroup.Wait()
    fmt.Println("\nExiting...")
}
```

Вызов `sync.Wait()` блокируется до тех пор, пока значение счетчика в соответствующей переменной `sync.WaitGroup` не станет равным нулю, что дает возможность всем горутинам завершить работу.

Выполнение `syncGo.go` приведет к результатам такого вида:

```
$ go run syncGo.go
Going to create 20 goroutines.
sync.WaitGroup{noCopy:sync.noCopy{}, state1:[12]uint8{0x0, 0x0, 0x0, 0x0,
0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0}, sema:0x0}
```

```
sync.WaitGroup{noCopy:sync.noCopy{}, state1:[12]uint8{0x0, 0x0, 0x0, 0x0,
0x14, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0}, sema:0x0}
19 7 8 9 10 11 12 13 14 15 16 17 0 1 2 5 18 4 6 3
Exiting...
$ go run syncGo.go -n 30
Going to create 30 goroutines.
sync.WaitGroup{noCopy:sync.noCopy{}, state1:[12]uint8{0x0, 0x0, 0x0, 0x0, 0x0, 0x0,
0x0, 0x0, 0x0, 0x0, 0x0, 0x0}, sema:0x0}
1 0 4 5 17 7 8 9 10 11 12 13 2 sync.WaitGroup{noCopy:sync.noCopy{},
state1:[12]uint8{0x0, 0x0, 0x0, 0x0, 0x17, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0},
sema:0x0}
29 15 6 27 24 25 16 22 14 23 18 26 3 19 20 28 21
Exiting...
$ go run syncGo.go -n 30
Going to create 30 goroutines.
sync.WaitGroup{noCopy:sync.noCopy{}, state1:[12]uint8{0x0, 0x0, 0x0, 0x0, 0x0, 0x0,
0x0, 0x0, 0x0, 0x0, 0x0, 0x0}, sema:0x0}
sync.WaitGroup{noCopy:sync.noCopy{}, state1:[12]uint8{0x0, 0x0, 0x0, 0x0, 0x1e,
0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0}, sema:0x0}
29 1 7 8 2 9 10 11 12 4 13 15 0 6 5 22 25 23 16 28 26 20 19 24 21 14 3 17 18 27
Exiting...
```

Результаты syncGo.go по-прежнему меняются от запуска к запуску, особенно при большом количестве горутин. В большинстве случаев это приемлемо; однако бывают ситуации, когда такое поведение нежелательно. Кроме того, при числе горутин, равном 30, некоторые из них завершили работу прежде, чем был выполнен второй оператор fmt.Printf("%#v\n", waitGroup). Наконец, обратите внимание: один из элементов поля state1 в sync.WaitGroup содержит счетчик, который увеличивается и уменьшается в соответствии с вызовами sync.Add() и sync.Done().

Что происходит, если количество вызовов Add() и Done() не совпадает

Когда количество вызовов sync.Add() и sync.Done() одинаково, в программе все хорошо. Однако в этом пункте вы увидите, что случится, если два числа не совпадают.

Если вызовов sync.Add() было выполнено больше, чем вызовов sync.Done(), то, поставив оператор waitGroup.Add(1) перед первым оператором fmt.Printf("%#v\n", waitGroup) в программе syncGo.go и вызвав команду go run, получим следующий результат:

```
$ go run syncGo.go
Going to create 20 goroutines.
sync.WaitGroup{noCopy:sync.noCopy{}, state1:[12]uint8{0x0, 0x0, 0x0, 0x0, 0x1, 0x0,
0x0, 0x0, 0x0, 0x0, 0x0, 0x0}, sema:0x0}
sync.WaitGroup{noCopy:sync.noCopy{}, state1:[12]uint8{0x0, 0x0, 0x0, 0x0, 0x15,
0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0}, sema:0x0}
```

```
19 10 11 12 13 17 18 8 5 4 6 14 1 0 7 3 2 15 9 16 fatal error: all
goroutines are asleep - deadlock!
goroutine 1 [semacquire]:
sync.runtime_Semacquire(0xc4200120bc)
    /usr/local/Cellar/go/1.9.3/libexec/src/runtime/sema.go:56 +0x39
sync.(*WaitGroup).Wait(0xc4200120b0)
    /usr/local/Cellar/go/1.9.3/libexec/src/sync/waitgroup.go:131 +0x72
main.main()
    /Users/mtsouk/Desktop/masterGo/ch/ch9/code/syncGo.go:28 +0x2d7
exit status 2
```

Сообщение об ошибке говорит само за себя: `fatal error: all goroutines are asleep - deadlock!` Так случилось потому, что мы дали программе указание ждать завершения `n+1` горутин, вызвав функцию `sync.Add(1)` `n+1` раз, в то время как всего `n` горутин выполнили `n` операторов `sync.Done()`. В результате вызов `sync.Wait()` будет бесконечно и безуспешно ожидать одного или нескольких вызовов `sync.Done()` — это, очевидно, тупиковая ситуация.

Если же вызовов `sync.Add()` было выполнено меньше, чем вызовов `sync.Done()`, которые можно эмулировать, добавив в программе `syncGo.go` оператор `waitGroup.Done()` после цикла `for`, то результат команды `go run` будет примерно таким:

```
$ go run syncGo.go
Going to create 20 goroutines.
sync.WaitGroup{noCopy:sync.noCopy{}, state1:[12]uint8{0x0, 0x0, 0x0, 0x0, 0x0, 0x0,
0x0, 0x0, 0x0, 0x0, 0x0, 0x0}, sema:0x0}
sync.WaitGroup{noCopy:sync.noCopy{}, state1:[12]uint8{0x0, 0x0, 0x0, 0x0, 0x12,
0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0}, sema:0x0}
19 6 1 2 9 7 8 15 13 0 14 16 17 3 11 4 5 12 18 10 panic: sync: negative
WaitGroup counter
goroutine 22 [running]:
sync.(*WaitGroup).Add(0xc4200120b0, 0xffffffffffffffff)
    /usr/local/Cellar/go/1.9.3/libexec/src/sync/waitgroup.go:75 +0x134
sync.(*WaitGroup).Done(0xc4200120b0)
    /usr/local/Cellar/go/1.9.3/libexec/src/sync/waitgroup.go:100 +0x34
main.main.func1(0xc4200120b0, 0x11)
    /Users/mtsouk/Desktop/masterGo/ch/ch9/code/syncGo.go:25 +0xd8
created by main.main
    /Users/mtsouk/Desktop/masterGo/ch/ch9/code/syncGo.go:21 +0x206
exit status 2
```

Опять же причина проблемы изложена предельно ясно: `panic: sync: negative WaitGroup counter`.

Несмотря на то что в обоих случаях сообщения об ошибках весьма содержательны и помогут решить реальную проблему, следует быть очень осторожным с количеством вызовов `sync.Add()` и `sync.Done()`, которые мы вводим в свои программы. Кроме того, обратите внимание, что при второй ошибке (`panic: sync: negative WaitGroup counter`) проблема может проявляться не всегда.

Каналы

Канал — это механизм коммуникации, который, помимо прочего, позволяет обмениваться данными между горутинами.

Однако здесь есть ряд правил. Во-первых, каждый канал позволяет обмениваться данными определенного типа, который называется *типом элемента* канала. Во-вторых, для правильной работы канала необходим кто-то, кто будет получать то, что отправляется через канал. Чтобы создать новый канал, нужно использовать ключевое слово `chan`, а чтобы его закрыть — вызвать функцию `close()`.

Наконец, еще одна очень важная деталь: используя канал в качестве аргумента функции, можно указать его направление — будет ли этот канал использоваться для отправки или получения данных. Я считаю, что, если назначение канала заранее известно, следует использовать эту возможность, поскольку это сделает программы надежнее и безопаснее. Вы не сможете случайно отправить данные в канал, из которого можно только получать данные, или получать данные из канала, в который данные можно только отправлять. В результате, если вы объявите, что канал — аргумент функции будет использоваться только для чтения, и попытаетесь записать в него данные, то получите сообщение об ошибке, которое, скорее всего, избавит вас от неприятных программных дефектов. Мы еще вернемся к этому вопросу позже.

В этой главе вы узнаете много нового о каналах. Однако, чтобы в полной мере оценить всю мощь и гибкость, которые каналы предоставляют разработчику Go, вам придется подождать до главы 10.

Запись в канал

Код, представленный в этом подразделе, покажет вам, как записывать данные в канал. Записать значение `x` в канал `c` очень просто: достаточно написать код `c <- x`. Стрелка показывает направление передачи значения, так что у вас не возникнет проблем с этим утверждением, если и `x`, и `c` имеют одинаковый тип. Пример кода, который рассмотрен в данном разделе, хранится в файле `writeCh.go`. Разделим его на три части.

Первый фрагмент кода `writeCh.go` выглядит так:

```
package main

import (
    "fmt"
    "time"
)

func writeToChannel(c chan int, x int) {
    fmt.Println(x)
```

```
    c <- x
    close(c)
    fmt.Println(x)
}
```

Ключевое слово chan используется для объявления того, что аргумент функции c является каналом, и после него должен быть указан тип канала (int). Оператор c <- x позволяет записать значение x в канал c, а функция close() закрывает канал, то есть делает невозможным запись в него.

Вторая часть writeCh.go содержит следующий код Go:

```
func main() {
    c := make(chan int)
```

В этом коде вы видите определение переменной канала с именем c, использование функции make() — впервые в этой главе, — а также ключевого слова chan. С каждым каналом связан определенный тип данных, здесь это int.

Остальной код writeCh.go выглядит так:

```
    go writeToChannel(c, 10)
    time.Sleep(1 * time.Second)
}
```

Здесь выполняется функция writeToChannel() как горутина и вызывается time.Sleep(), чтобы предоставить достаточно времени для выполнения функции writeToChannel().

Выполнение writeCh.go приведет к следующему результату:

```
$ go run writeCh.go
10
```

Странно, что функция writeToChannel() выводит это значение только один раз. Причиной такого неожиданного результата является то, что второй оператор fmt.Println(x) никогда не выполняется. Это объясняется очень просто, если понимать, как работают каналы: оператор c <- x блокирует выполнение остальной части функции writeToChannel(), так как никто не читает то, что было записано в канал c. Поэтому, когда оператор time.Sleep(1 * time.Second) завершает работу, программа заканчивает выполнение, не ожидая завершения writeToChannel().

Как читать данные из канала, вы узнаете в следующем подразделе.

Чтение из канала

В этом подразделе вы научитесь читать данные из канала. Чтобы прочитать из канала одно значение с именем c, нужно выполнить операцию <-c. В этом случае операция направлена из канала наружу.

Программа, которую мы рассмотрим, чтобы помочь вам понять, как читать данные из канала, называется readCh.go. Разделим ее на три части.

Первый фрагмент `readCh.go` содержит следующий код Go:

```
package main

import (
    "fmt"
    "time"
)

func writeToChannel(c chan int, x int) {
    fmt.Println("1", x)
    c <- x
    close(c)
    fmt.Println("2", x)
}
```

Реализация функции `writeToChannel()` такая же, как и раньше.

Вторая часть `readCh.go` выглядит так:

```
func main() {
    c := make(chan int)
    go writeToChannel(c, 10)
    time.Sleep(1 * time.Second)
    fmt.Println("Read:", <-c)
    time.Sleep(1 * time.Second)
```

В этом коде мы читаем данные из канала `c`, используя нотацию `<-c`. Если мы захотим сохранить это значение в переменной с именем `k`, вместо того чтобы просто вывести его на экран, мы можем использовать запись `k: = <-c`. Второй оператор `time.Sleep(1 * time.Second)` предоставляет время для чтения из канала.

Последняя часть кода `readCh.go` содержит следующий код Go:

```
    _, ok := <-c
    if ok {
        fmt.Println("Channel is open!")
    } else {
        fmt.Println("Channel is closed!")
    }
}
```

В этом коде показан прием, позволяющий определить, открыт ли данный канал. Этот код Go работает нормально, когда канал закрыт; но если канал открыт, то вследствие использования символа _ в выражении `_, ok: = <-c` код Go отбросит прочитанное из канала значение. Если хотите сохранить значение, полученное из канала, когда он открыт, используйте вместо _ правильное имя переменной.

Выполнение `readCh.go` приведет к следующим результатам:

```
$ go run readCh.go
1 10
Read: 10
2 10
```

```
Channel is closed!
$ go run readCh.go
1 10
2 10
Read: 10
Channel is closed!
```

Эти результаты все еще не являются детерминированными, зато теперь выполняются оба оператора `fmt.Println(x)` функции `writeToChannel()`, поскольку при чтении из канала данный канал разблокируется.

Прием данных из закрытого канала

В этом подразделе вы узнаете, что происходит, если попытаться прочитать данные из закрытого канала. Воспользуемся кодом Go из файла `readClose.go`, который мы разделим на две части.

Первая часть `readClose.go` выглядит так:

```
package main

import (
    "fmt"
)

func main() {
    willClose := make(chan int, 10)

    willClose <- -1
    willClose <- 0
    willClose <- 2

    <-willClose
    <-willClose
    <-willClose
```

В этой части программы мы создаем новый канал типа `int` с именем `willClose`, записываем в него данные и считываем все эти данные, ничего не делая с ними.

Вторая часть `readClose.go` содержит следующий код:

```
    close(willClose)
    read := <-willClose
    fmt.Println(read)
}
```

Здесь мы закрываем канал `willClose` и пытаемся прочитать из него данные, несмотря на то что мы очистили этот канал в предыдущей части программы.

Выполнение `readClose.go` приведет к следующим результатам:

```
$ go run readClose.go
0
```

Это означает, что при чтении данных из закрытого канала возвращается нулевое значение соответствующего типа, в данном случае `0`.[1]

Каналы как аргументы функции

Хоть мы и не обращались к этому свойству Go ни в `readCh.go`, ни в `writeCh.go`, Go позволяет указать направление канала при применении его в качестве аргумента функции, сообщив ей таким образом, будет ли канал использоваться для чтения или для записи. Эти два типа каналов называются однонаправленными каналами, тогда как по умолчанию каналы являются двунаправленными.

Рассмотрим код Go следующих двух функций:

```
func f1(c chan int, x int) {
    fmt.Println(x)
    c <- x
}

func f2(c chan<- int, x int) {
    fmt.Println(x)
    c <- x
}
```

Хотя обе функции реализуют одну и ту же функциональность, их определения немного различаются. Разница заключается в символе `<-`, который стоит справа от ключевого слова `chan` в определении функции `f2()`. Этот символ означает, что канал `c` может использоваться только для записи. Если в коде функции Go мы попытаемся прочитать данные из канала, предназначенного только для записи (только для отправки данных), то компилятор Go выдаст следующее сообщение об ошибке:

```
# command-line-arguments
a.go:19:11: invalid operation: range in (receive from send-only type chan<-int)
```

Подобным же образом можно написать следующие определения функций:

```
func f1(out chan<- int64, in <-chan int64) {
    fmt.Println(x)
    c <- x
}

func f2(out chan int64, in chan int64) {
    fmt.Println(x)
    c <- x
}
```

В определении функции `f2()` присутствует канал, доступный только для чтения, который называется `in`, и канал, доступный только для записи, с именем `out`.

[1] Попытка записи в закрытый канал приведет к панике.

Если попытаться случайно записать данные в канал, который является аргументом функции и предназначен только для чтения *(канал только для приема данных)*, и закрыть его, то получим следующее сообщение об ошибке:

```
# command-line-arguments
a.go:13:7: invalid operation: out <- i (send to receive-only type <-chan int)
a.go:15:7: invalid operation: close(out) (cannot close receive-only channel)
```

Конвейеры

Конвейер — это виртуальный метод, предназначенный для соединения горутин и каналов, так что выходные данные одной горутины становятся входными данными для другой горутины, а для передачи данных используются каналы.

Одним из преимуществ использования конвейеров является наличие постоянного потока данных, так что никакие горутины и каналы не должны ожидать, пока завершится все остальное, чтобы можно было начать выполнение. Кроме того, мы используем меньше переменных и, следовательно, меньше памяти, потому что не приходится сохранять все данные в виде переменных. Наконец, использование конвейеров упрощает разработку программ и делает их удобнее для поддержки.

Рассмотрим использование конвейеров на примере кода `pipeline.go`, разделив программу на шесть частей. Задача, выполняемая программой `pipeline.go`, состоит в том, чтобы генерировать случайные числа в заданном диапазоне и останавливаться, когда любое число в этой последовательности встретится во второй раз. При этом, прежде чем завершить работу, программа выводит на экран сумму всех случайных чисел, сгенерированных до того момента, когда впервые появилось повторяющееся случайное число. Для подключения каналов в программе нам понадобятся три функции. В этих функциях заключается логика программы. Данные будут передаваться по каналам конвейера.

Программа будет иметь два канала. Первый канал (канал *A*) будет использоваться для получения случайных чисел из первой функции и передачи их во вторую функцию. Второй канал (канал *B*) будет использоваться второй функцией для передачи приемлемых случайных чисел в третью функцию. Третья функция будет отвечать за получение данных из канала *B*, вычисления и представление результатов.

Первый фрагмент `pipeline.go` содержит следующий код Go:

```
package main

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

```
var CLOSEA = false

var DATA = make(map[int]bool)
```

Поскольку `second()` нужен способ сообщить функции `first()` о закрытии первого канала, я буду использовать для этого глобальную переменную с именем `CLOSEA`. Переменная `CLOSEA` проверяется только функцией `first()` и может быть изменена только `second()`.

Вторая часть `pipeline.go` содержит следующий код Go:

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

func first(min, max int, out chan<- int) {
    for {
        if CLOSEA {
            close(out)
            return
        }
        out <- random(min, max)
    }
}
```

В этом коде представлена реализация двух функций: `random()` и `first()`. С функцией `random()`, которая генерирует случайные числа в заданном диапазоне, вы уже знакомы. Главный интерес представляет функция `first()`, поскольку она продолжает выполнять цикл `for` до тех пор, пока логическая переменная (`CLOSEA`) не станет равной `true`, после чего функция закрывает свой канал `out`.

Третий фрагмент кода `pipeline.go` выглядит так:

```
func second(out chan<- int, in <-chan int) {
    for x := range in {
        fmt.Print(x, " ")
        _, ok := DATA[x]
        if ok {
            CLOSEA = true
        } else {
            DATA[x] = true
            out <- x
        }
    }
    fmt.Println()
    close(out)
}
```

Функция `second()` получает данные из канала `in` и передает их в канал `out`. Однако, как только `second()` обнаруживает случайное число, которое уже существует в отображении `DATA`, она присваивает глобальной переменной `CLOSEA` значение `true`

и прекращает передачу любых других чисел в канал `out`. После этого функция закрывает канал `out`.

Цикл range читает данные из канала и автоматически завершает работу, когда канал закрывается.

В четвертой части `pipeline.go` содержится следующий код Go:

```
func third(in <-chan int) {
    var sum int
    sum = 0
    for x2 := range in {
        sum = sum + x2
    }
    fmt.Printf("The sum of the random numbers is %d\n", sum)
}
```

Функция `third()` считывает данные из канала, переданного функции в качестве аргумента. Когда `second()` закрывает этот канал, цикл `for` прекращает получать данные и функция выводит результат на экран. На этом этапе вы уже, вероятно, поняли, что здесь всем управляет `second()`.

Пятый фрагмент кода `pipeline.go` выглядит так:

```
func main() {
    if len(os.Args) != 3 {
        fmt.Println("Need two integer parameters!")
        return
    }

    n1, _ := strconv.Atoi(os.Args[1])
    n2, _ := strconv.Atoi(os.Args[2])

    if n1 > n2 {
        fmt.Printf("%d should be smaller than %d\n", n1, n2)
        return
    }
```

Этот код используется для работы с аргументами командной строки программы.

Последняя часть программы `pipeline.go` выглядит следующим образом:

```
    rand.Seed(time.Now().UnixNano())
    A := make(chan int)
    B := make(chan int)

    go first(n1, n2, A)
    go second(B, A)
    third(B)
}
```

Здесь мы определяем требуемые каналы и выполняем две горутины и одну функцию. Функция `third()` не позволяет `main()` немедленно завершиться, поскольку она не выполняется как горутина.

Выполнение `pipeline.go` приведет к результатам такого вида:

```
$ go run pipeline.go 1 10
2 2
The sum of the random numbers is 2
$ go run pipeline.go 1 10
9 7 8 4 3 3
The sum of the random numbers is 31
$ go run pipeline.go 1 10
1 6 9 7 1
The sum of the random numbers is 23
$ go run pipeline.go 10 20
16 19 16
The sum of the random numbers is 35
$ go run pipeline.go 10 20
10 16 17 11 15 10
The sum of the random numbers is 69
$ go run pipeline.go 10 20
12 11 14 15 10 15
The sum of the random numbers is 62
```

Здесь важно, что, хотя функция `first()` генерирует случайные числа в своем темпе, а `second()` выводит их все на экран, нежелательные (повторяющиеся) случайные числа не будут отправлены в `third()` и, следовательно, не будут включены в итоговую сумму.

Состояние гонки

Код `pipeline.go` неидеален: он содержит логическую ошибку, которая в терминологии конкурентности называется состоянием гонки. Чтобы выявить эту ошибку, нужно выполнить следующую команду:

```
$ go run -race pipeline.go 1 10
2 2 ==================
WARNING: DATA RACE
Write at 0x00000122bae8 by goroutine 7:
  main.second()
      /Users/mtsouk/ch09/pipeline.go:34 +0x15c
Previous read at 0x00000122bae8 by goroutine 6:
  main.first()
      /Users/mtsouk/ch09/pipeline.go:21 +0xa3
Goroutine 7 (running) created at:
  main.main()
      /Users/mtsouk/ch09/pipeline.go:72 +0x2a1
```

```
Goroutine 6 (running) created at:
  main.main()
      /Users/mtsouk/ch09/pipeline.go:71 +0x275
==================
2
The sum of the random numbers is 2.
Found 1 data race(s)
exit status 66
```

Проблема заключается в том, что горутина, выполняющая функцию `second()`, может изменить значение переменной `CLOSEA` в тот момент, когда `first()` будет читать эту переменную. Поскольку невозможно определить, какое из данных событий произойдет первым, а какое — вторым, такое состояние считается состоянием гонки. Чтобы устранить состояние гонки, необходимо использовать сигнальный канал и ключевое слово `select`.

Подробнее об условиях гонки, сигнальных каналах и ключевом слове select вы прочитаете в главе 10.

Чтобы показать, какие изменения мы внесли в `pipeline.go`, воспользуемся командой `diff(1)` — новая версия программы называется `plNoRace.go`:

```
$ diff pipeline.go plNoRace.go
14a15,16
> var signal chan struct{}
>
21c23,24
<    if CLOSEA {
---
>    select {
>    case <-signal:
23a27
>    case out <- random(min, max):
25d28
<    out <- random(min, max)
31d33
<    fmt.Print(x, " ")
34c36
<        CLOSEA = true
---
>        signal <- struct{}{}
35a38
>        fmt.Print(x, " ")
61d63
<
66a69,70
>    signal = make(chan struct{})
>
```

Для того чтобы проверить логическую корректность `plNoRace.go`, можно воспользоваться следующей командой:

```
$ go run -race plNoRace.go 1 10
8 1 4 9 3
The sum of the random numbers is 25.
```

Сравнение моделей конкурентности в Go и Rust

Rust — очень популярный язык системного программирования, который также поддерживает конкурентное программирование. Далее кратко изложены некоторые характеристики Rust и модель конкурентности Rust:

- потоки Rust — это потоки UNIX, следовательно, они тяжелые, но позволяют делать много интересного;
- Rust поддерживает конкурентность на основе *обмена сообщениями* и на основе *совместного использования состояний* — то же самое, что в Go реализовано посредством каналов, мьютексов и общих переменных;
- благодаря строгой типизации и системе владения Rust обеспечивает безопасное изменяемое состояние потока. Эти правила применяются компилятором Rust;
- существуют структуры Rust, допускающие совместно используемые состояния;
- если поток начинает вести себя некорректно, с системой не случится фатальный сбой. Это контролируемая ситуация, которая может быть обработана;
- язык программирования Rust продолжает развиваться, из-за чего некоторые отказываются его использовать, чтобы не пришлось постоянно вносить изменения в существующий код.

Таким образом, у Rust есть гибкая модель конкурентности — еще более гибкая, чем в Go. Однако за эту гибкость приходится платить — необходимо мириться с Rust и его особенностями.

Сравнение моделей конкурентности в Go и Erlang

Erlang — это очень популярный язык конкурентного функционального программирования, разработанный для создания высокодоступных приложений. Далее приведены основные характеристики языка Erlang и реализованной в нем модели конкурентности:

- Erlang является сформировавшимся и протестированным языком программирования. Это также относится к его модели конкурентности;

- если вам не нравится, как работает код на Erlang, вы всегда можете попробовать *Elixir*, который основан на Erlang и использует Erlang VM, — его код выглядит более приятно;
- в Erlang используется только асинхронная коммуникация;
- в Erlang реализована система обработки ошибок, позволяющая создавать надежные конкурентные системы;
- процессы Erlang способны привести к сбою системы, но если этот сбой обрабатывается должным образом, то система может продолжить работать без проблем;
- как и в случае с горутинами, процессы Erlang являются изолированными, у них нет общих состояний. Единственный способ взаимодействия между процессами Erlang — передача сообщений;
- потоки в Erlang такие же легкие, как и горутины. Это означает, что можно создавать сколько угодно процессов.

Таким образом, Erlang и Elixir являются надежными и высокодоступными системами, если вас устраивает принцип конкурентности, реализованный в Erlang.

Дополнительные ресурсы

Советую вам посетить следующие полезные ресурсы:

- страницу документации пакета `sync`, которая находится по адресу https://golang.org/pkg/sync/;
- веб-сайт Rust по адресу https://www.rust-lang.org/;
- веб-сайт Erlang по адресу https://www.erlang.org/;
- и снова страницу документации пакета `sync`. Обратите особое внимание на типы `sync.Mutex` и `sync.RWMutex`, которые появятся в следующей главе.

Упражнения

- Создайте конвейер, который бы читал текстовые файлы, вычислял количество вхождений заданной фразы в каждом текстовом файле и подсчитывал общее количество вхождений этой фразы во всех файлах.
- Создайте конвейер для вычисления суммы квадратов всех натуральных чисел в заданном диапазоне.
- Удалите из программы `simple.go` оператор `time.Sleep(1 * time.Second)` и посмотрите, что произойдет. Почему так происходит?

- Измените код Go в файле `pipeline.go` таким образом, чтобы получить конвейер из пяти функций и соответствующего количества каналов.
- Измените код Go в файле `pipeline.go` таким образом, чтобы узнать, что произойдет, если вы забудете закрыть канал `out` функции `first()`.

Резюме

В этой главе вы разобрались со множеств уникальных свойств Go, в том числе с горутинами, каналами и конвейерами. Кроме того, вы поняли, как предоставить горутинам достаточно времени, чтобы они успели завершить свою работу, используя функциональность, предлагаемую пакетом `sync`. Наконец, вы узнали, что каналы могут служить аргументами функций Go. Это позволяет разработчикам создавать конвейеры, по которым могут передаваться данные.

В следующей главе тема конкурентности в Go продолжится и вы познакомитесь с грозным ключевым словом `select`. Это слово помогает каналам Go выполнять много интересных задач, и я думаю, что вы будете просто поражены его мощью.

Вы откроете для себя два способа, позволяющих отключить одну или несколько горутин, которые по какой-то причине остановили работу. Кроме того, вы узнаете о нулевых, сигнальных и буферизованных каналах, каналах каналов, а также о пакете `context`.

В главе 10 раскрыто понятие *общей памяти*, которая является традиционным способом обмена информацией между потоками в рамках одного процесса UNIX, что также применяется и к горутинам. Однако в среде программистов Go общая память менее популярна, поскольку Go предлагает более эффективные, безопасные и быстрые способы обмена данными между горутинами.

10 Конкурентность в Go: расширенные возможности

В предыдущей главе вы познакомились с горутинами, которые являются главным функционалом Go, а также с каналами и конвейерами. В этой главе мы продолжим с того места, на котором остановились в предыдущей, чтобы глубже изучить горутины и каналы, а также познакомиться с ключевым словом `select`, после чего перейдем к обсуждению общих переменных, а также типов `sync.Mutex` и `sync.RWMutex`.

В данной главе рассмотрены примеры кода, в которых продемонстровано использование *сигнальных*, *буферизованных* и *нулевых каналов*, а также *каналов каналов*. Кроме того, вы узнаете о двух методах прерывания выполнения горутины, если оно превысило заданный лимит, потому что невозможно гарантировать, что все горутины будут завершены за желаемое время.

В завершение главы обсудим пакет `atomic`, состояние гонки, стандартный Go-пакет `context` и пул рабочих потоков.

Эта глава посвящена следующим темам:

- ключевое слово `select`;
- как работает планировщик Go;
- два метода, которые позволяют прервать выполнение программы, которая длится больше времени, чем ожидалось;
- сигнальные каналы;
- буферизованные каналы;
- нулевые каналы;
- управляющая горутина;
- общая память и мьютексы;
- типы `sync.Mutex` и `sync.RWMutex`;
- пакет `context` и его расширенная функциональность;
- пакет `atomic`;
- пулы обработчиков;
- распознавание состояния гонки.

И снова о планировщике Go

Планировщик отвечает за эффективное распределение работы, которую следует выполнить, между доступными ресурсами. В этом разделе мы гораздо глубже, чем в предыдущей главе, рассмотрим работу планировщика Go. Как вы уже знаете, Go работает с использованием *планировщика m:n* (или *планировщика M:N*). Планировщик Go оперирует горутинами, которые легче потоков ОС, используя при этом потоки ОС. Однако сначала — теория и определения полезных терминов.

В Go используется модель *конкурентности fork-join*. Слово fork (вилка) в названии модели говорит о том, что дочерняя ветвь может быть создана в любой точке программы. Аналогично слово join («объединение») говорит о том, что в некоторой точке дочерняя ветвь заканчивается и объединяется с родительской. Точками такого объединения, в частности, являются операторы `sync.Wait()` и каналы, которые собирают результаты выполнения горутин, тогда как каждая новая горутина создает дочернюю ветвь.

Фаза fork в модели fork-join и системный вызов fork(2) в C — это совершенно разные понятия.

Стратегия справедливого планирования довольно проста концептуально и имеет простую реализацию. Согласно этой стратегии, вся нагрузка должна равномерно распределяться между доступными процессорами. Поначалу такая стратегия может показаться идеальной, поскольку приходится учитывать не так уж много нюансов, при этом сохраняя одинаковую нагрузку всех процессоров. Однако на поверку оказывается, что это не совсем так, поскольку большинство распределенных задач обычно зависят друг от друга, из-за чего некоторые процессоры оказываются недостаточно загружены, или, что то же самое, одни процессоры используются интенсивнее, чем другие.

Горутина в Go — это *задача*, а все, что происходит после вызова горутины, является *продолжением*. В *стратегии перехвата работы*, используемой планировщиком Go, недостаточно загруженный (логический) процессор ищет дополнительную работу, которую выполняют другие процессоры. Найдя такие задания, он перехватывает их у другого процессора или нескольких процессоров — отсюда и название. Кроме того, алгоритм перехвата работы в Go перехватывает продолжения и ставит их в очередь. *Останавливающее соединение*, как следует из названия, — это место, в котором поток выполнения останавливается в точке соединения и начинает искать себе другую работу.

Несмотря на то что и при перехвате задач, и при перехвате продолжений имеет место останавливающее соединение, продолжения случаются чаще, чем задачи; поэтому алгоритм Go работает не с задачами, а с продолжениями.

Основным недостатком перехвата продолжений является то, что это требует от компилятора языка программирования дополнительной работы. К счастью, язык Go предоставляет эту дополнительную возможность, и поэтому в его алгоритме перехвата работы используется алгоритм *перехвата продолжений*.

Одним из преимуществ перехвата продолжений является то, что он дает одинаковые результаты при использовании функций вместо горутин или одного потока с несколькими горутинами. Это закономерно, так как в каждый момент в обоих случаях выполняется только что-то одно.

Теперь вернемся к используемому в Go алгоритму планирования *m:n*. Строго говоря, в у нас всегда есть *m* горутин, которые выполняются и, следовательно, запланированы для запуска в *n* потоках ОС, использующих не более `GOMAXPROCS` логических процессоров. О том, что такое `GOMAXPROCS`, вы скоро узнаете.

Планировщик Go работает с использованием трех основных типов сущностей: потоков ОС (*M*), которые связаны с используемой операционной системой, горутин (*G*) и логических процессоров (*P*). Количество процессоров, которые могут использоваться программой Go, определяется значением переменной среды `GOMAXPROCS`: в любое время число процессоров не может превышать `GOMAXPROCS`.

Этот момент проиллюстрирован на рис. 10.1.

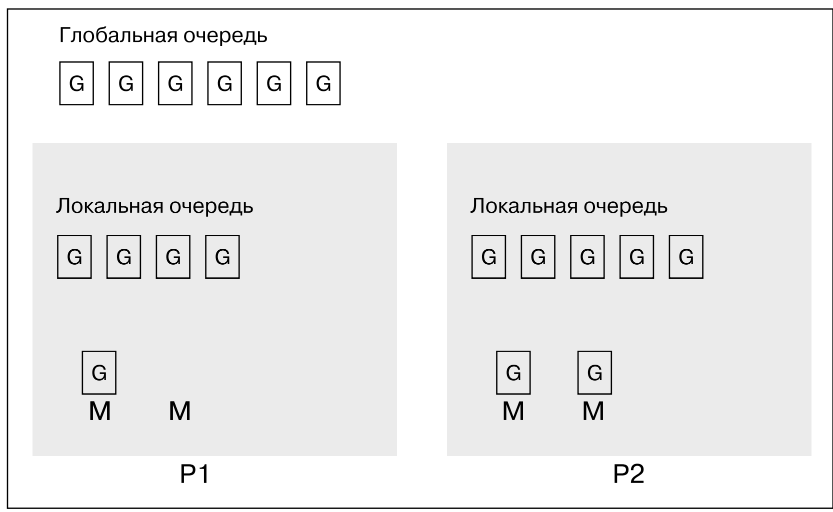


Рис. 10.1. Так работает планировщик Go

Как видно на рис. 10.1, существует два вида очередей: глобальная и локальная, причем каждая локальная очередь присоединена к соответствующему ло-

гическому процессору. Горутины из глобальной очереди назначаются очередям логических процессоров, где они и выполняются. Таким образом, планировщик Go должен проверить глобальную очередь, чтобы избежать выполнения горутин, расположенных только в локальной очереди какого-либо из логических процессоров. Однако планировщик не проверяет глобальную очередь постоянно. Следовательно, у глобальной очереди нет преимуществ по сравнению с локальной.

Кроме того, каждый логический процессор может иметь несколько потоков и возможен перехват между локальными очередями доступных логических процессоров. Имейте в виду, что планировщик Go может при необходимости создавать дополнительные потоки ОС. Однако потоки ОС весьма затратны, поэтому слишком активные действия с ними могут замедлить работу приложений Go.

Помните, что использование большего количества горутин в программе не панацея для повышения производительности, так как большое количество горутин в сочетании с многочисленными вызовами `sync.Add()`, `sync.Wait()` и `sync.Done()` может замедлить работу программы из-за дополнительных обслуживающих операций, которые должен выполнять планировщик Go.

Подобно большинству компонентов Go, планировщик Go постоянно развивается. Те, кто работает над планировщиком Go, все время стремятся повысить его производительность, внося небольшие изменения в его работу. Однако основные принципы остаются прежними.

Вам не нужно быть в курсе всего этого, чтобы писать код Go, использующий горутины. Но знание того, что происходит внутри кода, определенно поможет вам разобраться, в чем дело, если вдруг начнут происходить странные вещи или если вам станет интересно, как работает планировщик Go. Благодаря этому вы, безусловно, станете лучше как разработчик!

Переменная среды GOMAXPROCS

Переменная среды (и функция Go) `GOMAXPROCS` позволяет ограничить количество потоков операционной системы, которые могут одновременно выполнять код Go на уровне пользователя. Начиная с версии Go 1.5, по умолчанию значение `GOMAXPROCS` равно количеству логических ядер, доступных на UNIX-компьютере.

Если `GOMAXPROCS` меньше числа ядер на UNIX-компьютере, то это может привести к снижению производительности программы. Однако использование `GOMAXPROCS`, превышающего число доступных ядер, не обязательно заставит программы Go работать быстрее.

Чтобы программно узнать значение переменной среды `GOMAXPROCS`, нужно воспользоваться следующим кодом. Этот код вы найдете в программе с именем `maxprocs.go`:

```
package main

import (
    "fmt"
    "runtime"
)

func getGOMAXPROCS() int {
    return runtime.GOMAXPROCS(0)
}

func main() {
    fmt.Printf("GOMAXPROCS: %d\n", getGOMAXPROCS())
}
```

Выполнение `maxprocs.go` на машине с процессором Intel i7 даст следующий результат:

```
$ go run maxprocs.go
GOMAXPROCS: 8
```

Тем не менее мы можем изменить этот результат, если изменим значение переменной среды `GOMAXPROCS` перед выполнением программы, выполнив следующие команды в UNIX-оболочке bash(1):

```
$ go version
go version go1.12.3 darwin/amd64
$ export GOMAXPROCS=800; go run maxprocs.go
GOMAXPROCS: 800
$ export GOMAXPROCS=4; go run maxprocs.go
GOMAXPROCS: 4
```

Ключевое слово select

Как вы вскоре узнаете, ключевое слово `select` довольно мощное; оно может делать многое в самых разных ситуациях. Оператор `select` в Go похож на `switch`, но только для каналов. На практике это означает, что `select` позволяет горутине дождаться завершения нескольких операций коммуникации. Поэтому основное преимущество `select` состоит в том, что этот оператор дает возможность работать с несколькими каналами, используя один блок `select`. В результате, построив соответствующие блоки `select`, можно выполнять неблокирующие операции с каналами.

Самая большая проблема при использовании нескольких каналов и ключевого слова select — это взаимоблокировки. Это означает, что необходимо быть особенно осторожными при проектировании и разработке, чтобы избежать таких взаимоблокировок.

Мы рассмотрим использование ключевого слова `select` на примере программы `select.go`. Разделим ее на пять частей. Первая часть `select.go` содержит следующий код Go:

```
package main

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

Вторая часть кода `select.go` выглядит так:

```
func gen(min, max int, createNumber chan int, end chan bool) {
    for {
        select {
        case createNumber <- rand.Intn(max-min) + min:
        case <-end:
            close(end)
            return
        case <-time.After(4 * time.Second):
            fmt.Println("\ntime.After()!")
        }
    }
}
```

Итак, что в действительности происходит в коде этого блока `select`? В данном операторе `select` предусмотрено три варианта. Обратите внимание, что операторы `select` не требуют ветки `default`. В качестве «умной» ветки `default` в этом коде можно рассматривать третью ветку оператора `select`. Так происходит потому, что `time.After()` ожидает истечения заданного интервала, после чего передает значение текущего времени по возвращаемому каналу — это разблокирует оператор `select`, если все остальные каналы по какой-либо причине окажутся заблокированными.

Оператор `select` не выполняется последовательно, так как все его каналы проверяются одновременно. Если ни один из каналов, указанных в операторе `select`, не доступен, то оператор `select` будет заблокирован, пока не освободится один из каналов. Если доступны сразу несколько каналов оператора `select`,

то среда выполнения Go сделает случайный выбор из набора этих доступных каналов.

Среда выполнения Go пытается делать случайный выбор между доступными каналами как можно более равномерным и справедливым.

Третья часть `select.go` выглядит так:

```
func main() {
    rand.Seed(time.Now().Unix())
    createNumber := make(chan int)
    end := make(chan bool)
    if len(os.Args) != 2 {
        fmt.Println("Please give me an integer!")
        return
    }
```

Четвертая часть программы `select.go` содержит следующий код Go:

```
    n, _ := strconv.Atoi(os.Args[1])
    fmt.Printf("Going to create %d random numbers.\n", n)
    go gen(0, 2*n, createNumber, end)
    for i := 0; i < n; i++ {
        fmt.Printf("%d ", <-createNumber)
    }
```

Значение error, возвращаемое функцией strconv.Atoi(), не проверяется из соображения экономии места. Никогда не поступайте так в реальных приложениях.

Оставшийся код Go программы `select.go` выглядит так:

```
    time.Sleep(5 * time.Second)
    fmt.Println("Exiting...")
    end <- true
}
```

Главное назначение оператора `time.Sleep(5 * time.Second)` — предоставить функциям `time.After()` и `gen()` достаточно времени, чтобы отработать, вернуть результат и таким образом активизировать соответствующую ветвь оператора `select`.

Последний оператор функции `main()` завершает программу, активизируя ветвь `case <-end` оператора `select` в `gen()` и реализуя соответствующий код Go.

Выполнение `select.go` приведет к следующему:

```
$ go run select.go 10
Going to create 10 random numbers.
13 17 8 14 19 9 2 0 19 5
time.After()!
Exiting...
```

Главным преимуществом оператора select является то, что он может подключать, распределять нагрузку и управлять несколькими каналами. Поскольку каналы служат для связи между горутинами, select соединяет каналы, которые соединяют горутины. Таким образом, оператор select является одной из наиболее важных, если не самой важной, частью модели конкурентности в Go.

Принудительное завершение горутины

В этом разделе представлены два очень важных метода, позволяющих прервать горутину, если она выполняется слишком долго. Проще говоря, эти два способа избавят вас от необходимости ждать до бесконечности, пока программа завершит работу, и предоставят полный контроль над временем, в течение которого вы готовы ожидать завершения горутины. В обоих методах используются возможности удобного ключевого слова `select` в сочетании с функцией `time.After()`, которая уже встречалась в предыдущем разделе.

Принудительное завершение горутины, способ 1

Исходный код с примером первого способа хранится в файле `timeOut1.go`. Рассмотрим его, разделив на четыре части.

Первая часть `timeOut1.go` содержит следующий код Go:

```
package main

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

Второй фрагмент кода `timeOut1.go` выглядит так:

```
func main() {
    c1 := make(chan string)
    go func() {
        time.Sleep(time.Second * 3)
        c1 <- "c1 OK"
    }()
```

Вызов `time.Sleep()` используется для эмуляции времени, которое обычно занимает выполнение функции. В данном случае выполнение анонимной функции, которая

запускается как горутина, займет около трех секунд (`time.Second * 3`), после чего записывается сообщение в канал `c1`.

Третий фрагмент `timeOut1.go` содержит следующий код Go:

```
    select {
    case res := <-c1:
        fmt.Println(res)
    case <-time.After(time.Second * 1):
        fmt.Println("timeout c1")
    }
```

Цель вызова функции `time.After()` — ожидать в течение заданного времени. В данном случае нас интересует не значение, возвращаемое функцией `time.After()`, а факт завершения вызова функции `time.After()`, который означает, что доступное время ожидания истекло. В этом случае, поскольку значение, передаваемое в функцию `time.After()`, меньше, чем значение, используемое в вызове `time.Sleep()`, который был выполнен как горутина в предыдущем фрагменте кода, вы, скорее всего, получите сообщение о превышении времени ожидания.

Остальной код из `timeOut1.go` выглядит так:

```
    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")
    }
}
```

В этом коде запускается горутина, выполнение которой занимает около трех секунд из-за вызова `time.Sleep()`. Затем определяется интервал ожидания длительностью четыре секунды с использованием вызова `time.After(4 * time.Second)`. Если функция `time.After(4 * time.Second)` завершится позже, чем будет получено значение из канала `c2`, указанного в первом варианте блока `select`, то время выполнения горутины не будет превышено. В противном случае она превысит выделенный ей лимит времени! Однако при этом значение `time.After()` обеспечивает достаточно времени для возврата к вызову `time.Sleep()`, поэтому вы, скорее всего, не получите сообщение о превышении временного лимита.

Выполнение `timeOut1.go` приведет к результатам следующего вида:

```
$ go run timeOut1.go
timeout c1
c2 OK
```

Как и ожидалось, первая горутина не успела завершиться, в то время как у второй было достаточно времени, чтобы закончить работу.

Принудительное завершение горутины, способ 2

Исходный код, демонстрирующий второй способ прерывания горутин, хранится в файле `timeOut2.go`. Рассмотрим его, разделив на пять частей. На этот раз интервал ожидания будет предоставлен в качестве аргумента командной строки программы.

Первая часть `timeOut2.go` выглядит так:

```
package main

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

Во втором фрагменте `timeOut2.go` содержится следующий код Go:

```
func timeout(w *sync.WaitGroup, t time.Duration) bool {
    temp := make(chan int)
    go func() {
        defer close(temp)
        time.Sleep(5 * time.Second)
        w.Wait()
    }()

    select {
    case <-temp:
        return false
    case <-time.After(t):
        return true
    }
}
```

В этом коде временной интервал, который будет использоваться при вызове `time.After()`, является аргументом функции `timeout()`, следовательно, является переменной величиной. Как и в прошлый раз, логика прерывания в случае превышения временного лимита реализована в блоке `select`. Кроме того, вызов `w.Wait()` заставит функцию `timeout()` бесконечно ожидать соответствующую функцию `sync.Done()`, чтобы завершить работу. Когда `w.Wait()` закончит работу, будет выполнена первая ветвь оператора `select`.

Третья часть кода `timeOut2.go` выглядит так:

```
func main() {
    arguments := os.Args
    if len(arguments) != 2 {
        fmt.Println("Need a time duration!")
        return
    }

    var w sync.WaitGroup
    w.Add(1)

    t, err := strconv.Atoi(arguments[1])
    if err != nil {
        fmt.Println(err)
        return
    }
```

Четвертая часть программы `timeOut2.go`:

```
    duration := time.Duration(int32(t)) * time.Millisecond
    fmt.Printf("Timeout period is %s\n", duration)

    if timeout(&w, duration) {
        fmt.Println("Timed out!")
    } else {
        fmt.Println("OK!")
    }
```

Функция `time.Duration()` преобразует целочисленное значение в переменную `time.Duration`, которую мы будем использовать позже.

Оставшийся код Go из `timeOut2.go` выглядит так:

```
    w.Done()
    if timeout(&w, duration) {
        fmt.Println("Timed out!")
    } else {
        fmt.Println("OK!")
    }
}
```

После завершения вызова `w.Done()` предшествующая ему функция `timeout()` также завершится. Однако у второго вызова `timeout()` нет оператора `sync.Done()`, которого она могла бы ожидать.

Выполнение `timeOut2.go` приведет к результатам следующего вида:

```
$ go run timeOut2.go 10000
Timeout period is 10s
Timed out!
OK!
```

При таком выполнении `timeOut2.go` период ожидания длится больше, чем вызов анонимной горутины `time.Sleep(5 * time.Second)`. Однако без обязательного вызова `w.Done()` анонимная горутина не может завершиться, поэтому `time.After(t)` закончится раньше и функция `timeout()` первого оператора `if` вернет `true`. Во втором операторе `if` анонимной функции не приходится ждать, поэтому функция `timeout()` вернет `false`, так как `time.Sleep(5 * time.Second)` завершится раньше, чем `time.After(t)`.

```
$ go run timeOut2.go 100
Timeout period is 100ms
Timed out!
Timed out!
```

Однако при втором запуске программы период ожидания слишком мал, поэтому в обоих случаях вызова `timeout()` у функции не будет достаточно времени для завершения работы и она будет принудительно прервана. Поэтому, выбирая интервал ожидания, убедитесь, что это подходящее значение, иначе ваши результаты могут не соответствовать ожиданиям.

И снова о Go-каналах

Как только задействуется ключевое слово `select`, появляется несколько уникальных способов использования Go-каналов, позволяющих сделать гораздо больше, чем то, что вы видели в главе 9. В этом разделе вы узнаете о разных способах использования Go-каналов.

Напомню, что нулевым значением для каналов является `nil`, и если отправить сообщение в закрытый канал, то программа перейдет в режим паники. Но если вы попытаетесь прочитать данные из закрытого канала, то получите нулевое значение для данного типа канала. Таким образом, после закрытия канала вы больше не можете записывать в него данные, но все равно можете читать его.

Чтобы канал можно было закрыть, он не должен быть предназначен только для приема данных. Кроме того, нулевой канал всегда блокируется, то есть попытка чтения или записи с нулевого канала заблокирует канал. Это свойство каналов очень полезно в тех случаях, когда нужно отключить ветвь оператора `select`, — для этого достаточно присвоить переменной канала значение `nil`.

Наконец, при попытке закрыть нулевой канал программа поднимет панику. Рассмотрим это на примере программы `closeNilChannel.go`:

```
package main

func main() {
    var c chan string
    close(c)
}
```

Выполнение `closeNilChannel.go` приведет к следующему результату:

```
$ go run closeNilChannel.go
panic: close of nil channel
goroutine 1 [running]:
main.main()
    /Users/mtsouk/closeNilChannel.go:5 +0x2a
exit status 2
```

Сигнальные каналы

Сигнальный канал — это канал, который применяется только для передачи сигналов. Проще говоря, сигнальный канал можно использовать в тех случаях, когда вы хотите проинформировать другую программу о чем-либо. К сигнальным каналам не нужно прибегать для передачи данных.

Не следует путать сигнальные каналы с обработкой сигналов UNIX, о которой говорилось в главе 8, это совершенно разные вещи.

Пример кода, в котором используются сигнальные каналы, мы рассмотрим далее в этой главе.

Буферизованные каналы

Тема этого подраздела — буферизованные каналы. Это каналы, которые позволяют планировщику Go быстро размещать задания в очереди, чтобы обрабатывать больше запросов. Кроме того, их можно использовать в качестве *семафоров*, позволяющих ограничить пропускную способность приложения.

Представленный здесь метод работает так: все входящие запросы перенаправляются на канал, который обрабатывает их по очереди. Когда канал завершает обработку запроса, он отправляет исходному вызывающему объекту сообщение о том, что канал готов обработать новый запрос. Таким образом, емкость буфера канала ограничивает количество одновременных запросов, которые этот канал может хранить.

Мы рассмотрим этот метод на примере кода программы `bufChannel.go`. Разделим его на четыре части.

Первая часть кода `bufChannel.go` выглядит так:

```
package main

import (
    "fmt"
)
```

Второй фрагмент файла bufChannel.go содержит следующий код Go:

```
func main() {
    numbers := make(chan int, 5)
    counter := 10
```

Представленное здесь определение numbers позволяет хранить в этом канале до пяти целых чисел.

В третьей части bufChannel.go содержится следующий код Go:

```
    for i := 0; i < counter; i++ {
        select {
        case numbers <- i:
        default:
            fmt.Println("Not enough space for", i)
        }
    }
```

В этом коде мы попытались поместить в канал numbers десять чисел. Однако, поскольку в numbers есть место только для пяти целых чисел, сохранить в нем все десять целых чисел нам не удастся.

Остальной код Go из файла bufChannel.go выглядит так:

```
    for i := 0; i < counter+5; i++ {
        select {
            case num := <-numbers:
                fmt.Println(num)
            default:
                fmt.Println("Nothing more to be done!")
            break
        }
    }
}
```

В этом коде Go мы попытались прочитать содержимое канала numbers с помощью цикла for и оператора select. Пока в канале numbers есть что читать, будет выполняться первая ветвь оператора select. Когда канал numbers пустой, выполняется ветвь default.

Выполнение bufChannel.go приведет к результату следующего вида:

```
$ go run bufChannel.go
Not enough space for 5
Not enough space for 6
Not enough space for 7
Not enough space for 8
Not enough space for 9
0
1
2
3
4
```

```
Nothing more to be done!
Nothing more to be done!
Nothing more to be done!
Nothing more to be done!
Nothing more to be done!
Nothing more to be done!
Nothing more to be done!
Nothing more to be done!
Nothing more to be done!
Nothing more to be done!
```

Нулевые каналы

В этом разделе вы познакомитесь с нулевыми каналами. Это особый вид каналов, который всегда блокирован. Мы рассмотрим эти каналы на примере программы `nilChannel.go`. Разделим ее на четыре фрагмента кода.

Первая часть `nilChannel.go` выглядит так:

```
package main

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

Во второй части `nilChannel.go` содержится следующий код Go:

```
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)
        }
    }
}
```

Здесь на примере функции `add()` показано, как используется нулевой канал. Оператор `<-t.C` блокирует канал `C` таймера `t` на время, указанное в вызове `time.NewTimer()`. Не путайте канал `c`, который является аргументом функции, с каналом `t.C`, который принадлежит таймеру `t`. По истечении времени таймер отправляет значение в канал `t.C`, чем инициирует выполнение соответствующей ветви оператора

`select` — он присваивает каналу `c` значение `nil` и выводит на экран значение переменной `sum`.

Третий фрагмент кода `nilChannel.go` выглядит так:

```
func send(c chan int) {
    for {
        c <- rand.Intn(10)
    }
}
```

Цель функции `send()` — генерировать случайные числа и отправлять их в канал до тех пор, пока канал открыт.

Остальной код Go в `nilChannel.go` выглядит так:

```
func main() {
    c := make(chan int)
    go add(c)
    go send(c)

    time.Sleep(3 * time.Second)
}
```

Функция `time.Sleep()` нужна для того, чтобы у двух горутин было достаточно времени для выполнения.

Запуск `nilChannel.go` приведет к следующим результатам:

```
$ go run nilChannel.go
13167523
$ go run nilChannel.go
12988362
```

Поскольку количество выполнений первой ветви оператора `select` в функции `add()` не является фиксированным, запустив `nilChannel.go` несколько раз, мы получим разные результаты.

Каналы каналов

Канал каналов — это особая разновидность переменной канала, которая вместо обычных типов переменных работает с другими каналами. Тем не менее для канала каналов все равно нужно объявить тип данных. Для того чтобы определить канал каналов, нужно использовать ключевое слово `chan` два раза подряд, как показано в следующем выражении:

```
c1 := make(chan chan int)
```

Другие типы каналов, представленные в этой главе, более популярны и полезны, чем каналы каналов.

Мы рассмотрим использование каналов каналов на примере кода, который находится в файле `chSquare.go`. Разделим его на четыре части.

Первая часть `chSquare.go` выглядит так:

```
package main

import (
    "fmt"
    "os"
    "strconv"
    "time"
)

var times int
```

Вторая часть `chSquare.go` содержит следующий код Go:

```
func f1(cc chan chan int, f chan bool) {
    c := make(chan int)
    cc <- c
    defer close(c)

    sum := 0
    select {
    case x := <-c:
        for i := 0; i <= x; i++ {
            sum = sum + i
        }
        c <- sum
    case <-f:
        return
    }
}
```

Объявив обычный канал типа `int`, передаем его переменной канала каналов. Затем с помощью оператора `select` мы получаем возможность читать данные из обычного канала типа `int` или выходить из функции, используя сигнальный канал `f`.

Прочитав одно значение из канала `c`, мы запускаем цикл `for`, вычисляющий сумму всех целых чисел от `0` до целочисленного значения, которое мы только что прочитали. Затем отправляем вычисленное значение в канал `c` типа `int` и на этом завершаем работу.

В третьей части `chSquare.go` содержится следующий код Go:

```
func main() {
    arguments := os.Args
    if len(arguments) != 2 {
        fmt.Println("Need just one integer argument!")
        return
    }
```

```
    times, err := strconv.Atoi(arguments[1])
    if err != nil {
        fmt.Println(err)
        return
    }

    cc := make(chan chan int)
```

В последней строке этого фрагмента кода мы объявляем переменную канала каналов с именем `cc`. Эта переменная является звездой данной программы, потому что именно от нее здесь все зависит. Переменная `cc` передается в функцию `f1()` и используется в следующем цикле `for`.

Оставшийся код Go программы `chSquare.go` выглядит так:

```
    for i := 1; i < times+1; i++ {
        f := make(chan bool)
        go f1(cc, f)
        ch := <-cc
        ch <- i
        for sum := range ch {
            fmt.Print("Sum(", i, ")=", sum)
        }
        fmt.Println()
        time.Sleep(time.Second)
        close(f)
    }
}
```

Канал `f` является сигнальным каналом для завершения горутины, когда вся работа закончена. Инструкция `ch: = <-cc` позволяет получить из переменной канала каналов обычный канал, чтобы передать туда значение типа `int` с помощью оператора `ch <- i`. После этого мы читаем данные из канала, используя цикл `for`. Функция `f1()` запрограммирована на возврат одного значения, однако мы можем прочитать и несколько значений. Обратите внимание, что каждое значение `i` обслуживается своей горутиной.

Тип сигнального канала может быть любым, включая `bool`, примененный в предыдущем коде, и `struct{}`, который будет использоваться для сигнального канала в следующем разделе. Главным преимуществом сигнального канала типа `struct{}` является то, что в такой канал нельзя отправлять данные, это предотвращает возникновение ошибок.

Выполнение `chSquare.go` приведет к результатам такого вида:

```
$ go run chSquare.go 4
Sum(1)=1
Sum(2)=3
Sum(3)=6
Sum(4)=10
```

```
$ go run chSquare.go 7
Sum(1)=1
Sum(2)=3
Sum(3)=6
Sum(4)=10
Sum(5)=15
Sum(6)=21
Sum(7)=28
```

Выбор последовательности исполнения горутин

Вы не обязаны делать какие-либо предположения относительно последовательности выполнения горутин. Однако бывают случаи, когда необходимо контролировать этот порядок. В данном подразделе вы узнаете, как это делать с помощью сигнальных каналов.

Вы спросите: «Зачем создавать горутины, а затем выполнять их в заданном порядке, если то же самое гораздо легче сделать с помощью обычных функций?» Ответ прост: горутины способны работать одновременно и ожидать завершения других горутин, тогда как функции не могут этого делать, поскольку выполняются последовательно.

В этом подразделе мы рассмотрим программу Go, которая называется `defineOrder.go`. Разделим ее на пять частей. Первая часть `defineOrder.go` выглядит так:

```
package main

import (
    "fmt"
    "time"
)

func A(a, b chan struct{}) {
    <-a
    fmt.Println("A()!")
    time.Sleep(time.Second)
    close(b)
}
```

Функция `A()` заблокирована каналом, который хранится в параметре `a`. Как только этот канал будет разблокирован в `main()`, функция `A()` начнет работать. В конце она закроет канал `b`, тем самым разблокировав другую функцию — в данном случае `B()`.

Вторая часть `defineOrder.go` содержит следующий код Go:

```
func B(a, b chan struct{}) {
    <-a
    fmt.Println("B()!")
    close(b)
}
```

Логика функции `B()` такая же, как и у `A()`. Эта функция блокируется, пока не будет закрыт канал `a`. Затем она выполняет свою работу и закрывает канал `b`. Обратите внимание, что каналы `a` и `b` ссылаются на имена параметров функции.

Третий фрагмент кода `defineOrder.go` выглядит так:

```
func C(a chan struct{}) {
    <-a
    fmt.Println("C()!")
}
```

Функция `C()` заблокирована и ожидает закрытия канала `a`, чтобы начать работу.

В четвертой части `defineOrder.go` содержится следующий код:

```
func main() {
    x := make(chan struct{})
    y := make(chan struct{})
    z := make(chan struct{})
```

Эти три канала станут параметрами трех функций.

В последнем фрагменте `defineOrder.go` содержится следующий код Go:

```
    go C(z)
    go A(x, y)
    go C(z)
    go B(y, z)
    go C(z)

    close(x)
    time.Sleep(3 * time.Second)
}
```

Здесь программа выполняет все нужные функции, а потом закрывает канал `x` и засыпает на три секунды.

Выполнение `defineOrder.go` приведет к желаемому результату, несмотря на то что функция `C()` будет вызвана несколько раз:

```
$ go run defineOrder.go
A()!
B()!
C()!
C()!
C()!
```

Многократный вызов функции `C()` как горутины не вызовет проблем, потому что `C()` не закрывает никаких каналов. Но если вызвать `A()` или `B()` более одного раза, то, скорее всего, выведется сообщение об ошибке, например такое:

```
$ go run defineOrder.go
A()!
A()!
B()!
C()!
C()!
C()!
panic: close of closed channel
goroutine 7 [running]:
main.A(0xc420072060, 0xc4200720c0)
    /Users/mtsouk/Desktop/defineOrder.go:12 +0x9d
created by main.main
    /Users/mtsouk/Desktop/defineOrder.go:33 +0xfa
exit status 2
```

Как видим, здесь функция `A()` была вызвана дважды. Однако, когда `A()` закрывает канал, одна из ее горутин обнаруживает, что канал уже закрыт, и создает ситуацию паники, когда пытается закрыть этот канал снова. Если мы попытаемся более одного раза вызвать функцию `B()`, то получим похожую ситуацию паники.

Как не надо использовать горутины

В этом разделе вы познакомитесь с наивным способом сортировки натуральных чисел с помощью горутин. Программа, которую мы рассмотрим, называется `sillySort.go`. Разделим ее на две части. Первая часть `sillySort.go` выглядит так:

```go
package main

import (
    "fmt"
    "os"
    "strconv"
    "sync"
    "time"
)

func main() {
    arguments := os.Args

    if len(arguments) == 1 {
        fmt.Println(os.Args[0], "n1, n2, [n]")
        return
    }
```

```
    var wg sync.WaitGroup
    for _, arg := range arguments[1:] {
        n, err := strconv.Atoi(arg)
        if err != nil || n < 0 {
            fmt.Print(". ")
            continue
        }
```

Вторая часть `sillySort.go` содержит следующий код Go:

```
        wg.Add(1)
        go func(n int) {
            defer wg.Done()
            time.Sleep(time.Duration(n) * time.Second)
            fmt.Print(n, " ")
        }(n)
    }

    wg.Wait()
    fmt.Println()
}
```

Сортировка выполняется посредством вызова функции `time.Sleep()` — чем больше натуральное число, тем больше проходит времени перед выполнением оператора `fmt.Print()`!

Выполнение `sillySort.go` приведет к результатам такого вида:

```
$ go run sillySort.go a -1 1 2 3 5 0 100 20 60
. . 0 1 2 3 5 20 60 100
$ go run sillySort.go a -1 1 2 3 5 0 100 -1 a 20 hello 60
. . . . . 0 1 2 3 5 20 60 100
$ go run sillySort.go 0 0 10 2 30 3 4 30
0 0 2 3 4 10 30 30
```

Общая память и общие переменные

Общая память и *общие переменные* — наиболее распространенные способы взаимодействия потоков UNIX между собой.

Переменные *мьютекса* (сокращенное mutual exclusion — «взаимное исключение») используются главным образом для синхронизации потоков и для защиты общих данных, когда нужно исключить возможность одновременного выполнения нескольких операций записи. Мьютекс работает как буферизованный канал с емкостью, равной единице, так что в любой момент только одна горутина имеет доступ к общей переменной. Это означает, что две и более горутины не могут изменить эту переменную одновременно.

Код, который не может выполняться одновременно всеми процессами, потоками или в данном случае горутинами, называется *критическим разделом* конкурентной программы. Такой код должен быть защищен мьютексами. Таким образом,

определение критических разделов кода упростит весь процесс программирования, поэтому следует уделить особое внимание данной задаче.

Критический раздел не может быть встроен в другой критический раздел, если оба критических раздела используют одну и ту же переменную sync.Mutex или sync.RWMutex. Проще говоря, избегайте распределения мьютексов между функциями, так как это мешает понять, внедряете вы один критический раздел в другой или нет.

В следующих двух подразделах продемонстрировано использование типов `sync.Mutex` и `sync.RWMutex`.

Тип sync.Mutex

Тип `sync.Mutex` — это реализация мьютекса в Go. Определение этого типа находится в файле `mutex.go` каталога `sync` и выглядит так:

```
// Мьютекс (mutex) — сокращение mutual exclusion lock, предохранитель
// от взаимных блокировок.
// Нулевое значение для мьютекса означает открытый мьютекс.
// Однажды использованный мьютекс нельзя копировать.
type Mutex struct {
    state int32
    sema uint32
}
```

Если вам интересно, что делает этот код в стандартной библиотеке, помните, что Go является полностью открытым исходным кодом, так что вы можете спокойно открыть и прочитать его.

В определении типа `sync.Mutex` нет ничего необычного. Основная работа выполняется функциями `sync.Lock()` и `sync.Unlock()`, первая из которых блокирует, а вторая разблокирует мьютекс `sync.Mutex`. Когда мьютекс блокирован, это означает, что никто другой не может заблокировать этот мьютекс, пока он не будет освобожден с помощью функции `sync.Unlock()`.

Мы рассмотрим использование типа `sync.Mutex` на примере программы `mutex.go`. Разделим ее на пять частей.

Первый фрагмент кода `mutex.go` выглядит так:

```
package main

import (
    "fmt"
```

```
    "os"
    "strconv"
    "sync"
    "time"
)

var (
    m sync.Mutex
    v1 int
)
```

Во второй части `mutex.go` содержится следующий код Go:

```
func change(i int) {
    m.Lock()
    time.Sleep(time.Second)
    v1 = v1 + 1
    if v1%10 == 0 {
        v1 = v1 - 10*i
    }
    m.Unlock()
}
```

Важнейшей частью этой функции является код Go между операторами `m.Lock()` и `m.Unlock()`.

Третья часть `mutex.go`:

```
func read() int {
    m.Lock()
    a := v1
    m.Unlock()
    return a
}
```

Здесь, как и в предыдущем фрагменте, главный раздел функции находится между операторами `m.Lock()` и `m.Unlock()`.

Четвертый фрагмент кода `mutex.go` выглядит так:

```
func main() {
    if len(os.Args) != 2 {
        fmt.Println("Please give me an integer!")
        return
    }

    numGR, err := strconv.Atoi(os.Args[1])
    if err != nil {
        fmt.Println(err)
        return
    }
    var waitGroup sync.WaitGroup
```

Последняя часть mutex.go содержит следующий код Go:

```
    fmt.Printf("%d ", read())
    for i := 0; i < numGR; i++ {
        waitGroup.Add(1)
        go func(i int) {
            defer waitGroup.Done()
            change(i)
            fmt.Printf("-> %d", read())
        }(i)
    }

    waitGroup.Wait()
    fmt.Printf("-> %d\n", read())
}
```

Выполнение mutex.go приведет к результатам следующего вида:

```
$ go run mutex.go 21
0 -> 1-> 2-> 3-> 4-> 5-> 6-> 7-> 8-> 9-> -30-> -29-> -28-> -27-> -26->
-25-> -24-> -23-> -22-> -21-> -210-> -209-> -209
$ go run mutex.go 21
0 -> 1-> 2-> 3-> 4-> 5-> 6-> 7-> 8-> 9-> -130-> -129-> -128-> -127-> -126->
-125-> -124-> -123-> -122-> -121-> -220-> -219-> -219
$ go run mutex.go 21
0 -> 1-> 2-> 3-> 4-> 5-> 6-> 7-> 8-> 9-> -100-> -99-> -98-> -97-> -96->
-95-> -94-> -93-> -92-> -91-> -260-> -259-> -259
```

Если удалить операторы m.Lock() и m.Unlock() из функции change(), то программа выдаст результат, подобный следующему:

```
$ go run mutex.go 21
0 -> 1-> 6-> 7-> 5-> -60-> -59-> 9-> 2-> -58-> 3-> -52-> 4-> -57-> 8->
-55-> -90-> -54-> -89-> -53-> -56-> -51-> -89
$ go run mutex.go 21
0 -> 1-> 7-> 8-> 9-> 5-> -99-> 4-> 2-> -97-> -96-> 3-> -98-> -95-> -100->
-93-> -94-> -92-> -91-> -230-> 6-> -229-> -229
$ go run mutex.go 21
0 -> 3-> 7-> 8-> 9-> -120-> -119-> -118-> -117-> 1-> -115-> -114-> -116->
4-> 6-> -112-> 2-> -111-> 5-> -260-> -113-> -259-> -259
```

Причиной такого изменения результатов программы является то, что все процедуры изменяют общую переменную одновременно, из-за чего результаты выводятся в случайном порядке.

Что произойдет, если вы забудете разблокировать мьютекс

В этом пункте вы узнаете, что произойдет, если забыть разблокировать sync.Mutex. Для этого мы рассмотрим пример кода Go из файла forgetMutex.go. Разделим его на две части.

Первая часть forgetMutex.go содержит следующий код Go:

```
package main

import (
    "fmt"
    "sync"
)

var m sync.Mutex

func function() {
    m.Lock()
    fmt.Println("Locked!")
}
```

Причиной всех проблем в этой программе является то, что разработчик забыл снять блокировку с sync.Mutex. Однако если программа вызывает function() только один раз, то все будет отлично.

Вторая часть forgetMutex.go выглядит так:

```
func main() {
    var w sync.WaitGroup

    go func() {
        defer w.Done()
        function()
    }()
    w.Add(1)

    go func() {
        defer w.Done()
        function()
    }()
    w.Add(1)

    w.Wait()
}
```

В функции main(), которая генерирует только две горутины и ожидает их завершения, нет ничего плохого.

Выполнение forgetMutex.go приведет к следующим результатам:

```
$ go run forgetMutex.go
Locked!
fatal error: all goroutines are asleep - deadlock!
goroutine 1 [semacquire]:
sync.runtime_Semacquire(0xc42001209c)
    /usr/local/Cellar/go/1.12.3/libexec/src/runtime/sema.go:56 +0x39
sync.(*WaitGroup).Wait(0xc420012090)
    /usr/local/Cellar/go/1.12.3/libexec/src/sync/waitgroup.go:131 +0x72
```

```
main.main()
    /Users/mtsouk/forgetMutex.go:30 +0xb6
goroutine 5 [semacquire]:
sync.runtime_SemacquireMutex(0x115c6fc, 0x0)
    /usr/local/Cellar/go/1.12.3/libexec/src/runtime/sema.go:71 +0x3d
sync.(*Mutex).Lock(0x115c6f8)
    /usr/local/Cellar/go/1.12.3/libexec/src/sync/mutex.go:134 +0xee
main.function()
    /Users/mtsouk/forgetMutex.go:11 +0x2d
main.main.func1(0xc420012090)
    /Users/mtsouk/forgetMutex.go:20 +0x48
created by main.main
    /Users/mtsouk/forgetMutex.go:18 +0x58
exit status 2
```

Таким образом, если вы забудете разблокировать мьютекс `sync.Mutex`, то создадите ситуацию паники даже в самой простой программе. То же самое относится к мьютексам типа `sync.RWMutex`. Они рассмотрены в следующем подразделе.

Тип sync.RWMutex

Тип `sync.RWMutex` — это еще один вид мьютекса. Фактически это улучшенная версия `sync.Mutex`, определение которой находится в файле `rwmutex.go` каталога `sync`:

```
type RWMutex struct {
        w           Mutex  // held if there are pending writers
        writerSem   uint32 // semaphore for writers to wait for completing readers
        readerSem   uint32 // semaphore for readers to wait for completing writers
        readerCount int32  // number of pending readers
        readerWait  int32  // number of departing readers
}
```

Другими словами, тип `sync.RWMutex` основан на `sync.Mutex` с необходимыми дополнениями и улучшениями.

Теперь поговорим о том, что в `sync.RWMutex` улучшено по сравнению с `sync.Mutex`. Несмотря на то что только одной функции разрешается выполнять операции записи с мьютексом `sync.RWMutex`, у нас может быть несколько функций, владеющих мьютексом `sync.RWMutex` и выполняющих чтение. Однако следует учитывать, что, пока все функции, читающие мьютекс типа `sync.RWMutex`, не разблокируют этот мьютекс, вы не сможете заблокировать его для записи — не особенно большая цена за возможность разрешить несколько операций чтения.

Работать с мьютексом `sync.RWMutex` вам помогут функции `RLock()` и `RUnlock()`, которые блокируют и разблокируют мьютекс для чтения соответственно. Функции `Lock()` и `Unlock()`, используемые в мьютексе `sync.Mutex`, по-прежнему при-

меняются для того, чтобы заблокировать и разблокировать мьютекс `sync.RWMutex` для записи. Таким образом, вызов функции `RLock()`, которая блокирует мьютекс для чтения, должен быть связан с вызовом функции `RUnlock()`. Наконец, как вы, очевидно, понимаете, не следует изменять какие-либо общие переменные внутри блока кода, расположенного между `RLock()` и `RUnlock()`.

Мы рассмотрим применение и полезные свойства типа `sync.RWMutex` на примере кода Go, который находится в файле `rwMutex.go`. Эта программа содержит две отличающиеся версии одной и той же функции. Рассмотрим ее, разделив на шесть частей. В первой из этих функций для чтения данных используется мьютекс типа `sync.RWMutex`, а второй — мьютекс типа `sync.Mutex`. Разница в производительности между этими двумя функциями поможет вам лучше понять преимущества использования мьютекса `sync.RWMutex` для чтения.

Первая часть `rwMutex.go` содержит следующий код Go:

```
package main

import (
    "fmt"
    "os"
    "sync"
    "time"
)

var Password = secret{password: "myPassword"}

type secret struct {
    RWM      sync.RWMutex
    M        sync.Mutex
    password string
}
```

Структура `secret` содержит общую переменную, мьютекс `sync.RWMutex` и мьютекс `sync.Mutex`.

Во второй части файла `rwMutex.go` содержится следующий код:

```
func Change(c *secret, pass string) {
    c.RWM.Lock()
    fmt.Println("LChange")
    time.Sleep(10 * time.Second)
    c.password = pass
    c.RWM.Unlock()
}
```

Функция `Change()` изменяет общую переменную. Следовательно, в ней нужно использовать монопольную блокировку, для чего применяется функции `Lock()` и `Unlock()`. Если мы изменяем переменную, то монопольной блокировки не избежать!

Третья часть rwMutex.go выглядит так:

```
func show(c *secret) string {
    c.RWM.RLock()
    fmt.Print("show")
    time.Sleep(3 * time.Second)
    defer c.RWM.RUnlock()
    return c.password
}
```

В функции show() используются RLock() и RUnlock(), поскольку ее критический раздел нужен для чтения общей переменной. Таким образом, несмотря на то, что несколько горутин могут читать общую переменную, эту переменную нельзя изменить без использования функций Lock() и Unlock(). Однако пока какая-либо из горутин читает общую переменную с помощью мьютекса, Lock() будет заблокирована.

В четвертом фрагменте файла rwMutex.go содержится следующий код Go:

```
func showWithLock(c *secret) string {
    c.RWM.Lock()
    fmt.Println("showWithLock")
    time.Sleep(3 * time.Second)
    defer c.RWM.Unlock()
    return c.password
}
```

Единственное различие между кодом функций showWithLock() и show() состоит в том, что в showWithLock() применяется монопольная блокировка для чтения данных. Следовательно, только одна функция showWithLock() может прочитать поле password структуры secret.

В пятой части rwMutex.go содержится следующий код Go:

```
func main() {
    var showFunction = func(c *secret) string { return "" }
    if len(os.Args) != 2 {
        fmt.Println("Using sync.RWMutex!")
        showFunction = show
    } else {
        fmt.Println("Using sync.Mutex!")
        showFunction = showWithLock
    }

    var waitGroup sync.WaitGroup

    fmt.Println("Pass:", showFunction(&Password))
```

Оставшийся код rwMutex.go выглядит так:

```
    for i := 0; i < 15; i++ {
        waitGroup.Add(1)
```

```
        go func() {
            defer waitGroup.Done()
            fmt.Println("Go Pass:", showFunction(&Password))
        }()
    }

    go func() {
        waitGroup.Add(1)
        defer waitGroup.Done()
        Change(&Password, "123456")
    }()

    waitGroup.Wait()
    fmt.Println("Pass:", showFunction(&Password))
}
```

Если дважды выполнить `rwMutex.go` и воспользоваться утилитой командной строки `time(1)` для сравнения двух версий программы, то получим результат такого вида:

```
$ time go run rwMutex.go 10 >/dev/null
real       0m51.206s
user        0m0.130s
sys         0m0.074s
$ time go run rwMutex.go >/dev/null
real       0m22.191s
user        0m0.135s
sys         0m0.071s
```

Обратите внимание, что `> /dev/null` в конце обеих команд нужен для того, чтобы пропустить вывод. Следовательно, версия, в которой применяется мьютекс `sync.RWMutex`, работает гораздо быстрее, чем версия с `sync.Mutex`.

Пакет atomic

Атомарная операция — это операция, которая выполняется за один шаг относительно других потоков или в данном случае других горутин. Таким образом, атомарная операция не может быть прервана на середине.

В стандартную библиотеку Go входит пакет `atomic`, который иногда позволяет обойтись без использования мьютекса. Однако мьютексы более универсальны, чем атомарные операции. С помощью пакета `atomic` можно создавать атомные счетчики, доступные для нескольких горутин, не рискуя получить проблемы синхронизации и состояние гонки.

Обратите внимание, что в случае атомарной переменной все операции чтения и записи должны выполняться с применением атомарных функций, предоставляемых пакетом `atomic`. Мы рассмотрим использование пакета `atomic` на примере программы `atom.go`. Разделим ее на три части.

Первая часть atom.go выглядит так:

```
package main

import (
    "flag"
    "fmt"
    "sync"
    "sync/atomic"
)

type atomCounter struct {
    val int64
}

func (c *atomCounter) Value() int64 {
    return atomic.LoadInt64(&c.val)
}
```

Вторая часть atom.go содержит следующий код:

```
func main() {
    minusX := flag.Int("x", 100, "Goroutines")
    minusY := flag.Int("y", 200, "Value")
    flag.Parse()
    X := *minusX
    Y := *minusY

    var waitGroup sync.WaitGroup
    counter := atomCounter{}
```

Последняя часть atom.go:

```
    for i := 0; i < X; i++ {
        waitGroup.Add(1)
        go func(no int) {
            defer waitGroup.Done()
            for i := 0; i < Y; i++ {
                atomic.AddInt64(&counter.val, 1)
            }
        }(i)
    }

    waitGroup.Wait()
    fmt.Println(counter.Value())
}
```

Требуемая переменная изменяется с помощью функции atomic.AddInt64().

Выполнение atom.go приведет к следующим результатам:

```
$ go run atom.go
20000
$ go run atom.go -x 4000 -y 10
40000
```

Результаты `atom.go` подтверждают, что счетчик, применяемый в программе, безопасен. Интересно попробовать изменить программу `atom.go` так, чтобы изменить переменную `counter`, используя вместо `atomic.AddInt64()` обычную арифметику (`counter.val++`). В этом случае результаты программы были бы похожи на следующие:

```
$ go run atom.go -x 4000 -y 10
37613
$ go run atom.go
15247
```

Как видно из результатов, из-за `counter.val++` поток программы становится небезопасным.

В главе 12 мы рассмотрим похожий пример, в котором задействован написанный на Go HTTP-сервер.

Совместное использование памяти с помощью горутин

В последнем подразделе этого раздела показано, как можно организовать совместный доступ к данным с помощью выделенной для этого горутины. Несмотря на то что общая память является традиционным способом взаимодействия потоков между собой, в стандартный комплект поставки Go входят встроенные функции синхронизации, которые позволяют одной горутине владеть общей частью данных. Это означает, что другие горутины должны отправлять сообщения этой горутине, которая владеет общими данными, что предотвращает повреждение данных. Такая горутина называется *управляющей горутиной*. Согласно терминологии Go, это **не коммуникация посредством общей памяти, а общая память посредством коммуникации**.

Рассмотрим эту методику более подробно на примере программы `monitor.go`. Разделим ее на пять частей. Программа `monitor.go` генерирует случайные числа посредством управляющей горутины.

Первая часть `monitor.go` выглядит так:

```
package main

import (
    "fmt"
    "math/rand"
    "os"
    "strconv"
    "sync"
    "time"
)

var readValue = make(chan int)
var writeValue = make(chan int)
```

Канал `readValue` используется для чтения случайных чисел, а канал `writeValue` — для получения новых случайных чисел.

Вторая часть `monitor.go` содержит следующий код:

```
func set(newValue int) {
    writeValue <- newValue
}

func read() int {
    return <-readValue
}
```

Назначение функции `set()` — установить значение общей переменной, тогда как `read()` предназначена для чтения значения сохраненной переменной.

Третий фрагмент кода программы `monitor.go` выглядит так:

```
func monitor() {
    var value int
    for {
        select {
        case newValue := <-writeValue:
            value = newValue
            fmt.Printf("%d ", value)
        case readValue <- value:
        }
    }
}
```

Вся логика программы заключена в реализации функции `monitor()`, а точнее, в операторе `select`, который управляет работой всей программы.

Когда поступает запрос на чтение, функция `read()` пытается выполнить операцию чтения из канала `readValue`, который управляется функцией `monitor()`. Результатом операции является текущее значение, которое хранится в переменной `value`. И наоборот, когда мы хотим изменить сохраненное значение, то вызываем функцию `set()`. Она записывает данные в канал `writeValue`, который также обрабатывается оператором `select`. В результате никто не может обратиться к общей переменной в обход функции `monitor()`.

Четвертый фрагмент кода `monitor.go` выглядит так:

```
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()
```

Последняя часть `monitor.go` содержит следующий код Go:

```
    var w sync.WaitGroup

    for r := 0; r < n; r++ {
        w.Add(1)
        go func() {
            defer w.Done()
            set(rand.Intn(10 * n))
        }()
    }
    w.Wait()
    fmt.Printf("\nLast value: %d\n", read())
}
```

Выполнение `monitor.go` приведет к такому результату:

```
$ go run monitor.go 20
Going to create 20 random numbers.
89 88 166 42 149 89 20 84 44 178 184 28 52 121 62 91 31 117 140 106
Last value: 106
$ go run monitor.go 10
Going to create 10 random numbers.
30 16 66 70 65 45 31 57 62 26
Last value: 26
```

Я предпочитаю использовать управляющую горутину вместо традиционных методов общего доступа к памяти, потому что реализация с использованием управляющей горутины безопаснее, ближе к философии Go и намного чище.

И снова об операторе go

Горутины выполняются быстро, и на одной машине могут выполняться тысячи горутин, однако это имеет свою цену. В данном разделе мы поговорим об операторе `go`, его поведении и о том, что происходит, когда мы запускаем очередную горутину в программе Go.

Обратите внимание, что замкнутые переменные (closured variables) в горутинах определяются в момент фактического выполнения горутины, когда выполняется оператор `go`, создающий новую горутину. Это означает, что замкнутые переменные заменяются их значениями в тот момент, когда планировщик Go решит

выполнить соответствующий код. Эта особенность продемонстрирована в следующем коде Go, который хранится в файле `cloGo.go`:

```
package main

import (
    "fmt"
    "time"
)

func main() {
    for i := 0; i <= 20; i++ {
        go func() {
            fmt.Print(i, " ")
        }()
    }
    time.Sleep(time.Second)
    fmt.Println()
}
```

Было бы неплохо в этом месте прервать чтение и попробовать угадать, каким будет результат этого кода.

Многократное выполнение `cloGo.go` выявляет проблему, о которой мы говорили:

```
$ go run cloGo.go
9 21 21 21 21 21 21 21 21 21 21 21 21 21 21 21 21 21 21 21 21
$ go run cloGo.go
4 21 21 21 21 21 21 21 21 6 21 21 21 21 21 21 21 21 21 21 21
$ go run cloGo.go
6 21 6 6 21 21 21 21 21 21 21 21 21 21 21 21 21 21 21 6 21
```

Программа по большей части выводит число `21`, которое является последним значением переменной цикла `for`, и изредка — другие числа. Дело в том, что `i` —*замкнутая переменная* и поэтому определяется во время выполнения программы. Горутины создаются, но не запускаются, ожидая, пока их запустит планировщик Go. Тем временем цикл `for` заканчивается, поэтому значение `i` равно `21`. Кстати, эта же проблема относится и к каналам Go — будьте осторожны.

Существует довольно забавный и неожиданный способ решить эту проблему, в котором используется идиоматическая особенность Go. Это решение показано в программе `cloGoCorrect.go`. Она содержит следующий код:

```
package main

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

```
func main() {
    for i := 0; i <= 20; i++ {
        i := i
        go func() {
            fmt.Print(i, " ")
            }()
        }
        time.Sleep(time.Second)
        fmt.Println()
    }
}
```

Корректный, но странный оператор `i: = i` создает новый экземпляр переменной для горутины, благодаря чему результат работы `cloGoCorrect.go` выглядит примерно так:

```
$ go run cloGoCorrect.go
1 5 4 3 6 0 13 7 8 9 10 11 12 17 14 15 16 19 18 20 2
$ go run cloGoCorrect.go
5 2 20 13 6 7 1 9 10 11 0 3 17 14 15 16 4 19 18 8 12
```

Рассмотрим еще один необычный случай использования оператора `go`. Посмотрите на следующий код Go, который хранится в файле `endlessComp.go`:

```
package main

import (
    "fmt"
    "runtime"
)

func main() {
    var i byte
    go func() {
        for i = 0; i <= 255; i++ {
        }
    }()
    fmt.Println("Leaving goroutine!")
    runtime.Gosched()
    runtime.GC()

    fmt.Println("Good bye!")
}
```

Результат выполнения `endlessComp.go` вас удивит: эта программа никогда не заканчивается, потому что блокируется на неопределенный срок. Ее приходится останавливать вручную:

```
$ go run endlessComp.go
Leaving goroutine!
^Csignal: interrupt
```

Как вы уже догадались, главная причина проблемы связана со сборщиком мусора Go и принципом его работы. При вызове функции `runtime.Gosched()` планировщику дается задание выполнить другую горутину, после чего мы вызываем сборщик мусора Go, который пытается выполнить свою работу.

Чтобы сборщик мусора смог выполнить свою работу и заснуть, он должен перебрать все горутины. Однако проблема в том, что цикл `for` не закончится никогда, потому что переменная `for` имеет тип `byte`. Это означает, что `for` не позволяет системе делать что-либо еще, так как горутина, в которой находится цикл `for`, не заснет никогда. Этого неприятного обстоятельства не избежать, даже если у вашей машины несколько ядер.

Обратите внимание: если бы `for` не был пустым, то программа была бы выполнена и успешно завершилась, потому что сборщик мусора мог бы остановиться[1].

И наконец, имейте в виду, что для того чтобы горутина завершилась, нужен четкий сигнал о ее завершении. Проще всего завершить горутину с помощью обычного оператора `return`.

Распознавание состояния гонки

Состояние гонки по данным — это ситуация, когда два и более работающих элемента, таких как потоки и горутины, пытаются получить контроль над общим ресурсом или переменной программы либо изменить их. Строго говоря, гонка по данным происходит тогда, когда две или более инструкции обращаются к одному и тому же адресу памяти и хотя бы одна из них выполняет операцию записи. Если все операции являются чтением, то состояния гонки нет.

Использование флага `-race` при запуске или сборке исходного файла Go включает *детектор гонки* Go, что заставляет компилятор создать модифицированную версию исполняемого файла. Эта измененная версия может записывать все операции доступа к общим переменным, а также все события синхронизации, включая обращения к `sync.Mutex` и `sync.WaitGroup`. Проанализировав соответствующие события, детектор гонки выводит отчет, который может помочь распознать потенциальные проблемы, чтобы их можно было вовремя исправить.

Рассмотрим следующий код Go, который хранится в файле `raceC.go`. Мы разделим эту программу на три части.

Первая часть `raceC.go` выглядит так:

```
package main

import (
    "fmt"
```

[1] На момент подготовки русскоязычного издания уже вышла версия Go 1.14, в которой решена проблема блокировки планировщика «пустыми» циклами.

```
    "os"
    "strconv"
    "sync"
)

func main() {
    arguments := os.Args
    if len(arguments) != 2 {
        fmt.Println("Give me a natural number!")
        return
    }
    numGR, err := strconv.Atoi(os.Args[1])
    if err != nil {
        fmt.Println(err)
        return
    }
```

Вторая часть `raceC.go`:

```
    var waitGroup sync.WaitGroup
    var i int
    k := make(map[int]int)
    k[1] = 12

    for i = 0; i < numGR; i++ {
        waitGroup.Add(1)
        go func() {
            defer waitGroup.Done()
            k[i] = i
        }()
    }
```

Остальной код Go в `raceC.go` выглядит следующим образом:

```
    k[2] = 10
    waitGroup.Wait()
    fmt.Printf("k = %#v\n", k)
}
```

Посчитав недостаточным то, что к отображению `k` одновременно обращаются несколько горутин, я добавил еще один оператор, который обращается к отображению `k` перед вызовом функции `sync.Wait()`.

Если запустить `raceC.go`, то получим результат следующего вида, без каких-либо предупреждений или сообщений об ошибках:

```
$ go run raceC.go 10
k = map[int]int{7:10, 2:10, 10:10, 1:12}
$ go run raceC.go 10
k = map[int]int{2:10, 10:10, 1:12, 8:8, 9:9}
$ go run raceC.go 10
k = map[int]int{10:10, 1:12, 6:7, 7:7, 2:10}
```

Если выполнить raceC.go только один раз, то все будет выглядеть нормально, несмотря на то что мы не получим того, что ожидали при выводе содержимого отображения k. Однако многократное выполнение raceC.go говорит о том, что здесь что-то не так, — главным образом потому, что при каждом выполнении программы мы получаем разные результаты.

Из программы raceC.go и ее неожиданных результатов можно извлечь еще много интересного — если мы решим использовать детектор гонки Go для ее анализа:

```
$ go run -race raceC.go 10
==================
WARNING: DATA RACE
Read at 0x00c00001a0a8 by goroutine 6:
  main.main.func1()
      /Users/mtsouk/Desktop/mGo2nd/raceC.go:32 +0x66
Previous write at 0x00c00001a0a8 by main goroutine:
  main.main()
      /Users/mtsouk/Desktop/mGo2nd/raceC.go:28 +0x23f
Goroutine 6 (running) created at:
  main.main()
      /Users/mtsouk/Desktop/mGo2nd/raceC.go:30 +0x215
==================
==================
WARNING: DATA RACE
Write at 0x00c0000bc000 by goroutine 7:
  runtime.mapassign_fast64()
      /usr/local/Cellar/go/1.12.3/libexec/src/runtime/map_fast64.go:92
  main.main.func1()
      /Users/mtsouk/Desktop/mGo2nd/raceC.go:32 +0x8d
Previous write at 0x00c0000bc000 by goroutine 6:
  runtime.mapassign_fast64()
      /usr/local/Cellar/go/1.12.3/libexec/src/runtime/map_fast64.go:92
  main.main.func1()
      /Users/mtsouk/Desktop/mGo2nd/raceC.go:32 +0x8d
Goroutine 7 (running) created at:
  main.main()
      /Users/mtsouk/Desktop/mGo2nd/raceC.go:30 +0x215
Goroutine 6 (finished) created at:
  main.main()
      /Users/mtsouk/Desktop/mGo2nd/raceC.go:30 +0x215
==================
==================
WARNING: DATA RACE
Write at 0x00c0000bc000 by goroutine 8:
  runtime.mapassign_fast64()
      /usr/local/Cellar/go/1.12.3/libexec/src/runtime/map_fast64.go:92
  main.main.func1()
      /Users/mtsouk/Desktop/mGo2nd/raceC.go:32 +0x8d
Previous write at 0x00c0000bc000 by goroutine 6:
  runtime.mapassign_fast64()
```

```
      /usr/local/Cellar/go/1.12.3/libexec/src/runtime/map_fast64.go:92
  main.main.func1()
      /Users/mtsouk/Desktop/mGo2nd/raceC.go:32 +0x8d
Goroutine 8 (running) created at:
  main.main()
      /Users/mtsouk/Desktop/mGo2nd/raceC.go:30 +0x215
Goroutine 6 (finished) created at:
  main.main()
      /Users/mtsouk/Desktop/mGo2nd/raceC.go:30 +0x215
==================
k = map[int]int{1:1, 2:10, 3:3, 4:4, 5:5, 6:6, 7:7, 8:8, 9:9, 10:10}
Found 3 data race(s)
exit status 66
```

Итак, детектор гонки обнаружил три состояния гонки по данным. Каждое из них в его результатах начинается с сообщения `WARNING: DATA RACE`.

Первая ситуация гонки по данным происходит в функции `main.main.func1()` в строке 32, которая вызывается из строки 28 цикла `for`. Последний, в свою очередь, вызывается горутиной, созданной в строке 30. Эта проблема обозначена в сообщении `Previous write`. Изучив соответствующий код, легко понять, что реальная проблема заключается в том, что анонимная функция не принимает параметров. Следовательно, значение `i`, которое используется в цикле `for`, не может быть детерминистически определено, поскольку оно изменяется в цикле `for`, который является операцией записи.

Сообщение о втором состоянии гонки по данным гласит: `Write at 0x00c0000bc000 by goroutine 7`. Если мы прочитаем соответствующие результаты программы, то увидим, что здесь гонка по данным связана с операцией записи, и это происходит в отображении Go в строке 32 по крайней мере с двумя горутинами, которые запускаются в строке 30. Поскольку эти две горутины имеют одинаковое имя (`main.main.func1()`), это значит, что мы говорим об одной и той же горутине. Когда две горутины пытаются изменить одну и ту же переменную, возникает состояние гонки по данным. Третье состояние гонки по данным похоже на второе.

Запись main.main.func1() используется в Go для внутреннего присвоения имени анонимной функции. Если у нас разные анонимные функции, то их имена также будут разными.

Вы спросите: «Что я могу сделать уже сейчас, чтобы исправить проблемы, вызвавшие два состояния гонки по данным?»

Вы можете переписать функцию `main()` в `raceC.go` следующим образом:

```
func main() {
    arguments := os.Args
    if len(arguments) != 2 {
        fmt.Println("Give me a natural number!")
```

```
        return
    }
    numGR, err := strconv.Atoi(os.Args[1])
    if err != nil {
        fmt.Println(err)
        return
    }

    var waitGroup sync.WaitGroup
    var i int

    k := make(map[int]int)
    k[1] = 12

    for i = 0; i < numGR; i++ {
        waitGroup.Add(1)
        go func(j int) {
            defer waitGroup.Done()
            aMutex.Lock()
            k[j] = j
            aMutex.Unlock()
        }(i)
    }

    waitGroup.Wait()
    k[2] = 10
    fmt.Printf("k = %#v\n", k)
}
```

Переменная `aMutex` является глобальной переменной типа `sync.Mutex`, она определена вне функции `main()` и доступна из любой точки программы. Хотя это и не обязательно, наличие такой глобальной переменной избавляет от необходимости постоянно передавать ее в функции.

Сохранив новую версию `raceC.go` под именем `noRaceC.go` и выполнив ее, получим ожидаемый результат:

```
$ go run noRaceC.go 10
k = map[int]int{1:1, 0:0, 5:5, 3:3, 6:6, 9:9, 2:10, 4:4, 7:7, 8:8}
```

Обработка `noRaceC.go` с помощью детектора гонки Go приведет к следующему:

```
$ go run -race noRaceC.go 10
k = map[int]int{5:5, 7:7, 9:9, 1:1, 0:0, 4:4, 6:6, 8:8, 2:10, 3:3}
```

Обратите внимание, что при доступе к хеш-таблице `k` нужно использовать механизм блокировки. Если этого не делать и просто изменить реализацию анонимной функции, которая выполняется как горутина, то при выполнении `go run noRaceC.go` получим следующий результат:

```
$ go run noRaceC.go 10
fatal error: concurrent map writes
goroutine 10 [running]:
```

```
runtime.throw(0x10ca0bd, 0x15)
    /usr/local/Cellar/go/1.9.3/libexec/src/runtime/panic.go:605 +0x95
fp=0xc420024738 sp=0xc420024718 pc=0x10276b5
runtime.mapassign_fast64(0x10ae680, 0xc420074180, 0x5, 0x0)
    /usr/local/Cellar/go/1.9.3/libexec/src/runtime/hashmap_fast.go:607
+0x3d2 fp=0xc420024798 sp=0xc420024738 pc=0x100b582
main.main.func1(0xc420010090, 0xc420074180, 0x5)
    /Users/mtsouk/ch10/code/noRaceC.go:35 +0x6b fp=0xc4200247c8
sp=0xc420024798 pc=0x1096f5b
runtime.goexit()
    /usr/local/Cellar/go/1.9.3/libexec/src/runtime/asm_amd64.s:2337 +0x1
fp=0xc4200247d0 sp=0xc4200247c8 pc=0x1050c21
created by main.main
    /Users/mtsouk/ch10/code/noRaceC.go:32 +0x15a
goroutine 1 [semacquire]:
sync.runtime_Semacquire(0xc42001009c)
    /usr/local/Cellar/go/1.9.3/libexec/src/runtime/sema.go:56 +0x39
sync.(*WaitGroup).Wait(0xc420010090)
    /usr/local/Cellar/go/1.9.3/libexec/src/sync/waitgroup.go:131 +0x72
main.main()
    /Users/mtsouk/ch10/code/noRaceC.go:40 +0x17a
goroutine 12 [runnable]:
sync.(*WaitGroup).Done(0xc420010090)
    /usr/local/Cellar/go/1.9.3/libexec/src/sync/waitgroup.go:99 +0x43
main.main.func1(0xc420010090, 0xc420074180, 0x7)
    /Users/mtsouk/ch10/code/noRaceC.go:37 +0x79
created by main.main
    /Users/mtsouk/ch10/code/noRaceC.go:32 +0x15a
exit status 2
```

Корень проблемы ясен: одновременная запись в хеш-таблицу.

Пакет context

Главное назначение пакета `context` — определение типа `Context` и поддержка *аннулирования*. Да, вы все правильно поняли: бывают случаи, когда по какой-либо причине необходимо отказаться от того, что вы делаете. Однако очень хорошо, когда есть возможность добавить дополнительную информацию о ваших решениях об аннулировании. Именно это позволяет сделать пакет `context`.

Если вы посмотрите в исходный код пакета `context`, то поймете, что его реализация довольно проста — даже реализация типа `Context` проста, тем не менее, пакет `context` очень важен.

В течение некоторого времени пакет context был внешним пакетом Go; впервые он стал стандартным в версии Go 1.7. Таким образом, если у вас более старая версия Go, то вы не сможете выполнить код из этого раздела, не загрузив предварительно пакет context или не установив более новую версию Go.

Тип `Context` — это интерфейс, имеющий четыре метода: `Deadline()`, `Done()`, `Err()` и `Value()`. К счастью, мы не обязаны реализовывать все функции интерфейса `Context` — достаточно лишь изменить переменную `Context`, используя такие функции, как `context.WithCancel()`, `context.WithDeadline()` и `context.WithTimeout()`. Эти три функции возвращают производный (дочерний) объект `Context` и функцию `CancelFunc`. Вызов функции `CancelFunc` удаляет ссылку родителя на дочерний объект и останавливает все связанные с ним таймеры. Следовательно, после этого сборщик мусора Go может свободно собрать дочерние горутины, у которых больше нет связанных с ними родительских горутин. Чтобы сборка мусора работала правильно, родительская горутина должна хранить ссылку на все дочерние горутины. Если дочерняя горутина завершится без ведома родительской, начинается утечка памяти, которая продолжается до тех пор, пока родительская горутина также не будет аннулирована.

Рассмотрим простой пример использования пакета `context` с применением кода Go из файла `simpleContext.go`. Разделим его на шесть частей.

Первый фрагмент файла `simpleContext.go` содержит следующий код:

```
package main

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

Вторая часть `simpleContext.go` выглядит так:

```
func f1(t int) {
    c1 := context.Background()
    c1, cancel := context.WithCancel(c1)
    defer cancel()

    go func() {
        time.Sleep(4 * time.Second)
        cancel()
    }()
```

Функция `f1()` принимает всего один аргумент, который является временной задержкой, — все остальное определено внутри функции. Обратите внимание, что тип переменной `cancel` — `context.CancelFunc`.

Для инициализации пустого объекта `Context` нужно вызвать `context.Background()`. Функция `context.WithCancel()` использует существующий объект `Context` и создает его потомка с возможностью аннулирования. Функция `context.WithCancel()` также создает канал `Done`, который можно закрыть либо при вызове функции

cancel(), как показано в предыдущем коде, либо когда закрывается канал Done родительского контекста.

Третья часть simpleContext.go содержит остальной код функции f1():

```
    select {
    case <-c1.Done():
        fmt.Println("f1():", c1.Err())
        return
    case r := <-time.After(time.Duration(t) * time.Second):
        fmt.Println("f1():", r)
    }
    return
}
```

Здесь мы видим использование функции Done() для переменной типа Context. При вызове этой функции происходит аннулирование. Возвращаемое значение Context.Done() является каналом, потому что в противном случае мы не смогли бы применить его в операторе select.

Четвертая часть simpleContext.go содержит следующий код Go:

```
func f2(t int) {
    c2 := context.Background()
    c2, cancel := context.WithTimeout(c2, time.Duration(t)*time.Second)
    defer cancel()

    go func() {
        time.Sleep(4 * time.Second)
        cancel()
    }()

    select {
    case <-c2.Done():
        fmt.Println("f2():", c2.Err())
        return
    case r := <-time.After(time.Duration(t) * time.Second):
        fmt.Println("f2():", r)
    }
    return
}
```

В этом фрагменте показано применение функции context.WithTimeout(), которая принимает два аргумента: один типа Context, а второй — типа time.Duration. По окончании времени ожидания автоматически вызывается функция cancel().

Пятая часть simpleContext.go выглядит так:

```
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()
```

```
    go func() {
        time.Sleep(4 * time.Second)
        cancel()
    }()

    select {
    case <-c3.Done():
        fmt.Println("f3():", c3.Err())
        return
    case r := <-time.After(time.Duration(t) * time.Second):
        fmt.Println("f3():", r)
    }
    return
}
```

В этом коде Go показано использование функции `context.WithDeadline()`, которая принимает два аргумента: `Context` и время в будущем, соответствующее крайнему сроку окончания операции. По истечении этого времени автоматически вызывается функция `cancel()`.

Оставшийся код Go файла `simpleContext.go` выглядит так:

```
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)
}
```

Назначение функции `main()` — все инициализировать.

Выполнение `simpleContext.go` приведет к результатам следующего вида:

```
$ go run simpleContext.go 4
Delay: 4
f1(): context canceled
f2(): 2019-04-11 18:18:43.327345 +0300 EEST m=+8.004588898
f3(): 2019-04-11 18:18:47.328073 +0300 EEST m=+12.005483099
$ go run simpleContext.go 2
Delay: 2
f1(): 2019-04-11 18:18:53.972045 +0300 EEST m=+2.005231943
f2(): context deadline exceeded
```

```
f3(): 2019-04-11 18:18:57.974337 +0300 EEST m=+6.007690061
$ go run simpleContext.go 10
Delay: 10
f1(): context canceled
f2(): context canceled
f3(): context canceled
```

Длинные строки этих выходных данных являются возвращаемыми значениями, получаемыми при вызовах функции `time.After()`. Они соответствуют нормальной работе программы. Дело в том, что работа программы аннулируется при задержках в ее выполнении.

Использование пакета `context` действительно так просто, как показано здесь, несмотря на то что представленный код не выполняет сколь-нибудь серьезной работы с интерфейсом `Context`. Однако код Go, который мы рассмотрим в следующем подразделе, представляет более реалистичный пример.

Расширенный пример использования пакета context

Гораздо лучше и глубже функциональность пакета `context` видна на примере кода Go из программы `useContext.go`. Разделим ее на пять частей. В этом примере мы создадим HTTP-клиент, который не желает слишком долго ждать ответа от HTTP-сервера, — вполне обычный сценарий. Фактически, поскольку почти все HTTP-клиенты поддерживают такую функциональность, мы рассмотрим другой способ отмены HTTP-запроса в главе 12.

Программа `useContext.go` принимает два аргумента командной строки: URL-адрес сервера, к которому она будет подключаться, и время, в течение которого представленная утилита должна ожидать отклика сервера. Если передать программе только один аргумент командной строки, то время ожидания составит пять секунд.

Первый фрагмент кода `useContext.go` выглядит так:

```
package main

import (
    "context"
    "fmt"
    "io/ioutil"
    "net/http"
    "os"
    "strconv"
    "sync"
    "time"
)

var (
    myUrl    string
```

```
    delay    int = 5
    w        sync.WaitGroup
)

type myData struct {
    r    *http.Response
    err    error
}
```

И myURL, и delay являются глобальными переменными, поэтому к ним можно обращаться из любой точки кода. Кроме того, здесь еще есть переменная типа sync.WaitGroup с именем w, также с глобальной областью действия, и определена структура с именем myData для хранения ответа веб-сервера и переменной error на случай, если где-то возникнет ошибка. Вторая часть useContext.go содержит следующий код Go:

```
func connect(c context.Context) error {
    defer w.Done()
    data := make(chan myData, 1)

    tr := &http.Transport{}
    httpClient := &http.Client{Transport: tr}

    req, _ := http.NewRequest("GET", myUrl, nil)
```

Этот код Go отвечает за HTTP-соединение.

Подробнее о разработке HTTP-серверов и клиентов вы прочитаете в главе 12.

В третьей части useContext.go содержится такой код Go:

```
    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
        }
    }()
```

Четвертый фрагмент useContext.go содержит следующий код Go:

```
    select {
    case <-c.Done():
        tr.CancelRequest(req)
        <-data
```

```
        fmt.Println("The request was cancelled!")
        return c.Err()
    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 := ioutil.ReadAll(resp.Body)
        if err != nil {
            fmt.Println("Error select:", err)
            return err
        }
        fmt.Printf("Server Response: %s\n", realHTTPData)
    }
    return nil
}
```

Остальной код `useContext.go` является реализацией функции `main()` и выглядит так:

```
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
    }

    fmt.Println("Delay:", delay)
    c := context.Background()
    c, cancel := context.WithTimeout(c, time.Duration(delay)*time.Second)
    defer cancel()

    fmt.Printf("Connecting to %s \n", myUrl)
    w.Add(1)
    go connect(c)
    w.Wait()
    fmt.Println("Exiting...")
}
```

Период ожидания определяется методом `context.WithTimeout()`. Функция `connect()`, которая выполняется как горутина, завершится либо нормально, либо посредством выполнения `cancel()`. Обратите внимание, что принято использовать `context.Background()` в функции `main()`, или `init()` пакета, или в тестах.

Несмотря на то что вам не обязательно знать о серверной части операции, следует видеть, как Go-версия веб-сервера может замедляться случайным образом. Так, генератор случайных чисел решает, насколько медленным будет веб-сервер, — реальные веб-серверы могут оказаться слишком загруженными и поэтому не ответить. Или же могут возникнуть проблемы с сетью, которые вызовут задержку. Рассмотрим исходный файл `slowWWW.go`. Его содержимое выглядит так:

```
package main

import (
    "fmt"
    "math/rand"
    "net/http"
    "os"
    "time"
)

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

func myHandler(w http.ResponseWriter, r *http.Request) {
    delay := random(0, 15)
    time.Sleep(time.Duration(delay) * time.Second)

    fmt.Fprintf(w, "Serving: %s\n", r.URL.Path)
    fmt.Fprintf(w, "Delay: %d\n", delay)
    fmt.Printf("Served: %s\n", r.Host)
}

func main() {
    seed := time.Now().Unix()
    rand.Seed(seed)

    PORT := ":8001"
    arguments := os.Args
    if len(arguments) == 1 {
        fmt.Println("Using default port number: ", PORT)
    } else {
        PORT = ":" + arguments[1]
    }

    http.HandleFunc("/", myHandler)
    err := http.ListenAndServe(PORT, nil)
    if err != nil {
```

```
        fmt.Println(err)
        return
    }
}
```

Как видите, в файле slowWWW.go можно не применять пакет context, потому что именно веб-клиент решает, сколько времени он готов ждать ответа.

Код функции myHandler() определяет, насколько медленно будет работать программа веб-сервера. Задержка может составлять от 0 до 14 секунд, что определяется вызовом функции random(0, 15).

Если использовать веб-сервер slowWWW.go с помощью такого инструмента, как wget(1), то получим результат следующего вида:

```
$ wget -qO- http://localhost:8001/
Serving: /
Delay: 4
$ wget -qO- http://localhost:8001/
Serving: /
Delay: 13
```

Так происходит потому, что значение времени ожидания по умолчанию для wget(1) больше, чем у slowWWW.go. Если выполнить useContext.go, когда slowWWW.go уже запущен в другой оболочке UNIX, после обработки с помощью удобной утилиты time(1) получим следующий результат:

```
$ time go run useContext.go http://localhost:8001/ 1
Delay: 1
Connecting to http://localhost:8001/
Get http://localhost:8001/: net/http: request canceled
The request was cancelled!
Exiting...
real    0m1.374s
user    0m0.304s
sys     0m0.117s
$ time go run useContext.go http://localhost:8001/ 10
Delay: 10
Connecting to http://localhost:8001/
Get http://localhost:8001/: net/http: request canceled
The request was cancelled!
Exiting...
real    0m10.381s
user    0m0.314s
sys     0m0.125s
$ time go run useContext.go http://localhost:8001/ 15
Delay: 15
Connecting to http://localhost:8001/
Server Response: Serving: /
Delay: 13
Exiting...
```

```
real    0m13.379s
user    0m0.309s
sys     0m0.118s
```

Как видно из этих результатов, только третья команда действительно получила ответ от HTTP-сервера — для двух первых команд время ожидания истекло раньше.

Еще один пример использования пакета context

В этом подразделе вы еще ближе познакомитесь с пакетом `context` и узнаете, насколько это мощный и уникальный пакет стандартной библиотеки Go. На этот раз мы создадим контекст, в котором вместо функции `context.Background()` будет использоваться `context.TODO()`. Обе эти функции возвращают ненулевой пустой контекст, однако назначение у них разное. Мы также продемонстрируем применение функции `context.WithValue()`. Программа, которую мы рассмотрим, называется `moreContext.go`. Разделим ее на четыре части.

Первая часть `moreContext.go` выглядит так:

```
package main

import (
    "context"
    "fmt"
)

type aKey string
```

Вторая часть `moreContext.go` содержит следующий код Go:

```
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)
    }
}
```

Эта функция извлекает значение из контекста и проверяет, существует ли такое значение.

Третья часть `moreContext.go` содержит следующий код Go:

```
func main() {
    myKey := aKey("mySecretValue")
    ctx := context.WithValue(context.Background(), myKey, "mySecretValue")

    searchKey(ctx, myKey)
```

Функция `context.WithValue()` предоставляет способ связывания значения с объектом `Context`.

Обратите внимание, что контексты не должны храниться в структурах — их следует передавать в функции в виде отдельных аргументов. Рекомендуется передавать контекст в качестве первого аргумента функции.

Последняя часть `moreContext.go` выглядит так:

```
    searchKey(ctx, aKey("notThere"))
    emptyCtx := context.TODO()
    searchKey(emptyCtx, aKey("notThere"))
}
```

В данном случае мы заявляем, что, несмотря на то что мы намерены использовать контекст операции, мы все еще не уверены в этом и потому используем функцию `context.TODO()`. К счастью, `TODO()` распознается средствами статического анализа, что позволяет им определить, правильно ли распространяются контексты.

Выполнение `moreContext.go` приведет к следующим результатам:

```
$ go run moreContext.go
found value: mySecretValue
key not found: notThere
key not found: notThere
```

Запомните: никогда не передавайте нулевой контекст — используйте функцию `context.TODO()` для создания подходящего контекста. И учтите, что `context.TODO()` следует использовать, если вы не уверены в том, какой объект `Context` хотите применить.

Пулы обработчиков

Пул обработчиков — это множество потоков, предназначенных для обработки назначаемых им заданий. Веб-сервер Apache и Go-пакет `net/http` работают приблизительно так: основной процесс принимает все входящие запросы, которые затем перенаправляются рабочим процессам для обработки. Как только рабочий процесс завершает свою работу, он готов к обслуживанию нового клиента.

Однако здесь есть главное различие: пул обработчиков использует не потоки, а горутины. Кроме того, потоки обычно не умирают после обработки запросов, потому что затраты на завершение потока и создание нового слишком высоки, тогда как горутина прекращает существовать после завершения работы. Как вы вскоре увидите, пулы обработчиков в Go реализованы с помощью буферизованных каналов, поскольку они позволяют ограничить число одновременно выполняемых горутин.

Рассмотрим программу `workerPool.go`. Разделим ее на пять частей. Эта программа решает простую задачу: она обрабатывает целые числа и выводит их квадраты

посредством отдельной горутины для обслуживания каждого запроса. Несмотря на простоту workerPool.go, код Go программы вполне можно использовать как шаблон для реализации более сложных задач.

Это расширенный метод, он поможет вам разрабатывать серверные процессы на Go, способные принимать и обслуживать несколько клиентов с использованием горутин.

Первая часть workerPool.go выглядит так:

```
package main

import (
    "fmt"
    "os"
    "strconv"
    "sync"
    "time"
)

type Client struct {
    id        int
    integer    int
}

type Data struct {
    job        Client
    square    int
}
```

Здесь мы видим метод с применением структуры Client для назначения уникального идентификатора каждому запросу, который мы намерены обработать. Структура Data нужна для группировки данных Client с реальными результатами, сгенерированными программой. Проще говоря, структура Client содержит входные данные каждого запроса, а структура Data — результаты запроса.

Вторая часть workerPool.go содержит следующий код Go:

```
var (
    size    = 10
    clients = make(chan Client, size)
    data = make(chan Data, size)
)

func worker(w *sync.WaitGroup) {
    for c := range clients {
        square := c.integer * c.integer
        output := Data{c, square}
```

```
        data <- output
        time.Sleep(time.Second)
    }
    w.Done()
}
```

Этот код состоит из двух интересных частей. Первая часть создает три глобальные переменные. Буферизованные каналы `clients` и `data` используются для получения новых клиентских запросов и записи результатов соответственно. Если вы хотите, чтобы программа работала быстрее, можете увеличить значение параметра `size`.

Вторая часть — это реализация функции `worker()`, которая читает канал `clients` и получает оттуда новые запросы на обслуживание. После завершения обработки результат записывается в канал `data`. Задержка, которая вводится с помощью оператора `time.Sleep(time.Second)`, не является необходимой, но дает лучшее представление о том, как будут выводиться на экран результаты программы.

Наконец, не забудьте использовать в функции `worker()` указатель для параметра `sync.WaitGroup`, иначе переменная `sync.WaitGroup` скопируется и будет бесполезной.

Третья часть `workerPool.go` содержит следующий код Go:

```
func makeWP(n int) {
    var w sync.WaitGroup
    for i := 0; i < n; i++ {
        w.Add(1)
        go worker(&w)
    }
    w.Wait()
    close(data)
}

func create(n int) {
    for i := 0; i < n; i++ {
        c := Client{i, i}
        clients <- c
    }
    close(clients)
}
```

В этом коде реализованы две функции — `makeWP()` и `create()`. Назначение функции `makeWP()` состоит в том, чтобы сгенерировать необходимое количество горутин `worker()` для обработки всех запросов. Функция `w.Add(1)` вызывается из `makeWP()`, однако `w.Done()` вызывается из `worker()` после того, как обработчик завершит свою работу. Назначение функции `create()` — правильно создать все запросы, используя тип `Client`, а затем записать их в канал `clients` для обработки. Обратите внимание, что канал `clients` читается функцией `worker()`.

Четвертый фрагмент кода `workerPool.go` выглядит так:

```
func main() {
    fmt.Println("Capacity of clients:", cap(clients))
    fmt.Println("Capacity of data:", cap(data))

    if len(os.Args) != 3 {
        fmt.Println("Need #jobs and #workers!")
        os.Exit(1)
    }

    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 считываем аргументы командной строки. Однако сначала здесь показано, как можно использовать функцию `cap()`, чтобы определить пропускную способность канала.

Если количество обработчиков больше, чем размер буферизованного канала `clients`, то количество создаваемых горутин будет равно размеру канала `clients`. Аналогично если количество заданий больше, чем количество обработчиков, то задания будут обрабатываться меньшими группами.

Программа позволяет определить количество обработчиков и количество заданий с помощью аргументов командной строки. Однако, чтобы изменить размер каналов `clients` и `data`, нужно внести изменения в исходный код программы.

Остальной код `workerPool.go` выглядит так:

```
        go create(nJobs)
        finished := make(chan interface{})
        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
    }()
    makeWP(nWorkers)
    fmt.Printf(": %v\n", <-finished)
}
```

Сначала мы вызываем функцию `create()`, чтобы имитировать клиентские запросы, которые следует обработать. Для чтения канала `data` и вывода результатов на экран применяется анонимная горутина. Канал `finished` используется для блокировки программы до тех пор, пока анонимная программа не закончит чтение канала `data`. Поэтому канал `finished` не нуждается в конкретном типе.

Затем мы вызываем функцию `makeWP()`, которая и выполняет обработку запросов. Оператор `<-finished` в блоках `fmt.Printf()` не позволяет программе завершиться, пока кто-нибудь не напишет что-либо в канал `finished`. Этот «кто-то» на самом деле — анонимная горутина в `main()`. Кроме того, хотя анонимная горутина и записывает значение `true` в канал `finished`, мы могли бы записать туда значение `false` и получить тот же результат, который разблокирует функцию `main()`. Не верите — попробуйте сами!

Выполнение `workerPool.go` приведет к результатам следующего вида:

```
$ go run workerPool.go 15 5
Capacity of clients: 10
Capacity of data: 10
Client ID: 0    int: 0        square: 0
Client ID: 4    int: 4        square: 16
Client ID: 1    int: 1        square: 1
Client ID: 3    int: 3        square: 9
Client ID: 2    int: 2        square: 4
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
Client ID: 10    int: 10        square: 100
Client ID: 11    int: 11        square: 121
Client ID: 12    int: 12        square: 144
Client ID: 13    int: 13        square: 169
Client ID: 14    int: 14        square: 196
```

Если вы хотите обслуживать каждый запрос, не ожидая ответа от него в функции `main()`, как это сделано в `workerPool.go`, то у вас будет меньше поводов для беспокойства. Простой способ применять горутины для обработки запросов и получения ответа от них в функции `main()` — использовать общую память или управляющий процесс, который будет собирать данные, а не просто выводить их на экран.

Наконец, работа программы `workerPool.go` сильно упрощена, поскольку функция `worker()` не может завершиться с ошибкой. Это придется учитывать при работе с реальными компьютерными сетями или с другими видами ресурсов, которые могут дать сбой.

Дополнительные ресурсы

Следующие ресурсы будут вам очень полезны:

- посетите страницу документации пакета `sync`, которую вы найдете по адресу https://golang.org/pkg/sync/;
- посетите страницу документации пакета `context` по адресу https://golang.org/pkg/context/;
- чтобы больше узнать о реализации планировщика Go, посетите страницу https://golang.org/src/runtime/proc.go;
- страницу документации пакета `atomic` вы найдете по адресу https://golang.org/pkg/sync/atomic/;
- просмотрите проектную документацию планировщика Go по адресу https://golang.org/s/go11sched.

Упражнения

- Попробуйте реализовать конкурентную версию `wc(1)`, которая бы использовала буферизованный канал.
- Затем попробуйте реализовать конкурентную версию `wc(1)`, которая бы использовала общую память.
- Наконец, попробуйте реализовать конкурентную версию `wc(1)`, в которой бы использовалась управляющая горутина.
- Измените код Go файла `workerPool.go` таким образом, чтобы сохранять результаты в файле. При работе с файлом примените мьютекс и критический раздел или управляющую горутину, которая будет записывать данные на диск.
- Что произойдет с программой `workerPool.go`, если значение глобальной переменной `size` станет равным 1? Почему?
- Измените код Go программы `workerPool.go` таким образом, чтобы реализовать функциональность утилиты командной строки `wc(1)`.
- Измените код Go программы `workerPool.go` таким образом, чтобы размер буферизованных каналов `clients` и `data` можно было задавать в виде аргументов командной строки.
- Попробуйте написать конкурентную версию утилиты командной строки `find(1)`, в которой бы применялась управляющая горутина.
- Измените код `simpleContext.go` таким образом, чтобы анонимная функция, используемая в функциях `f1()`, `f2()` и `f3()`, стала отдельной функцией. Какая основная проблема возникает при таком изменении кода?

- Измените код Go программы `simpleContext.go` таким образом, чтобы функции `f1()`, `f2()` и `f3()` использовали созданную извне переменную `Context` вместо определения собственной такой переменной.
- Измените код Go программы `useContext.go` таким образом, чтобы вместо функции `context.WithTimeout()` использовать `context.WithDeadline()` или `context.WithCancel()`.
- Наконец, попробуйте реализовать конкурентную версию утилиты командной строки `find(1)` с помощью мьютекса `sync.Mutex`.

Резюме

В этой главе рассмотрено много важных тем, связанных с горутинами и каналами. Однако в основном это касалось функциональности оператора `select`. Благодаря возможностям оператора `select` каналы являются предпочтительным способом Go для организации взаимодействия между компонентами конкурентной программы Go, в которой задействовано несколько горутин. Кроме того, в этой главе продемонстрировано использование стандартного Go-пакета `context`, который иногда бывает незаменим.

В конкурентном программировании есть много правил; однако самое важное из них заключается в том, что следует избегать общих данных, если только у вас нет на то веских причин! Общие данные являются причиной всех ошибок при конкурентном программировании.

Главное, что следует запомнить, прочитав эту главу, — раньше общая память была единственным способом обмена данными между потоками одного процесса, сейчас же Go предлагает более удобные способы коммуникации. Поэтому подумайте с точки зрения Go, прежде чем принять решение об использовании общей памяти в вашем коде Go. Если же вам действительно нужно это использовать, вы можете задействовать управляющую горутину.

Основные темы следующей главы — тестирование, оптимизация и профилирование кода средствами Go. Кроме этого, вы познакомитесь с тестированием кода Go, кросс-компиляцией и поиском недоступного кода в Go.

В конце вы узнаете, как документировать код Go и генерировать документацию в формате HTML с помощью утилиты `godoc`.

11 Тестирование, оптимизация и профилирование кода

В предыдущей главе мы обсудили конкурентность в Go, мьютексы, пакет `atomic`, различные типы каналов, состояние гонки и то, как оператор `select` позволяет использовать каналы в качестве связующего звена для управления горутинами и предоставления им возможности обмена информацией.

Темы, которые рассмотрены в этой главе, очень практичны и важны, особенно если вы хотите повысить производительность ваших программ Go и быстро обнаруживать ошибки. Эта глава прежде всего посвящена оптимизации, тестированию, документированию и профилированию кода.

Оптимизация кода — это процесс, при котором один или несколько разработчиков стремятся ускорить выполнение определенных частей программы, повысить их эффективность или использовать меньше ресурсов. Проще говоря, оптимизация кода заключается в устранении недостатков программы.

Тестирование кода — это проверка того, что код выполняет то, что от него требуется. В этой главе вы познакомитесь с методом тестирования кода в Go. Лучше всего писать код для тестирования программ на этапе разработки, поскольку это помогает выявить ошибки в коде на самых ранних стадиях.

Профилирование кода означает измерение определенных аспектов программы, что позволяет составить подробно представление о том, как работает код. Результаты профилирования кода помогают принять решение о том, какие части кода нуждаются в изменении.

Я надеюсь, что вы уже осознали важность документирования кода для описания тех решений, которые вы приняли в процессе разработки реализации программы. В этой главе вы увидите, как Go помогает сгенерировать документацию для реализуемых модулей.

Документация настолько важна, что некоторые разработчики сначала пишут документацию и только потом код! Однако действительно важно, чтобы документация программы говорила о том, что делает функциональность программы.

В этой главе рассмотрены следующие темы:

- профилирование кода Go;
- утилита `go tool pprof`;
- использование веб-интерфейса профилирования Go;
- тестирование кода Go;
- команда `go test`;
- утилита `go tool trace`;
- удобный пакет `testing/quick`;
- бенчмаркинг кода Go;
- кросс-компиляция;
- тестирование покрытия кода;
- автоматизированное создание документации для кода Go; создание примеров функций;
- поиск недоступного кода Go в программах.

Оптимизация

Оптимизация кода — это и искусство, и наука. Не существует четко определенного способа, который бы помог вам оптимизировать код Go или любой другой код на любом языке программирования. Вам придется хорошо подумать и многое сделать, если вы хотите, чтобы ваш код заработал быстрее.

Убедитесь, что оптимизируете код, который не содержит ошибок: нет смысла оптимизировать ошибки. Если в вашей программе есть ошибки, ее нужно сначала отладить.

Если вы действительно интересуетесь оптимизацией кода, то советую прочитать книгу «Компиляторы. Принципы, технологии и инструментарий»[1], в которой основное внимание уделяется строению компилятора. Кроме того, все тома серии «Искусство программирования»[2] Дональда Кнута являются отличными источниками знаний по всем аспектам программирования.

---

[1] *Ахо А. В., Лам М. С., Сети Р., Ульман Д. Д.* Компиляторы. Принципы, технологии и инструментарий, 2-е изд. — М.: Вильямс, 2017.

[2] *Кнут. Д.* Искусство программирования. В 4 т. — М.: Вильямс.

Всегда помните, что сказал Кнут об оптимизации:

> *«Настоящая проблема заключается в том, что программисты тратят слишком много времени на повышение эффективности не в тех местах и не в тот момент; преждевременная оптимизация – корень всех бед в программировании (или по крайней мере большей их части)».*

Также помните, что сказал об оптимизации Джо Армстронг, один из разработчиков Erlang:

> *«Сначала сделай так, чтобы это работало, потом сделай так, чтобы оно было красивым, и только потом, если это действительно нужно, сделай так, чтобы оно работало быстро. В 90 % случаев, если ты сделаешь это красивым, оно само по себе будет работать быстро. Поэтому просто сделай это по-настоящему красивым!»*

К тому же лишь небольшая часть программы должна быть оптимизирована. В таких случаях для реализации определенных функций Go можно использовать язык ассемблера. Это по-настоящему хороший вариант, он окажет огромное влияние на производительность ваших программ.

Оптимизация кода Go

Как уже отмечалось, оптимизация кода — это процесс, в ходе которого мы пытаемся обнаружить части кода, влияющие на производительность всей программы, и стараемся сделать так, чтобы они работали быстрее или использовали меньше ресурсов.

Один из следующих разделов этой главы, посвященный *бенчмаркингу*, поможет вам понять, что именно происходит внутри программы при выполнении кода и какие параметры программы оказывают наиболее сильное влияние на ее производительность. Однако не стоит недооценивать важность здравого смысла. Проще говоря, если одна из функций выполняется в 10 000 раз чаще, чем остальные функции программы, попробуйте для начала оптимизировать ее.

Общий совет по оптимизации: оптимизируйте только тот код, который не содержит ошибок. Другими словами, следует оптимизировать только рабочий код. Поэтому сначала напишите правильный код, пускай он даже работает медленно. В конце концов, самая распространенная ошибка, которую допускают программисты, — это попытка оптимизировать первую же версию своего кода. А это влечет за собой остальные ошибки!

Напомню: оптимизация кода — это одновременно и искусство, и наука, а следовательно, довольно сложная задача. Следующий раздел, посвященный профилированию кода Go, определенно поможет вам в оптимизации кода, потому что основная цель профилирования — найти недостатки в коде, чтобы оптимизировать самые важные части программы.

Профилирование кода Go

Профилирование — это динамический анализ программы, в ходе которого измеряются различные величины, связанные с выполнением программы, что позволяет лучше понять поведение программы. В этом разделе вы узнаете, как профилировать код Go, чтобы лучше понять, как он работает, и повысить его производительность. Иногда профилирование кода даже помогает найти ошибки!

Для начала мы воспользуемся интерфейсом командной строки профилировщика Go. Затем мы будем использовать веб-интерфейс профилировщика Go.

Запомните самое важное: если вы хотите профилировать код Go, для этого вам нужно импортировать стандартный Go пакет `runtime/pprof`, прямо или косвенно. Чтобы открыть страницу справки инструмента `pprof`, нужно выполнить команду `go tool pprof -help`. В ответ вы получите много интересных сведений.

Стандартный Go-пакет net/http/pprof

В комплект поставки Go входит стандартный Go-пакет низкого уровня `runtime/pprof`, однако существует также пакет высокого уровня `net/http/pprof`. Именно его следует использовать, если вы хотите профилировать веб-приложение, написанное на Go. В этой главе не будет обсуждаться создание HTTP-серверов на Go, подробнее о пакете `net/http/pprof` вы узнаете в главе 12.

Простой пример профилирования

Go поддерживает два вида профилирования: *профилирование процессора* и *профилирование памяти*. Выполнять для одного приложения оба вида профилирования одновременно не рекомендуется, поскольку они плохо совместимы. Однако рассмотренное далее приложение `profileMe.go` является исключением из правила, поскольку используется для иллюстрации обоих методов.

Мы сохраним профилируемый код Go в файле `profileMe.go` и рассмотрим его, разделив на пять частей.

Первая часть `profileMe.go` содержит следующий код Go:

```
package main

import (
```

```
    "fmt"
    "math"
    "os"
    "runtime"
    "runtime/pprof"
    "time"
)

func fibo1(n int) int64 {
    if n == 0 || n == 1 {
        return int64(n)
    }
    time.Sleep(time.Millisecond)
    return int64(fibo2(n-1)) + int64(fibo2(n-2))
}
```

Обратите внимание, что для создания данных профилирования необходимо импортировать в вашу программу пакет `runtime/pprof`, явно или неявно. Функция `time.Sleep()` вызывается из функции `fibo1()` для того, чтобы немного ее замедлить. Зачем это нужно, вы узнаете в конце этого подраздела.

Второй фрагмент кода `profileMe.go` выглядит так:

```
func fibo2(n int) int {
    fn := make(map[int]int)
    for i := 0; i <= n; i++ {
        var f int
        if i <= 2 {
            f = 1
        } else {
            f = fn[i-1] + fn[i-2]
        }
        fn[i] = f
    }
    time.Sleep(50 * time.Millisecond)
    return fn[n]
}
```

Этот код представляет собой реализацию еще одной функции Go, в которой использован другой алгоритм определения чисел из последовательности Фибоначчи.

В третьей части `profileMe.go` содержится следующий код Go:

```
func N1(n int) bool {
    k := math.Floor(float64(n/2 + 1))
    for i := 2; i < int(k); i++ {
        if (n % i) == 0 {
            return false
        }
```

```
    }
    return true
}

func N2(n int) bool {
    for i := 2; i < n; i++ {
        if (n % i) == 0 {
            return false
        }
    }
    return true
}
```

Функции `N1()` и `N2()` используются для того, чтобы определить, является ли данное целое число простым. Первая функция оптимизирована, поскольку в ней цикл `for` выполняет примерно вдвое меньше итераций, чем в цикле `for` функции `N2()`. Поскольку обе функции относительно медленные, здесь нет необходимости обращаться к `time.Sleep()`.

Четвертый фрагмент кода `profileMe.go` выглядит так:

```
func main() {
    cpuFile, err := os.Create("/tmp/cpuProfile.out")
    if err != nil {
        fmt.Println(err)
        return
    }
    pprof.StartCPUProfile(cpuFile)
    defer pprof.StopCPUProfile()
    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()
```

Вызов функции `os.Create()` нужен для получения файла, в который записываются данные профилирования. Вызов `pprof.StartCPUProfile()` начинает профилирование процессора для программы, а вызов `pprof.StopCPUProfile()` останавливает его.

Если вы хотите многократно создавать и использовать временные файлы и каталоги, обратите внимание на функции `ioutil.TempFile()` и `ioutil.TempDir()` соответственно.

Последняя часть `profileMe.go` выглядит так:

```
    // Профилирование памяти!
    memory, err := os.Create("/tmp/memoryProfile.out")
    if err != nil {
        fmt.Println(err)
        return
    }
    defer memory.Close()
    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
    }
}
```

В этой последней части программы показано, как работает метод *профилирования памяти*. Он очень похож на профилирование процессора, и для записи данных профилирования также нужен файл.

Выполнение `profileMe.go` приведет к следующим результатам:

```
$ go run profileMe.go
Total primes: 9592
Total primes: 9592
```

```
1 2 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181 6765 10946
17711 28657 46368 75025 121393 196418 317811 514229 832040 1346269 2178309
3524578 5702887 9227465 14930352 24157817 39088169 63245986 102334155
165580141 267914296 433494437 701408733 1134903170 1836311903 2971215073
4807526976 7778742049 12586269025 20365011074 32951280099 53316291173
86267571272 139583862445 225851433717 365435296162 591286729879
956722026041 1548008755920 2504730781961 4052739537881 6557470319842
10610209857723 17167680177565 27777890035288 44945570212853 72723460248141
117669030460994 190392490709135 308061521170129 498454011879264
806515533049393 1304969544928657 2111485077978050 3416454622906707
5527939700884757 8944394323791464 14472334024676221 23416728348467685
37889062373143906 61305790721611591 99194853094755497 160500643816367088
259695496911122585 420196140727489673 679891637638612258
1100087778366101931 1779979416004714189
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181 6765 10946
17711 28657 46368 75025 121393 196418 317811 514229 832040 1346269 2178309
3524578 5702887 9227465 14930352 24157817 39088169 63245986 102334155
165580141 267914296 433494437 701408733 1134903170 1836311903 2971215073
4807526976 7778742049 12586269025 20365011074 32951280099 53316291173
86267571272 139583862445 225851433717 365435296162 591286729879
956722026041 1548008755920 2504730781961 4052739537881 6557470319842
10610209857723 17167680177565 27777890035288 44945570212853 72723460248141
117669030460994 190392490709135 308061521170129 498454011879264
806515533049393 1304969544928657 2111485077978050 3416454622906707
5527939700884757 8944394323791464 14472334024676221 23416728348467685
37889062373143906 61305790721611591 99194853094755497 160500643816367088
259695496911122585 420196140727489673 679891637638612258
1100087778366101931 1779979416004714189
```

Помимо этого, программа будет записывать данные профилирования в два файла:

```
$ cd /tmp
$ ls -l *Profile*
-rw-r--r-- 1 mtsouk wheel 1557 Apr 24 16:37 cpuProfile.out
-rw-r--r-- 1 mtsouk wheel 438 Apr 24 16:37 memoryProfile.out
```

Только после того, как данные профилирования будут собраны, можно приступить к их анализу. Итак, теперь можно запустить профилировщик командной строки, чтобы проверить данные процессора:

```
$ go tool pprof /tmp/cpuProfile.out
Type: cpu
Time: Apr 24, 2019 at 4:37pm (EEST)
Duration: 19.59s, Total samples = 4.46s (22.77%)
Entering interactive mode (type "help" for commands, "o" for options)
(pprof)
```

Если нажать help в оболочке профилировщика, то получим следующий результат:

```
(pprof) help
  Commands:
    callgrind       Outputs a graph in callgrind format
    comments        Output all profile comments
    disasm          Output assembly listings annotated with samples
    dot             Outputs a graph in DOT format
    eog             Visualize graph through eog
    evince          Visualize graph through evince
    gif             Outputs a graph image in GIF format
    gv              Visualize graph through gv
    kcachegrind     Visualize report in KCachegrind
    list            Output annotated source for functions matching regexp
    pdf             Outputs a graph in PDF format
    peek            Output callers/callees of functions matching regexp
    png             Outputs a graph image in PNG format
    proto           Outputs the profile in compressed protobuf format
    ps              Outputs a graph in PS format
    raw             Outputs a text representation of the raw profile
    svg             Outputs a graph in SVG format
    tags            Outputs all tags in the profile
    text            Outputs top entries in text form
    top             Outputs top entries in text form
    topproto        Outputs top entries in compressed protobuf format
    traces          Outputs all profile samples in text form
    tree            Outputs a text rendering of call graph
    web             Visualize graph through web browser
    weblist         Display annotated source in a web browser
    o/options       List options and their current values
    quit/exit/^D    Exit pprof
  Options:
    call_tree       Create a context-sensitive call tree
    compact_labels  Show minimal headers
    divide_by       Ratio to divide all samples before visualization
    drop_negative   Ignore negative differences
    edgefraction    Hide edges below <f>*total
    focus           Restricts to samples going through a node matching regexp
    hide            Skips nodes matching regexp
    ignore          Skips paths going through any nodes matching regexp
    mean            Average sample value over first value (count)
    nodecount       Max number of nodes to show
    nodefraction    Hide nodes below <f>*total
    noinlines       Ignore inlines.
    normalize       Scales profile based on the base profile.
    output          filename for file-based outputs
    prune_from      Drops any functions below the matched frame.
    relative_percentages Show percentages relative to focused subgraph
    sample_index    Sample value to report (0-based index or name)
    show            Only show nodes matching regexp
```

```
    show_from       Drops functions above the highest matched frame.
    source_path     Search path for source files
    tagfocus        Restricts to samples with tags in range or matched by
    regexp
    taghide         Skip tags matching this regexp
    tagignore       Discard samples with tags in range or matched by regexp
    tagshow         Only consider tags matching this regexp
    trim            Honor nodefraction/edgefraction/nodecount defaults
    trim_path       Path to trim from source paths before search
    unit            Measurement units to display
  Option groups (only set one per group):
    cumulative
      cum             Sort entries based on cumulative weight
      flat            Sort entries based on own weight
    granularity
      addresses       Aggregate at the address level.
      filefunctions   Aggregate at the function level.
      files           Aggregate at the file level.
      functions       Aggregate at the function level.
      lines           Aggregate at the source code line level.
 :    Clear focus/ignore/hide/tagfocus/tagignore
 type "help <cmd|option>" for more information
(pprof)
```

Найдите время для ознакомления со всеми командами утилиты go tool pprof и проверьте их.

Команда `top` возвращает десять лучших записей в текстовой форме:

```
(pprof) top
Showing nodes accounting for 4.42s, 99.10% of 4.46s total
Dropped 14 nodes (cum <= 0.02s)
Showing top 10 nodes out of 19
      flat  flat%   sum%        cum   cum%
     2.69s 60.31% 60.31%      2.69s 60.31%  main.N2
     1.41s 31.61% 91.93%      1.41s 31.61%  main.N1
     0.19s  4.26% 96.19%      0.19s  4.26%  runtime.nanotime
     0.10s  2.24% 98.43%      0.10s  2.24%  runtime.usleep
     0.03s  0.67% 99.10%      0.03s  0.67%  runtime.memclrNoHeapPointers
         0     0% 99.10%      4.14s 92.83%  main.main
         0     0% 99.10%      0.03s  0.67%  runtime.(*mheap).alloc
         0     0% 99.10%      0.03s  0.67%  runtime.largeAlloc
         0     0% 99.10%      4.14s 92.83%  runtime.main
         0     0% 99.10%      0.03s  0.67%  runtime.makeslice
```

Функции, представленные в первых строках выходных состояний, отвечают за 99,10 % общего времени выполнения программы.

В частности, функция `main.N2` отвечает за `60,31%` всего времени выполнения программы.

Команда `top10 --cum` возвращает совокупное время выполнения для каждой функции:

```
(pprof) top10 --cum
Showing nodes accounting for 4390ms, 98.43% of 4460ms total
Dropped 14 nodes (cum <= 22.30ms)
Showing top 10 nodes out of 19
      flat  flat%   sum%        cum   cum%
         0     0%     0%     4140ms 92.83%   main.main
         0     0%     0%     4140ms 92.83%   runtime.main
    2690ms 60.31% 60.31%     2690ms 60.31%   main.N2
    1410ms 31.61% 91.93%     1410ms 31.61%   main.N1
         0     0% 91.93%      290ms  6.50%   runtime.mstart
         0     0% 91.93%      270ms  6.05%   runtime.mstart1
         0     0% 91.93%      270ms  6.05%   runtime.sysmon
     190ms  4.26% 96.19%      190ms  4.26%   runtime.nanotime
     100ms  2.24% 98.43%      100ms  2.24%   runtime.usleep
         0     0% 98.43%       50ms  1.12%   runtime.systemstack
```

А как узнать, что происходит с каждой отдельной функцией? Для этого можно использовать команду `list`, в которой указать имя функции и имя пакета, и тогда мы получим более развернутую информацию о производительности этой функции:

```
(pprof) list main.N1
Total: 4.18s
ROUTINE ======= main.N1 in /Users/mtsouk/ch11/code/profileMe.go
    1.41s      1.41s  (flat, cum) 31.61% of Total
        .          .      32:    return fn[n]
        .          .      33: }
        .          .      34:
        .          .      35:func N1(n int) bool {
        .          .      36:    k := math.Floor(float64(n/2 + 1))
     60ms       60ms      37:    for i := 2; i < int(k); i++ {
    1.35s      1.35s      38:        if (n % i) == 0 {
        .          .      39:            return false
        .          .      40:        }
        .          .      41:    }
        .          .      42:    return true
        .          .      43:}
   (pprof)
```

Как видно из этого результата, цикл `for` функции `main.N1` отвечает почти за все время выполнения этой функции. В частности, оператор `if (n % i) == 0` занимает `1.35s` из `1.41s` времени выполнения всей функции.

Из оболочки профилировщика Go также можно создать PDF-файл данных профилирования, воспользовавшись командой `pdf`:

```
(pprof) pdf
Generating report in profile001.pdf
```

Обратите внимание, что для создания PDF-файла, который потом можно просмотреть с помощью любого PDF-ридера, вам понадобится программа *Graphviz*.

И последнее предупреждение: если ваша программа выполняется слишком быстро, то у профилировщика не будет достаточно времени, чтобы сделать необходимые измерения, и при загрузке файла данных вы можете увидеть сообщение `Total samples = 0`. В этом случае вам не удастся получить сколько-нибудь полезную информацию в результате профилирования. Именно поэтому в некоторых функциях `profileMe.go` мы использовали `time.Sleep()`:

```
$ go tool pprof /tmp/cpuProfile.out
Type: cpu
Time: Apr 24, 2019 at 4:37pm (EEST)
Duration: 19.59s, Total samples = 4.46s (22.77%)
Entering interactive mode (type "help" for commands, "o" for options)
(pprof)
```

Удобный внешний пакет для профилирования

В этом подразделе продемонстрировано использование внешнего пакета Go, который настраивает среду профилирования. Он гораздо удобнее, чем стандартный Go-пакет `runtime/pprof`. Мы рассмотрим внешний пакет на примере программы `betterProfile.go`. Разделим ее на три части.

Первая часть `betterProfile.go` выглядит так:

```
package main

import (
    "fmt"
    "github.com/pkg/profile"
)

var VARIABLE int

func N1(n int) bool {
    for i := 2; i < n; i++ {
        if (n % i) == 0 {
            return false
        }
    }
    return true
}
```

В этом коде показано использование внешнего Go-пакета, который размещен по адресу github.com/pkg/profile. Вы можете его скачать с помощью команды `go get`:

```
$ go get github.com/pkg/profile
```

Второй фрагмент файла `bestProfile.go` содержит следующий код Go:

```
func Multiply(a, b int) int {
    if a == 1 {
        return b
    }
    if a == 0 || b == 0 {
        return 0
    }
    if a < 0 {
        return -Multiply(-a, b)
    }
    return b + Multiply(a-1, b)
}

func main() {
    defer profile.Start(profile.ProfilePath("/tmp")).Stop()
```

Для *профилирования процессора* при помощи пакета `github.com/pkg/profile`, разработанного Дэйвом Чейни (Dave Cheney), достаточно добавить в приложение Go всего один оператор. Для того чтобы выполнить *профилирование памяти*, вместо этого оператора нужно вставить следующий:

```
defer profile.Start(profile.MemProfile).Stop()
```

Оставшийся код Go программы выглядит так:

```
    total := 0
    for i := 2; i < 200000; i++ {
        n := N1(i)
        if n {
            total++
        }
    }

    fmt.Println("Total: ", total)
    total = 0
    for i := 0; i < 5000; i++ {
        for j := 0; j < 400; j++ {
            k := Multiply(i, j)
            VARIABLE = k
            total++
        }
    }
    fmt.Println("Total: ", total)
}
```

Выполнение `betterProfile.go` приведет к следующим результатам:

```
$ go run betterProfile.go
2019/04/24 16:44:05 profile: cpu profiling enabled, /tmp/cpu.pprof
Total: 17984
Total: 2000000
2019/04/24 16:44:33 profile: cpu profiling disabled, /tmp/cpu.pprof
```

Пакет github.com/pkg/profile нужен для сбора данных; обработка данных выполняется так же, как и раньше.

```
$ go tool pprof /tmp/cpu.pprof
Type: cpu
Time: Apr 24, 2019 at 4:44pm (EEST)
Duration: 27.40s, Total samples = 25.10s (91.59%)
Entering interactive mode (type "help" for commands, "o" for options)
(pprof)
```

Веб-интерфейс Go-профилировщика

К счастью, в Go версии 1.10 у команды `go tool pprof` появился пользовательский веб-интерфейс.

Чтобы пользовательский веб-интерфейс смог работать, необходимо установить Graphviz, а ваш веб-браузер должен поддерживать JavaScript. В целях безопасности используйте Chrome или Firefox.

Интерактивный профилировщик Go запускается с помощью такой команды:

```
$ go tool pprof -http=[host]:[port] aProfile
```

Пример профилирования с использованием веб-интерфейса

Для изучения веб-интерфейса профилировщика Go мы будем использовать данные, полученные при выполнении `profileMe.go`, поскольку для этого не нужно создавать отдельный пример кода.

Как вы уже знаете, для этого сначала нужно выполнить следующую команду:

```
$ go tool pprof -http=localhost:8080 /tmp/cpuProfile.out
Main binary filename not available.
```

На рис. 11.1 представлен начальный экран пользовательского веб-интерфейса профилировщика Go после выполнения предыдущей команды.

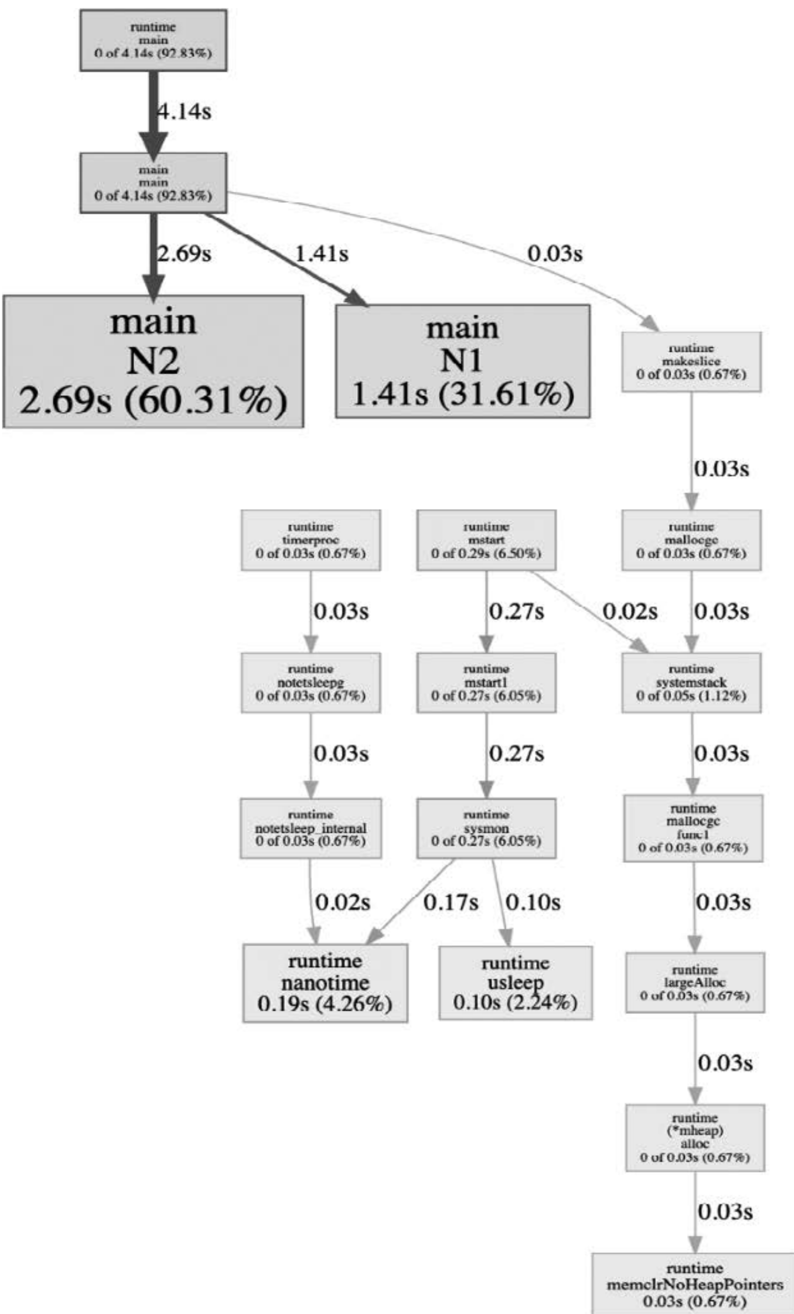


Рис. 11.1. Веб-интерфейс профилировщика Go

Аналогично на рис. 11.2 показан URL-адрес профилировщика Go http://localhost:8080/ui/source, по которому выводится аналитическая информация для каждой функции программы.

```
unknown cpu
localhost:8080/ui/source
pprof   VIEW   SAMPLE   REFINE   Search regexp                      unknown cpu

main.N2
/Users/mtsouk/Desktop/mGo2nd/Mastering-Go-Second-Edition/ch11/profileMe.go

 Total:       2.69s      2.69s (flat, cum) 60.31%
     41           .          .                  }
     42           .          .                  return true
     43           .          .          }
     44           .          .
     45           .          .          func N2(n int) bool {
     46        90ms       90ms                  for i := 2; i < n; i++ {
     47       2.60s      2.60s                          if (n % i) == 0 {
     48           .          .                                  return false
     49           .          .                          }
     50           .          .                  }
     51           .          .                  return true
     52           .          .          }

main.N1
/Users/mtsouk/Desktop/mGo2nd/Mastering-Go-Second-Edition/ch11/profileMe.go

 Total:       1.41s      1.41s (flat, cum) 31.61%
     32           .          .                  return fn[n]
     33           .          .          }
     34           .          .
     35           .          .          func N1(n int) bool {
     36           .          .                  k := math.Floor(float64(n/2 + 1))
     37        60ms       60ms                  for i := 2; i < int(k); i++ {
     38       1.35s      1.35s                          if (n % i) == 0 {
     39           .          .                                  return false
     40           .          .                          }
     41           .          .                  }
     42           .          .                  return true
     43           .          .          }

runtime.nanotime
/usr/local/Cellar/go/1.12.4/libexec/src/runtime/sys_darwin.go

 Total:       190ms      190ms (flat, cum)  4.26%
    222           .          .          func open_trampoline()
    223           .          .
    224           .          .          //go:nosplit
    225           .          .          //go:cgo_unsafe_args
    226           .          .          func nanotime() int64 {
    227       190ms      190ms                  var r struct {
    228           .          .                          t            int64  // raw timer
    229           .          .                          numer, denom uint32 // conversion factors. nanoseconds = t * numer / denom.
    230           .          .                  }
    231           .          .                  libcCall(unsafe.Pointer(funcPC(nanotime_trampoline)), unsafe.Pointer(&r))
    232           .          .                  // Note: Apple seems unconcerned about overflow here. See

runtime.usleep
/usr/local/Cellar/go/1.12.4/libexec/src/runtime/sys_darwin.go

 Total:       100ms      100ms (flat, cum)  2.24%
    202           .          .
    203           .          .          //go:nosplit
    204           .          .          //go:cgo_unsafe_args
    205           .          .          func usleep(usec uint32) {
```

Рис. 11.2. Использование URL-адреса /source профилировщика Go

Поскольку я не могу показать каждую страницу веб-интерфейса профилировщика Go, вам следует познакомиться с ним самостоятельно — это отличный инструмент для проверки работы ваших программ.

Коротко об основах Graphviz

Graphviz — это очень удобное сочетание утилит и компьютерного языка, который позволяет рисовать сложные графики. Строго говоря, Graphviz — это набор инструментов для манипулирования направленными и ненаправленными *графовыми* структурами и создания графовых структур.

У Graphviz есть собственный язык *DOT* — простой, изящный и мощный. Преимущество Graphviz состоит в том, что вы можете писать его код в простом текстовом редакторе. Замечательным вторичным эффектом этого свойства является возможность легко разрабатывать сценарии, которые генерируют код Graphviz. Кроме того, большинство языков программирования, включая Python, Ruby, C++ и Perl, имеют собственные интерфейсы для создания файлов Graphviz с использованием собственного кода.

Вы не обязаны знать все о Graphviz, чтобы использовать веб-интерфейс профилировщика Go. Но полезно понимать, как работает Graphviz и как выглядит его код.

Следующий код Graphviz, который сохраняется как `graph.dot`, кратко иллюстрирует работу Graphviz и внешний вид языка Graphviz:

```
digraph G
{
    graph [dpi = 300, bgcolor = "gray"];
    rankdir = LR;
    node [shape=record, width=.2, height=.2, color="white" ];
    node0 [label = "<p0>; |<p1>|<p2>|<p3>|<p4>| | ", height = 3];
    node[ width=2 ];
    node1 [label = "{<e> r0 | 123 | <p> }", color="gray" ];
    node2 [label = "{<e> r10 | 13 | <p> }" ];
    node3 [label = "{<e> r11 | 23 | <p> }" ];
    node4 [label = "{<e> r12 | 326 | <p> }" ];
    node5 [label = "{<e> r13 | 1f3 | <p> }" ];
    node6 [label = "{<e> r20 | 143 | <p> }" ];
    node7 [label = "{<e> r40 | b23 | <p> }" ];
    node0:p0 -> node1:e [dir=both color="red:blue"];
    node0:p1 -> node2:e [dir=back arrowhead=diamond];
    node2:p -> node3:e;
    node3:p -> node4:e [dir=both arrowtail=box color="red"];
    node4:p -> node5:e [dir=forward];
    node0:p2 -> node6:e [dir=none color="orange"];
    node0:p4 -> node7:e;
}
```

Атрибут `color` меняет цвет узла, а атрибут `shape` — форму узла. Кроме того, атрибут `dir`, который можно применить к ребрам, определяет, будет ли у ребра две

стрелки, одна или ни одной. Стиль стрелки определяется атрибутами `arrowhead` и `arrowtail`.

Скомпилировав этот код с использованием одного из инструментов командной строки Graphviz, получим изображение в формате PNG. Для этого нужно выполнить следующую команду в любой оболочке UNIX:

```
$ dot -T png graph.dot -o graph.png
$ ls -l graph.png
-rw-r--r--@ 1 mtsouk staff 94862 Apr 24 16:48 graph.png
```

На рис. 11.3 показан графический файл, сгенерированный в результате выполнения этой команды:

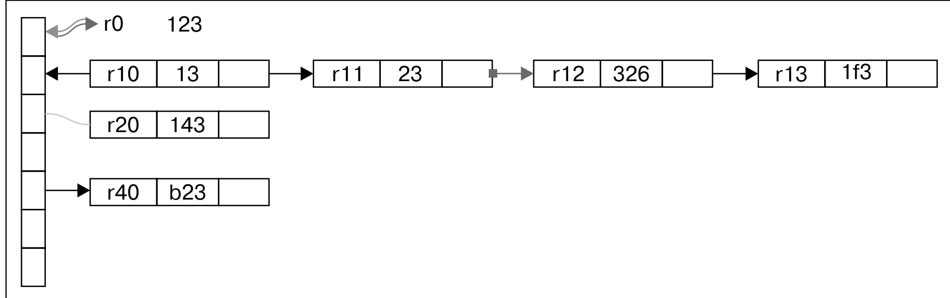


Рис. 11.3. Использование Graphviz для создания графов

Таким образом, если мы хотим визуализировать какую-либо структуру, определенно следует подумать об использовании Graphviz и его инструментов, особенно если нам нужно автоматизировать эти действия с помощью собственных сценариев.

Утилита go tool trace

Утилита `go tool trace` — это средство просмотра файлов трассировки, которые можно сгенерировать одним из трех способов:

- с помощью пакета `runtime/trace`;
- с помощью пакета `net/http/pprof`;
- с помощью команды `go test -trace`.

В этом разделе мы будем использовать только первый способ. Выходные данные следующей команды помогут вам понять, что делает *трассировщик выполнения программы* Go:

```
$ go doc runtime/trace
package trace // import "runtime/trace"
```

```
Package trace contains facilities for programs to generate traces for the Go
execution tracer.
Tracing runtime activities
The execution trace captures a wide range of execution events such as
goroutine creation/blocking/unblocking, syscall enter/exit/block, GC-related
events, changes of heap size, processor start/stop, etc. A precise
nanosecond-precision timestamp and a stack trace is captured for most
events. The generated trace can be interpreted using `go tool trace`.
The trace tool computes the latency of a task by measuring the time
between the task creation and the task end and provides latency
distributions for each task type found in the trace.
func IsEnabled() bool
func Log(ctx context.Context, category, message string)
func Logf(ctx context.Context, category, format string, args
...interface{})
func Start(w io.Writer) error
func Stop()
func WithRegion(ctx context.Context, regionType string, fn func())
type Region struct{ ... }
func StartRegion(ctx context.Context, regionType string) *Region
type Task struct{ ... }
func NewTask(pctx context.Context, taskType string) (ctx context.Context, task *Task)
```

В главе 2 мы говорили о сборщике мусора Go и познакомились с Go-утилитой `gColl.go`, которая позволила увидеть некоторые переменные сборщика мусора Go. В этом разделе мы соберем еще больше информации о работе `gColl.go` с помощью утилиты `go tool trace`.

Сначала мы рассмотрим модифицированную версию программы `gColl.go`, которая дает команду Go собирать данные о производительности. Эта утилита хранится в файле `goGC.go`. Рассмотрим ее, разделив на три части.

Первая часть `goGC.go` выглядит так:

```
package main

import (
    "fmt"
    "os"
    "runtime"
    "runtime/trace"
    "time"
)

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)
    fmt.Println("-----")
}
```

Как мы уже знаем, чтобы начать сбор данных для утилиты `go tool trace`, сначала нужно импортировать стандартный Go-пакет `runtime/trace`.

Второй фрагмент кода `goGC.go` содержит следующий код Go:

```
func main() {
    f, err := os.Create("/tmp/traceFile.out")
    if err != nil {
        panic(err)
    }
    defer f.Close()
    err = trace.Start(f)
    if err != nil {
        fmt.Println(err)
        return
    }
    defer trace.Stop()
```

Эта часть программы получает данные для утилиты `go tool trace` и не имеет ничего общего с функциональностью реальной программы. Сначала мы создаем новый файл, который будет содержать данные трассировки для утилиты `go tool trace`. Затем с помощью функции `trace.Start()` мы запускаем процесс трассировки. После того как она закончит работу, мы вызываем функцию `trace.Stop()`. Вызов этой функции с использованием ключевого слова `defer` означает, что мы хотим прекратить трассировку, когда программа завершит работу.

Использование утилиты go tool trace — двухэтапный процесс, который требует дополнительного кода Go. Сначала мы собираем данные, а затем отображаем и обрабатываем их.

Оставшийся код Go выглядит так:

```
    var mem runtime.MemStats
    printStats(mem)
    for i := 0; i < 3; i++ {
        s := make([]byte, 50000000)
        if s == nil {
            fmt.Println("Operation failed!")
        }
    }
    printStats(mem)
    for i := 0; i < 5; i++ {
        s := make([]byte, 100000000)
        if s == nil {
            fmt.Println("Operation failed!")
        }
        time.Sleep(time.Millisecond)
    }
    printStats(mem)
}
```

Выполнение программы goGC.go приведет к следующим результатам, а также к созданию нового файла с именем /tmp/traceFile.out, в котором содержится информация о трассировке:

```
$ go run goGC.go
mem.Alloc: 108592
mem.TotalAlloc: 108592
mem.HeapAlloc: 108592
mem.NumGC: 0
-----
mem.Alloc: 109736
mem.TotalAlloc: 150127000
mem.HeapAlloc: 109736
mem.NumGC: 3
-----
mem.Alloc: 114672
mem.TotalAlloc: 650172952
mem.HeapAlloc: 114672
mem.NumGC: 8
-----
$ cd /tmp
$ ls -l traceFile.out
-rw-r--r-- 1 mtsouk wheel 10108 Apr 24 16:51 /tmp/traceFile.out
$ file /tmp/traceFile.out
/tmp/traceFile.out: data
```

Утилита go tool trace использует веб-интерфейс, который автоматически запускается при выполнении следующей команды:

```
$ go tool trace /tmp/traceFile.out
2019/04/24 16:52:06 Parsing trace...
2019/04/24 16:52:06 Splitting trace...
2019/04/24 16:52:06 Opening browser. Trace viewer is listening on
http://127.0.0.1:50383
```

На рис. 11.4 показан начальный вид веб-интерфейса утилиты go tool trace при просмотре файла трассировки /tmp/traceFile.out.

Теперь нужно выбрать ссылку View trace. Она приведет нас к виду веб-интерфейса утилиты go tool trace, использующей данные из /tmp/traceFile.out, который показан на рис. 11.5.

Как видно на рис. 11.5, сборщик мусора Go работает в собственной горутине, но он работает не постоянно. Кроме того, здесь мы видим количество горутин, задействованных программой. Чтобы узнать об этом больше, можно выбрать отдельные части интерактивного представления. Поскольку нас интересует сборщик мусора, то полезна информация о том, как часто и как долго работает сборщик мусора.

Обратите внимание, что, хотя go tool trace — очень удобная и мощная утилита, она не решает все проблемы производительности. Бывают случаи, когда уместнее использовать go tool pprof, особенно если вы хотите узнать, на что программа тратит большую часть своего времени, исследуя ее отдельные функции.

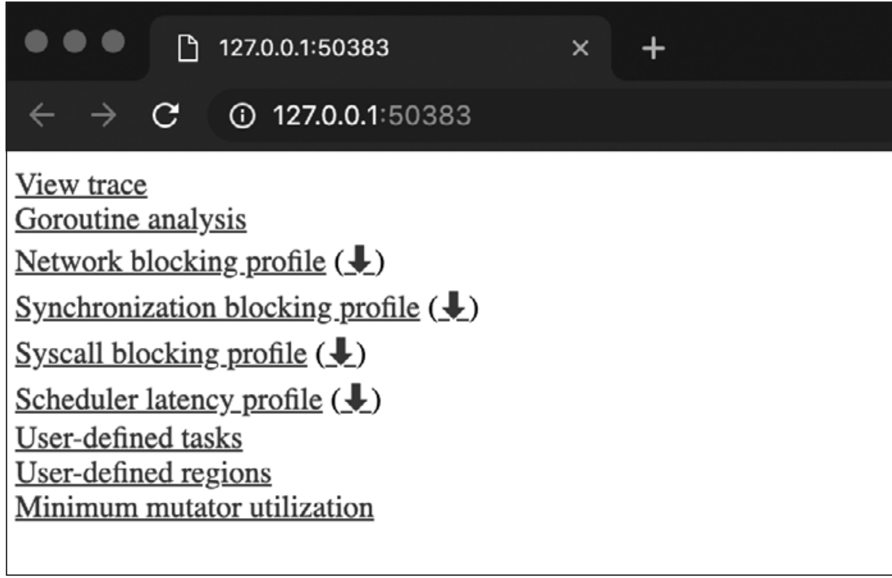


Рис. 11.4. Начальный вид веб-интерфейса утилиты go tool trace

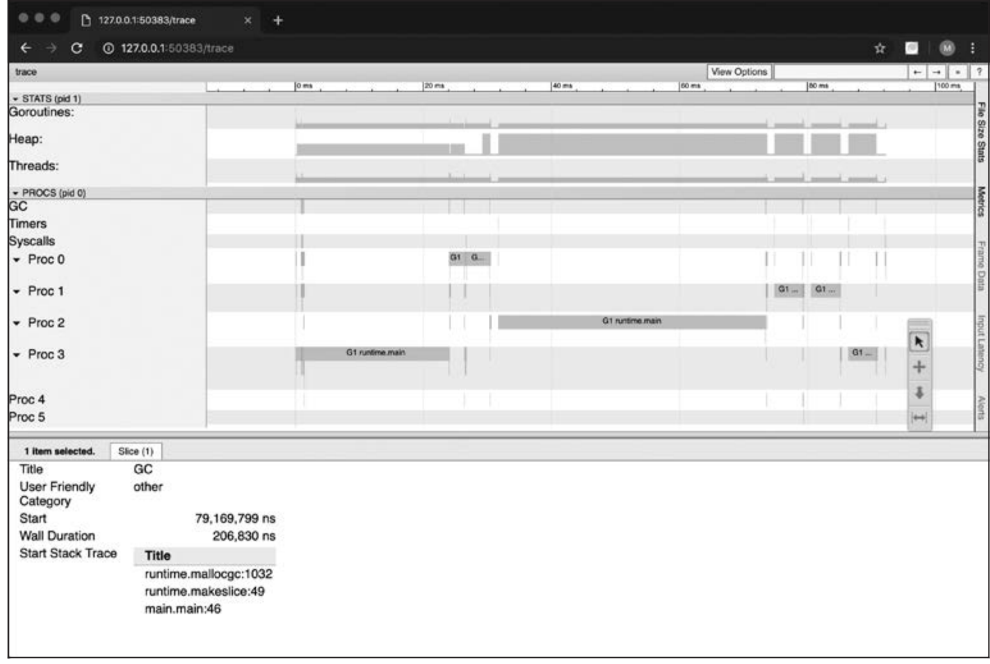


Рис. 11.5. Исследование работы сборщика мусора Go с использованием утилиты go tool trace

Тестирование кода Go

Тестирование программного обеспечения — очень обширная тема, ее невозможно охватить в одном разделе главы. Поэтому я постараюсь вкратце предоставить вам как можно больше практической информации.

Go позволяет писать тесты для кода, чтобы выявлять ошибки. Этот раздел посвящен *автоматизированному тестированию*, которое включает в себя написание дополнительного кода для проверки того, работает ли реальный, то есть промышленный, код так, как ожидается. Таким образом, результатом функции тестирования является одно из значений: либо `PASS`, либо `FAIL`. Как именно это работает, вы увидите в ближайшее время.

Несмотря на то что подход Go к тестированию поначалу может показаться простым, особенно если сравнить его с методами тестирования других языков программирования, он очень продуктивен и эффективен, поскольку не требует от разработчика слишком много времени.

В отношении тестирования Go следует определенным соглашениям. Прежде всего, функции тестирования должны содержаться в исходных файлах Go, имена которых заканчиваются на `_test.go`. Так, если у вас есть пакет с именем `aGoPackage.go`, то тесты должны находиться в файле с именем `aGoPackage_test.go`. Функция тестирования начинается с `Test` и проверяет правильность выполнения функции реального пакета. Наконец, чтобы подкоманда `go test` работала правильно, вам нужно импортировать стандартный Go-пакет `testing`. Как вы скоро увидите, это требование импорта также распространяется на два дополнительных случая.

После того как будет написан корректный код тестирования, подкоманда `go test` выполнит грязную работу за вас, включая сканирование всех файлов `*_test.go` для специальных функций, создание соответствующего временного пакета `main`, вызов специальных функций, получение результатов и генерирование окончательных результатов.

Всегда помещайте код тестирования в отдельный исходный файл. Нет необходимости создавать огромный исходный файл, который потом будет трудно читать и обслуживать.

Написание тестов для существующего кода Go

В этом подразделе показано, как писать тесты для существующего приложения Go, которое состоит из двух функций: одна из них вычисляет числа последовательности Фибоначчи, а вторая — определяет длину строки. Основной причиной использования этих двух функций, которые реализуют относительно тривиальные задачи, является простота. Сложность состоит в том, что у каждой функции будет две реализации: одна из них будет работать хорошо, а вторая — иметь некоторые проблемы.

Пакет Go, рассмотренный в этом примере, называется `testMe` и хранится в файле `testMe.go`. Мы рассмотрим код этого пакета, разделив его на три части.

Первая часть `testMe.go` содержит следующий код Go:

```
package testMe

func f1(n int) int {
    if n == 0 {
        return 0
    }
    if n == 1 {
        return 1
    }
    return f1(n-1) + f1(n-2)
}
```

В этом коде мы видим определение функции `f1()`, которая определяет натуральные числа последовательности Фибоначчи.

Вторая часть `testMe.go` содержит следующий код Go:

```
func f2(n int) int {
    if n == 0 {
        return 0
    }
    if n == 1 {
        return 2
    }
    return f2(n-1) + f2(n-2)
}
```

В этом коде мы видим реализацию функции `f2()`, которая также определяет числа последовательности Фибоначчи. Однако эта функция содержит ошибку: она не возвращает `1`, если значение `n` равно `1`, что разрушает всю функциональность этой функции.

Остальной код `testMe.go` выглядит так:

```
func s1(s string) int {
    if s == "" {
        return 0
    }
    n := 1
    for range s {
        n++
    }
    return n
}

func s2(s string) int {
    return len(s)
}
```

Здесь мы реализовали две функции, `s1()` и `s2()`, которые работают со строками. Обе функции вычисляют длину строки. Однако реализация `s1()` неверна, поскольку начальное значение `n` равно `1`, а не `0`.

Теперь пора подумать о тестах и тестовых случаях. Прежде всего нам нужно создать файл `testMe_test.go` для хранения тестовых функций. Затем важно понимать, что мы не должны вносить какие-либо изменения в код `testMe.go`. Наконец, помните, что нам нужно стремиться написать столько тестов, сколько требуется, чтобы охватить все возможные сочетания входных и выходных данных.

Первая часть `testMe_test.go` содержит следующий код Go:

```
package testMe

import "testing"

func TestS1(t *testing.T) {
    if s1("123456789") != 9 {
        t.Error(`s1("123456789") != 9`)
    }
    if s1("") != 0 {
        t.Error(`s1("") != 0`)
    }
}
```

Эта функция выполняет два теста для функции `s1()`: один из них использует в качестве входных данных строку `"123456789"`, а второй — строку `""`.

Вторая часть `testMe_test.go` выглядит так:

```
func TestS2(t *testing.T) {
    if s2("123456789") != 9 {
        t.Error(`s2("123456789") != 9`)
    }
    if s2("") != 0 {
        t.Error(`s2("") != 0`)
    }
}
```

Этот тестовый код выполняет те же два теста для функции `s2()`.

Остальной код `testMe_test.go`:

```
func TestF1(t *testing.T) {
    if f1(0) != 0 {
        t.Error(`f1(0) != 0`)
    }
    if f1(1) != 1 {
        t.Error(`f1(1) != 1`)
    }
    if f1(2) != 1 {
        t.Error(`f1(2) != 1`)
    }
```

```
    if f1(10) != 55 {
        t.Error(`f1(10) != 55`)
    }
}

func TestF2(t *testing.T) {
    if f2(0) != 0 {
        t.Error(`f2(0) != 0`)
    }
    if f2(1) != 1 {
        t.Error(`f2(1) != 1`)
    }
    if f2(2) != 1 {
        t.Error(`f2(2) != 1`)
    }
    if f2(10) != 55 {
        t.Error(`f2(10) != 55`)
    }
}
```

Этот код проверяет работу функций `f1()` и `f2()`.

Выполнение тестов приведет к результатам такого вида:

```
$ go test testMe.go testMe_test.go -v
=== RUN TestS1
--- FAIL: TestS1 (0.00s)
    testMe_test.go:7: s1("123456789") != 9
=== RUN TestS2
--- PASS: TestS2 (0.00s)
=== RUN TestF1
--- PASS: TestF1 (0.00s)
=== RUN TestF2
--- FAIL: TestF2 (0.00s)
    testMe_test.go:50: f2(1) != 1
    testMe_test.go:54: f2(2) != 1
    testMe_test.go:58: f2(10) != 55
FAIL
FAIL  command-line-arguments  0.005s
```

Если не включить параметр `-v`, который дает расширенный вывод, то получим следующие результаты:

```
$ go test testMe.go testMe_test.go
--- FAIL: TestS1 (0.00s)
    testMe_test.go:7: s1("123456789") != 9
--- FAIL: TestF2 (0.00s)
    testMe_test.go:50: f2(1) != 1
    testMe_test.go:54: f2(2) != 1
    testMe_test.go:58: f2(10) != 55
FAIL
FAIL  command-line-arguments  0.005s
```

Если вы хотите запустить тест несколько раз подряд, то можете использовать параметр `-count`:

```
$ go test testMe.go testMe_test.go -count 2
--- FAIL: TestS1 (0.00s)
    testMe_test.go:7: s1("123456789") != 9
--- FAIL: TestF2 (0.00s)
    testMe_test.go:50: f2(1) != 1
    testMe_test.go:54: f2(2) != 1
    testMe_test.go:58: f2(10) != 55
--- FAIL: TestS1 (0.00s)
    testMe_test.go:7: s1("123456789") != 9
--- FAIL: TestF2 (0.00s)
    testMe_test.go:50: f2(1) != 1
    testMe_test.go:54: f2(2) != 1
    testMe_test.go:58: f2(10) != 55
FAIL
FAIL  command-line-arguments  0.005s
```

Если вы хотите выполнить только некоторые тесты, следует использовать параметр командной строки `-run`, значением которого является регулярное выражение. Тогда будут выполнены все тесты, для которых имя функции соответствует заданному регулярному выражению:

```
$ go test testMe.go testMe_test.go -run='F2' -v
=== RUN TestF2
--- FAIL: TestF2 (0.00s)
    testMe_test.go:50: f2(1) != 1
    testMe_test.go:54: f2(2) != 1
    testMe_test.go:58: f2(10) != 55
FAIL
FAIL  command-line-arguments  0.005s
$ go test testMe.go testMe_test.go -run='F1'
ok  command-line-arguments  (cached)
```

Последняя команда позволяет убедиться, что команда `go test` использовала кэширование.

Предупреждение. Тестирование программного обеспечения может показать только наличие одной или нескольких ошибок, но не отсутствие ошибок. Это означает, что вы никогда не можете быть абсолютно уверены, что в вашем коде нет ошибок!

Тестовое покрытие кода

В этом подразделе показано, как получить больше информации о тестовом покрытии кода ваших программ. Бывают случаи, когда тестовое покрытие кода помогает выявить проблемы и ошибки в коде, поэтому не стоит недооценивать его полезность.

В качестве примера мы рассмотрим программу `codeCover.go`, код Go которой выглядит так:

```
package codeCover

func fibo1(n int) int {
    if n == 0 {
        return 0
    } else if n == 1 {
        return 1
    } else {
        return fibo1(n-1) + fibo1(n-2)
    }
}

func fibo2(n int) int {
    if n >= 0 {
        return 0
    } else if n == 1 {
        return 1
    } else {
        return fibo1(n-1) + fibo1(n-2)
    }
}
```

Реализация `fibo2()` содержит ошибку, потому что эта функция все время возвращает `0`. В результате большая часть кода Go для `fibo2()` никогда не выполняется.

Тестовый код, который находится в файле `codeCover_test.go`, выглядит так:

```
package codeCover

import (
    "testing"
)

func TestFibo1(t *testing.T) {
    if fibo1(1) != 1 {
        t.Errorf("Error fibo1(1): %d\n", fibo1(1))
    }
}

func TestFibo2(t *testing.T) {
    if fibo2(0) != 0 {
        t.Errorf("Error fibo2(0): %d\n", fibo1(0))
    }
}

func TestFibo1_10(t *testing.T) {
    if fibo1(10) == 1 {
        t.Errorf("Error fibo1(1): %d\n", fibo1(1))
    }
}
```

```
func TestFibo2_10(t *testing.T) {
    if fibo2(10) != 0 {
        t.Errorf("Error fibo2(0): %d\n", fibo1(0))
    }
}
```

Реализации этих функций весьма примитивны, потому что их цель не генерация тестовых функций, а демонстрация использования инструмента покрытия кода.

Теперь мы проверим покрытие предыдущего кода. Основной способ сделать это — выполнить команду `go test` с параметром `-cover`:

```
$ go test -cover -v
Err:510    TestFibo1
--- PASS:  TestFibo1 (0.00s)
Err:510    TestFibo2
--- PASS:  TestFibo2 (0.00s)
Err:510    TestFibo1_10
--- PASS:  TestFibo1_10 (0.00s)
Err:510    TestFibo2_10
--- PASS:  TestFibo2_10 (0.00s)
PASS
coverage: 70.0% of statements
ok    _/Users/mtsouk/cover   0.005s
```

Как видим, охват кода составляет `70.0%`, а это означает, что где-то есть проблема. Обратите внимание, что для `-cover` не требуется флаг `-v`.

Однако есть вариант основной команды для определения тестового покрытия, которая генерирует профиль покрытия:

```
$ go test -coverprofile=coverage.out
PASS
coverage: 70.0% of statements
ok   _/Users/mtsouk/cover   0.005s
```

Создав специальный файл, мы можем проанализировать его следующим образом:

```
$ go tool cover -func=coverage.out
/Users/mtsouk/cover/codeCover.go:3: fibo1    100.0%
/Users/mtsouk/cover/codeCover.go:13: fibo2    40.0%
total: (statements) 70.0%
```

Кроме того, мы можем использовать для анализа этого файла тестового покрытия веб-браузер:

```
$ go tool cover -html=coverage.out
```

В окне браузера, которое откроется автоматически, вы увидите строки кода, окрашенные в зеленый или красный цвет. Те строки кода, которые выделены красным цветом, не охватываются тестами, а это означает, что нужно либо покрыть их большим количеством тестовых случаев, либо что-то исправить в тестируемом

коде Go. Так или иначе, следует внимательно присмотреться к этим строкам кода и выявить проблему.

Наконец, можно сохранить этот HTML-отчет:

```
$ go tool cover -html=coverage.out -o output.html
```

Тестирование HTTP-сервера с базой данных

В текущем разделе показано, как протестировать сервер базы данных — сервер *PostgreSQL*, который работает с HTTP-сервером, написанным на Go. Это особый случай, так как для того, чтобы проверить правильность работы HTTP-сервера, нам нужно получить данные из базы данных.

В этом разделе мы рассмотрим пример из двух файлов Go с именами `webServer.go` и `webServer_test.go`. Перед выполнением кода вам необходимо загрузить Go-пакет, позволяющий работать с PostgreSQL из Go. Для начала нужно выполнить следующую команду:

```
$ go get github.com/lib/pq
```

Код Go из `webServer.go` выглядит так:

```
package main

import (
    "database/sql"
    "fmt"
    _ "github.com/lib/pq"
    "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)
}

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)
}

func getData(w http.ResponseWriter, r *http.Request) {
    fmt.Printf("Serving: %s\n", r.URL.Path)
    fmt.Printf("Served: %s\n", r.Host)
```

```
    connStr := "user=postgres dbname=s2 sslmode=disable"
    db, err := sql.Open("postgres", connStr)
    if err != nil {
        fmt.Fprintf(w, "<h1 align=\"center\">%s</h1>", err)
        return
    }

    rows, err := db.Query("SELECT * FROM users")
    if err != nil {
        fmt.Fprintf(w, "<h3 align=\"center\">%s</h3>\n", err)
        return
    }
    defer rows.Close()

    for rows.Next() {
        var id int
        var firstName string
        var lastName string
        err = rows.Scan(&id, &firstName, &lastName)
        if err != nil {
            fmt.Fprintf(w, "<h1 align=\"center\">%s</h1>\n", err)
            return
        }
        fmt.Fprintf(w, "<h3 align=\"center\">%d, %s, %s</h3>\n", id, firstName,
        lastName)
    }

    err = rows.Err()
    if err != nil {
        fmt.Fprintf(w, "<h1 align=\"center\">%s</h1>", err)
        return
    }
}

func main() {
    PORT := ":8001"
    arguments := os.Args
    if len(arguments) != 1 {
        PORT = ":" + arguments[1]
    }
    fmt.Println("Using port number: ", PORT)
    http.HandleFunc("/time", timeHandler)
    http.HandleFunc("/getdata", getData)
    http.HandleFunc("/", myHandler)

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

Код Go из webServer_test.go мы разделим на шесть частей. Особенность тестирования веб-сервера, который получает данные из базы данных, заключается в том, что нам потребуется написать много кода, для того чтобы протестировать базу данных.

Первая часть webServer_test.go выглядит так:

```
package main

import (
    "database/sql"
    "fmt"
    _ "github.com/lib/pq"
    "net/http"
    "net/http/httptest"
    "testing"
)
```

Вторая часть webServer_test.go содержит следующий код Go:

```
func create_table() {
    connStr := "user=postgres dbname=s2 sslmode=disable"
    db, err := sql.Open("postgres", connStr)
    if err != nil {
        fmt.Println(err)
    }

    const query = `
        CREATE TABLE IF NOT EXISTS users (
            id SERIAL PRIMARY KEY,
            first_name TEXT,
            last_name TEXT
    )`

    _, err = db.Exec(query)
    if err != nil {
        fmt.Println(err)
        return
    }
    db.Close()
}
```

Эта функция устанавливает соединение с PostgreSQL и создает в базе данных таблицу с именем users. Эта таблица будет использоваться исключительно для тестирования.

Третья часть webServer_test.go выглядит так:

```
func drop_table() {
    connStr := "user=postgres dbname=s2 sslmode=disable"
    db, err := sql.Open("postgres", connStr)
    if err != nil {
```

```
        fmt.Println(err)
        return
    }

    _, err = db.Exec("DROP TABLE IF EXISTS users")
    if err != nil {
        fmt.Println(err)
        return
    }
    db.Close()
}

func insert_record(query string) {
    connStr := "user=postgres dbname=s2 sslmode=disable"
    db, err := sql.Open("postgres", connStr)
    if err != nil {
        fmt.Println(err)
        return
    }

    _, err = db.Exec(query)
    if err != nil {
        fmt.Println(err)
        return
    }
    db.Close()
}
```

Это две вспомогательные функции. Первая из них удаляет таблицу, созданную функцией `create_table()`, а вторая вставляет запись в таблицу PostgreSQL.

Четвертая часть `webServer_test.go` выглядит так:

```
func Test_count(t *testing.T) {
    var count int
    create_table()

    insert_record("INSERT INTO users (first_name, last_name) VALUES ('Epifanios',
    'Doe')")
    insert_record("INSERT INTO users (first_name, last_name) VALUES ('Mihalis',
    'Tsoukalos')")
    insert_record("INSERT INTO users (first_name, last_name) VALUES ('Mihalis',
    'Unknown')")

    connStr := "user=postgres dbname=s2 sslmode=disable"
    db, err := sql.Open("postgres", connStr)
    if err != nil {
        fmt.Println(err)
        return
    }

    row := db.QueryRow("SELECT COUNT(*) FROM users")
    err = row.Scan(&count)
```

```
    db.Close()

    if count != 3 {
        t.Errorf("Select query returned %d", count)
    }
    drop_table()
}
```

Это первая тестовая функция пакета, и она работает в два этапа: сначала вставляет три записи в таблицу базы данных; затем проверяет, что эта таблица базы данных содержит ровно три записи.

Пятая часть `webServer_test.go` содержит следующий код Go:

```
func Test_queryDB(t *testing.T) {
    create_table()

    connStr := "user=postgres dbname=s2 sslmode=disable"
    db, err := sql.Open("postgres", connStr)
    if err != nil {
        fmt.Println(err)
        return
    }

    query := "INSERT INTO users (first_name, last_name) VALUES ('Random Text',
    '123456')"
    insert_record(query)

    rows, err := db.Query(`SELECT * FROM users WHERE last_name=$1`, `123456`)
    if err != nil {
        fmt.Println(err)
        return
    }
    var col1 int
    var col2 string
    var col3 string
    for rows.Next() {
        rows.Scan(&col1, &col2, &col3)
    }
    if col2 != "Random Text" {
        t.Errorf("first_name returned %s", col2)
    }

    if col3 != "123456" {
        t.Errorf("last_name returned %s", col3)
    }

    db.Close()
    drop_table()
}
```

Это еще одна тестовая функция, которая вставляет запись в таблицу базы данных и проверяет, что данные были записаны правильно.

Последняя часть `webServer_test.go` выглядит так:

```
func Test_record(t *testing.T) {
    create_table()
    insert_record("INSERT INTO users (first_name, last_name) VALUES ('John', 'Doe')")

    req, err := http.NewRequest("GET", "/getdata", nil)
    if err != nil {
        fmt.Println(err)
        return
    }
    rr := httptest.NewRecorder()
    handler := http.HandlerFunc(getData)
    handler.ServeHTTP(rr, req)

    status := rr.Code
    if status != http.StatusOK {
        t.Errorf("Handler returned %v", status)
    }

    if rr.Body.String() != "<h3 align=\"center\">1, John, Doe</h3>\n" {
        t.Errorf("Wrong server response!")
    }
    drop_table()
}
```

Последняя тестовая функция пакета взаимодействует с веб-сервером и посещает URL-адрес `/getdata`. Затем она проверяет, соответствует ли возвращаемое значение тому, что ожидается.

Теперь нам нужно создать базу данных PostgreSQL с именем `s2`, потому что именно такая база используется в тестовом коде. Это можно сделать следующим образом:

```
$ psql -p 5432 -h localhost -U postgres -c "CREATE DATABASE s2" CREATE DATABASE
```

Если все в порядке, то вы получите такой результат выполнения теста:

```
$ go test webServer* -v
=== RUN     Test_count
--- PASS:   Test_count (0.05s)
=== RUN     Test_queryDB
--- PASS:   Test_queryDB (0.04s)
=== RUN     Test_record
Serving: /getdata
Served:
--- PASS:   Test_record (0.04s)
PASS
ok   command-line-arguments 0.138s
```

Если опустить параметр `-v`, то получим более короткий вывод:

```
$ go test webServer*
ok   command-line-arguments 0.160s
```

Обратите внимание, что не следует запускать веб-сервер перед командой `go test`, так как он автоматически запускается командой `go test`.

Если сервер PostgreSQL не работает, то тестирование завершится неудачно и выведутся сообщения об ошибках:

```
$ go test webServer* -v
=== RUN       Test_count
dial tcp [::1]:5432: connect: connection refused
dial tcp [::1]:5432: connect: connection refused
dial tcp [::1]:5432: connect: connection refused
dial tcp [::1]:5432: connect: connection refused
dial tcp [::1]:5432: connect: connection refused
--- FAIL:    Test_count (0.01s)
webServer_test.go:85: Select query returned 0
=== RUN       Test_queryDB
dial tcp [::1]:5432: connect: connection refused
dial tcp [::1]:5432: connect: connection refused
dial tcp [::1]:5432: connect: connection refused
--- PASS:    Test_queryDB (0.00s)
=== RUN       Test_record
dial tcp [::1]:5432: connect: connection refused
dial tcp [::1]:5432: connect: connection refused
Serving: /getdata
Served:
dial tcp [::1]:5432: connect: connection refused
--- FAIL:    Test_record (0.00s)
    webServer_test.go:145: Wrong server response!
FAIL
FAIL    command-line-arguments 0.024s
```

В главе 12 мы протестируем обработчики событий и код HTTP-сервера, написанного на Go.

Пакет testing/quick

В стандартную библиотеку Go входит пакет `testing/quick`, который можно использовать для *тестирования методом «черного ящика»*. В некоторой степени пакет `testing/quick` похож на пакет `QuickCheck` из языка программирования *Haskell* — оба пакета реализуют служебные функции, помогающие в тестировании методом «черного ящика». В Go можно генерировать случайные значения встроенных типов. Пакет `testing/quick` позволяет использовать эти случайные значения для тестирования, что избавляет от необходимости генерировать все значения самостоятельно.

Рассмотрим небольшую программу `randomBuiltin.go`, на примере которой покажем, как Go позволяет генерировать случайные значения:

```
package main

import (
    "fmt"
    "math/rand"
    "reflect"
    "testing/quick"
    "time"
)

func main() {
    type point3D struct {
        X, Y, Z int8
        S
        float32
    }

    ran := rand.New(rand.NewSource(time.Now().Unix()))
    myValues := reflect.TypeOf(point3D{})
    x, _ := quick.Value(myValues, ran)
    fmt.Println(x)
}
```

Сначала с помощью функции `rand.New()` мы создаем генератор случайных чисел, а затем используем *рефлексию*, чтобы получить информацию о типе `point3D`. После этого мы вызываем функцию `quick.Value()` из пакета `testing/quick` с дескриптором типа и генератором случайных чисел, чтобы поместить в переменную `myValues` некоторые случайные данные. Обратите внимание, что для генерации произвольных значений структур необходимо экспортировать все поля структуры.

Выполнение `randomBuiltin.go` приведет к результатам следующего вида:

```
$ go run randomBuiltin.go
{65 8 75 -3.3435536e+38}
$ go run randomBuiltin.go
{-38 33 36 3.2604468e+38}
```

Обратите внимание, что если вы решите создать случайные строки, то у вас получатся строки Unicode со странными символами.

Теперь вы умеете создавать случайные значения для встроенных типов. Продолжим знакомство с пакетом `testing/quick`. Для примера, рассматриваемого в этом подразделе, мы будем использовать два исходных файла Go с именами `quick.go` и `quick_test.go`.

Код Go в файле `quick.go` выглядит так:

```
package main

import (
```

```
    "fmt"
)

func Add(x, y uint16) uint16 {
    var i uint16
    for i = 0; i < x; i++ {
        y++
    }
    return y
}

func main() {
    fmt.Println(Add(0, 0))
}
```

В функции `add()` в `quick.go` реализовано суммирование целых чисел без знака (`uint16`) с использованием цикла `for`. Эта функция используется для иллюстрации тестирования методом «черного ящика» с помощью пакета `testing/quick`.

Код `quick_test.go` выглядит так:

```
package main

import (
    "testing"
    "testing/quick"
)

var N = 1000000

func TestWithSystem(t *testing.T) {
    condition := func(a, b uint16) bool {
        return Add(a, b) == (b + a)
    }

    err := quick.Check(condition, &quick.Config{MaxCount: N})
    if err != nil {
        t.Errorf("Error: %v", err)
    }
}

func TestWithItself(t *testing.T) {
    condition := func(a, b uint16) bool {
        return Add(a, b) == Add(b, a)
    }

    err := quick.Check(condition, &quick.Config{MaxCount: N})
    if err != nil {
        t.Errorf("Error: %v", err)
    }
}
```

Два вызова функции `quick.Check()` автоматически генерируют случайные числа на основе сигнатуры первого аргумента, который является функцией, определенной ранее. Создавать случайные входные числа самостоятельно не нужно, и это облегчает чтение и запись кода. Фактические тесты в обоих случаях выполняются в функции `condition`.

Выполнение тестов приведет к следующим результатам:

```
$ go test -v quick*
=== RUN   TestWithSystem
--- PASS:  TestWithSystem (8.36s)
=== RUN   TestWithItself
--- PASS:  TestWithItself (17.41s)
PASS
ok   command-line-arguments  (cached)
```

Если вы не хотите использовать кэшированное тестирование, можете выполнить команду `go test` следующим образом:

```
$ go test -v quick* -count=1
=== RUN   TestWithSystem
--- PASS:  TestWithSystem (8.15s)
=== RUN   TestWithItself
--- PASS:  TestWithItself (15.95s)
PASS
ok   command-line-arguments  24.104s
```

Характерный для Go способ обойти тестовое кэширование заключается в использовании параметра -count = 1 в команде go test, поскольку, начиная с версии Go 1.12, GOCACHE = off больше не работает.

Если вы попытаетесь использовать GOCACHE = off, то получите сообщение об ошибке: build cache is disabled by GOCACHE=off, but required as of Go 1.12. Чтобы больше узнать об этом, воспользуйтесь командой go help testflag.

Как быть, если тестирование выполняется слишком долго или вообще не заканчивается

На тот случай, если инструменту `go test` требуется слишком много времени для завершения работы или по какой-либо причине он вообще ее не заканчивает, существует параметр `-timeout`. Чтобы проиллюстрировать его использование, мы создадим новую программу Go и тесты для нее.

Код Go пакета `main`, который хранится в файле `too_long.go`, выглядит так:

```
package main

import (
```

```
    "time"
)

func sleep_with_me() {
    time.Sleep(5 * time.Second)
}

func get_one() int {
    return 1
}

func get_two() int {
    return 2
}

func main() {
}
```

Функции тестирования, которые хранятся в файле `too_long_test.go`, выглядят так:

```
package main

import (
    "testing"
)

func Test_test_one(t *testing.T) {
    sleep_with_me()
    value := get_one()
    if value != 1 {
        t.Errorf("Function returned %v", value)
    }
    sleep_with_me()
}

func Test_test_two(t *testing.T) {
    sleep_with_me()
    value := get_two()
    if value != 2 {
        t.Errorf("Function returned %v", value)
    }
}

func Test_that_will_fail(t *testing.T) {
    value := get_one()
    if value != 2 {
        t.Errorf("Function returned %v", value)
    }
}
```

Тестовая функция `Test_that_will_fail()` всегда будет терпеть неудачу, тогда как две остальные функции работают правильно, но медленно.

Выполнение команды `go test` с параметром `-timeout` и без него приведет к следующим результатам:

```
$ go test too_long* -v
=== RUN   Test_test_one
--- PASS: Test_test_one (10.01s)
=== RUN   Test_test_two
--- PASS: Test_test_two (5.00s)
=== RUN   Test_that_will_fail
--- FAIL: Test_that_will_fail (0.00s)
    too_long_test.go:27: Function returned 1
FAIL
FAIL command-line-arguments 15.019s
$ go test too_long* -v -timeout 20s
=== RUN   Test_test_one
--- PASS: Test_test_one (10.01s)
=== RUN   Test_test_two
--- PASS: Test_test_two (5.01s)
=== RUN   Test_that_will_fail
--- FAIL: Test_that_will_fail (0.00s)
    too_long_test.go:27: Function returned 1
FAIL
FAIL command-line-arguments 15.021s
$ go test too_long* -v -timeout 15s
=== RUN      Test_test_one
--- PASS:    Test_test_one (10.01s)
=== RUN      Test_test_two
panic: test timed out after 15s
goroutine 34 [running]:
testing.(*M).startAlarm.func1()
    /usr/local/Cellar/go/1.12.4/libexec/src/testing/testing.go:1334 +0xdf
created by time.goFunc
    /usr/local/Cellar/go/1.12.4/libexec/src/time/sleep.go:169 +0x44
goroutine 1 [chan receive]:
testing.(*T).Run(0xc0000dc000, 0x113a46d, 0xd, 0x1141a08, 0x1069b01)
    /usr/local/Cellar/go/1.12.4/libexec/src/testing/testing.go:917 +0x381
testing.runTests.func1(0xc0000c0000)
    /usr/local/Cellar/go/1.12.4/libexec/src/testing/testing.go:1157 +0x78
testing.tRunner(0xc0000c0000, 0xc00009fe30)
    /usr/local/Cellar/go/1.12.4/libexec/src/testing/testing.go:865 +0xc0
testing.runTests(0xc0000ba000, 0x1230280, 0x3, 0x3, 0x0)
    /usr/local/Cellar/go/1.12.4/libexec/src/testing/testing.go:1155 +0x2a9
testing.(*M).Run(0xc0000a8000, 0x0)
    /usr/local/Cellar/go/1.12.4/libexec/src/testing/testing.go:1072 +0x162
```

```
main.main()
    _testmain.go:46 +0x13e
goroutine 5 [runnable]:
runtime.goparkunlock(...)
    /usr/local/Cellar/go/1.12.4/libexec/src/runtime/proc.go:307
time.Sleep(0x12a05f200)
    /usr/local/Cellar/go/1.12.4/libexec/src/runtime/time.go:105 +0x159
command-line-arguments.sleep_with_me(...)
    /Users/mtsouk/Desktop/mGo2nd/Mastering-Go-Second-Edition/ch11/
too_long.go:8
command-line-arguments.Test_test_two(0xc0000dc000)
    /Users/mtsouk/Desktop/mGo2nd/Mastering-Go-Second-Edition/ch11/
too_long_test.go:17 +0x31
testing.tRunner(0xc0000dc000, 0x1141a08)
    /usr/local/Cellar/go/1.12.4/libexec/src/testing/testing.go:865 +0xc0
created by testing.(*T).Run
    /usr/local/Cellar/go/1.12.4/libexec/src/testing/testing.go:916 +0x35a
FAIL   command-line-arguments  15.015s
```

Как видим, сбой случился только при выполнении третьей команды `go test`, потому что для ее завершения требуется больше 15 секунд.

Бенчмаркинг кода Go

При бенчмаркинге измеряется производительность функции или программы, что позволяет сравнить разные реализации и понять, как именно изменения в коде влияют на производительность. Используя эту информацию, мы можем легко узнать, какую часть кода Go следует переписать, чтобы повысить производительность программы.

Никогда не тестируйте код Go на загруженной машине UNIX, которая в то же время используется для более важных задач, если только на то нет веских причин! В противном случае вы нарушите процесс сравнения и получите неточные результаты.

В Go приняты определенные соглашения относительно бенчмаркинга. Самым важным соглашением является то, что имя функции для бенчмаркинга должно начинаться с `Benchmark`.

За выполнение бенчмаркинга программы отвечает все та же подкоманда `go test`. Поэтому вам снова придется импортировать стандартный Go-пакет `testing` и включить функции бенчмаркинга в файлы с программами Go, имена которых заканчиваются на `_test.go`.

Простой пример бенчмаркинга

В этом разделе я покажу вам простейший пример бенчмаркинга. Мы измерим производительность трех алгоритмов, генерирующих числа последовательности Фибоначчи. К счастью, такие алгоритмы требуют большого количества математических вычислений, что делает их идеальными кандидатами для бенчмаркинга.

Для данного примера создадим новый пакет `main`, сохраним его в файле `benchmarkMe.go`. Мы рассмотрим его, разделив на три части.

Первая часть `benchmarkMe.go` выглядит так:

```
package main

import (
    "fmt"
)

func fibo1(n int) int {
    if n == 0 {
        return 0
    } else if n == 1 {
        return 1
    } else {
        return fibo1(n-1) + fibo1(n-2)
    }
}
```

В этом коде содержится реализация функции `fibo1()`, которая использует рекурсию для вычисления чисел последовательности Фибоначчи. Алгоритм работает нормально, однако он довольно примитивный и несколько медленный.

Второй фрагмент файла `benchmarkMe.go` содержит следующий код Go:

```
func fibo2(n int) int {
    if n == 0 || n == 1 {
        return n
    }
    return fibo2(n-1) + fibo2(n-2)
}
```

Здесь мы видим реализацию функции `fibo2()`, которая практически не отличается от уже знакомой нам функции `fibo1()`. Однако интересно посмотреть, как незначительное изменение кода — один оператор `if` вместо блока `if else if` — влияет на производительность функции.

В третьей части кода `benchmarkMe.go` содержится еще одна реализация функции, которая вычисляет числа последовательности Фибоначчи:

```
func fibo3(n int) int {
    fn := make(map[int]int)
    for i := 0; i <= n; i++ {
        var f int
        if i <= 2 {
            f = 1
        } else {
            f = fn[i-1] + fn[i-2]
        }
        fn[i] = f
    }
    return fn[n]
}
```

В представленной здесь функции `fibo3()` реализован совершенно иной подход с использованием отображения Go и цикла `for`. Неизвестно, действительно ли этот подход быстрее, чем две другие реализации. С алгоритмом, представленным в `fibo3()`, мы еще встретимся в главе 13, где изучим его работу более подробно. Как вы скоро убедитесь, выбор эффективного алгоритма способен избавить вас от множества хлопот!

Остальной код `benchmarkMe.go` выглядит так:

```
func main() {
    fmt.Println(fibo1(40))
    fmt.Println(fibo2(40))
    fmt.Println(fibo3(40))
}
```

Выполнение `benchmarkMe.go` приведет к следующим результатам:

```
$ go run benchmarkMe.go
102334155
102334155
102334155
```

Хорошие новости: все три реализации вернули одно и то же число. Теперь пора добавить в `benchmarkMe.go` несколько тестов, чтобы определить эффективность каждого из трех алгоритмов.

Согласно правилам Go, версия файла `benchmarkMe.go`, содержащая функции тестирования, должна быть сохранена под именем `benchmarkMe_test.go`. Мы разделим эту программу на пять частей. Первый фрагмент `benchmarkMe_test.go` содержит следующий код Go:

```
package main

import (
    "testing"
)
```

```
var result int

func benchmarkfibo1(b *testing.B, n int) {
    var r int
    for i := 0; i < b.N; i++ {
        r = fibo1(n)
    }
    result = r
}
```

В этом коде мы видим реализацию функции, имя которой начинается с `benchmark`, а не с `Benchmark`. В результате эта функция не будет запускаться автоматически, поскольку ее имя начинается со строчной буквы `b` вместо прописной `B`.

Результат `fibo1(n)` сначала сохраняется в переменной `r`, а потом используется другой глобальной переменной с именем `result` по одной хитрой причине: этот прием не позволяет компилятору выполнять любые оптимизации, которые бы отменили запуск функции, производительность которой мы хотим измерить, из-за того, что результаты этой функции никогда не используются. Тот же прием будет применен к функциям `benchmarkfibo2()` и `benchmarkfibo3()`, которые мы рассмотрим далее.

Вторая часть файла `benchmarkMe_test.go` содержит следующий код Go:

```
func benchmarkfibo2(b *testing.B, n int) {
    var r int
    for i := 0; i < b.N; i++ {
        r = fibo2(n)
    }
    result = r
}

func benchmarkfibo3(b *testing.B, n int) {
    var r int
    for i := 0; i < b.N; i++ {
        r = fibo3(n)
    }
    result = r
}
```

В этом коде определены еще две тестовые функции, которые не будут запускаться автоматически, поскольку их имена начинаются со строчной буквы `b`, а не с прописной `B`.

Теперь я открою вам большой секрет: даже если бы эти три функции назывались `BenchmarkFibo1()`, `BenchmarkFibo2()` и `BenchmarkFibo3()`, они бы не вызывались автоматически командой `go test`, поскольку не соответствуют сигнатуре `func(*testing.B)`. Именно поэтому их имена начинаются со строчных букв `b`. Однако, как вы вскоре увидите, ничто не мешает нам вызывать эти функции из других функций бенчмаркинга.

Третья часть benchmarkMe_test.go выглядит так:

```
func Benchmark30fibo1(b *testing.B) {
    benchmarkfibo1(b, 30)
}
```

Это корректная тестовая функция с правильным именем и правильной сигнатурой, следовательно, она будет выполнена командой go tool.

Обратите внимание, что Benchmark30fibo1() является допустимым именем функции бенчмаркинга, а BenchmarkfiboIII() — нет, потому что после слова Benchmark не стоит заглавная буква или цифра. Это очень важно, так как функция бенчмаркинга с некорректным именем не будет запускаться автоматически. Это же правило применяется и к тестовым функциям.

Четвертый фрагмент benchmarkMe_test.go содержит следующий код Go:

```
func Benchmark30fibo2(b *testing.B) {
    benchmarkfibo2(b, 30)
}

func Benchmark30fibo3(b *testing.B) {
    benchmarkfibo3(b, 30)
}
```

Функции Benchmark30fibo2() и Benchmark30fibo3() аналогичны функции Benchmark30fibo1().

Последняя часть файла benchmarkMe_test.go выглядит так:

```
func Benchmark50fibo1(b *testing.B) {
    benchmarkfibo1(b, 50)
}

func Benchmark50fibo2(b *testing.B) {
    benchmarkfibo2(b, 50)
}

func Benchmark50fibo3(b *testing.B) {
    benchmarkfibo3(b, 50)
}
```

Здесь мы видим еще три функции бенчмаркинга, которые вычисляют 50-е число в последовательности Фибоначчи.

Учтите, что каждый бенчмаркинг по умолчанию выполняется в течение не менее одной секунды. Если функция бенчмаркинга завершает работу меньше чем за одну секунду, то значение b.N увеличивается и функция запускается снова. В первый раз значение b.N равно 1, затем оно становится равным 2, 5, 10, 20, 50 и т. д. Так сделано потому, что чем быстрее функция, тем больше раз ее нужно запустить, чтобы получить точные результаты.

Выполнение benchmarkMe_test.go приведет к следующим результатам:

```
$ go test -bench=. benchmarkMe.go benchmarkMe_test.go
goos: darwin
goarch: amd64
Benchmark30fibo1-8    300     4494213 ns/op
Benchmark30fibo2-8    300     4463607 ns/op
Benchmark30fibo3-8    500000     2829 ns/op
Benchmark50fibo1-8    1          67272089954 ns/op
Benchmark50fibo2-8    1          67300080137 ns/op
Benchmark50fibo3-8    300000     4138 ns/op
PASS
ok    command-line-arguments   145.827s
```

Здесь два важных момента: во-первых, значение параметра -bench определяет, какие именно функции бенчмаркинга будут выполняться. Использованное значение . — это регулярное выражение, которое соответствует всем допустимым функциям бенчмаркинга. Во-вторых, если опустить параметр -bench, то ни одна функция бенчмаркинга не будет выполнена.

Итак, о чем говорят эти результаты? Прежде всего, -8 в конце каждой тестовой функции (Benchmark10fibo1-8) означает количество горутин, использованных во время ее выполнения, что, по сути, является значением переменной среды GOMAXPROCS. Напомню, что мы обсуждали переменную среды GOMAXPROCS еще в главе 10. Аналогично вы можете увидеть здесь значения GOOS и GOARCH, которые соответствуют операционной системе и архитектуре вашей машины.

Во втором столбце результатов показано, сколько раз была выполнена соответствующая функция. Более быстрые функции выполняются чаще, чем медленные. Например, функция Benchmark30fibo3() была выполнена 500 000 раз, а функция Benchmark50fibo2() — только один раз! В третьем столбце отображается среднее время каждого выполнения функции.

Как видим, функции fibo1() и fibo2() действительно медленнее, чем функция fibo3(). Если вы хотите включить в результаты бенчмаркинга статистику выделения памяти, то выполните следующую команду:

```
$ go test -benchmem -bench=. benchmarkMe.go benchmarkMe_test.go
goos: darwin
goarch: amd64
Benchmark30fibo1-8 300      4413791 ns/op      0 B/op      0 allocs/op
Benchmark30fibo2-8 300      4430097 ns/op      0 B/op      0 allocs/op
Benchmark30fibo3-8 500000   2774 ns/op         2236 B/op   6 allocs/op
Benchmark50fibo1-8 1        71534648696 ns/op  0 B/op      0 allocs/op
Benchmark50fibo2-8 1        72551120174 ns/op  0 B/op      0 allocs/op
Benchmark50fibo3-8 300000   4612 ns/op         2481 B/op  10 allocs/op
PASS
ok   command-line-arguments  150.500s
```

Этот результат аналогичен результату бенчмаркинга без параметра командной строки `-benchmem`, но здесь появились еще два столбца. В четвертом столбце показано среднее количество памяти, которое выделялось при каждом выполнении функции тестирования. В пятом — количество операций выделения памяти, применяемых для выделения количества памяти, указанного в четвертом столбце. Так, для каждого вызова функции `Benchmark50fibo3()` выделялся в среднем 2481 байт за десять операций выделения.

Как видим, функции `fibo1()` и `fibo2()` не нуждаются в каком-либо особом виде памяти, кроме ожидаемого, потому что ни одна из них не использует какую-либо структуру данных, — в отличие от функции `fibo3()`, в которой используется хеш-таблица. Отсюда и отличные от нуля значения в четвертом и пятом столбцах для `Benchmark10fibo3-8`.

Неправильно определенные функции бенчмаркинга

Рассмотрим код Go следующей тестовой функции:

```
func BenchmarkFiboI(b *testing.B) {
    for i := 0; i < b.N; i++ {
        _ = fibo1(i)
    }
}
```

Функция `BenchmarkFibo()` имеет правильное имя и сигнатуру. Однако, к сожалению, эта функция бенчмаркинга некорректна, и после выполнения команды `go test` вы не получите от нее никакого результата.

Причина в том, что из-за использования цикла `for`, при увеличении значения `b.N` описанным выше способом, время выполнения этой функции берчмаркинга также увеличивается. Это не позволяет функции `BenchmarkFiboI()` сойтись к какому-либо определенному числу, так что функция не завершается и, следовательно, не возвращает результат.

По аналогичной причине следующая функция бенчмаркинга также реализована неправильно:

```
func BenchmarkfiboII(b *testing.B) {
    for i := 0; i < b.N; i++ {
        _ = fibo2(b.N)
    }
}
```

А в реализации следующих двух функций бенчмаркинга нет ничего плохого:

```
func BenchmarkFiboIV(b *testing.B) {
    for i := 0; i < b.N; i++ {
```

```
        _ = fibo3(10)
    }
}

func BenchmarkFiboIII(b *testing.B) {
    _ = fibo3(b.N)
}
```

Бенчмаркинг буферизованной записи

В этом разделе вы узнаете, как размер буфера записи влияет на производительность всей операции записи. Для этого мы рассмотрим код Go из файла `writeBU.go`. Разделим его на пять частей.

Программа `writingBU.go` генерирует фиктивные файлы со случайно сгенерированными данными. Аргументами программы являются размер буфера и размер выходного файла.

Первая часть `writingBU.go` выглядит так:

```
package main

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

var BUFFERSIZE int
var FILESIZE int

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

Вторая часть файла `writingBU.go` содержит следующий код Go:

```
func createBuffer(buf *[]byte, count int) {
    *buf = make([]byte, count)
    if count == 0 {
        return
    }
    for i := 0; i < count; i++ {
        intByte := byte(random(0, 100))
        if len(*buf) > count {
            return
        }
        *buf = append(*buf, intByte)
    }
}
```

Третья часть `writingBU.go` содержит такой код Go:

```
func Create(dst string, b, f int) error {
    _, err := os.Stat(dst)
    if err == nil {
        return fmt.Errorf("File %s already exists.", dst)
    }
    destination, err := os.Create(dst)
    if err != nil {
        return err
    }
    defer destination.Close()
        if err != nil {
        panic(err)
    }
    buf := make([]byte, 0)
    for {
        createBuffer(&buf, b)
        buf = buf[:b]
        if _, err := destination.Write(buf); err != nil {
            return err
        }
        if f < 0 {
            break
        }
        f = f - len(buf)
    }
    return err
}
```

Основную работу в программе выполняет функция `Create()`, и именно для нее необходимо выполнить бенчмаркинг.

Обратите внимание, что если бы размер буфера и размер файла не были частью сигнатуры функции `Create()`, то у нас возникла бы проблема при написании функции бенчмаркинга для `Create()`. Это потому, что было бы необходимо использовать глобальные переменные `BUFFERSIZE` и `FILESIZE`, которые инициализируются в функции `main()` программы `writingBU.go`. Сделать это в файле `writingBU_test.go` затруднительно. Следовательно, нужно думать о бенчмаркинге функции еще при написании кода самой этой функции.

Четвертый фрагмент кода `writingBU.go` выглядит так:

```
func main() {
    if len(os.Args) != 3 {
        fmt.Println("Need BUFFERSIZE FILESIZE!")
        return
    }
    output := "/tmp/randomFile"
    BUFFERSIZE, _ = strconv.Atoi(os.Args[1])
    FILESIZE, _ = strconv.Atoi(os.Args[2])
```

```
    err := Create(output, BUFFERSIZE, FILESIZE)
    if err != nil {
        fmt.Println(err)
    }
```

Остальной код Go программы `writingBU.go` выглядит так:

```
    err = os.Remove(output)
    if err != nil {
        fmt.Println(err)
    }
}
```

Несмотря на то что вызов `os.Remove()`, который удаляет временный файл, находится внутри функции `main()`, которая не вызывается функциями бенчмаркинга, `os.Remove()` легко можно вызвать из функций бенчмаркинга, так что в этом нет проблемы.

Двойной запуск `writeBU.go` на компьютере с macOS Mojave и жестким диском SSD с использованием утилиты `time(1)` для проверки скорости выполнения программы привел к следующим результатам:

```
$ time  go run writingBU.go 1 100000
real    0m1.193s
user    0m0.349s
sys     0m0.809s
$ time  go run writingBU.go 10 100000
real    0m0.283s
user    0m0.195s
sys     0m0.228s
```

Итак, несмотря на очевидный факт, что размер буфера записи играет ключевую роль в производительности программы, мы должны быть более конкретными и точными. Поэтому мы напишем функции бенчмаркинга, которые сохраним в файле `writingBU_test.go`.

В первой части `writingBU_test.go` содержится следующий код Go:

```
package main

import (
    "fmt"
    "os"
    "testing"
)

var ERR error

func benchmarkCreate(b *testing.B, buffer, filesize int) {
    var err error
    for i := 0; i < b.N; i++ {
```

```
        err = Create("/tmp/random", buffer, filesize)
    }
    ERR = err
    err = os.Remove("/tmp/random")
    if err != nil {
        fmt.Println(err)
    }
}
```

Как вы, вероятно, помните, такая функция бенчмаркинга не является допустимой.

Второй фрагмент кода `writeBU_test.go` выглядит так:

```
func Benchmark1Create(b *testing.B) {
    benchmarkCreate(b, 1, 1000000)
}

func Benchmark2Create(b *testing.B) {
    benchmarkCreate(b, 2, 1000000)
}
```

Остальной код `writingBU_test.go` следующий:

```
func Benchmark4Create(b *testing.B) {
    benchmarkCreate(b, 4, 1000000)
}

func Benchmark10Create(b *testing.B) {
    benchmarkCreate(b, 10, 1000000)
}

func Benchmark1000Create(b *testing.B) {
    benchmarkCreate(b, 1000, 1000000)
}
```

Здесь мы создали пять тестовых функций, которые будут проверять производительность функции `benchmarkCreate()`. Она, в свою очередь, проверяет производительность `Create()` для разного размера буфера записи.

Выполнение команды `go test` для файлов `writingBU.go` и `writingBU_test.go` приведет к результатам следующего вида:

```
$ go test -bench=. writingBU.go writingBU_test.go
goos: darwin
goarch: amd64
Benchmark1Create-8          1   6001864841 ns/op
Benchmark2Create-8          1   3063250578 ns/op
Benchmark4Create-8          1   1557464132 ns/op
Benchmark10Create-8    100000        11136 ns/op
Benchmark1000Create-8  200000         5532 ns/op
PASS
ok    command-line-arguments  21.847s
```

Следующая команда проверяет также выделение памяти для функций бенчмаркинга:

```
$ go test -bench=. writingBU.go writingBU_test.go -benchmem
goos: darwin
goarch: amd64
Benchmark1Create-8 1    6209493161 ns/op  16000840 B/op  2000017 allocs/op
Benchmark2Create-8 1    3177139645 ns/op   8000584 B/op  1000013 allocs/op
Benchmark4Create 1      1632772604 ns/op   4000424 B/op   500011 allocs/op
Benchmark10Create-8 100000   11238 ns/op       336 B/op        7 allocs/op
Benchmark1000Create-8 200000  5122 ns/op       303 B/op        5 allocs/op
PASS
ok   command-line-arguments  24.031s
```

Теперь пора интерпретировать результаты выполнения двух команд бенчмаркинга.

Очевидно, что использование буфера записи размером 1 байт неэффективно и только замедляет работу. Кроме того, такой размер буфера требует слишком много операций с памятью, что еще больше замедляет программу.

Использование буфера записи размером 2 байта ускоряет программу в два раза, и это хорошо. Однако это еще очень медленно. То же самое относится и к буферу записи размером 4 байта. Все сильно ускоряется, когда мы увеличиваем буфер записи до 10 байт. Наконец, результаты бенчмаркинга показывают, что использование буфера записи размером 1000 байт не ускоряет работу в 100 раз по сравнению с использованием буфера размером 10 байт. Следовательно, золотая середина между скоростью и размером буфера записи находится где-то между этими двумя значениями размера буфера.

Обнаружение недоступного кода Go

Код Go, который не может быть выполнен, является логической ошибкой, и поэтому разработчикам или компилятору Go в обычном режиме его довольно сложно найти. Проще говоря, в недоступном коде нет ничего плохого, кроме того, что этот код не может быть выполнен.

Рассмотрим следующий код Go, который хранится в файле `cannotReach.go`:

```
package main

import (
    "fmt"
)
```

```
func f1() int {
    fmt.Println("Entering f1()")
    return -10
    fmt.Println("Exiting f1()")
    return -1
}

func f2() int {
    if true {
        return 10
    }
    fmt.Println("Exiting f2()")
    return 0
}

func main() {
    fmt.Println(f1())
    fmt.Println("Exiting program...")
}
```

В коде Go программы `cannotReach.go` нет ничего синтаксически некорректного. Мы можем выполнить `cannotReach.go`, не получив от компилятора вывода об ошибках:

```
$ go run cannotReach.go
Entering f1()
-1
Exiting program...
```

Однако обратите внимание, что функция `f2()` в программе никогда не используется. Очевидно, что следующий код Go функции `f2()` никогда не выполняется, потому что условие в предыдущем операторе `if` всегда истинно:

```
fmt.Println("Exiting f2()")
return 0
```

Что же можно сделать? Мы можем выполнить `go vet`:

```
$ go vet cannotReach.go
# command-line-arguments
./cannotReach.go:10:2: unreachable code
```

Результат говорит о том, что в строке 10 программы есть недоступный код. Теперь удалим из функции `f1()` инструкцию `return -10` и снова запустим `go vet`:

```
$ go vet cannotReach.go
```

Теперь сообщений об ошибках нет, несмотря на то что в функции `f2()` все еще остается недоступный код. Это означает, что `go vet` не фиксирует все возможные типы логических ошибок.

Кросс-компиляция

Кросс-компиляция — это процесс создания двоичного исполняемого файла для архитектуры, отличной от той, на которой вы работаете.

Главное преимущество, которое дает кросс-компиляция, заключается в том, что нам не требуется второй или третий компьютер, чтобы создавать исполняемые файлы для разных архитектур. В целом достаточно всего одной машины для разработки. К счастью, в Go есть встроенная поддержка кросс-компиляции.

В этом разделе для иллюстрации процесса кросс-компиляции мы рассмотрим пример кода Go, который находится в файле `xCompile.go`. Этот код выглядит так:

```
package main

import (
    "fmt"
    "runtime"
)

func main() {
    fmt.Print("You are using ", runtime.Compiler, " ")
    fmt.Println("on a", runtime.GOARCH, "machine")
    fmt.Println("with Go version", runtime.Version())
}
```

Выполнение `xCompile.go` на компьютере с macOS Mojave приведет к следующим результатам:

```
$ go run xCompile.go
You are using gc on a amd64 machine
with Go version go1.12.4
```

Для кросс-компиляции исходного файла Go необходимо присвоить переменным среды `GOOS` и `GOARCH` значения, соответствующие той операционной системе и архитектуре, для которых компилируется программа. Это не так сложно, как кажется.

Итак, процесс кросс-компиляции выглядит так:

```
$ env GOOS=linux GOARCH=arm go build xCompile.go
$ file xCompile
xCompile: ELF 32-bit LSB executable, ARM, EABI5 version 1 (SYSV),
statically linked, not stripped
$ ./xCompile
-bash: ./xCompile: cannot execute binary file
```

Первая команда генерирует двоичный файл, который работает на машинах с Linux и архитектурой ARM. Результат выполнения команды `file(1)` подтверждает, что сгенерированный двоичный файл действительно предназначен для другой архитектуры.

Поскольку компьютер с Debian Linux, который используется в этом примере, имеет процессор Intel, нам придется еще раз выполнить команду go build со значением GOARCH:

```
$ env GOOS=linux GOARCH=386 go build xCompile.go
$ file xCompile
xCompile: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV),
statically linked, with debug_info, not stripped
```

Выполнение сгенерированного двоичного исполняемого файла на компьютере с Linux даст следующий ожидаемый результат:

```
$ ./xCompile
You are using gc on a 386 machine
with Go version go1.12.4
$ go version
go version go1.3.3 linux/amd64
$ go run xCompile.go
You are using gc on a amd64 machine
with Go version go1.3.3
```

Следует отметить, что двоичный файл, полученный в результате кросс-компиляции xCompile.go, выводит версию Go, примененную для его компиляции. Второе, на что следует обратить внимание, — архитектура машины с Linux — это на самом деле amd64, а не 386, которая использовалась при кросс-компиляции.

Список доступных значений переменных среды GOOS и GOARCH размещен по адресу https://golang.org/doc/install/source. Однако имейте в виду, что не все пары GOOS и GOARCH актуальны.

Создание примеров функций

Частью процесса документирования является создание примеров кода, которые демонстрируют использование некоторых или всех функций и типов, существующих в пакете. Примеры функций имеют много преимуществ, в том числе они являются готовыми тестами, которые можно выполнить с помощью команды go test. Таким образом, если функция примера содержит строку // Output:, то инструмент go test проверит, соответствует ли вычисленный результат значениям, найденным после строки // Output:.

Кроме того, примеры, размещенные в документации к пакету, действительно полезны. Подробнее документация рассмотрена в следующем разделе. Наконец, функции примеров, представленные на сервере документации Go (https://golang.org/pkg/io/#example_Copy), позволяют экспериментировать с примерами кода. Интерактивная среда Go на сайте https://play.golang.org/ также поддерживает эту функцию.

Поскольку за примеры программы отвечает подкоманда `go test`, необходимо импортировать стандартный Go-пакет `testing` и включить примеры функций в файлы Go, которые заканчиваются на `_test.go`. Кроме того, имя каждой функции-примера должно начинаться со слова `Example`. Наконец, функции примеров не принимают входных аргументов и не возвращают результатов.

Теперь создадим несколько функций примеров для следующего пакета, который сохраним в файле `ex.go`:

```
package ex

func F1(n int) int {
    if n == 0 {
        return 0
    }
    if n == 1 || n == 2 {
        return 1
    }
    return F1(n-1) + F1(n-2)
}

func S1(s string) int {
    return len(s)
}
```

Исходный файл `ex.go` содержит реализацию двух функций с именами `F1()` и `S1()`.

Обратите внимание, что в программу `ex.go` не нужно импортировать пакет `fmt`.

Как вы уже знаете, функции примеров должны быть включены в файл `ex_test.go`. Разделим его на три части.

Первая часть `ex_test.go` выглядит так:

```
package ex

import (
    "fmt"
)
```

Вторая часть кода `ex_test.go` содержит следующий код Go:

```
func ExampleF1() {
    fmt.Println(F1(10))
    fmt.Println(F1(2))
    // Output:
    // 55
    // 1
}
```

Остальной код Go файла `ex_test.go`:

```
func ExampleS1() {
    fmt.Println(S1("123456789"))
    fmt.Println(S1(""))
    // Output:
    // 8
    // 0
}
```

Выполнение команды `go test` для пакета `ex.go` приведет к результатам следующего вида:

```
$ go test ex.go ex_test.go -v
=== RUN ExampleF1
--- PASS: ExampleF1 (0.00s)
=== RUN ExampleS1
--- FAIL: ExampleS1 (0.00s)
got:
9
0
want:
8
0
FAIL
FAIL   command-line-arguments   0.006s
```

Как видим, судя по данным, которые выводятся после комментария `// Output:`, здесь что-то не так с функцией `S1()`.

От кода Go до машинного кода

В этом разделе подробно показано, как код Go преобразуется в машинный код. В качестве примера мы рассмотрим программу Go, которая называется `machineCode.go`. Разделим ее на две части.

Первая часть `machineCode.go` выглядит так:

```
package main

import (
    "fmt"
)

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

Вторая и последняя часть machineCode.go:

```
func main() {
    hello()
}
```

Теперь посмотрим, как выглядит программа machineCode.go, переведенная в машинный код:

```
$ GOSSAFUNC=main GOOS=linux GOARCH=amd64 go build -gcflags "-S"
machineCode.go
# runtime
dumped SSA to /usr/local/Cellar/go/1.12.4/libexec/src/runtime/ssa.html
# command-line-arguments
dumped SSA to ./ssa.html
os.(*File).close STEXT dupok nosplit size=26 args=0x18 locals=0x0
    0x0000 00000 (<autogenerated>:1)  TEXT os.(*File).close(SB),
DUPOK|NOSPLIT|ABIInternal, $0-24
    0x0000 00000 (<autogenerated>:1)  FUNCDATA $0, gclocals
e6397a44f8e1b6e77d0f200b4fba5269(SB)
    0x0000 00000 (<autogenerated>:1)  FUNCDATA $1, gclocals
69c1753bd5f81501d95132d08af04464(SB)
    0x0000 00000 (<autogenerated>:1)  FUNCDATA $3, gclocals
9fb7f0986f647f17cb5
...
```

Судя по первым строкам, есть два файла, которые содержат все сгенерированные выходные данные: /usr/local/Cellar/go/1.12.4/libexec/src/runtime/ssa.html и ./ssa.html. Стоит отметить, что если у вас установлена другая версия Go или другой комплект установки Go, то первый файл будет расположен в другом месте. Форма *статического одиночного присваивания* (Static Single Assignment, SSA) — это метод описания низкоуровневых операций, который очень близок к машинным инструкциям. Однако заметьте, что SSA, в отличие от машинного кода, действует так, как будто у него бесконечное количество регистров.

Открыв этот файл в любом браузере, вы увидите много полезной, но низкоуровневой информации о программе. Наконец, обратите внимание, что значением параметра GOSSAFUNC является функция Go, которую вы хотите *дизассемблировать*.

К сожалению, более подробное обсуждение SSA выходит за рамки данной книги.

Использование ассемблера в Go

В этом подразделе речь пойдет об *ассемблере* и Go, а также о том, как использовать ассемблер для реализации функций Go. Для начала мы рассмотрим следующую программу Go, которая хранится в файле add_me.go:

```
package main

import (
```

```
    "fmt"
)

func add(x, y int64) int64 {
    return x + y
}

func main() {
    fmt.Println(add(1, 2))
}
```

Выполнив следующую команду, мы увидим ассемблерную реализацию функции add():

```
$ GOOS=darwin GOARCH=amd64 go tool compile -S add_me.go
"".add STEXT nosplit size=19 args=0x18 locals=0x0
    0x0000 00000 (add_me.go:7)  TEXT      "".add(SB), NOSPLIT|ABIInternal, $0-24
    0x0000 00000 (add_me.go:7)  FUNCDATA      $0, gclocals
                                              33cdeccccebe80329f1fdbee7f5874cb(SB)
    0x0000 00000 (add_me.go:7)  FUNCDATA      $1, gclocals
                                              33cdeccccebe80329f1fdbee7f5874cb(SB)
    0x0000 00000 (add_me.go:7)  FUNCDATA      $3, gclocals
                                              33cdeccccebe80329f1fdbee7f5874cb(SB)
    0x0000 00000 (add_me.go:8)  PCDATA        $2, $0
    0x0000 00000 (add_me.go:8)  PCDATA        $0, $0
    0x0000 00000 (add_me.go:8)  MOVQ          "".y+16(SP), AX
    0x0005 00005 (add_me.go:8)  MOVQ          "".x+8(SP), CX
    0x000a 00010 (add_me.go:8)  ADDQ          CX, AX
    0x000d 00013 (add_me.go:8)  MOVQ          AX, "".~r2+24(SP)
    0x0012 00018 (add_me.go:8)  RET
    0x0000 48 8b 44 24 10 48 8b 4c 24 08 48 01 c8 48 89 44
                                              H.D$.H.L$.H..H.D
    0x0010 24 18 c3             $..
    "".main STEXT size=150 args=0x0 locals=0x58
```

Последняя строка не является частью ассемблерной реализации функции add(). Кроме того, строки FUNCDATA также не имеют ничего общего с ассемблерной реализацией функции, они добавляются компилятором Go.

Теперь внесем некоторые изменения в представленный выше ассемблерный код, чтобы он выглядел следующим образом:

```
TEXT  add(SB),$0
    MOVQ x+0(FP), BX
    MOVQ y+8(FP), BP
    ADDQ BP, BX
    MOVQ BX, ret+16(FP)
    RET
```

Этот ассемблерный код мы сохраним в файле add_amd64.s, чтобы использовать в качестве реализации функции add().

Новая версия `add_me.go` будет выглядеть так:

```
package main

import (
    "fmt"
)

func add(x, y int64) int64

func main() {
    fmt.Println(add(1, 2))
}
```

Это означает, что `add_me.go` будет использовать ассемблерную реализацию функции `add()`. Применять ассемблерную реализацию функции `add()` очень просто:

```
$ go build
$ ls -l
total 4136
-rw-r--r--@   1 mtsouk  staff       93 Apr 18 22:49 add_amd64.s
-rw-r--r--@   1 mtsouk  staff      101 Apr 18 22:59 add_me.go
-rwxr-xr-x    1 mtsouk  staff  2108072 Apr 18 23:00 asm
$ ./asm
3
$ file asm
asm: Mach-O    64-bit executable x86_64
```

Единственная сложность состоит в том, что ассемблерный код не является переносимым. К сожалению, более подробное обсуждение использования ассемблера в Go выходит за рамки данной книги — советую изучить ассемблер и архитектуру вашего процессора.

Генерация документации

Go предоставляет инструмент `godoc`, который позволяет просматривать документацию пакетов — при условии, что в файлы пакетов включена дополнительная информация.

Общая рекомендация: следует стремиться документировать все, за исключением очевидного. Другими словами, не говорите: «Здесь я создаю новую переменную типа int». Лучше сообщите о назначении этой переменной! Однако по-настоящему хороший код обычно не нуждается в документации.

Правило написания документации в Go довольно простое и понятное: чтобы задокументировать какой-либо объект, нужно поместить непосредственно перед

его объявлением одну или несколько обычных строк комментариев, которые начинаются с символов //. Это соглашение применяется для документирования функций, переменных, констант и даже пакетов.

Кроме того, вы скоро заметите, что первая строка документации пакета, независимо от его размера, появляется в списке пакетов godoc, как это сделано на сайте https://golang.org/pkg/. Поэтому описание должно быть довольно емким и полным.

Помните, что комментарии, начинающиеся со слова BUG (что-то там), появятся в разделе Bugs документации пакета, даже если они не предшествуют объявлению объекта.

Если вам нужен пример, посмотрите на исходный код и страницу документации пакета bytes, которые вы найдете по адресам https://golang.org/src/bytes/bytes.go и https://golang.org/pkg/bytes/ соответственно. Наконец, все комментарии, не связанные с объявлением верхнего уровня, в выводе, генерируемом утилитой godoc, опускаются.

Рассмотрим следующий код Go, который хранится в файле documentMe.go:

```
// This package is for showcasing the documentation capabilities of Go
// It is a naive package!
package documentMe
// Pie is a global variable
// This is a silly comment!
const Pie = 3.1415912
// The S1() function finds the length of a string
// It iterates over the string using range
func S1(s string) int {
    if s == "" {
        return 0
    }
    n := 0
    for range s {
        n++
    }
    return n
}
// The F1() function returns the double value of its input integer
// A better function name would have been Double()!
func F1(n int) int {
    return 2 * n
}
```

Как сказано в предыдущем разделе, для создания примеров к этим функциям нам нужно создать файл documentMe_test.go. Содержимое documentMe_test.go выглядит так:

```
package documentMe

import (
```

```
    "fmt"
)

func ExampleS1() {
    fmt.Println(S1("123456789"))
    fmt.Println(S1(""))
    // Output:
    // 9
    // 0
}

func ExampleF1() {
    fmt.Println(F1(10))
    fmt.Println(F1(2))
    // Output:
    // 1
    // 55
}
```

Чтобы просмотреть документацию к файлу documentMe.go, как говорилось в главе 6, нам необходимо установить этот пакет на свой компьютер. Для этого нужно выполнить следующие команды из оболочки UNIX:

```
$ mkdir ~/go/src/documentMe
$ cp documentMe* ~/go/src/documentMe/
$ ls -l ~/go/src/documentMe/
total 16
-rw-r--r--@ 1 mtsouk staff 542 Mar 6 21:11 documentMe.go
-rw-r--r--@ 1 mtsouk staff 223 Mar 6 21:11 documentMe_test.go
$ go install documentMe
$ cd ~/go/pkg/darwin_amd64
$ ls -l documentMe.a
-rw-r--r-- 1 mtsouk staff 1626 Mar 6 21:11 documentMe.a
```

А затем следует выполнить утилиту godoc:

```
$ godoc -http=":8080"
```

Обратите внимание, что если этот порт уже используется и пользователь имеет права root, то вы получите следующее сообщение об ошибке:

```
$ godoc -http=":22"
2019/08/19 15:18:21 ListenAndServe :22: listen tcp :22: bind: address already in use
```

Однако если у пользователя нет привилегий root, то сообщение об ошибке будет следующим (даже если данный порт уже используется):

```
$ godoc -http=":22"
2019/03/06 21:03:05 ListenAndServe :22: listen tcp :22: bind: permission denied
```

Когда все будет готово, вы сможете просматривать созданную HTML-документацию в любом браузере. Для этого нужно открыть URL http://localhost:8080/pkg/.

На рис. 11.6 показан корневой каталог сервера `godoc`, о котором мы только что говорили. Здесь, наряду с другими пакетами Go, присутствует пакет `documentMe`, который мы создали в файле `documentMe.go`.

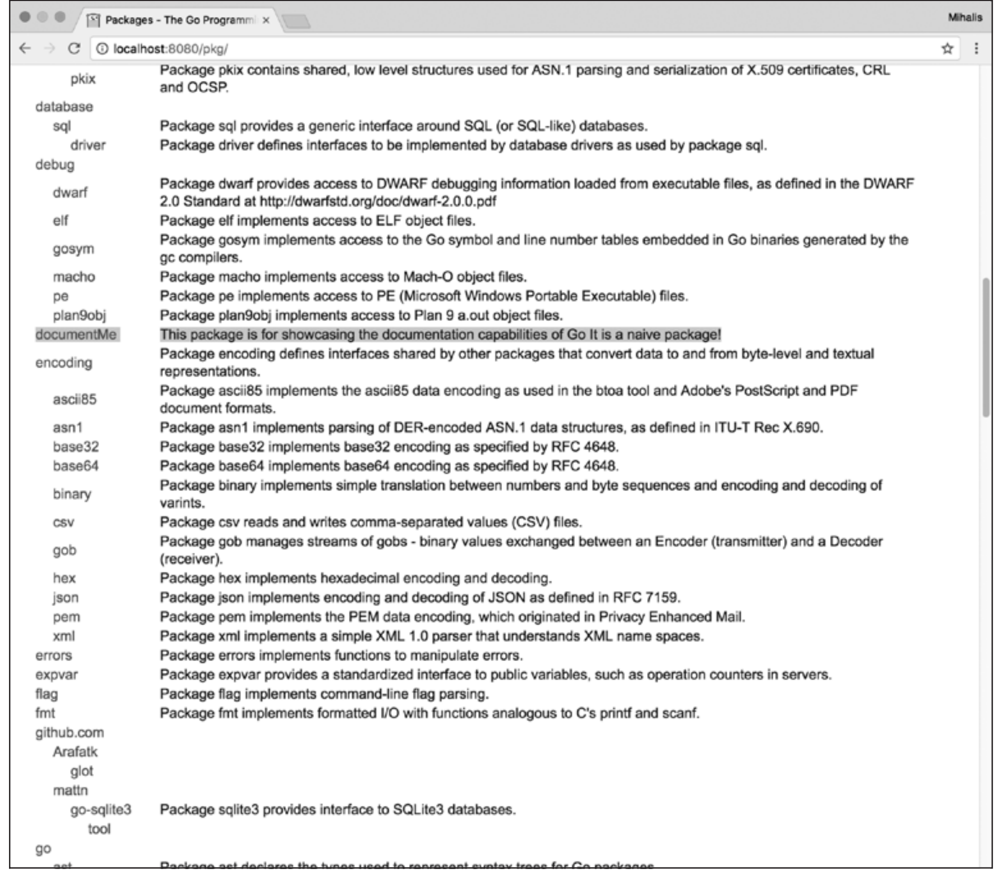


Рис. 11.6. Корневой каталог сервера godoc

На рис. 11.7 показан корневой каталог документации пакета `documentMe`, реализованного в исходном файле `documentMe.go`.

Аналогично на рис. 11.8 более подробно показана документация функции `S1()` пакета `documentMe.go`, которая также включает в себя пример кода.

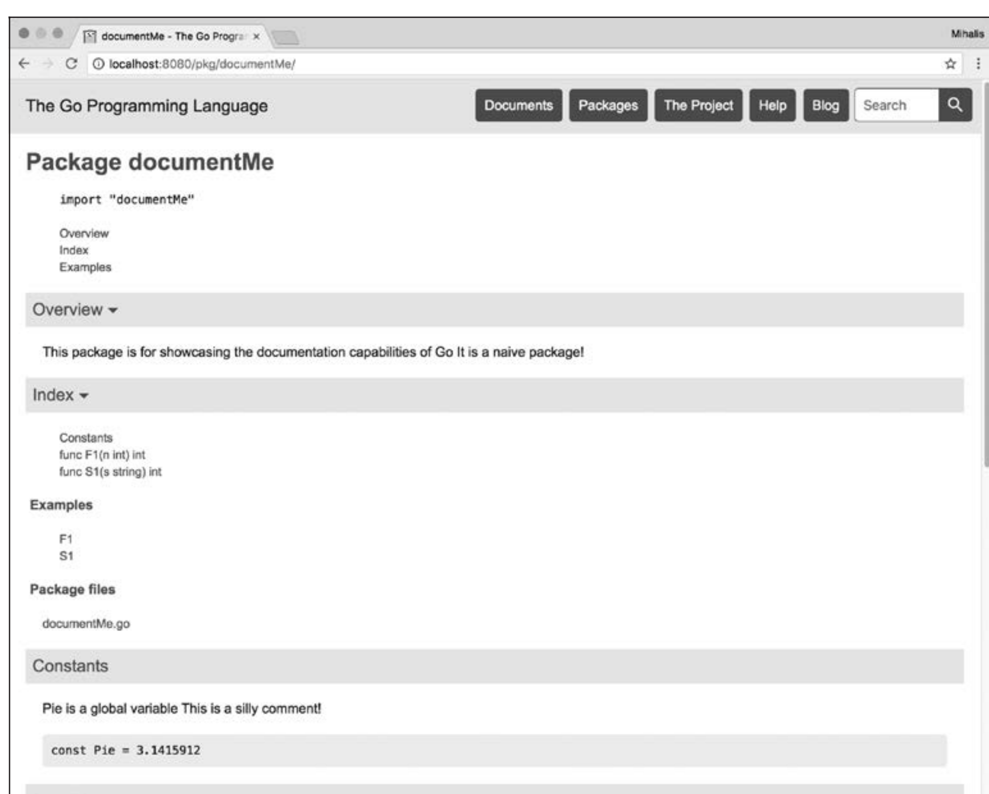


Рис. 11.7. Начальная страница файла documentMe.go

Выполнение команды `go test` приведет к следующим результатам, которые позволяют выявить потенциальные проблемы и ошибки в коде:

```
$ go test -v documentMe*
=== RUN ExampleS1
--- PASS: ExampleS1 (0.00s)
=== RUN ExampleF1
--- FAIL: ExampleF1 (0.00s)
got:
20
4
want:
1
55
FAIL
FAIL    command-line-arguments  0.005s
```

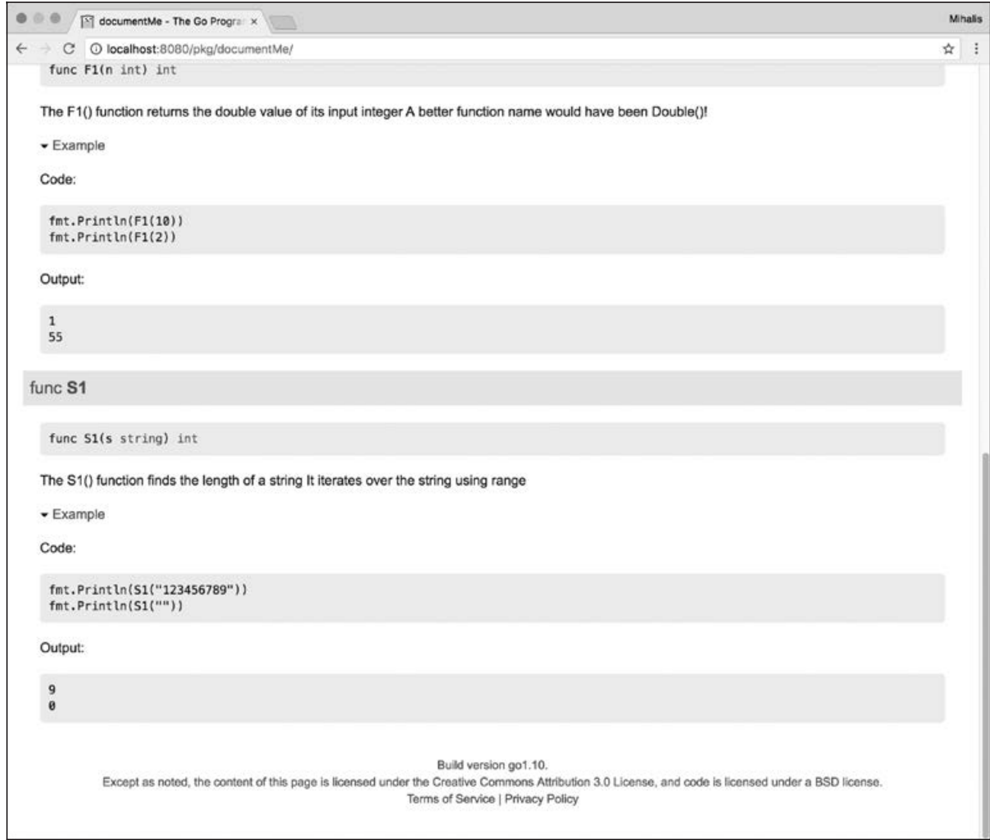


Рис. 11.8. Страница документации и пример функции S1()

Использование образов Docker

В этом разделе вы узнаете, как создать образ Docker, содержащий базу данных PostgreSQL и код Go. Этот образ не будет идеальным, но на его примере вы увидите, как это делается. Как вы уже знаете, все должно начинаться с `Dockerfile`, который в данном случае будет следующим:

```
FROM ubuntu:18.04

RUN apt-get update && apt-get install -y gnupg
RUN apt-key adv --keyserver hkp://p80.pool.sks-keyservers.net:80 --recv-
keys B97B0AFCAA1A47F044F244A07FCC7D46ACCC4CF8
RUN echo "deb http://apt.postgresql.org/pub/repos/apt/ precise-pgdg main" >
/etc/apt/sources.list.d/pgdg.list
```

```
RUN apt-get update && apt-get install -y software-properties-common
postgresql-9.3 postgresql-client-9.3 postgresql-contrib-9.3
RUN apt-get update && apt-get install -y git golang vim

USER postgres
RUN /etc/init.d/postgresql start &&\
    psql --command "CREATE USER docker WITH SUPERUSER PASSWORD 'docker';" &&\
    createdb -O docker docker

RUN echo "host all  all   0.0.0.0/0  md5" >>
/etc/postgresql/9.3/main/pg_hba.conf
RUN echo "listen_addresses='*'" >> /etc/postgresql/9.3/main/postgresql.conf

USER root
RUN mkdir files
COPY webServer.go files
WORKDIR files
RUN go get github.com/lib/pq
RUN go build webServer.go
RUN ls -l
USER postgres
CMD ["/usr/lib/postgresql/9.3/bin/postgres", "-D",
"/var/lib/postgresql/9.3/main", "-c",
"config_file=/etc/postgresql/9.3/main/postgresql.conf"]
```

Это довольно сложный `Dockerfile`. Он начинается с базового образа Docker из Docker Hub и загружает дополнительное программное обеспечение. Этого более или менее достаточно, чтобы создать настраиваемый образ Docker, который соответствует вашим потребностям.

Выполнение `Dockerfile` сгенерирует много выходных данных, включая следующие:

```
$ docker build -t go_postgres .
Sending build context to Docker daemon 9.216kB
Step 1/19 : FROM ubuntu:18.04
    ---> 94e814e2efa8
Step 2/19 : RUN apt-get update && apt-get install -y gnupg
...
Step 15/19 : RUN go get github.com/lib/pq
    ---> Running in 17aede1c97d8
Removing intermediate container 17aede1c97d8
    ---> 1878408c6e06
Step 16/19 : RUN go build webServer.go
    ---> Running in 39cfe8af63d5
Removing intermediate container 39cfe8af63d5
    ---> 1b4d638242ae
...
```

```
Removing intermediate container 9af6a391c1e4
    ---> c24937079367
Successfully built c24937079367
Successfully tagged go_postgres:latest
```

В последней части `Dockerfile` мы также соберем исполняемый файл из исходной программы `webServer.go`, используя операционную систему образа Docker, в данном случае Ubuntu Linux.

Выполнение команды `docker images` приведет к результатам следующего вида:

```
$ docker images
REPOSITORY     TAG       IMAGE ID         CREATED           SIZE
go_postgres    latest    c24937079367     34 seconds ago    831MB
```

Теперь мы можем выполнить этот образ Docker и подключиться к его оболочке `bash(1)`:

```
$ docker run --name=my_go -dt go_postgres:latest
23de1ab0d3f5517d5dbf8c599a68075574a8ed9217aa3cb4899ea2f92412a833
$ docker exec -it my_go bash
postgres@23de1ab0d3f5:/files$
```

Дополнительные ресурсы

Советую посетить следующие веб-ссылки:

- посетите веб-сайт Graphviz, расположенный по адресу http://graphviz.org;
- посетите страницу документации пакета `testing`, расположенную по адресу https://golang.org/pkg/testing/;
- если вы не знаете о Дональде Кнуте (Donald Knuth) и его работах, советую узнать о нем больше на странице https://en.wikipedia.org/wiki/Donald_Knuth;
- чтобы ближе познакомиться с утилитой `godoc`, посетите ее страницу документации https://godoc.org/golang.org/x/tools/cmd/godoc;
- посетите страницу документации стандартного Go-пакета `runtime/pprof` по адресу https://golang.org/pkg/runtime/pprof/;
- просмотрите код Go пакета `net/http/pprof`, посетив его страницу https://golang.org/src/net/http/pprof/pprof.go;
- страницу документации пакета `net/http/pprof` вы найдете по адресу https://golang.org/pkg/net/http/pprof/;
- чтобы больше узнать об инструменте `pprof`, посетите страницу разработки https://github.com/google/pprof;

- просмотрите видеоролик *Advanced Testing with Go*, размещенный на GopherCon 2017 Митчеллом Хашимото (Mitchell Hashimoto): https://www.youtube.com/watch?v=8hQG7QlcLBk;
- почитайте исходный код пакета `testing` по адресу https://golang.org/src/testing/testing.go;
- дополнительную информацию о пакете `testing/quick` вы найдете на странице https://golang.org/pkg/testing/quick/;
- чтобы больше узнать о пакете `profile`, посетите веб-страницу https://github.com/pkg/profile;
- *Manual for the Plan 9 assembler*: https://9p.io/sys/doc/asm.html;
- страницу документации пакета `arm64` вы найдете по адресу https://golang.org/pkg/cmd/internal/obj/arm64/;
- чтобы больше узнать о Go и SSA, посетите страницу https://golang.org/src/cmd/compile/internal/ssa/;
- библиография SSA: http://www.dcs.gla.ac.uk/~jsinger/ssa.html;
- чтобы ближе познакомиться с инструментом `go fix`, посетите его веб-страницу https://golang.org/cmd/fix/.

Упражнения

- Напишите тестовые функции для программы `byWord.go`, которую мы разработали в главе 8.
- Напишите функции бенчмаркинга для программы `readSize.go`, которую мы разработали в главе 8.
- Попробуйте исправить проблемы в Go-программах `documentMe.go` и `documentMe_test.go`.
- Используя текстовый интерфейс утилиты `go tool pprof`, просмотрите файл `memoryProfile.out`, созданный программой `profileMe.go`.
- Измените `webServer.go` и `webServer_test.go` так, чтобы они позволяли работать с другими базами данных, такими как MySQL и SQLite3.
- Измените файл `Dockerfile` из последнего раздела таким образом, чтобы включить в него `webServer_test.go`.
- Измените `Dockerfile` таким образом, чтобы можно было выполнить `go test -v` для веб-сервера. Изменения должны позволять корректно создавать пользователей и базы данных PostgreSQL.
- Используя веб-интерфейс утилиты `go tool pprof`, проверьте файл `memoryProfile.out`, созданный с помощью `profileMe.go`.

Резюме

В этой главе мы рассмотрели тестирование, оптимизацию и профилирование кода. В конце главы вы узнали, как найти в программе недоступный код и как кросс-компилировать код Go, как использовать команду `go test` для тестирования и бенчмаркинга кода Go, а также для составления дополнительной документации с использованием примеров функций.

Несмотря на то что здесь представлено далеко не полное описание профилировщика Go и команды `go tool trace`, следует понимать, что при изучении таких тем, как профилирование и трассировка кода, ничто не заменит самостоятельные эксперименты и испытания новых методов!

В следующей главе рассмотрено сетевое программирование на Go, то есть программирование приложений, работающих в компьютерных сетях TCP/IP, в том числе в Интернете. Следующая глава посвящена таким темам, как пакет `net/http`, создание веб-клиентов и веб-серверов на Go, структуры `http.Response` и `http.Request`, профилирование HTTP-серверов, gRPC и превышение времени ожидания сетевых подключений.

Кроме того, в главе 12 рассмотрены протоколы *IPv4* и *IPv6*, а также инструменты Wireshark и tshark, которые применяются для перехвата и анализа сетевого трафика.

12 Основы сетевого программирования на Go

В предыдущей главе речь шла о бенчмаркинге кода Go с использованием функций бенчмаркинга, о тестировании в Go, функциях примеров, покрытии кода, кросс-компиляции и профилировании кода Go, а также о создании документации в Go и образов Docker, содержащих требуемое программное обеспечение.

Эта глава посвящена сетевому и веб-программированию. Вы узнаете, как создавать веб-приложения, работающие в компьютерных сетях и Интернете. Однако, чтобы узнать, как разрабатывать приложения для TCP и UDP, вам придется подождать до главы 13.

Обратите внимание, что для того, чтобы успешно усвоить содержание этой и следующей главы, вам понадобятся некие базовые знания о протоколе HTTP, работе с сетями и функционировании компьютерных сетей.

В этой главе рассмотрены следующие темы:

- что такое протокол TCP/IP и почему он так важен;
- протоколы IPv4 и IPv6;
- утилита командной строки `netcat`;
- *DNS*-поиск в Go;
- пакет `net/http`;
- структуры `http.Response`, `http.Request` и `http.Transport`;
- создание веб-серверов на Go;
- программирование веб-клиентов на Go;
- создание веб-сайтов на Go;
- *gRPC* и Go;
- тип `http.NewServeMux`;
- *Wireshark* и *tshark*;
- разрыв HTTP-соединений, ожидание которых занимает слишком много времени, на стороне сервера и на стороне клиента.

Что такое net/http, net и http.RoundTripper

Центральная тема этой главы — пакет `net/http`, который предоставляет функции, позволяющие разрабатывать мощные веб-серверы и веб-клиенты. Методы `http.Set()` и `http.Get()` применяются для выполнения запросов HTTP и HTTPS, а функция `http.ListenAndServe()` — для создания веб-серверов, передавая ей IP-адрес и порт TCP, который будет прослушивать сервер. Также в состав этого пакета входят функции, которые обрабатывают входящие запросы.

Кроме пакета `net/http`, в некоторых программах, представленных в этой главе, мы будем использовать пакет `net`. Однако функциональность `net` более подробно раскрыта в главе 13.

Наконец, вам будет полезно узнать, что `http.RoundTripper` — это *интерфейс*, который позволяет создавать элементы Go, способные выполнять HTTP-транзакции. Проще говоря, это означает, что такие элементы Go способны получать отклики `http.Response` для заданных запросов `http.Request`. О том, что такое `http.Response` и `http.Request`, вы скоро узнаете.

Тип http.Response

Определение структуры `http.Response`, которое находится в файле https://golang.org/src/net/http/response.go, выглядит так:

```
type Response struct {
    Status      string    // e.g. "200 OK"
    StatusCode  int       // e.g. 200
    Proto       string    // e.g. "HTTP/1.0"
    ProtoMajor  int       // e.g. 1
    ProtoMinor  int       // e.g. 0
    Header Header
    Body io.ReadCloser
    ContentLength int64
    TransferEncoding []string
    Close bool
    Uncompressed bool
    Trailer Header
    Request *Request
    TLS *tls.ConnectionState
}
```

Цель этого довольно сложного типа `http.Response` — ответ на HTTP-запрос. В исходном файле https://golang.org/src/net/http/response.go содержится подробное описание назначения каждого из полей этой структуры, как для большинства структурных типов, входящих в состав стандартной библиотеки Go.

Тип http.Request

Назначение типа `http.Request` — представление для HTTP-запроса, уже полученного сервером или готового к отправке от HTTP-клиента на сервер.

Структурный тип `http.Request`, определенный в файле https://golang.org/src/net/http/request.go, выглядит так:

```
type Request struct {
    Method string
    URL *url.URL
    Proto      string    // "HTTP/1.0"
    ProtoMajor int       // 1
    ProtoMinor int       // 0
    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
    ctx context.Context
}
```

Тип http.Transport

Определение структуры `http.Transport`, которое находится по адресу `https://golang.org/src/net/http/transport.go`, выглядит так:

```
type Transport struct {
    idleMu      sync.Mutex
    wantIdle    bool
    idleConn    map[connectMethodKey][]*persistConn
    idleConnCh  map[connectMethodKey]chan *persistConn
    idleLRU     connLRU
    reqMu       sync.Mutex
    reqCanceler map[*Request]func(error)
    altMu       sync.Mutex
    altProto    atomic.Value
    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)
    DialTLS func(network, addr string) (net.Conn, error)
    TLSClientConfig *tls.Config
    TLSHandshakeTimeout time.Duration
    DisableKeepAlives bool
    DisableCompression bool
    MaxIdleConns int
    MaxIdleConnsPerHost int
    IdleConnTimeout time.Duration
    ResponseHeaderTimeout time.Duration
    ExpectContinueTimeout time.Duration
    TLSNextProto map[string]func(authority string, c *tls.Conn) RoundTripper
    ProxyConnectHeader Header
    MaxResponseHeaderBytes int64
    nextProtoOnce sync.Once
    h2transport    *http2Transport
}
```

Как видим, структура `http.Transport` довольно сложна и содержит очень большое количество полей. К счастью, нам не придется применять структуру `http.Transport` во всех программах, а когда мы все же будем ее использовать, не нужно будет каждый раз иметь дело со всеми ее полями.

Структура `http.Transport` реализует интерфейс `http.RoundTripper`, поддерживает протоколы HTTP, HTTPS и HTTP-прокси. Обратите внимание, что `http.Transport` — довольно низкоуровневый тип данных, в то время как структура `http.Client`, которая также используется в этой главе, реализует HTTP-клиент высокого уровня.

Что такое TCP/IP

TCP/IP — это семейство протоколов, на которых работает Интернет. Их название происходит от двух самых известных протоколов: *TCP* и *IP*.

TCP расшифровывается как Transmission Control Protocol — протокол управления передачей. Программное обеспечение TCP передает данные между машинами, используя сегменты, которые называются *TCP-пакетами*. Главное преимущество TCP состоит в том, что это надежный протокол, который гарантирует, что пакет был доставлен, и программисту не требуется писать для этого дополнительный код. Если доставка пакета не подтверждается, то TCP отправляет этот пакет повторно. Пакет TCP также может использоваться для установления соединений, передачи данных, отправки подтверждений и закрытия соединений.

Когда между двумя машинами устанавливается TCP-соединение, создается виртуальный полнодуплексный канал, похожий на телефонный звонок. Эти машины постоянно связываются, чтобы убедиться, что данные отправлены и получены

правильно. Если по какой-либо причине соединение не удается, две машины пытаются найти проблему и сообщить об этом соответствующему приложению.

IP расшифровывается как Internet Protocol. Главное достоинство IP в том, что он по своей природе не является надежным протоколом. IP инкапсулирует данные, которые передаются по сети TCP/IP, так как отвечает за доставку пакетов с исходного хоста на хост назначения в соответствии с IP-адресами. IP обеспечивает способ адресации для эффективной доставки пакета в точку назначения. Несмотря на то что существуют специальные выделенные устройства — маршрутизаторы, осуществляющие IP-маршрутизацию, каждое TCP/IP-устройство выполняет некоторую базовую маршрутизацию самостоятельно.

Протокол *UDP* (User Datagram Protocol, протокол передачи датаграмм) основан на IP, следовательно, также ненадежен. Вообще, протокол UDP проще, чем TCP, потому, что UDP по своей природе не является надежным. В результате UDP-сообщения могут быть потеряны, продублированы или доставлены не в должной последовательности. Кроме того, пакеты могут прибывать быстрее, чем получатель способен их обрабатывать. Поэтому протокол UDP используется тогда, когда скорость важнее надежности.

Что такое IPv4 и IPv6

Первая версия протокола IP сейчас называется *IPv4*, чтобы отличать ее от последней версии этого протокола — *IPv6*.

Главная проблема IPv4 состоит в том, что у него заканчиваются IP-адреса. Именно поэтому создан протокол IPv6. Это произошло из-за того, что на адрес IPv4 выделяется всего 32 бита, что позволяет использовать в общей сложности 2^{32} (4 294 967 296) IP-адресов. В IPv6, напротив, представление адреса занимает 128 бит. Адреса IPv4 имеют формат 10.20.32.245 (четыре части, разделенные точками), а формат адреса IPv6 — 3fce:1706:4523:3:150:f8ff:fe21:56cf (восемь частей, разделенных двоеточиями).

Утилита командной строки nc(1)

Утилита `nc(1)`, которая также называется `netcat(1)`, очень удобна для тестирования серверов и клиентов TCP/IP. В этом разделе рассмотрены самые распространенные варианты этой утилиты.

Например, утилиту `nc(1)` можно применять в качестве клиента для сервиса TCP, который работает на компьютере с IP-адресом 10.10.1.123 и прослушивает номер порта `1234`:

```
$ nc 10.10.1.123 1234
```

По умолчанию nc(1) использует протокол TCP. Но если выполнить nc(1) с флагом -u, то nc(1) будет использовать протокол UDP.

Если установить параметр -l, то netcat(1) будет действовать как сервер, то есть начнет прослушивать соединения с заданным номером порта.

Наконец, netcat(1) с параметрами -v и -vv будет генерировать подробный отчет, который пригодится при устранении неполадок в сетевых подключениях.

Утилита netcat(1) также помогает тестировать HTTP-приложения, однако подробно эта тема рассмотрена в главе 13 на примере разрабатотки собственных клиентов и серверов TCP и UDP. Поэтому в данной главе к утилите netcat(1) обратимся только один раз.

Чтение конфигурации сетевых интерфейсов

Конфигурация сети состоит из четырех основных элементов, таких как IP-адрес интерфейса, сетевая маска интерфейса, DNS-серверы, обеспечивающие маршрутизацию к данному компьютеру, а также шлюз или маршрутизатор, используемый по умолчанию для данного компьютера. Но есть проблема: мы не можем получить все эти данные, используя «родной», переносимый код Go. Другими словами, не существует переносимого кода, позволяющего получить DNS-конфигурацию и информацию о шлюзе по умолчанию на компьютере с UNIX.

Поэтому в данном разделе показано, как считывать конфигурацию сетевых интерфейсов UNIX-машины на Go. Для этого мы рассмотрим переносимые утилиты, которые позволяют получить информацию о сетевых интерфейсах.

Исходный код первой утилиты, которая называется netConfig.go, мы разделим на три части. Первая часть netConfig.go содержит следующий код Go:

```
package main

import (
    "fmt"
    "net"
)

func main() {
    interfaces, err := net.Interfaces()
    if err != nil {
        fmt.Println(err)
        return
    }
```

Функция net.Interfaces() возвращает все интерфейсы текущего компьютера в виде среза, содержащего элементы типа net.Interface. Этот срез мы будем использовать для получения желаемой информации.

Во второй части кода netConfig.go содержится следующий код Go:

```
    for _, i := range interfaces {
        fmt.Printf("Interface: %v\n", i.Name)
        byName, err := net.InterfaceByName(i.Name)
        if err != nil {
            fmt.Println(err)
        }
```

В этом коде мы перебираем все элементы среза типа net.Interface, чтобы получить нужную информацию.

Остальной код Go из файла netConfig.go выглядит так:

```
        addresses, err := byName.Addrs()
        for k, v := range addresses {
            fmt.Printf("Interface Address #%v: %v\n", k, v.String())
        }
        fmt.Println()
    }
}
```

Выполнение netConfig.go на компьютере с macOS Mojave и с Go версии 1.12.4 приводит к следующим результатам:

```
$ go run netConfig.go
Interface: lo0
Interface Address #0 : 127.0.0.1/8
Interface Address #1 : ::1/128
Interface Address #2 : fe80::1/64
Interface: gif0
Interface: stf0
Interface: XHC20
Interface: en0
Interface Address #0 : fe80::1435:19cd:ece8:f532/64
Interface Address #1 : ece8:f532/64
Interface: p2p0
Interface: awdl0
Interface Address #0 : fe80::888:68ff:fe01:99c/64
Interface: en1
Interface: en2
Interface: bridge0
Interface: utun0
Interface Address #0 : fe80::7fd3:e1ba:a4b1:fe22/64
```

Как видим, утилита netConfig.go возвращает довольно много данных, поскольку современные компьютеры обычно имеют множество сетевых интерфейсов и программа поддерживает протоколы IPv4 и IPv6.

Выполнение netConfig.go на компьютере с Debian Linux и с версией Go 1.7.4 приводит к следующим результатам:

```
$ go run netConfig.go
Interface: lo
```

```
Interface Address #0: 127.0.0.1/8
Interface Address #1: ::1/128
Interface: dummy0
Interface: eth0
Interface Address #0: 10.74.193.253/24
Interface Address #1: 2a01:7e00::f03c:91ff:fe69:1381/64
Interface Address #2: fe80::f03c:91ff:fe69:1381/64
Interface: teql0
Interface: tunl0
Interface: gre0
Interface: gretap0
Interface: erspan0
Interface: ip_vti0
Interface: ip6_vti0
Interface: sit0
Interface: ip6tnl0
Interface: ip6gre0
```

Обратите внимание, что главная причина, по которой сетевой интерфейс может не иметь сетевого адреса, заключается в том, что этот интерфейс не работает. Это, по сути, означает, что он в данный момент не настроен.

Не ко всем перечисленным интерфейсам подключено реальное аппаратное сетевое устройство. Более репрезентативный пример — интерфейс lo0, который представляет собой устройство обратной связи. Устройство обратной связи — это специальный виртуальный сетевой интерфейс, который используется компьютером для связи по сети с самим собой.

Код Go для второй утилиты, которая называется `netCapabilities.go`, мы также разделим на три части. Назначение `netCapabilities.go` — раскрыть возможности всех сетевых интерфейсов, обнаруженных в системе UNIX.

В утилите `netCapabilities.go` используются поля структуры `net.Interface`, которая определяется следующим образом:

```
type Interface struct {
    Index        int
    MTU          int
    Name         string
    HardwareAddr HardwareAddr
    Flags        Flags
}
```

Первая часть кода Go утилиты `netCapabilities.go` выглядит так:

```
package main

import (
    "fmt"
    "net"
)
```

Вторая часть кода netCapabilities.go содержит следующий код Go:

```
func main() {
    interfaces, err := net.Interfaces()
    if err != nil {
        fmt.Print(err)
        return
    }
```

Последняя часть netCapabilities.go содержит такой код Go:

```
    for _, i := range interfaces {
        fmt.Printf("Name: %v\n", i.Name)
        fmt.Println("Interface Flags:", i.Flags.String())
        fmt.Println("Interface MTU:", i.MTU)
        fmt.Println("Interface Hardware Address:", i.HardwareAddr)
        fmt.Println()
    }
}
```

Запуск netCapabilities.go на компьютере с macOS Mojave приведет к следующим результатам:

```
$ go run netCapabilities.go
Name : lo0
Interface Flags: up|loopback|multicast
Interface MTU: 16384
Interface Hardware Address:
Name : gif0
Interface Flags: pointtopoint|multicast
Interface MTU: 1280
Interface Hardware Address:
Name : stf0
Interface Flags: 0
Interface MTU: 1280
Interface Hardware Address:
Name : XHC20
Interface Flags: 0
Interface MTU: 0
Interface Hardware Address:
Name : en0
Interface Flags: up|broadcast|multicast
Interface MTU: 1500
Interface Hardware Address: b8:e8:56:34:a1:c8
Name : p2p0
Interface Flags: up|broadcast|multicast
Interface MTU: 2304
Interface Hardware Address: 0a:e8:56:34:a1:c8
Name : awdl0
Interface Flags: up|broadcast|multicast
Interface MTU: 1484
```

```
Interface Hardware Address: 0a:88:68:01:09:9c
Name : en1
Interface Flags: up|broadcast|multicast
Interface MTU: 1500
Interface Hardware Address: 72:00:00:9d:b2:b0
Name : en2
Interface Flags: up|broadcast|multicast
Interface MTU: 1500
Interface Hardware Address: 72:00:00:9d:b2:b1
Name : bridge0
Interface Flags: up|broadcast|multicast
Interface MTU: 1500
Interface Hardware Address: 72:00:00:9d:b2:b0
Name : utun0
Interface Flags: up|pointtopoint|multicast
Interface MTU: 2000
Interface Hardware Address:
```

Результаты запуска netCapabilities.go на компьютере с Debian Linux будут аналогичными.

Наконец, если вы действительно хотите узнать параметры шлюза, используемого на компьютере по умолчанию, вы можете выполнить команду netstat -nr либо из оболочки, либо с помощью функции exec.Command(), получив результат через канал или из функции exec.CombinedOutput() и обработав его как текст средствами Go. Правда, такое решение нельзя назвать ни элегантным, ни идеальным!

Выполнение DNS-поиска

Аббревиатура DNS расшифровывается как Domain Name System — система доменных имен — и означает способ преобразования IP-адреса в имя, такое как packt.com, и обратно из имени в IP-адрес. Логика утилиты DNS.go, которая рассмотрена в этом разделе, довольно проста: если заданный аргумент командной строки является корректным IP-адресом, то программа обработает его как IP-адрес. В противном случае программа будет считать, что ей передано имя хоста, которое необходимо преобразовать в один или несколько IP-адресов.

Мы разделим код утилиты DNS.go на три части. Первая часть программы содержит следующий код Go:

```
package main

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

```
func lookIP(address string) ([]string, error) {
    hosts, err := net.LookupAddr(address)
    if err != nil {
        return nil, err
    }
    return hosts, nil
}

func lookHostname(hostname string) ([]string, error) {
    IPs, err := net.LookupHost(hostname)
    if err != nil {
        return nil, err
    }
    return IPs, nil
}
```

Функция `lookIP()` получает в качестве входного аргумента IP-адрес и возвращает список имен, которые соответствуют этому IP-адресу, полученный с помощью функции `net.LookupAddr()`.

Функция `lookHostname()`, наоборот, получает в качестве входного аргумента имя хоста и возвращает список соответствующих IP-адресов с помощью `net.LookupHost()`.

Во второй части `DNS.go` содержится следующий код Go:

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

    input := arguments[1]
    IPaddress := net.ParseIP(input)
```

Функция `net.ParseIP()` выполняет синтаксический анализ строки как адреса IPv4 или IPv6. Если IP-адрес оказывается недействительным, то `net.ParseIP()` возвращает `nil`.

Остальной код Go утилиты `DNS.go` выглядит так:

```
    if IPaddress == nil {
        IPs, err := lookHostname(input)
        if err == nil {
            for _, singleIP := range IPs {
                fmt.Println(singleIP)
            }
        }
    } else {
        hosts, err := lookIP(input)
        if err == nil {
```

```
            for _, hostname := range hosts {
                fmt.Println(hostname)
            }
        }
    }
}
```

Выполнение `DNS.go` с различными вариантами входных данных приводит к следующим результатам:

```
$ go run DNS.go 127.0.0.1
localhost
$ go run DNS.go 192.168.1.1
cisco
$ go run DNS.go packtpub.com
83.166.169.231
$ go run DNS.go google.com
2a00:1450:4001:816::200e
216.58.210.14
$ go run DNS.go www.google.com
2a00:1450:4001:816::2004
216.58.214.36
$ go run DNS.go cnn.com
2a04:4e42::323
2a04:4e42:600::323
2a04:4e42:400::323
2a04:4e42:200::323
151.101.193.67
151.101.1.67
151.101.129.67
151.101.65.67
```

Обратите внимание, что результаты выполнения команды `go run DNS.go 192.168.1.1` взяты из моего файла `/etc/hosts`, поскольку в моем файле `/etc/hosts` имя хоста `cisco` является псевдонимом для IP-адреса `192.168.1.1`.

Как видно из результатов выполнения последней команды, иногда у одного имени хоста (`cnn.com`) может быть несколько публичных IP-адресов. Пожалуйста, обратите особое внимание на слово *«публичных»*, потому что, несмотря на то что у www.google.com несколько IP-адресов, у этого хоста только один публичный IP-адрес (`216.58.214.36`).

Получение NS-записей домена

Один из самых популярных DNS-запросов связан с поиском *серверов доменных имен* (name servers), данные о которых хранятся в *NS-записях* этого домена. Рассмотрим эту функциональность на примере кода из файла `Nsrecords.go`. Разделим его на две части.

Первая часть NSrecords.go выглядит так:

```
package main

import (
    "fmt"
    "net"
    "os"
)

func main() {
    arguments := os.Args
    if len(arguments) == 1 {
        fmt.Println("Need a domain name!")
        return
    }
```

Здесь мы проверяем, получила ли программа хотя бы один аргумент командной строки, чтобы у нее было с чем работать.

Остальной код Go программы NSrecords.go выглядит так:

```
    domain := arguments[1]
    NSs, err := net.LookupNS(domain)
    if err != nil {
        fmt.Println(err)
        return
    }

    for _, NS := range NSs {
        fmt.Println(NS.Host)
    }
}
```

Основную работу делает функция net.LookupNS(). Она возвращает NS-записи домена в виде переменной среза типа net.NS. Именно поэтому программа выводит поле Host для каждого элемента среза net.NS. Выполнение NSrecords.go приводит к результатам следующего вида:

```
$ go run NSrecords.go mtsoukalos.eu
ns5.linode.com.
ns4.linode.com.
ns1.linode.com.
ns2.linode.com.
ns3.linode.com.
$ go run NSrecords.go www.mtsoukalos.eu
lookup www.mtsoukalos.eu on 8.8.8.8:53: no such host
```

Чтобы проверить правильность этих результатов, можно воспользоваться утилитой host(1):

```
$ host -t ns www.mtsoukalos.eu
www.mtsoukalos.eu has no NS record
$ host -t ns mtsoukalos.eu
mtsoukalos.eu name server ns3.linode.com.
mtsoukalos.eu name server ns1.linode.com.
mtsoukalos.eu name server ns4.linode.com.
mtsoukalos.eu name server ns2.linode.com.
mtsoukalos.eu name server ns5.linode.com.
```

Получение MX-записей домена

Другой очень популярный DNS-запрос связан с получением *MX-записей* домена. MX-записи указывают на *почтовые серверы* (mail servers) домена. Чтобы показать, как выполнить эту задачу на Go, мы рассмотрим код утилиты `MXrecords.go`. Первая часть `MXrecords.go` содержит следующий код Go:

```
package main

import (
    "fmt"
    "net"
    "os"
)

func main() {
    arguments := os.Args
    if len(arguments) == 1 {
        fmt.Println("Need a domain name!")
        return
    }
```

Во второй части `MXrecords.go` содержится такой код Go:

```
    domain := arguments[1]
    MXs, err := net.LookupMX(domain)
    if err != nil {
        fmt.Println(err)
        return
    }

    for _, MX := range MXs {
        fmt.Println(MX.Host)
    }
}
```

Код `MXrecords.go` работает аналогично коду утилиты `NXrecords.go`, представленному в предыдущем разделе. Единственное отличие состоит в том, что в `MXrecords.go` вместо функции `net.LookupNS()` используется `net.LookupMX()`.

Выполнение `MXrecords.go` приведет к результатам следующего вида:

```
$ go run MXrecords.go golang.com
aspmx.l.google.com.
alt3.aspmx.l.google.com.
alt1.aspmx.l.google.com.
alt2.aspmx.l.google.com.
$ go run MXrecords.go www.mtsoukalos.eu
lookup www.mtsoukalos.eu on 8.8.8.8:53: no such host
```

Как и в прошлый раз, мы можем убедиться в правильности этих результатов с помощью утилиты `host(1)`:

```
$ host -t mx golang.com
golang.com mail is handled by 2 alt3.aspmx.l.google.com.
golang.com mail is handled by 1 aspmx.l.google.com.
golang.com mail is handled by 2 alt1.aspmx.l.google.com.
golang.com mail is handled by 2 alt2.aspmx.l.google.com.
$ host -t mx www.mtsoukalos.eu
www.mtsoukalos.eu has no MX record
```

Создание веб-сервера на Go

Используя некоторые функции стандартной библиотеки Go, можно создавать собственные веб-серверы.

Написанный на Go веб-сервер позволяет выполнять многие задачи эффективно и безопасно. Однако если вам нужен действительно мощный веб-сервер, который будет поддерживать модульность, несколько веб-сайтов и виртуальные хосты, то лучше использовать такой веб-сервер, как Apache, Nginx или Candy, который также написан на Go.

В этом разделе в качестве примера рассмотрим программу Go, которая называется `www.go`. Разделим ее на пять частей. Первая часть `www.go` содержит ожидаемые объявления импорта:

```
package main

import (
    "fmt"
    "net/http"
    "os"
    "time"
)
```

Пакет `time` для работы веб-сервера необязателен. Однако в данном случае он необходим, поскольку сервер будет отправлять своим клиентам время и дату.

Второй фрагмент кода `www.go` содержит следующий код Go:

```
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 реализация первой функции-обработчика в данной программе. Функция-обработчик обслуживает один или несколько URL-адресов, в зависимости от конфигурации, указанной в коде Go, — мы можем создать столько функций-обработчиков, сколько пожелаем.

В третьей части `www.go` содержится следующий код Go:

```
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)
}
```

В этом коде Go представлена реализация второй функции-обработчика в этой программе. Эта функция генерирует динамическое содержимое.

Четвертый фрагмент кода нашего веб-сервера посвящен аргументам командной строки и определению поддерживаемых URL-адресов:

```
func main() {
    PORT := ":8001"
    arguments := os.Args
    if len(arguments) == 1 {
        fmt.Println("Using default port number: ", PORT)
    } else {
        PORT = ":" + arguments[1]
    }

    http.HandleFunc("/time", timeHandler)
    http.HandleFunc("/", myHandler)
```

Функция `http.HandleFunc()` связывает URL-адрес с функцией-обработчиком. Самое важное, что все URL-адреса, кроме `/time`, обслуживаются функцией `myHandler()`, поскольку ее первый аргумент, равный `/`, соответствует любому URL-адресу, который не подходит никакому другому обработчику.

Последняя часть программы `www.go` выглядит так:

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

Чтобы запустить веб-сервер, необходимо воспользоваться функцией `http.ListenAndServe()`, указав соответствующий номер порта.

Выполнение `www.go` и подключение к созданному веб-серверу приведет к следующим результатам:

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

Несмотря на то что результаты программы содержат интересные данные, я полагаю, что вы предпочли бы увидеть реальный результат работы программы в окне вашего любимогобраузера. На рис. 12.1 показан результат работы функции `myHandler()` веб-сервера в Google Chrome.

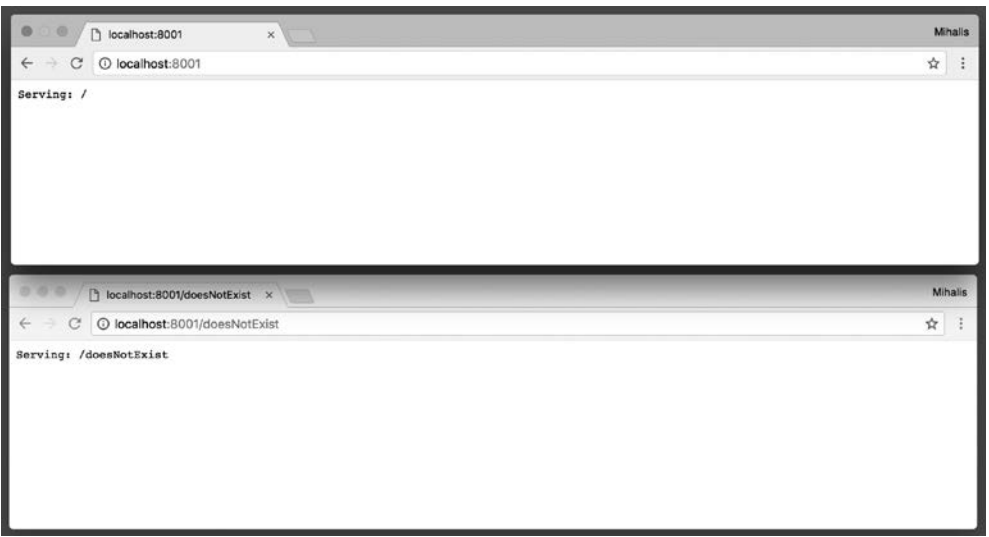


Рис 12.1. Начальная страница веб-сервера www.go

На рис. 12.2 видно, что `www.go` позволяет также генерировать динамические страницы. В данном случае это веб-страница, зарегистрированная по адресу `/time`, которая отображает текущую дату и время.

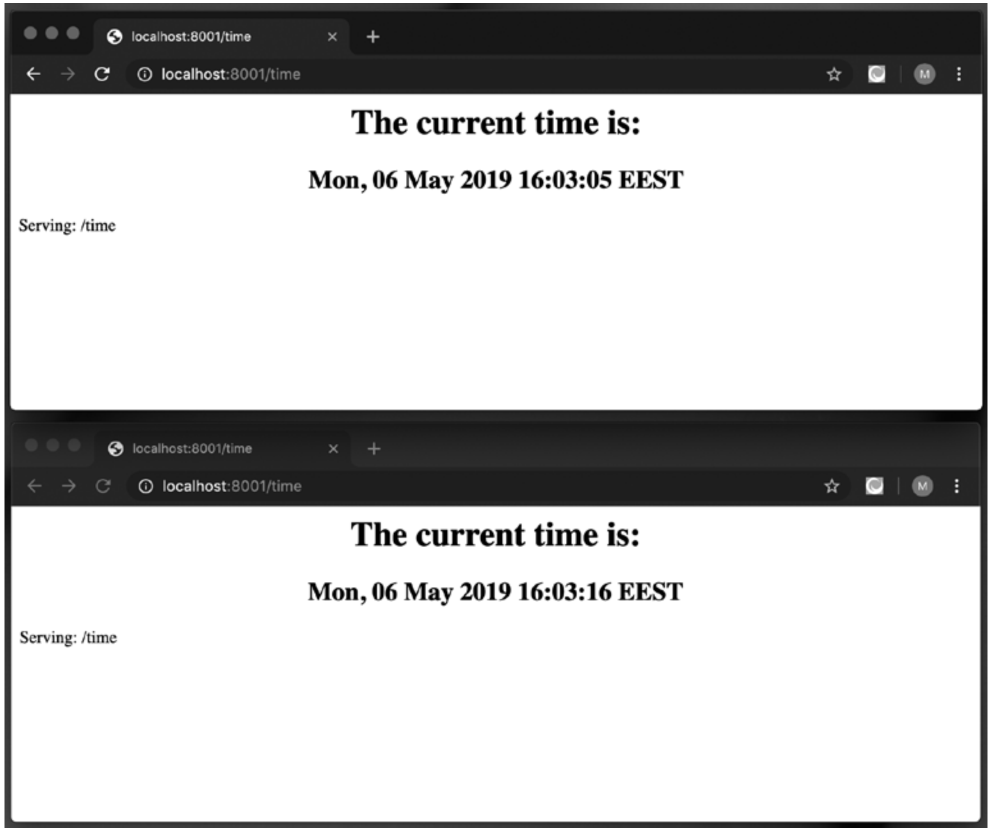


Рис. 12.2. Получение текущей даты и времени от веб-сервера www.go

Использование пакета atomic

В этом разделе вы узнаете, как использовать пакет `atomic` в среде HTTP-сервера. Для этого рассмотрим программу `atomWWW.go`, которую разделим на три части.

Первая часть `atomWWW.go` выглядит так:

```
package main

import (
    "fmt"
    "net/http"
    "runtime"
    "sync/atomic"
)

var count int32
```

Переменную, которая используется пакетом `atomic`, мы сделали глобальной, чтобы она была доступной из любой точки кода.

Во второй части `atomWWW.go` содержится следующий код Go:

```
func handleAll(w http.ResponseWriter, r *http.Request) {
    atomic.AddInt32(&count, 1)
}

func getCounter(w http.ResponseWriter, r *http.Request) {
    temp := atomic.LoadInt32(&count)
    fmt.Println("Count:", temp)
    fmt.Fprintf(w, "<h1 align=\"center\">%d</h1>", count)
}
```

Счетчик `atomic`, применяемый в этой программе, связан с глобальной переменной `count` и помогает подсчитать общее количество клиентов, которые обслужил веб-сервер.

Последняя часть `atomWWW.go` выглядит так:

```
func main() {
    runtime.GOMAXPROCS(runtime.NumCPU() - 1)
    http.HandleFunc("/getCounter", getCounter)
    http.HandleFunc("/", handleAll)
    http.ListenAndServe(":8080", nil)
}
```

Тестирование `atomWWW.go` с помощью утилиты `ab(1)` помогает проверить, насколько эффективен пакет `atomic`, поскольку оно показывает, что переменная `count` доступна всем клиентам без каких-либо ограничений. Прежде всего запустим `atomWWW.go`:

```
$ go run atomWWW.go
Count: 1500
```

Затем выполним `ab(1)`:

```
$ ab -n 1500 -c 100 http://localhost:8080/
This is ApacheBench, Version 2.3 <$Revision: 1826891 $>
Copyright 1996 Adam Twiss, Zeus Technology Ltd, http://www.zeustech.net/
Licensed to The Apache Software Foundation, http://www.apache.org/
Benchmarking localhost (be patient)
Completed 150 requests
Completed 300 requests
Completed 450 requests
Completed 600 requests
Completed 750 requests
Completed 900 requests
Completed 1050 requests
Completed 1200 requests
Completed 1350 requests
Completed 1500 requests
```

```
Finished 1500 requests
Server Software:
Server Hostname:        localhost
Server Port:            8080
Document Path:          /
Document Length:        0 bytes
Concurrency Level:      100
Time taken for tests:   0.098 seconds
Complete requests:      1500
Failed requests:        0
Total transferred:      112500 bytes
HTML transferred:       0 bytes
Requests per second:    15238.64 [#/sec] (mean)
Time per request:       6.562 [ms] (mean)
Time per request:       0.066 [ms] (mean, across all concurrent requests)
Transfer rate:          1116.11 [Kbytes/sec] received
Connection Times (ms)
            min      mean[+/-sd]     median    max
Connect:      0        3    0.5        3        5
Processing:   2        3    0.6        3        6
Waiting:      0        3    0.6        3        5
Total:        4        6    0.6        6        9
Percentage of the requests served within a certain time (ms)
    50%       6
    66%       6
    75%       7
    80%       7
    90%       7
    95%       7
    98%       8
    99%       8
   100%       9 (longest request)
```

Показанная выше команда ab(1) отправляет 1500 запросов, при этом количество одновременных запросов равно 100.

После выполнения ab(1) посетите адрес /getCounter, чтобы получить текущее значение переменной count:

```
$ wget -qO- http://localhost:8080/getCounter
<h1 align="center">1500</h1>%
```

Профилирование HTTP-сервера

Как вы уже знаете из главы 11, для того чтобы профилировать приложение Go с собственным HTTP-сервером, следует использовать стандартный пакет Go net/http/pprof. Для этого при импорте net/http/pprof в папку /debug/pprof/URL устанавливаются различные обработчики. Вскоре вы ближе познакомитесь с этой технологией. Пока достаточно вспомнить, что для профилирования веб-приложений с помощью HTTP-сервера необходимо использовать пакет net/http/pprof, тогда

как для профилирования остальных видов приложений применяется стандартный Go-пакет `runtime/pprof`.

Обратите внимание, что если ваш профилировщик работает с адресом http://localhost:8080, то вы автоматически получите поддержку следующих веб-ссылок:

- http://localhost:8080/debug/pprof/goroutine;
- http://localhost:8080/debug/pprof/heap;
- http://localhost:8080/debug/pprof/threadcreate;
- http://localhost:8080/debug/pprof/block;
- http://localhost:8080/debug/pprof/mutex;
- http://localhost:8080/debug/pprof/profile;
- http://localhost:8080/debug/pprof/trace?seconds=5.

Следующая программа, которую мы рассмотрим, применяет `www.go` в качестве отправной точки и добавляет необходимый код Go, чтобы ее можно было профилировать.

Эта программа называется `wwwProfile.go`. Разделим ее на четыре части.

Заметьте, что для регистрации поддерживаемых программой путей в `wwwProfile.go` используется переменная `http.NewServeMux`. Главная причина этого заключается в том, что для того, чтобы использовать `http.NewServeMux`, нужно задать конечные точки HTTP вручную. Также обратите внимание, что мы можем определять подмножество поддерживаемых конечных точек HTTP. Если вы решите не использовать `http.NewServeMux`, то конечные точки HTTP будут зарегистрированы автоматически. Для этого вам также придется импортировать пакет `net/http/pprof`, поставив перед ним символ `_`.

Первая часть `wwwProfile.go` содержит следующий код Go:

```
package main

import (
    "fmt"
    "net/http"
    "net/http/pprof"
    "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)
}

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)
}
```

Реализации этих двух функций-обработчиков точно такие же, как и раньше.

Второй фрагмент кода `wwwProfile.go` выглядит так:

```
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)
```

В этом коде Go мы определяем URL-адреса, которые будут поддерживаться веб-сервером с помощью функций `http.NewServeMux()` и `HandleFunc()`.

Третья часть `wwwProfile.go` содержит следующий код Go:

```
    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)
```

В этом коде Go определены конечные точки HTTP, связанные с профилированием. Без них мы не сможем профилировать веб-приложение.

Остальной код Go выглядит так:

```
    err := http.ListenAndServe(PORT, r)
    if err != nil {
        fmt.Println(err)
        return
    }
}
```

Этот код запускает веб-сервер Go и позволяет ему обслуживать соединения с HTTP-клиентами. Вы, вероятно, заметили, что второй параметр функции `http.ListenAndServe()` больше не равен нулю.

Как видим, в `wwwProfile.go` не определена конечная точка HTTP `/debug/pprof/goroutine`. Это оправданно, поскольку в `wwwProfile.go` отсутствуют горутины.

Выполнение `wwwProfile.go` приведет к результатам следующего вида:

```
$ go run wwwProfile.go 1234
Using port number: :1234
Served time for: localhost:1234
```

Использование профилировщика Go для получения данных — довольно простая задача. Для ее выполнения нужно запустить следующую команду, которая автоматически откроет оболочку профилировщика Go:

```
$ go tool pprof http://localhost:1234/debug/pprof/profile
Fetching profile over HTTP from http://localhost:1234/debug/pprof/profile
Saved profile in /Users/mtsouk/pprof/pprof.samples.cpu.003.pb.gz
Type: cpu
Time: Mar 27, 2018 at 10:04pm (EEST)
Duration: 30s, Total samples = 21.04s (70.13%)
Entering interactive mode (type "help" for commands, "o" for options)
(pprof) top
Showing nodes accounting for 19.94s, 94.77% of 21.04s total
Dropped 159 nodes (cum <= 0.11s)
Showing top 10 nodes out of 75
      flat  flat%   sum%        cum   cum%
    13.73s 65.26% 65.26%     13.73s 65.26%  syscall.Syscall
     1.58s  7.51% 72.77%      1.58s  7.51%  runtime.kevent
     1.36s  6.46% 79.23%      1.36s  6.46%  runtime.mach_semaphore_signal
     1.02s  4.85% 84.08%      1.02s  4.85%  runtime.usleep
     0.80s  3.80% 87.88%      0.80s  3.80%  runtime.mach_semaphore_wait
     0.53s  2.52% 90.40%      2.11s 10.03%  runtime.netpoll
     0.44s  2.09% 92.49%      0.44s  2.09%  internal/poll.convertErr
     0.26s  1.24% 93.73%      0.26s  1.24%  net.(*TCPConn).Read
     0.18s  0.86% 94.58%      0.18s  0.86%  runtime.freedefer
     0.04s  0.19% 94.77%      1.05s  4.99%  runtime.runqsteal
(pprof)
```

Теперь мы можем применять данные профилирования и анализировать их с помощью инструмента `go pprof`, как вы научились в главе 11.

Чтобы узнать результаты профилирования, посетите страницу http://ИМЯ_ХОСТА:НОМЕР_ПОРТА/debug/pprof/. Если ИМЯ_ХОСТА — localhost, а НОМЕР_ПОРТА — 1234, получим страницу http://localhost:1234/debug/pprof/.

Протестировать производительность веб-серверного приложения можно с помощью утилиты `ab(1)`, которая более известна как инструмент бенчмаркинга HTTP-сервера Apache для создания трафика для сервера и выполнения бенчмаркинга `wwwProfile.go`. Это также позволит инструменту `go pprof` собрать более точные данные — для этого нужно выполнить `ab(1)` следующим образом:

```
$ ab -k -c 10 -n 100000 "http://127.0.0.1:1234/time"
This is ApacheBench, Version 2.3 <$Revision: 1807734 $>
Copyright 1996 Adam Twiss, Zeus Technology Ltd, http://www.zeustech.net/
```

```
Licensed to The Apache Software Foundation, http://www.apache.org/
Benchmarking 127.0.0.1 (be patient)
Completed 10000 requests
Completed 20000 requests
Completed 30000 requests
Completed 40000 requests
Completed 50000 requests
Completed 60000 requests
Completed 70000 requests
Completed 80000 requests
Completed 90000 requests
Completed 100000 requests
Finished 100000 requests

Server Software:
Server Hostname:        127.0.0.1
Server Port:            1234
Document Path:          /time
Document Length:        114 bytes
Concurrency Level:      10
Time taken for tests:   2.114 seconds
Complete requests:      100000
Failed requests:        0
Keep-Alive requests:    100000
Total transferred:      25500000 bytes
HTML transferred:       11400000 bytes
Requests per second:    47295.75 [#/sec] (mean)
Time per request:       0.211 [ms] (mean)
Time per request:       0.021 [ms] (mean, across all concurrent requests)
Transfer rate:          11777.75 [Kbytes/sec] received
Connection Times (ms)
            min   mean[+/-sd]   median    max
Connect:      0    0    0.0        0        0
Processing:   0    0    0.7        0       13
Waiting:      0    0    0.7        0       13
Total:        0    0    0.7        0       13
Percentage of the requests served within a certain time (ms)
  50%      0
  66%      0
  75%      0
  80%      0
  90%      0
  95%      0
  98%      0
  99%      0
 100%     13 (longest request)
```

Можно ли использовать пакет net/http/pprof для профилирования приложений командной строки? Да, можно! Однако пакет net/http/pprof особенно полезен для профилирования работающего веб-приложения и сбора оперативных данных. Именно поэтому он представлен в данной главе.

Создание веб-сайта на Go

Помните ли вы приложение `keyValue.go` из главы 4 и `kvSaveLoad.go` из главы 8? В этом подразделе вы узнаете, как создать веб-интерфейс для приложения `keyValue.go`, используя возможности стандартной библиотеки Go. Для этого мы рассмотрим файл `kvWeb.go`, который разделим на шесть частей.

Главное различие между программой `kvWeb.go` и `www.go`, разработанной ранее в этой главе, заключается в том, что в `kvWeb.go` для обработки HTTP-запросов используется тип `http.NewServeMux`, поскольку этот тип гораздо более универсальный и подходит для нетривиальных веб-приложений.

Первая часть `kvWeb.go` выглядит так:

```
package main

import (
    "encoding/gob"
    "fmt"
    "html/template"
    "net/http"
    "os"
)

type myElement struct {
    Name     string
    Surname  string
    Id       string
}

var DATA = make(map[string]myElement)
var DATAFILE = "/tmp/dataFile.gob"
```

Этот код нам уже встречался в программе `kvSaveLoad.go` в главе 8.

Вторая часть `kvWeb.go` содержит следующий код Go:

```
func save() error {
    fmt.Println("Saving", DATAFILE)
    err := os.Remove(DATAFILE)
    if err != nil {
        fmt.Println(err)
    }

    saveTo, err := os.Create(DATAFILE)
    if err != nil {
        fmt.Println("Cannot create", DATAFILE)
        return err
    }
    defer saveTo.Close()
```

```
    encoder := gob.NewEncoder(saveTo)
    err = encoder.Encode(DATA)
    if err != nil {
        fmt.Println("Cannot save to", DATAFILE)
        return err
    }
    return nil
}

func load() error {
    fmt.Println("Loading", DATAFILE)
    loadFrom, err := os.Open(DATAFILE)
    defer loadFrom.Close()
    if err != nil {
        fmt.Println("Empty key/value store!")
        return err
    }

    decoder := gob.NewDecoder(loadFrom)
    decoder.Decode(&DATA)
    return nil
}

func ADD(k string, n myElement) bool {
    if k == "" {
        return false
    }

    if LOOKUP(k) == nil {
        DATA[k] = n
        return true
    }
    return false
}

func DELETE(k string) bool {
    if LOOKUP(k) != nil {
        delete(DATA, k)
        return true
    }
    return false
}

func LOOKUP(k string) *myElement {
    _, ok := DATA[k]
    if ok {
        n := DATA[k]
        return &n
```

```
    } else {
        return nil
    }
}

func CHANGE(k string, n myElement) bool {
    DATA[k] = n
    return true
}

func PRINT() {
    for k, d := range DATA {
        fmt.Printf("key: %s value: %v\n", k, d)
    }
}
```

Этот код Go вам также должен быть знаком, так как вы уже встречали его в программе `kvSaveLoad.go` в главе 8.

Третья часть кода `kvWeb.go` выглядит так:

```
func homePage(w http.ResponseWriter, r *http.Request) {
    fmt.Println("Serving", r.Host, "for", r.URL.Path)
    myT := template.Must(template.ParseGlob("home.gohtml"))
    myT.ExecuteTemplate(w, "home.gohtml", nil)
}

func listAll(w http.ResponseWriter, r *http.Request) {
    fmt.Println("Listing the contents of the KV store!")

    fmt.Fprintf(w, "<a href=\"/\" style=\"margin-right: 20px;\">Home sweet home!</a>")
    fmt.Fprintf(w, "<a href=\"/list\" style=\"margin-right: 20px;\">
            List all elements!</a>")
    fmt.Fprintf(w, "<a href=\"/change\" style=\"margin-right: 20px;\">
            Change an element!</a>")
    fmt.Fprintf(w, "<a href=\"/insert\" style=\"margin-right: 20px;\">
            Insert new element!</a>")

    fmt.Fprintf(w, "<h1>The contents of the KV store are:</h1>") fmt.Fprintf(w, "<ul>")
    for k, v := range DATA {
        fmt.Fprintf(w, "<li>")
        fmt.Fprintf(w, "<strong>%s</strong> with value: %v\n", k, v)
        fmt.Fprintf(w, "</li>")
    }

    fmt.Fprintf(w, "</ul>")
}
```

Функция `listAll()` не использует шаблоны Go для генерации динамического вывода. Вместо этого вывод функции генерируется по ходу выполнения программы с помощью Go. Это можно считать исключением из правил, поскольку обычно

веб-приложения лучше работают с HTML-шаблонами и стандартным Go-пакетом `html/template`.

Четвертая часть `kvWeb.go` содержит следующий код Go:

```
func changeElement(w http.ResponseWriter, r *http.Request) {
    fmt.Println("Changing an element of the KV store!")
    tmpl := template.Must(template.ParseFiles("update.gohtml"))
    if r.Method != http.MethodPost {
        tmpl.Execute(w, nil)
        return
    }

    key := r.FormValue("key")
    n := myElement{
        Name:       r.FormValue("name"),
        Surname:    r.FormValue("surname"),
        Id:         r.FormValue("id"),
    }

    if !CHANGE(key, n) {
        fmt.Println("Update operation failed!")
    } else {
        err := save()
        if err != nil {
            fmt.Println(err)
            return
        }
        tmpl.Execute(w, struct{ Success bool }{true})
    }
}
```

В этом коде Go показано, как можно читать значения из полей HTML-формы с помощью `FormValue()`. Функция `template.Must()` — это вспомогательная функция, которая гарантирует, что предоставленный файл шаблона не содержит ошибок.

Пятый фрагмент в `kvWeb.go` содержит следующий код Go:

```
func insertElement(w http.ResponseWriter, r *http.Request) {
    fmt.Println("Inserting an element to the KV store!")
    tmpl := template.Must(template.ParseFiles("insert.gohtml"))
    if r.Method != http.MethodPost {
        tmpl.Execute(w, nil)
        return
    }

    key := r.FormValue("key")
    n := myElement{
        Name:       r.FormValue("name"),
        Surname:    r.FormValue("surname"),
        Id:         r.FormValue("id"),
    }
```

```
    if !ADD(key, n) {
        fmt.Println("Add operation failed!")
    } else {
        err := save()
        if err != nil {
            fmt.Println(err)
            return
        }
        tmpl.Execute(w, struct{ Success bool }{true})
    }
}
```

Остальной код Go выглядит так:

```
func main() {
    err := load()
    if err != nil {
        fmt.Println(err)
    }

    PORT := ":8001"
    arguments := os.Args
    if len(arguments) == 1 {
        fmt.Println("Using default port number: ", PORT)
    } else {
        PORT = ":" + arguments[1]
    }

    http.HandleFunc("/", homePage)
    http.HandleFunc("/change", changeElement)
    http.HandleFunc("/list", listAll)
    http.HandleFunc("/insert", insertElement)
    err = http.ListenAndServe(PORT, nil)
    if err != nil {
        fmt.Println(err)
    }
}
```

Функция `main()` в `kvWeb.go` намного проще, чем `main()` в `kvSaveLoad.go` из главы 8, потому что эти две программы имеют совершенно разную структуру.

Теперь пора взглянуть на файлы `gohtml`, необходимые для этого проекта, начиная с `home.gohtml`:

```
<!doctype html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>A Key Value Store!</title>
</head>
<body>

<a href="/" style="margin-right: 20px;">Home sweet home!</a>
<a href="/list" style="margin-right: 20px;">List all elements!</a>
```

```
<a href="/change" style="margin-right: 20px;">Change an element!</a>
<a href="/insert" style="margin-right: 20px;">Insert new element!</a>

<h2>Welcome to the Go KV store!</h2>

</body>
</html>
```

Файл home.gohtml является статическим, то есть его содержимое не изменяется. Однако остальные файлы gohtml отображают информацию динамически.

Содержимое update.gohtml выглядит так:

```
<!doctype html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>A Key Value Store!</title>
</head>
<body>

<a href="/" style="margin-right: 20px;">Home sweet home!</a>
<a href="/list" style="margin-right: 20px;">List all elements!</a>
<a href="/change" style="margin-right: 20px;">Change an element!</a>
<a href="/insert" style="margin-right: 20px;">Insert new element!</a>

{{if .Success}}
    <h1>Element updated!</h1>
{{else}}
    <h1>Please fill in the fields:</h1>

    <form method="POST">
        <label>Key:</label><br />
        <input type="text" name="key"><br />
        <label>Name:</label><br />
        <input type="text" name="name"><br />
        <label>Surname:</label><br />
        <input type="text" name="surname"><br />
        <label>Id:</label><br />
        <input type="text" name="id"><br />
        <input type="submit">
    </form>
{{end}}

</body>
</html>
```

В основном этот код представляет собой HTML. Самая интересная его часть — оператор `if`, который указывает, что следует обновлять: форму или сообщение `Element updated!`.

Наконец, содержимое файла insert.gohtml:

```
<!doctype html>
<html lang="en">
```

```
<head>
    <meta charset="UTF-8">
    <title>A Key Value Store!</title>
</head>
<body>

<a href="/" style="margin-right: 20px;">Home sweet home!</a>
<a href="/list" style="margin-right: 20px;">List all elements!</a>
<a href="/change" style="margin-right: 20px;">Change an element!</a>
<a href="/insert" style="margin-right: 20px;">Insert new element!</a>

{{if .Success}}
    <h1>Element inserted!</h1>
{{else}}
    <h1>Please fill in the fields:</h1>
    <form method="POST">
        <label>Key:</label><br />
        <input type="text" name="key"><br />
        <label>Name:</label><br />
        <input type="text" name="name"><br />
        <label>Surname:</label><br />
        <input type="text" name="surname"><br />
        <label>Id:</label><br />
        <input type="text" name="id"><br />
        <input type="submit">
    </form>
{{end}}

</body>
</html>
```

Как видим, `insert.gohtml` и `update.gohtml` идентичны, кроме текста внутри тега `<title>`.

Выполнение `kvWeb.go` в оболочке UNIX выведет следующее:

```
$ go run kvWeb.go
Loading /tmp/dataFile.gob
Using default port number: :8001
Serving localhost:8001 for /
Serving localhost:8001 for /favicon.ico
Listing the contents of the KV store!
Serving localhost:8001 for /favicon.ico
Inserting an element to the KV store!
Serving localhost:8001 for /favicon.ico
Inserting an element to the KV store!
Add operation failed!
Serving localhost:8001 for /favicon.ico
Inserting an element to the KV store!
Serving localhost:8001 for /favicon.ico
Inserting an element to the KV store!
Saving /tmp/dataFile.gob
Serving localhost:8001 for /favicon.ico
```

```
Inserting an element to the KV store!
Serving localhost:8001 for /favicon.ico
Changing an element of the KV store!
Serving localhost:8001 for /favicon.ico
```

Кроме того, здесь особенно интересно то, как мы можем взаимодействовать с `kvWeb.go` из браузера.

На рис. 12.3 показана начальная страница веб-сайта, определенная в `home.gohtml`.

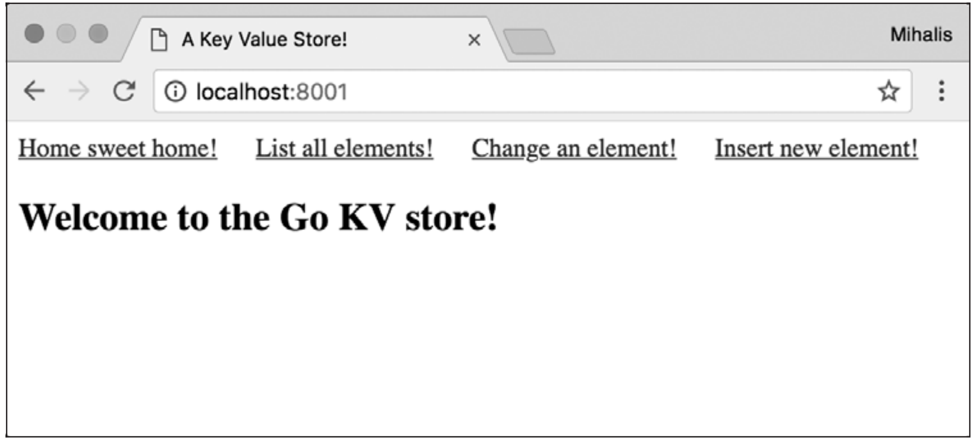


Рис. 12.3. Статическая начальная страница нашего веб-приложения

Содержимое хранилища «ключ — значение» видно на рис. 12.4.

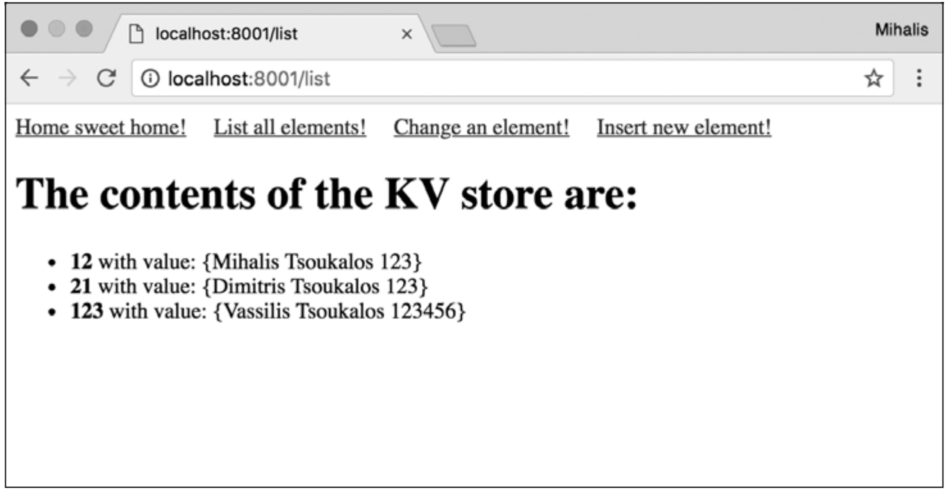


Рис. 12.4. Содержимое хранилища «ключ — значение»

На рис. 12.5 — внешний вид веб-страницы, которая позволяет добавлять новые данные в хранилище «ключ — значение» с помощью веб-интерфейса веб-приложения `kvWeb.go`.

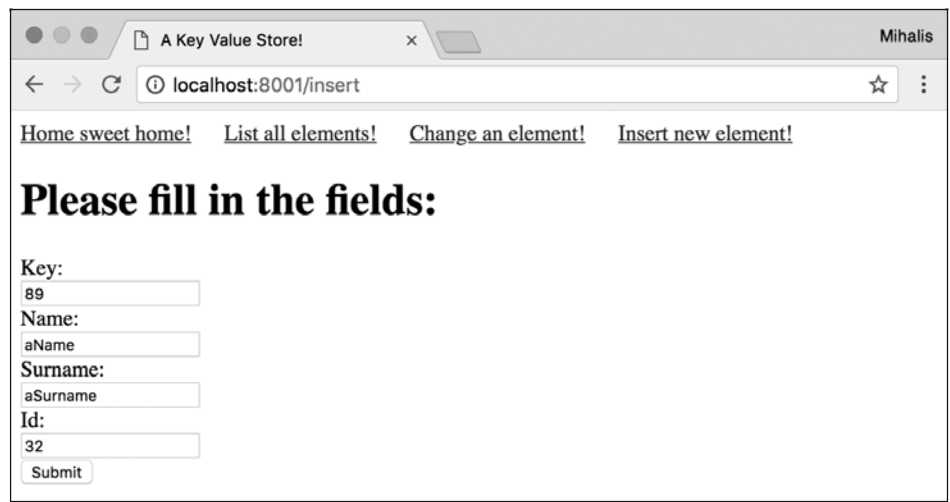


Рис. 12.5. Добавление новой записи в хранилище «ключ — значение»

Как можно изменить значение существующего ключа с помощью веб-интерфейса веб-приложения `kvWeb.go`, вы видите на рис. 12.6.

Рис. 12.6. Изменение значения ключа в хранилище «ключ — значение»

Веб-приложение `kvWeb.go` далеко от совершенства, поэтому не стесняйтесь улучшать его — это хорошее упражнение!

В этом подразделе показано, как можно разрабатывать на Go целые веб-сайты и веб-приложения. Хотя требования к вашим приложениям наверняка будут другими, методы останутся теми же, что и в kvWeb.go. Обратите внимание, что сайты, написанные вручную, более безопасны, чем сайты, созданные с использованием популярных систем управления контентом.

HTTP-трассировка

Go поддерживает HTTP-трассировку с помощью стандартного пакета `net/http/httptrace`. Он позволяет поэтапно отслеживать прохождение HTTP-запроса. Рассмотрим использование стандартного Go-пакета `net/http/httptrace` на примере программы `httpTrace.go`. Разделим ее на пять частей.

Первая часть `httpTrace.go` выглядит так:

```
package main

import (
    "fmt"
    "io"
    "net/http"
    "net/http/httptrace"
    "os"
)
```

Как и следовало ожидать, чтобы сделать возможной HTTP-трассировку, нам нужно импортировать пакет `net/http/httptrace`.

Вторая часть `httpTrace.go` содержит следующий код Go:

```
func main() {
    if len(os.Args) != 2 {
        fmt.Printf("Usage: URL\n")
        return
    }

    URL := os.Args[1]
    client := http.Client{}
```

Здесь мы читаем аргументы командной строки и создаем новую переменную `http.Client`.

Объект `http.Client` заслуживает отдельного обсуждения. Он позволяет отправить запрос на сервер и получить ответ.

Его поле Transport позволяет задавать различные параметры HTTP-запроса вместо установленных по умолчанию.

Обратите внимание, что в реальном программном обеспечении никогда не следует использовать для объекта http.Client значения по умолчанию, поскольку эти значения не определяют задержки запросов, которые могли бы значительно снизить производительность программ и горутин. Кроме того, структура объектов http.Client позволяет безопасно применять их в конкурентных программах.

Третий фрагмент httpTrace.go содержит следующий код Go:

```
req, _ := http.NewRequest("GET", URL, nil)
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")
    },
}
```

Этот код отвечает за отслеживание HTTP-запросов. Объект httptrace.ClientTrace определяет события, которые нас интересуют. Когда происходит одно из таких событий, выполняется соответствующий код. Более подробную информацию о поддерживаемых событиях и их назначении вы найдете в документации пакета net/http/httptrace.

Четвертая часть утилиты httpTrace.go выглядит так:

```
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 новый контекст, полученный на основе заданного родительского контекста; метод `http.DefaultTransport.RoundTrip()` служит оберткой для объекта `http.DefaultTransport.RoundTrip`, чтобы он отслеживал текущий запрос.

Обратите внимание, что HTTP-трассировка в Go разработана только для отслеживания событий `http.Transport.RoundTrip`. Однако, поскольку при обработке одного HTTP-запроса возможно несколько перенаправлений URL-адресов, вам необходимо определить текущий запрос.

Остальной код Go программы `httpTrace.go` выглядит так:

```
    response, err := client.Do(req)
    if err != nil {
        fmt.Println(err)
        return
    }

    io.Copy(os.Stdout, response.Body)
}
```

Последняя часть программы отвечает за фактическое выполнение запроса к веб-серверу с помощью функции `Do()`, а также за получение HTTP-данных и вывод их на экран.

Выполнение `httpTrace.go` позволяет получить следующую очень важную информацию:

```
$ go run httpTrace.go http://localhost:8001/
Requesting data from server!
DNS Info: {Addrs:[{IP:::1 Zone:} {IP:127.0.0.1 Zone:}] Err:<nil>
Coalesced:false}
Dial start
Dial done
Got Conn: {Conn:0xc420142000 Reused:false WasIdle:false IdleTime:0s}
Wrote headers
First response byte!
DNS Info: {Addrs:[{IP:::1 Zone:} {IP:127.0.0.1 Zone:}] Err:<nil>
Coalesced:false}
Dial start
Dial done
Got Conn: {Conn:0xc420142008 Reused:false WasIdle:false IdleTime:0s}
Wrote headers
First response byte!
Serving: /
```

Учтите, что, поскольку `httpTrace.go` выводит полный HTML-отклик HTTP-сервера, при тестировании реального веб-сервера мы можем получить много выходных данных. Именно поэтому мы воспользовались веб-сервером, разработанным ранее с помощью `www.go`.

Если у вас есть время, взгляните на исходный код пакета net/http/httptrace по адресу https://golang.org/src/net/http/httptrace/trace.go. Тогда вы сразу поймете, что net/http/httptrace — низкоуровневый пакет, для реализации функций которого используются пакеты context, reflect и internal/nettrace. Помните, что вы можете сделать это для любого кода стандартной библиотеки, потому что Go — это проект с полностью открытым исходным кодом.

Тестирование HTTP-обработчиков

В этом подразделе вы научитесь тестировать HTTP-обработчики в Go. Мы начнем с кода Go из файла `www.go` и будем менять его по мере необходимости.

Новую версию `www.go` мы назовем `testWWW.go.` Рассмотрим ее, разделив на три части.

Первая часть кода `testWWW.go` выглядит так:

```
package main

import (
    "fmt"
    "net/http"
    "os"
)

func CheckStatusOK(w http.ResponseWriter, r *http.Request) {
    w.WriteHeader(http.StatusOK)
    fmt.Fprintf(w, `Fine!`)
}
```

Вторая часть `testWWW.go` содержит следующий код Go:

```
func StatusNotFound(w http.ResponseWriter, r *http.Request) {
    w.WriteHeader(http.StatusNotFound)
}

func MyHandler(w http.ResponseWriter, r *http.Request) {
    fmt.Fprintf(w, "Serving: %s\n", r.URL.Path)
    fmt.Printf("Served: %s\n", r.Host)
}
```

Остальной код Go программы `testWWW.go`:

```
func main() {
    PORT := ":8001"
    arguments := os.Args
    if len(arguments) == 1 {
        fmt.Println("Using default port number: ", PORT)
    } else {
        PORT = ":" + arguments[1]
    }
```

```
    http.HandleFunc("/CheckStatusOK", CheckStatusOK)
    http.HandleFunc("/StatusNotFound", StatusNotFound)
    http.HandleFunc("/", MyHandler)

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

Теперь нам нужно начать тестирование `testWWW.go`, для чего следует создать файл `testWWW_test.go`. Содержимое этого файла представлено в четырех частях.

Первая часть `testWWW_test.go` содержит следующий код Go:

```
package main

import (
    "fmt"
    "net/http"
    "net/http/httptest"
    "testing"
)
```

Обратите внимание, что для тестирования веб-приложений в Go необходимо импортировать стандартный Go-пакет `net/http/httptest`.

Вторая часть `testWWW_test.go` содержит следующий код:

```
func TestCheckStatusOK(t *testing.T) {
    req, err := http.NewRequest("GET", "/CheckStatusOK", nil)
    if err != nil {
        fmt.Println(err)
        return
    }

    rr := httptest.NewRecorder()
    handler := http.HandlerFunc(CheckStatusOK)
    handler.ServeHTTP(rr, req)
```

Функция `httptest.NewRecorder()` возвращает объект `httptest.ResponseRecorder` и используется для записи HTTP-ответа.

Третья часть `testWWW_test.go` выглядит так:

```
    status := rr.Code
    if status != http.StatusOK {
        t.Errorf("handler returned %v", status)
    }

    expect := `Fine!`
    if rr.Body.String() != expect {
        t.Errorf("handler returned %v", rr.Body.String())
    }
}
```

Сначала мы убеждаемся, что код ответа соответствует ожидаемому, а затем — что тело ответа также совпадает с ожидаемым.

Остальной код Go программы `testWWW_test.go` выглядит так:

```
func TestStatusNotFound(t *testing.T) {
    req, err := http.NewRequest("GET", "/StatusNotFound", nil)
    if err != nil {
        fmt.Println(err)
        return
    }

    rr := httptest.NewRecorder()
    handler := http.HandlerFunc(StatusNotFound)
    handler.ServeHTTP(rr, req)

    status := rr.Code
    if status != http.StatusNotFound {
        t.Errorf("handler returned %v", status)
    }
}
```

Эта тестовая функция проверяет, что `StatusNotFound()` основного пакета работает должным образом.

Выполнение двух тестовых функций `testWWW_test.go` приведет к следующим результатам:

```
$ go test testWWW.go testWWW_test.go -v --count=1
=== RUN   TestCheckStatusOK
--- PASS: TestCheckStatusOK (0.00s)
=== RUN   TestStatusNotFound
--- PASS: TestStatusNotFound (0.00s)
PASS
ok   command-line-arguments  (cached)
```

Создание веб-клиента на Go

В этом разделе вы узнаете больше о разработке веб-клиентов на Go. Утилита веб-клиента, которую мы рассмотрим, называется `webClient.go`. Разделим ее на четыре части.

Первая часть `webClient.go` содержит следующий код Go:

```
package main

import (
    "fmt"
    "io"
    "net/http"
    "os"
    "path/filepath"
)
```

Во второй части webClient.go мы читаем URL-адрес, заданный в качестве аргумента командной строки:

```
func main() {
    if len(os.Args) != 2 {
        fmt.Printf("Usage: %s URL\n", filepath.Base(os.Args[0]))
        return
    }

    URL := os.Args[1]
```

Главное происходит в третьей части кода webClient.go:

```
    data, err := http.Get(URL)

    if err != nil {
        fmt.Println(err)
        return
```

Всю работу выполняет функция http.Get(). Это очень удобно, если мы не хотим иметь дело с параметрами и опциями. Однако этот вариант вызова не обеспечивает гибкости. Обратите внимание, что http.Get() возвращает переменную http.Response.

Остальной код Go выглядит так:

```
    } else {
        defer data.Body.Close()
        _, err := io.Copy(os.Stdout, data.Body)
        if err != nil {
            fmt.Println(err)
            return
        }
    }
}
```

Здесь содержимое поля Body структуры http.Response копируется в стандартный поток вывода.

Выполнение webClient.go выведет следующий результат (обратите внимание, что здесь представлена лишь небольшая часть информации):

```
$ go run webClient.go http://www.mtsoukalos.eu/ | head -20
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML+RDFa 1.0//EN"
  "http://www.w3.org/MarkUp/DTD/xhtml-rdfa-1.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en"
version="XHTML+RDFa 1.0" dir="ltr"
  xmlns:content="http://purl.org/rss/1.0/modules/content/"
  xmlns:dc="http://purl.org/dc/terms/"
  xmlns:foaf="http://xmlns.com/foaf/0.1/"
  xmlns:og="http://ogp.me/ns#"
  xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
  xmlns:sioc="http://rdfs.org/sioc/ns#"
  xmlns:sioct="http://rdfs.org/sioc/types#"
```

```
  xmlns:skos="http://www.w3.org/2004/02/skos/core#"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema#">
<head profile="http://www.w3.org/1999/xhtml/vocab">
  <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<meta name="viewport" content="width=device-width, initial-scale=1" />
<link rel="shortcut icon" href="http://www.mtsoukalos.eu/misc/favicon.ico"
type="image/vnd.microsoft.icon" />
<meta name="HandheldFriendly" content="true" />
<meta name="MobileOptimized" content="width" />
<meta name="Generator" content="Drupal 7 (http://drupal.org)" />
```

Основная проблема `webClient.go` заключается в том, что процесс практически не контролируется — вы либо получаете весь вывод HTML, либо вовсе ничего!

Как усовершенствовать наш веб-клиент Go

Поскольку веб-клиент, созданный ранее в разделе, довольно примитивен и совершенно не гибок, в этом подразделе показан более элегантный способ чтения URL-адресов без помощи функции `http.Get()` и с дополнительными параметрами. Утилита, которую мы рассмотрим, называется `advancedWebClient.go`. Разделим ее на пять частей.

Первый фрагмент файла `advancedWebClient.go` содержит следующий код Go:

```go
package main

import (
    "fmt"
    "net/http"
    "net/http/httputil"
    "net/url"
    "os"
    "path/filepath"
    "strings"
    "time"
)
```

Вторая часть файла `advancedWebClient.go` выглядит так:

```go
func main() {
    if len(os.Args) != 2 {
        fmt.Printf("Usage: %s URL\n", filepath.Base(os.Args[0]))
        return
    }

    URL, err := url.Parse(os.Args[1])
    if err != nil {
        fmt.Println("Error in parsing:", err)
        return
    }
```

Третья часть файла `advancedWebClient.go` содержит следующий код Go:

```
c := &http.Client{
    Timeout: 15 * time.Second,
}
request, err := http.NewRequest("GET", URL.String(), nil)
if err != nil {
    fmt.Println("Get:", err)
    return
}
httpData, err := c.Do(request)
if err != nil {
    fmt.Println("Error in Do():", err)
    return
}
```

Функция `http.NewRequest()` возвращает объект `http.Request` с указанием метода, URL-адреса и, возможно, тела запроса. Функция `http.Do()` отправляет HTTP-запрос (`http.Request`) посредством `http.Client` и получает HTTP-ответ (`http.Response`). Таким образом, `http.Do()` делает то же самое, что `http.Get()`, но более подробно.

Строку `"GET"` в вызове `http.NewRequest()` можно заменить на `http.MethodGet`.

Четвертая часть файла `advancedWebClient.go` содержит следующий код Go:

```
fmt.Println("Status code:", httpData.Status)
header, _ := httputil.DumpResponse(httpData, false)
fmt.Print(string(header))

contentType := httpData.Header.Get("Content-Type")
characterSet := strings.SplitAfter(contentType, "charset=")
if len(characterSet) > 1 {
    fmt.Println("Character Set:", characterSet[1])
}

if httpData.ContentLength == -1 {
    fmt.Println("ContentLength is unknown!")
} else {
    fmt.Println("ContentLength:", httpData.ContentLength)
}
```

В этом коде показано, как начать поиск ответа сервера, чтобы найти то, что мы хотим.

Последняя часть утилиты `advancedWebClient.go` выглядит так:

```
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 этом коде показано, как можно определить размер HTTP-ответа сервера. Если мы хотим вывести на экран HTML-код, то можем вывести содержимое буферной переменной `r`.

Выполнение `advancedWebClient.go` для посещения веб-страницы приведет к таким результатам — намного более подробным, чем раньше:

```
$ go run advancedWebClient.go http://www.mtsoukalos.eu
Status code: 200 OK
HTTP/1.1 200 OK
Accept-Ranges: bytes
Age: 0
Cache-Control: no-cache, must-revalidate
Connection: keep-alive
Content-Language: en
Content-Type: text/html; charset=utf-8
Date: Sat, 24 Mar 2018 18:52:17 GMT
Expires: Sun, 19 Nov 1978 05:00:00 GMT
Server: Apache/2.4.25 (Debian) PHP/5.6.33-0+deb8u1 mod_wsgi/4.5.11
Python/2.7
Vary: Accept-Encoding
Via: 1.1 varnish (Varnish/5.0)
X-Content-Type-Options: nosniff
X-Frame-Options: SAMEORIGIN
X-Generator: Drupal 7 (http://drupal.org)
X-Powered-By: PHP/5.6.33-0+deb8u1
X-Varnish: 886025
Character Set: utf-8
ContentLength is unknown!
EOF
Calculated response data length: 50176
```

Выполнение `advancedWebClient.go` для посещения другого URL-адреса выдаст немного другой результат:

```
$ go run advancedWebClient.go http://www.google.com
Status code: 200 OK
HTTP/1.1 200 OK
Cache-Control: private, max-age=0
Content-Type: text/html; charset=ISO-8859-7
Date: Sat, 24 Mar 2018 18:52:38 GMT
Expires: -1
P3p: CP="This is not a P3P policy! See g.co/p3phelp for more info."
Server: gws
Set-Cookie: 1P_JAR=2018-03-24-18; expires=Mon, 23-Apr-2018 18:52:38 GMT;
```

```
path=/; domain=.google.gr
Set-Cookie:
NID=126=csX1_koD30SJcC_ljAfcM2V8kTfRkppmAamLjINLfclracMxuk6JGe4glc0Pjs8uD00
bqGaxkSW-J-ZNDJexG2ZX9pNB9E_dRc2y1KZ05V7pk0bOczE2FtS1zb50Uofl; expires=Sun,
23-Sep-2018 18:52:38 GMT; path=/; domain=.google.gr; HttpOnly
X-Frame-Options: SAMEORIGIN
X-Xss-Protection: 1; mode=block
Character Set: ISO-8859-7
ContentLength in unknown!
EOF
Calculated response data length: 10240
```

Если попробовать выполнить `advancedWebClient.go` для некорректного URL-адреса, то получим следующий результат:

```
$ go run  advancedWebClient.go http://www.google
Error in  Do(): Get http://www.google: dial tcp: lookup www.google: no such host
$ go run  advancedWebClient.go www.google.com
Error in  Do(): Get www.google.com: unsupported protocol scheme ""
```

Не стесняйтесь вносить в `advancedWebClient.go` любые изменения, чтобы результат соответствовал вашим требованиям!

Задержки HTTP-соединений

Из этого раздела вы узнаете, как разрывать сетевые соединения, выполнение которых занимает слишком много времени. Напомню, что подобный способ рассмотрен в главе 10 на примере стандартного Go-пакет `context`. Этот метод был представлен в файле `useContext.go`.

Способ, описанный в данном разделе, реализован гораздо проще. Соответствующий код хранится в файле `clientTimeOut.go`. Разделим файл на четыре части. Эта утилита принимает два аргумента командной строки: URL-адрес и время ожидания в секундах. Обратите внимание, что второй параметр является необязательным.

Первая часть кода `clientTimeOut.go` выглядит так:

```
package main

import (
    "fmt"
    "io"
    "net"
    "net/http"
    "os"
    "path/filepath"
    "strconv"
    "time"
)

var timeout = time.Duration(time.Second)
```

Второй фрагмент `clientTimeOut.go` содержит следующий код Go:

```
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
}
```

Мы рассмотрим `SetDeadline()` более подробно в следующем подразделе. Функция `Timeout()` будет использоваться полем `Dial` переменной `http.Transport`.

В третьей части `clientTimeOut.go` содержится следующий код:

```
func main() {
    if len(os.Args) == 1 {
        fmt.Printf("Usage: %s URL TIMEOUT\n", filepath.Base(os.Args[0]))
        return
    }

    if len(os.Args) == 3 {
        temp, err := strconv.Atoi(os.Args[2])
        if err != nil {
            fmt.Println("Using Default Timeout!")
        } else {
            timeout = time.Duration(time.Duration(temp) * time.Second)
        }
    }

    URL := os.Args[1]
    t := http.Transport{ Dial: Timeout, }
```

Остальной код Go утилиты `clientTimeOut.go` выглядит так:

```
    client := http.Client{
        Transport: &t,
    }
    data, err := client.Get(URL)
    if err != nil {
        fmt.Println(err)
        return
    } else {
        defer data.Body.Close()
        _, err := io.Copy(os.Stdout, data.Body)
        if err != nil {
            fmt.Println(err)
            return
        }
    }
}
```

Мы протестируем веб-клиент `clientTimeOut.go` с помощью веб-сервера `slowWWW.go`, разработанного в главе 10.

Если дважды запустить `clientTimeOut.go`, то получим следующее:

```
$ go run clientTimeOut.go http://localhost:8001
Serving: /
Delay: 0
$ go run clientTimeOut.go http://localhost:8001
Get http://localhost:8001: read tcp [::1]:57397->[::1]:8001: i/o timeout
```

Как видно из полученных данных, первый запрос без проблем подключился к нужному веб-серверу. Однако второй запрос `http.Get()` занял больше времени, чем планировалось, его время ожидания истекло, и он был прерван.

Подробнее о SetDeadline

Функция `SetDeadline()` используется в пакете `net` для того, чтобы задать максимальную длительность операций чтения и записи для данного сетевого соединения. Специфика работы функции `SetDeadline()` такова, что ее необходимо вызвать перед любой операцией чтения или записи. Следует помнить, что максимальная длительность операций необходима в Go для реализации принудительного прерывания операций. Поэтому не нужно переопределять период ожидания каждый раз, когда приложение получает или отправляет данные.

Установка периода ожидания на стороне сервера

В этом подразделе вы узнаете, как устанавливать период ожидания соединения на стороне сервера. Это необходимо сделать потому, что в некоторых ситуациях клиентам для завершения HTTP-соединения требуется намного больше времени, чем ожидалось. Обычно это происходит по одной из двух причин: из-за ошибок в клиентском программном обеспечении или когда серверный процесс подвергается атаке!

Технология, которую мы рассмотрим, реализована в программе `serverTimeOut.go`, которую мы разделим на четыре части. Первая часть `serverTimeOut.go` выглядит так:

```
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)
}
```

Во второй части `serverTimeOut.go` содержится следующий код Go:

```
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)
}
```

Третий фрагмент `serverTimeOut.go` содержит следующий код Go:

```
func main() {
    PORT := ":8001"
    arguments := os.Args
    if len(arguments) == 1 {
        fmt.Printf("Listening on http://0.0.0.0%s\n", PORT)
    } else {
        PORT = ":" + arguments[1]
        fmt.Printf("Listening on http://0.0.0.0%s\n", PORT)
    }

    m := http.NewServeMux()
    srv := &http.Server{
        Addr:         PORT,
        Handler:         m,
        ReadTimeout:     3 * time.Second,
        WriteTimeout:    3 * time.Second,
    }
```

В данном случае мы использовали структуру `http.Server`, поля которой поддерживают два вида задержек. Первый из них называется `ReadTimeout`, а второй — `WriteTimeout`. Значение поля `ReadTimeout` определяет максимальную продолжительность, разрешенную для операций чтения всего запроса, включая его тело.

Значение поля `WriteTimeout` устанавливает максимально допустимое время ожидания ответа. Проще говоря, это время от конца прочитанного заголовка запроса до конца записи ответа.

Остальной код Go программы `serverTimeOut.go` выглядит так:

```
    m.HandleFunc("/time", timeHandler)
    m.HandleFunc("/", myHandler)

    err := srv.ListenAndServe()
    if err != nil {
```

```
        fmt.Println(err)
        return
    }
}
```

Теперь мы выполним программу `serverTimeOut.go` и будем взаимодействовать с ней с помощью `nc(1)`:

```
$ go run serverTimeOut.go
Listening on http://0.0.0.0:8001
```

Команду `nc(1)`, которая в данном случае играет роль фиктивного HTTP-клиента, который подключается к `serverTimeOut.go`, необходимо выполнить в следующем формате:

```
$ time  nc localhost 8001
real    0m3.012s
user    0m0.001s
sys     0m0.002s
```

Поскольку мы не давали никаких команд, HTTP-сервер завершил соединение. Утилита `time(1)` позволяет узнать время, которое потребовалось серверу для того, чтобы закрыть соединение.

Еще один способ определить период ожидания

В этом подразделе представлен еще один способ задать максимально допустимый период ожидания HTTP-соединения, на этот раз на стороне клиента. Как вы увидите, это самая простая форма определения периода ожидания, потому что здесь достаточно просто использовать объект `http.Client` и присвоить его полю `Timeout` желаемое значение периода ожидания.

Утилита, на примере которой мы рассмотрим последний способ определения периода ожидания, называется `anotherTimeOut.go`. Разделим ее на четыре части.

Первая часть `anotherTimeOut.go` выглядит так:

```
package main

import (
    "fmt"
    "io"
    "net/http"
    "os"
    "strconv"
    "time"
)

var timeout = time.Duration(time.Second)
```

Вторая часть anotherTimeOut.go содержит следующий код Go:

```
func main() {
    if len(os.Args) == 1 {
        fmt.Println("Please provide a URL")
        return
    }

    if len(os.Args) == 3 {
        temp, err := strconv.Atoi(os.Args[2])
        if err != nil {
            fmt.Println("Using Default Timeout!")
        } else {
            timeout = time.Duration(time.Duration(temp) * time.Second)
        }
    }

    URL := os.Args[1]
```

Третий фрагмент anotherTimeOut.go содержит такой код Go:

```
    client := http.Client{
        Timeout: timeout,
    }
    client.Get(URL)
```

Именно здесь в поле Timeout переменной http.Client определяется период ожидания.

Последняя часть кода anotherTimeOut.go выглядит так:

```
    data, err := client.Get(URL)
    if err != nil {
        fmt.Println(err)
        return
    } else {
        defer data.Body.Close()
        _, err := io.Copy(os.Stdout, data.Body)
        if err != nil {
            fmt.Println(err)
            return
        }
    }
}
```

Выполнение anotherTimeOut.go и взаимодействие с веб-сервером slowWWW.go, разработанным в главе 10, выведет такой результат:

```
$ go run anotherTimeOut.go http://localhost:8001
Get http://localhost:8001: net/http: request canceled (Client.Timeout
exceeded while awaiting headers)
$ go run anotherTimeOut.go http://localhost:8001 15
Serving: /
Delay: 8
```

Инструменты Wireshark и tshark

В этом разделе кратко представлены мощные утилиты *Wireshark* и *tshark*. Wireshark — это графическое приложение, которое является главным инструментом для анализа сетевого трафика практически любого типа. Несмотря на то что Wireshark — очень мощный инструмент, в некоторых случаях нужно что-то более легкое, без графического интерфейса, с возможностью выполнить удаленно. В таких ситуациях можно использовать tshark — версию Wireshark с интерфейсом командной строки.

К сожалению, более подробная информация о Wireshark и tshark не рассматривается в данной главе.

Go и gRPC

gRPC — это протокол, основанный на HTTP/2, который позволяет легко создавать сервисы. gRPC может использовать буферы протокола для определения *языка описания интерфейса*, а также для указания формата передаваемых сообщений. Клиенты и серверы gRPC могут быть написаны на любом языке программирования, при этом клиенты не обязательно должны быть созданы на том же языке, что и их серверы.

Процесс, который здесь описан, состоит из трех этапов. Первый — создание файла описания интерфейса, второй — разработка клиента gRPC, а третий — разработка сервера gRPC, который будет работать с клиентом gRPC.

Определение файла описания интерфейса

Как вы уже знаете из предыдущего раздела, прежде чем начать разрабатывать клиент и сервер gRPC для нашего сервиса, необходимо определить некоторые структуры данных и протоколы, которые они будут использовать.

Для сериализации структурированных данных мы воспользуемся методом *Protocol Buffers (protobuf)*. Поскольку в protobuf используется двоичный формат, он занимает меньше места, чем текстовые форматы сериализации, такие как JSON и XML. Однако данные в формате protobuf необходимо закодировать, чтобы их мог использовать компьютер, и декодировать, чтобы их мог прочесть человек.

Таким образом, чтобы использовать protobuf в приложениях, нам нужно загрузить необходимые инструменты, которые позволят работать с этим форматом, — забавно, что большинство инструментов для protobuf написаны на Go, потому что он отлично подходит для создания инструментов командной строки!

На компьютере с macOS необходимые инструменты можно загрузить с помощью *Homebrew*:

```
$ brew install protobuf
```

Чтобы обеспечить поддержку protobuf в Go, необходимо выполнить еще одну операцию, поскольку protobuf не поддерживает Go по умолчанию. Для этого нужно выполнить следующую команду:

```
$ go get -u github.com/golang/protobuf/protoc-gen-go
```

Кроме прочего, эта команда загружает исполняемый файл `protoc-gen-go` и помещает его в каталог `~/go/bin`, который на моем компьютере является значением переменной `$GOPATH/bin`. Однако, чтобы компилятор `protoc` нашел данный файл, необходимо добавить этот каталог в переменную окружения `PATH`, что в оболочках `bash(1)` и `zsh(1)` можно сделать так:

```
$ export PATH=$PATH:~/go/bin
```

После того как все необходимые инструменты будут настроены, нужно определить структуры и функции, которые будут использоваться клиентом и сервером gRPC. В нашем случае файл описания интерфейса, который хранится под именем `api.proto`, выглядит так:

```
syntax = "proto3";

package message_service;

message Request {
    string text    = 1;
    string subtext = 2;
}

message Response {
    string text    = 1;
    string subtext = 2;
}

service MessageService {
    rpc SayIt (Request) returns (Response);
}
```

Сервер и клиент gRPC, которые мы разрабатываем, будут поддерживать этот протокол, который более или менее определяет простейшую службу обмена сообщениями с двумя основными типами, `Request` и `Response`, и одной функцией `SayIt()`.

Однако мы еще не закончили, поскольку `api.proto` должен обрабатываться и компилироваться инструментом protobuf. В данном случае компилятор protobuf, который находится в `/usr/local/bin/protoc`. Это можно сделать так:

```
$ protoc -I . --go_out=plugins=grpc:. api.proto
```

После выполнения этих команд на вашем компьютере появится дополнительный файл с именем `api.pb.go`:

```
$ ls -l api.pb.go/
-rw-r--r-- 1 mtsouk staff 7320 May 4 18:31 api.pb.go
```

Итак, для Go компилятор protobuf генерирует файлы .pb.go, которые, кроме прочего, содержат тип для каждого типа сообщений в файле описания интерфейса и интерфейс Go, который должен быть реализован на сервере gRPC. Для других языков программирования вместо расширения .pb.go применяются иные файловые расширения.

Файл api.pb.go начинается со следующих строк:

```
// Code generated by protoc-gen-go. DO NOT EDIT.
// source: api.proto

package message_service

import (
    context "context"
    fmt "fmt"
    proto "github.com/golang/protobuf/proto"
    grpc "google.golang.org/grpc"
    codes "google.golang.org/grpc/codes"
    status "google.golang.org/grpc/status"
    math "math"
)
```

Здесь говорится о том, что не следует редактировать файл api.pb.go самостоятельно и что пакет называется message_service. Чтобы ваши программы Go могли найти все файлы, связанные с protobuf, желательно поместить их в ваш собственный репозиторий GitHub. В нашем примере это будет репозиторий https://github.com/mactsouk/protobuf. Это означает, что вам нужно получить этот репозиторий с помощью следующей команды:

```
$ go get github.com/mactsouk/protobuf
```

Если вы однажды обновите api.proto или другой подобный файл на своем компьютере, то не забудьте совершить два действия: во-первых, обновить репозиторий GitHub, а во-вторых, выполнить команду go get -u -v, после которой указать адрес удаленного репозитория на GitHub, чтобы загрузить оттуда обновления на ваш локальный компьютер.

Теперь мы готовы продолжить разработку кода Go для клиента и сервера gRPC.

Обратите внимание, что если на вашем компьютере не окажется необходимого пакета Go, то при попытке скомпилировать файл описания интерфейса выведется сообщение об ошибке:

```
$ protoc -I . --go_out=plugins=grpc:. api.proto
protoc-gen-go: program not found or is not executable
--go_out: protoc-gen-go: Plugin failed with status code 1.
```

gRPC-клиент

В этом подразделе мы разработаем gRPC-клиент на Go, который будет храниться в `gClient.go`. Разделим файл на три части.

Первая часть `gClient.go` выглядит так:

```
package main

import (
    "fmt"
    p "github.com/mactsouk/protobuf"
    "golang.org/x/net/context"
    "google.golang.org/grpc"
)

var port = ":8080"
```

Обратите внимание: вам не нужно загружать внешние пакеты Go, поскольку вы уже сделали это, когда выполнили показанную ниже команду, чтобы скомпилировать файл языка описания интерфейса и создать выходные файлы для Go:

```
$ go get -u github.com/golang/protobuf/protoc-gen-go
```

Учтите, что, когда задан параметр `-u`, `go get` обновляет именованные пакеты вместе с их зависимостями. Если использовать параметр `-v`, то мы будем лучше понимать, что на самом деле происходит, так как при наличии этого параметра команда выводит дополнительную отладочную информацию.

Во второй части `gClient.go` содержится следующий код Go:

```
func AboutToSayIt(ctx context.Context, m p.MessageServiceClient,
                  text string) (*p.Response, error) {
    request := &p.Request{
        Text:    text,
        Subtext:    "New Message!",
    }
    r, err := m.SayIt(ctx, request)
    if err != nil {
        return nil, err
    }
    return r, nil
}
```

Вы можете назвать `AboutToSayIt()` как угодно, но сигнатура функции должна содержать параметр `context.Context`, а также параметр `MessageServiceClient`, чтобы впоследствии можно было вызывать функцию `SayIt()`. Обратите внимание, что для клиента не нужно реализовывать функции языка описания интерфейса — достаточно просто вызвать их.

Последняя часть `gClient.go` выглядит так:

```
func main() {
    conn, err := grpc.Dial(port, grpc.WithInsecure())
```

```
    if err != nil {
        fmt.Println("Dial:", err)
        return
    }

    client := p.NewMessageServiceClient(conn)
    r, err := AboutToSayIt(context.Background(), client, "My Message!")
    if err != nil {
        fmt.Println(err)
    }

    fmt.Println("Response Text:", r.Text)
    fmt.Println("Response SubText:", r.Subtext)
}
```

Чтобы подключиться к gRPC-серверу и создать новый клиент с помощью функции `NewMessageServiceClient()`, нужно вызвать `grpc.Dial()`. Имя последней функции зависит от значения оператора `package` в файле `api.proto`. Сообщение, отправляемое на gRPC-сервер, жестко закодировано в файле `gClient.go`.

Нет смысла запускать gRPC-клиент без gRPC-сервера, поэтому для запуска клиента вам нужно дождаться следующего раздела. Но если вы хотите увидеть, как программа выдаст сообщение об ошибке, то можете попробовать:

```
$ go run gClient.go
rpc error: code = Unavailable desc = all SubConns are in TransientFailure,
latest connection error: connection error: desc = "transport: Error while
dialing dial tcp :8080: connect: connection refused"
panic: runtime error: invalid memory address or nil pointer dereference
[signal SIGSEGV: segmentation violation code=0x1 addr=0x8 pc=0x13d8afe]
goroutine 1 [running]:
main.main()
    /Users/mtsouk/ch12/gRPC/gClient.go:41 +0x22e
exit status 2
```

gRPC-сервер

В этом подразделе вы узнаете, как разработать gRPC-сервер на Go. Программа, которую мы рассмотрим, называется `gServer.go`. Разделим ее на четыре части.

Первая часть `gServer.go` выглядит так:

```
package main

import (
    "fmt"
    p "github.com/mactsouk/protobuf"
    "golang.org/x/net/context"
    "google.golang.org/grpc"
    "net"
)
```

Поскольку Go-пакет с языком описания интерфейса носит довольно длинное имя `message_service`, я ввел для него более короткий и удобный псевдоним `p`.

Пакеты `golang.org/x/net/context` и `google.golang.org/grpc` были загружены ранее, вместе с другими зависимостями пакета `github.com/golang/protobuf/protoc-gen-go`, следовательно, вам не нужно их скачивать.

Начиная с версии Go 1.7, пакет golang.org/x/net/context входит в стандартную библиотеку под именем context. Таким образом, если хотите, можете заменить этот пакет на context.

Во второй части `gServer.go` содержится следующий код Go:

```
type MessageServer struct {
}

var port = ":8080"
```

Эта пустая структура нужна для того, чтобы впоследствии можно было создать gRPC-сервер в коде Go.

В третьей части `gServer.go` реализован интерфейс:

```
func (MessageServer) SayIt(ctx context.Context, r *p.Request) (*p.Response, error)
{
    fmt.Println("Request Text:", r.Text)
    fmt.Println("Request SubText:", r.Subtext)

    response := &p.Response{
        Text:    r.Text,
        Subtext:    "Got it!",
    }

    return response, nil
}
```

Сигнатура функции `SayIt()` зависит от данных в файле языка описания интерфейса и находится в файле `api.pb.go`.

Функция `SayIt()` возвращает содержимое поля `Text` клиенту, если изменяется содержимое поля `Subtext`.

Последняя часть `gServer.go` выглядит так:

```
func main() {
    server := grpc.NewServer()
    var messageServer MessageServer
    p.RegisterMessageServiceServer(server, messageServer)
    listen, err := net.Listen("tcp", port)
    if err != nil {
        fmt.Println(err)
        return
```

```
    }
    fmt.Println("Serving requests...")
    server.Serve(listen)
}
```

Чтобы проверить соединение, вначале нужно запустить `gServer.go`:

```
$ go run gServer.go
Serving requests...
```

Запустив `gClient.go` в то время, пока работает `gServer.go`, получим такой результат:

```
$ go run gClient.go
Response Text: My Message!
Response SubText: Got it!
```

Во время работы `gClient.go` мы увидим следующий вывод от `gServer.go`:

```
Request Text: My Message!
Request SubText: New Message!
```

Программа `gClient.go` завершится автоматически, однако `gServer.go` нужно завершить вручную.

Дополнительные ресурсы

Следующие ресурсы определенно расширят ваш кругозор, поэтому, пожалуйста, найдите время, чтобы с ними ознакомиться:

- официальная веб-страница веб-сервера Apache: http://httpd.apache.org/;
- официальная веб-страница веб-сервера Nginx: http://nginx.org/;
- если вы хотите больше узнать об Интернете и протоколе TCP/IP, его многочисленных сервисах, начните с чтения *RFC*-документов. Один из источников этих документов находится по адресу https://www.rfc-archive.org/;
- посетите сайт Wireshark и tshark по адресу https://www.wireshark.org/, чтобы больше узнать об этих инструментах;
- посетите страницу документации стандартного Go-пакета `net`, которую вы найдете по адресу https://golang.org/pkg/net/. Это один из самых больших документов в официальной документации Go;
- посетите страницу документации Go-пакета `net/http` по адресу https://golang.org/pkg/net/http/;
- если вы хотите создать сайт без написания кода Go, то попробуйте использовать написанную на Go утилиту Hugo. Подробнее о фреймворке Hugo вы узнаете на сайте https://gohugo.io/. Однако любому программисту на Go будет интересно

взглянуть на код Go этого фреймворка, который вы найдете по адресу https://github.com/gohugoio/hugo;

- посетите страницу документации пакета `net/http/httptrace`: https://golang.org/pkg/net/http/httptrace/;
- страницу документации пакета `net/http/pprof` вы найдете по адресу https://golang.org/pkg/net/http/pprof/;
- посетите страницу руководства утилиты командной строки `nc(1)`, чтобы больше узнать о ее возможностях и параметрах командной строки;
- утилита `httpstat`, разработанная Дейвом Чейни, размещена по адресу `https://github.com/davecheney/httpstat`. Это хороший пример использования Go-пакета `net/http/httptrace` для HTTP-трассировки;
- более подробную информацию о веб-сервере Candy вы найдете по адресу https://github.com/caddyserver/cadd;
- чтобы больше узнать о protobuf, посетите страницы https://opensource.google.com/projects/protobuf и https://developers.google.com/protocol-buffers/;
- документацию Protocol Buffers Language Guide вы найдете по адресу https://developers.google.com/protocol-buffers/docs/proto3;
- страницу документации Go-пакета `protobuf` вы найдете по адресу https://github.com/golang/protobuf;
- чтобы получить больше информации об утилите `ab(1)`, посетите ее страницу руководства по адресу https://httpd.apache.org/docs/2.4/programs/ab.html.

Упражнения

- Напишите на Go свой веб-клиент, не заглядывая в код этой главы.
- Объедините код программ `MXrecords.go` и `NSrecords.go`, чтобы получить общую утилиту, которая бы выполняла обе задачи на основе аргументов командной строки.
- Измените код утилит `MXrecords.go` и `NSrecords.go` так, чтобы они принимали в качестве входных данных в том числе и IP-адреса.
- Создайте версию `MXrecords.go` и `NSrecords.go` с помощью пакетов `cobra` и `viper`.
- Создайте свое gRPC-приложение на основе собственного языка описания интерфейса.
- Измените код `gServer.go` таким образом, чтобы использовать горутины и сохранять количество обслуженных клиентов.
- Измените программу `advancedWebClient.go` таким образом, чтобы сохранять HTML-вывод во внешнем файле.

- Попробуйте самостоятельно реализовать на Go простую версию `ab(1)`, применяя горутины и каналы.
- Измените программу `kvWeb.go` таким образом, чтобы она поддерживала операции `DELETE` и `LOOKUP`, присутствующие в исходной версии хранилища «ключ — значение».
- Измените программу `httpTrace.go` таким образом, чтобы она поддерживала флаг, запрещающий выполнение оператора `io.Copy(os.Stdout, response.Body)`.

Резюме

В этой главе вы узнали о коде Go для программирования веб-клиентов и веб-серверов, а также для создания веб-сайта на Go. Вы также познакомились со структурами `http.Response`, `http.Request` и `http.Transport`, которые позволяют определять параметры HTTP-соединения.

Кроме того, вы научились разрабатывать gRPC-приложения и научились с помощью кода Go получать сетевую конфигурацию машины UNIX и выполнять в программе Go DNS-поиск, включая получение NS- и MX-записей домена.

Наконец, я рассказал о Wireshark и tshark, двух очень популярных утилитах, которые позволяют захватывать и, что самое главное, анализировать сетевой трафик. В начале этой главы я также упомянул утилиту `nc(1)`.

В следующей главе этой книги тема сетевого программирования на Go продолжится. Однако на этот раз представлен код Go более низкого уровня, который позволит вам разрабатывать клиенты и серверы TCP, а также процессы клиентов и серверов UDP. Кроме того, вы узнаете о создании клиентов и серверов RCP.

13 Сетевое программирование: создание серверов и клиентов

В предыдущей главе обсуждались темы, связанные с сетевым программированием, такие как разработка HTTP-клиентов, HTTP-серверов и веб-приложений, выполнение DNS-поиска и период ожидания HTTP-соединений.

Эта глава выведет вас на следующий уровень: вы узнаете, как работать с протоколом HTTPS, как программировать собственные TCP-клиенты и TCP-серверы, а также собственные клиенты и серверы UDP.

Кроме того, в этой главе приводятся два примера программирования конкурентного TCP-сервера. Первый пример — относительно простой, так как конкурентный TCP-сервер будет просто вычислять и возвращать числа последовательности Фибоначчи. Однако во втором примере мы возьмем за основу код приложения `keyValue.go` из главы 4 и преобразуем хранилище «ключ — значение» в конкурентное TCP-приложение, которое способно работать без веб-браузера. В этой главе рассмотрены следующие темы:

- работа с HTTPS-трафиком;
- стандартный Go-пакет `net`;
- разработка TCP-клиентов и TCP-серверов;
- разработка конкурентного TCP-сервера;
- разработка UDP-клиентов и UDP-серверов;
- модификация программы `kvSaveLoad.go` из главы 8, чтобы она могла обслуживать запросы через TCP-соединения;
- запуск сервера TCP/IP в образе Docker;
- создание RCP-клиента и RCP-сервера.

Работа с HTTPS-трафиком

Прежде чем мы создадим сервер TCP/IP на Go, вы узнаете, как работать в Go с протоколом HTTPS, который является безопасной версией протокола HTTP. Обратите внимание, что по умолчанию для HTTPS номер TCP-порта — 443, но при желании можно использовать любой другой номер порта, если указать его в URL-адресе.

Создание сертификатов

Чтобы повторить и выполнить примеры кода из этого подраздела, вам сначала необходимо создать сертификаты, которые требует HTTPS. На компьютере с macOS Mojave для этого нужно выполнить следующие команды:

```
$ openssl genrsa -out server.key 2048
Generating RSA private key, 2048 bit long modulus
.....................+++
.......................+++
e is 65537 (0x10001)
$ openssl ecparam -genkey -name secp384r1 -out server.key
$ openssl req -new -x509 -sha256 -key server.key -out server.crt -days 3650
```

Последняя команда задаст вам несколько вопросов, которые здесь не отображаются. Что вводить в ответ — неважно, можно просто оставить большинство ответов пустыми. После выполнения этих команд будут созданы следующие два файла:

```
$ ls -l server.crt
-rw-r--r-- 1 mtsouk   staff  501 May 16 09:42 server.cr
$ ls -l server.key
-rw-r--r-- 1 mtsouk   staff  359 May 16 09:42 server.key
```

Обратите внимание: если сертификат является самозаверяющим, как тот, который мы только что создали, для работы HTTPS-клиента нужно использовать параметр `InsecureSkipVerify: true` в структуре `http.Transport` — скоро вы узнаете, как это сделать.

Теперь нужно создать сертификат для клиента — необходимо выполнить следующую команду:

```
$ openssl req -x509 -nodes -newkey rsa:2048 -keyout client.key -out
client.crt -days 3650 -subj "/"
Generating a 2048 bit RSA private key
........................+++
..........................................+++
writing new private key to 'client.key'
-----
```

Эта команда создаст еще два файла:

```
$ ls -l client.*
-rw-r--r-- 1 mtsouk    staff    924 May 16 22:17 client.crt
-rw-r--r-- 1 mtsouk    staff    1704 May 16 22:17 client.key
```

Теперь мы готовы продолжить. Поговорим о создании HTTPS-клиента.

HTTPS-клиент

В настоящее время большинство веб-сайтов работают не с HTTP, а с HTTPS. Поэтому в данном подразделе вы узнаете, как создать HTTPS-клиент. Программа, которую мы рассмотрим, называется `httpsClient.go`. Разделим ее на три части.

В зависимости от архитектуры развертывания программы Go могут просто передавать HTTP-данные, а некоторые другие сервисы (например, веб-сервер Nginx или облачные сервисы) могут предоставлять данные протокола защиты информации Secure Sockets Layer (SSL).

В первой части `httpsClient.go` содержится следующий код:

```
package main

import (
    "crypto/tls"
    "fmt"
    "io/ioutil"
    "net/http"
    "os"
    "path/filepath"
    "strings"
)
```

Самым важным из этих пакетов является `crypto/tls`, который в соответствии со своей документацией частично реализует протокол безопасности на *транспортном уровне* (Transport Layer Security, TLS) версии 1.2 согласно RFC 5246 и TLS 1.3 согласно RFC 8446.

Вторая часть `httpsClient.go` выглядит так:

```
func main() {
    if len(os.Args) != 2 {
        fmt.Printf("Usage: %s URL\n", filepath.Base(os.Args[0]))
        return
```

```
    }
    URL := os.Args[1]

    tr := &http.Transport{
        TLSClientConfig: &tls.Config{},
    }
    client := &http.Client{Transport: tr}
    response, err := client.Get(URL)
    if err != nil {
        fmt.Println(err)
        return
    }
    defer response.Body.Close()
```

Структура `http.Transport` настроена на использование TLS посредством параметра `TLSClientConfig`, в котором содержится другая структура с именем `tls.Config`, которая в данный момент использует значения по умолчанию.

В последней части HTTPS-клиента `httpsClient.go` содержится следующий код, отвечающий за чтение ответа от HTTPS-сервера и вывод его на экран:

```
    content, _ := ioutil.ReadAll(response.Body)
    s := strings.TrimSpace(string(content))

    fmt.Println(s)
}
```

Выполнение `httpsClient.go` для чтения защищенного веб-сайта приведет к результатам такого вида:

```
$ go run httpsClient.go https://www.google.com
<!doctype html><html itemscope="" itemtype="http://schema.org/WebPage"
lang="el"><head><meta content="text/html; charset=UTF-8" http-
equiv="Content-Type"><meta
content="/images/branding/googleg/1x/googleg_standard_color_128dp.png"
.
.
.
```

Однако в некоторых случаях выполнение `httpsClient.go` может привести к сбою, в зависимости от сертификата сервера:

```
$ go run httpsClient.go https://www.mtsoukalos.eu/
Get https://www.mtsoukalos.eu/: x509: certificate signed by unknown
authority
```

Решение этой проблемы — использование параметра `InsecureSkipVerify: true` при инициализации `http.Transport`.

Если хотите, попробуйте сейчас сделать это сами или дождитесь, когда мы будем рассматривать программу `TLSclient.go`.

Простой HTTPS-сервер

В этом подразделе мы рассмотрим код Go для HTTPS-сервера. Реализация простого HTTPS-сервера хранится в файле `https.go`. Разделим его на три части.

Первая часть `https.go` выглядит так:

```
package main

import (
    "fmt"
    "net/http"
)

var PORT = ":1443"
```

Во второй части `https.go` содержится следующий код:

```
func Default(w http.ResponseWriter, req *http.Request) {
    fmt.Fprintf(w, "This is an example HTTPS server!\n")
}
```

Эта функция будет обрабатывать все входящие HTTPS-соединения.

Последняя часть `https.go` выглядит так:

```
func main() {
    http.HandleFunc("/", Default)
    fmt.Println("Listening to port number", PORT)

    err := http.ListenAndServeTLS(PORT, "server.crt", "server.key", nil)
    if err != nil {
        fmt.Println("ListenAndServeTLS: ", err)
        return
    }
}
```

Функция `ListenAndServeTLS()` аналогична `ListenAndServe()`, использованной в предыдущей главе. Основное различие между ними состоит в том, что `ListenAndServeTLS()` ожидает HTTPS-соединения, в то время как `ListenAndServe()` не обслуживает HTTPS-клиенты. Кроме того, `ListenAndServeTLS()` принимает больше аргументов, чем `ListenAndServe()`, поскольку использует файл сертификата, а также файл ключа.

Выполнение программы `https.go` и подключение к ней клиента `httpsClient.go` приведет к таким результатам для клиента `httpsClient.go`:

```
$ go run httpsClient.go https://localhost:1443
Get https://localhost:1443: x509: certificate is not valid for any names,
but wanted to match localhost
```

Помните, что использование самозаверенного сертификата не позволит `httpsClient.go` подключиться к нашему HTTPS-серверу. Это является проблемой

при реализации как клиента, так и сервера. В данном случае результат `https.go` будет следующим:

```
$ go run https.go
Listening to port number :1443
2019/05/17 10:11:21 http: TLS handshake error from [::1]:56716: remote
error: tls: bad certificate
```

HTTPS-сервер, разработанный в этом разделе, использует HTTPS через SSL, а это не самый безопасный вариант. Лучше использовать TLS — в следующем подразделе рассмотрена реализация на Go HTTPS-сервера, использующего TLS.

Разработка TLS-сервера и TLS-клиента

В этом подразделе мы намерены реализовать TLS-сервер с именем `TLSserver.go`. Рассмотрим его, разделив на четыре части. Этот HTTPS-сервер будет лучше, чем сервер `https.go`, реализованный в предыдущем разделе.

Первая часть `TLSserver.go` выглядит так:

```
package main

import (
    "crypto/tls"
    "crypto/x509"
    "fmt"
    "io/ioutil"
    "net/http"
)

var PORT = ":1443"

type handler struct {
}
```

Во второй части `TLSserver.go` содержится следующий код:

```
func (h *handler) ServeHTTP(w http.ResponseWriter, req *http.Request) {
    w.Write([]byte("Hello world!\n"))
}
```

Это функция-обработчик веб-сервера, которая обрабатывает все клиентские соединения.

Третья часть `TLSserver.go` выглядит так:

```
func main() {
    caCert, err := ioutil.ReadFile("client.crt")
    if err != nil {
        fmt.Println(err)
        return
    }
```

```
    caCertPool := x509.NewCertPool()
    caCertPool.AppendCertsFromPEM(caCert)
    cfg := &tls.Config{
        ClientAuth: tls.RequireAndVerifyClientCert,
        ClientCAs: caCertPool,
    }
```

Go-пакет `x509` анализирует ключи и сертификаты в кодировке *X.509*. Более подробную информацию об этом пакете можно найти по адресу https://golang.org/pkg/crypto/x509/.

Последняя часть `TLSserver.go` содержит следующий код:

```
    srv := &http.Server{
        Addr:       PORT,
        Handler:    &handler{},
        TLSConfig:  cfg,
    }
    fmt.Println("Listening to port number", PORT)
    fmt.Println(srv.ListenAndServeTLS("server.crt", "server.key"))
}
```

Вызов функции `ListenAndServeTLS()` запускает HTTPS-сервер — его полная конфигурация содержится в структуре `http.Server`.

Если запустить `httpsClient.go` для `TLSserver.go`, то получим на стороне клиента такой результат:

```
$ go run httpsClient.go https://localhost:1443
Get https://localhost:1443: x509: certificate is not valid for any names,
but wanted to match localhost
```

В этом случае вывод сервера будет таким:

```
$ go run TLSserver.go
Listening to port number :1443
2019/05/17 10:05:11 http: TLS handshake error from [::1]:56569: remote
error: tls: bad certificate
```

Как уже отмечалось, для того чтобы HTTPS-клиент `httpsClient.go` успешно взаимодействовал с `TLSserver.go`, который использует самозаверенный сертификат, необходимо добавить в структуру `http.Transport` параметр `InsecureSkipVerify: true`. Версия `httpsClient.go`, которая работает с `TLSserver.go` и содержит `InsecureSkipVerify: true`, сохранена в файле `TLSclient.go`. Разделим его на четыре части.

Первая часть `TLSclient.go` выглядит так:

```
package main

import (
    "crypto/tls"
```

```
    "crypto/x509"
    "fmt"
    "io/ioutil"
    "net/http"
    "os"
    "path/filepath"
)
```

Во второй части `TLSclient.go` содержится следующий код:

```
func main() {
    if len(os.Args) != 2 {
        fmt.Printf("Usage: %s URL\n", filepath.Base(os.Args[0]))
        return
    }
    URL := os.Args[1]

    caCert, err := ioutil.ReadFile("server.crt")
    if err != nil {
        fmt.Println(err)
        return
    }
```

Третья часть `TLSclient.go` выглядит так:

```
    caCertPool := x509.NewCertPool()
    caCertPool.AppendCertsFromPEM(caCert)
    cert, err := tls.LoadX509KeyPair("client.crt", "client.key")

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

    client := &http.Client{
        Transport: &http.Transport{
            TLSClientConfig: &tls.Config{
                RootCAs:            caCertPool,
                InsecureSkipVerify: true,
                Certificates:       []tls.Certificate{cert},
            },
        },
    }

    resp, err := client.Get(URL)
    if err != nil {
        fmt.Println(err)
        return
    }
```

Здесь показано использование `InsecureSkipVerify`.

В последней части TLSclient.go содержится следующий код Go:

```
    htmlData, err := ioutil.ReadAll(resp.Body)
    if err != nil {
        fmt.Println(err)
        return
    }

    defer resp.Body.Close()
    fmt.Printf("%v\n", resp.Status)
    fmt.Printf(string(htmlData))
}
```

Если подключить TLSclient.go к TLSserver.go, то получим ожидаемый результат:

```
$ go run TLSclient.go https://localhost:1443
200 OK
Hello world!
```

Если же подключить TLSclient.go к https.go, то также получим ожидаемый результат. Это означает, что TLSclient.go — довольно хорошая реализация HTTPS-клиента:

```
$ go run TLSclient.go https://localhost:1443
200 OK
This is an example HTTPS server!
```

Стандартный Go-пакет net

На этом мы закончим разговор об HTTPS. Пора рассмотреть основные протоколы TCP/IP, такие как TCP, IP и UDP.

Чтобы создать на Go TCP- или UDP-клиент либо сервер, необходимо использовать функции, предлагаемые пакетом `net`. Для подключения к сети в качестве клиента применяется функция `net.Dial()`, а для того, чтобы программа Go принимала сетевые подключения и работала как сервер, — функция `net.Listen()`. Обе функции — и `net.Dial()`, и `net.Listen()` — возвращают значение типа `net.Conn`. Этот тип реализует интерфейсы `io.Reader` и `io.Writer`. Первым параметром обеих функций является тип сети, но на этом их сходство заканчивается.

TCP-клиент

Как вы уже знаете из предыдущей главы, основной характеристикой протокола TCP является надежность. TCP-заголовок каждого пакета включает в себя поля для *порта источника* и *порта приемника*. Эти два поля в сочетании с IP-адресами источника и приемника однозначно идентифицируют каждое TCP-соединение. В этом разделе мы разработаем TCP-клиент, который называется `TCPclient.go`.

Разделим его на четыре части. Первая часть `TCPclient.go` содержит следующий код Go:

```
package main

import (
    "bufio"
    "fmt"
    "net"
    "os"
    "strings"
)
```

Второй фрагмент кода `TCPclient.go` выглядит так:

```
func main() {
    arguments := os.Args
    if len(arguments) == 1 {
        fmt.Println("Please provide host:port.")
        return
    }

    CONNECT := arguments[1]
    c, err := net.Dial("tcp", CONNECT)
    if err != nil {
        fmt.Println(err)
        return
    }
```

Функция `net.Dial()` необходима для подключения к удаленному серверу. Первый параметр функции `net.Dial()` определяет тип сети, а второй — адрес сервера, который также должен включать в себя 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` (UNIX-сокеты), `unixgram` и `unixpacket`.

В третьей части `TCPclient.go` содержится следующий код:

```
    for {
        reader := bufio.NewReader(os.Stdin)
        fmt.Print(">> ")
        text, _ := reader.ReadString('\n')
        fmt.Fprintf(c, text+"\n")
```

Этот код предназначен для получения данных, вводимых пользователем, что можно проверить, прочитав файл `os.Stdin`. Игнорирование значения `error`, возвращаемого функцией `reader.ReadString()`, — идея не очень хорошая. Но здесь мы таким образом немного экономим место. Конечно, никогда не следует поступать так в реальном программном обеспечении.

Последняя часть `TCPclient.go` выглядит так:

```
        message, _ := bufio.NewReader(c).ReadString('\n')
        fmt.Print("->: " + message)
```

```
        if strings.TrimSpace(string(text)) == "STOP" {
            fmt.Println("TCP client exiting...")
            return
        }
    }
}
```

Если для тестирования `TCPclient.go` подключить эту программу к TCP-серверу, реализованному с использованием `netcat(1)`, то получим такой результат:

```
$ go run TCPclient.go 8001
dial tcp: address 8001: missing port in address
$ go run TCPclient.go localhost:8001
>> Hello from TCPclient.go!
->: Hi from nc!
>> STOP
->:
TCP client exiting...
```

Результат выполнения первой команды показывает, что произойдет, если не передать имя хоста как аргумент командной строки `TCPclient.go`. Результат TCP-сервера `netcat(1)`, который должен выводиться в начале, выглядит так:

```
$ nc -l 127.0.0.1 8001
Hello from TCPclient.go!
Hi from nc!
STOP
```

Обратите внимание, что клиент для протокола TCP и UDP может быть очень обобщенным, то есть он может общаться со многими типами серверов, которые поддерживают его протокол. Как вы скоро увидите, это не относится к серверным приложениям, которые должны реализовывать предварительно организованную функциональность, используя заранее согласованный протокол.

Другая версия TCP-клиента

В Go есть еще одно семейство функций, которое также позволяет разрабатывать TCP-клиенты и серверы. В этом подразделе вы узнаете, как запрограммировать TCP-клиент с помощью этих функций.

Наш TCP-клиент называется `otherTCPclient.go`. Разделим его на четыре части. Первый фрагмент кода `otherTCPclient.go` выглядит так:

```
package main

import (
    "bufio"
```

```
    "fmt"
    "net"
    "os"
    "strings"
)
```

Во второй части otherTCPclient.go содержится следующий код:

```
func main() {
    arguments := os.Args
    if len(arguments) == 1 {
        fmt.Println("Please provide a server:port string!")
        return
    }

    CONNECT := arguments[1]

    tcpAddr, err := net.ResolveTCPAddr("tcp4", CONNECT)
    if err != nil {
        fmt.Println("ResolveTCPAddr:", err.Error())
        return
    }
```

Функция net.ResolveTCPAddr() возвращает адрес конечной точки TCP (тип TCPAddr) и может использоваться только для сетей TCP.

В третьей части otherTCPclient.go содержится следующий код:

```
conn, err := net.DialTCP("tcp4", nil, tcpAddr)
if err != nil {
    fmt.Println("DialTCP:", err.Error())
    return
}
```

Функция net.DialTCP() эквивалентна функции net.Dial(), только для сетей TCP.

Остальной код otherTCPclient.go выглядит так:

```
    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
        }
    }
}
```

Выполнение `otherTCPclient.go` во взаимодействии с TCP-сервером приведет к такому результату:

```
$ go run otherTCPclient.go localhost:8001
>> Hello from otherTCPclient.go!
->: Hi from netcat!
>> STOP
->:
TCP client exiting...
```

В данном примере для TCP-сервера использована утилита `netcat(1)`, которая выполняется следующим образом:

```
$ nc -l 127.0.0.1 8001
Hello from otherTCPclient.go!
Hi from netcat!
STOP
```

TCP-сервер

В этом разделе мы разработаем TCP-сервер, который возвращает клиенту текущую дату и время в одном сетевом пакете. На практике это означает, что после установки клиентского соединения сервер получит время и дату из системы UNIX и вернет эти данные клиенту.

Утилита, которую мы рассмотрим, называется `TCPserver.go`. Разделим ее на четыре части.

Первая часть `TCPserver.go` выглядит так:

```
package main

import (
    "bufio"
    "fmt"
    "net"
    "os"
    "strings"
    "time"
)
```

Вторая часть `TCPserver.go` содержит следующий код Go:

```
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()
```

Функция net.Listen() прослушивает соединения. Если ее второй аргумент не содержит IP-адрес, а только номер порта, то net.Listen() будет прослушивать все доступные IP-адреса локального компьютера.

Третий фрагмент TCPserver.go выглядит так:

```
    c, err := l.Accept()
    if err != nil {
        fmt.Println(err)
        return
    }
```

Функция Accept() ожидает следующего соединения и возвращает порождающую переменную Conn. Единственная проблема этого TCP-сервера состоит в том, что он способен обслуживать только первый TCP-клиент, который к нему подключится, потому что вызов Accept() находится за пределами следующего цикла for. Остальной код Go программы TCPserver.go выглядит так:

```
    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))
    }
}
```

Выполнение TCPserver.go во взаимодействии с приложением TCP-клиента приведет к результатам такого вида:

```
$ go run TCPserver.go 8001
-> HELLO
Exiting TCP server!
```

На стороне клиента результат будет таким:

```
$ nc 127.0.0.1 8001
HELLO
2019-05-18T22:50:31+03:00
STOP
```

Когда утилита `TCPserver.go` пытается использовать TCP-порт, который уже занят другим процессом UNIX, появляется такое сообщение об ошибке:

```
$ go run TCPserver.go 9000
listen tcp :9000: bind: address already in use
```

Наконец, если утилита `TCPserver.go` попытается использовать TCP-порт в диапазоне от 1 до 1024, для чего в системах UNIX требуются привилегии `root`, появится такое сообщение об ошибке:

```
$ go run TCPserver.go 80
listen tcp :80: bind: permission denied
```

Другая версия TCP-сервера

В этом подразделе вы познакомитесь с альтернативной реализацией TCP-сервера на Go. На этот раз TCP-сервер реализует *сервис Echo*, который просто возвращает клиенту те же данные, которые были отправлены им. Программа, которую мы рассмотрим, называется `otherTCPserver.go`. Разделим ее на четыре части.

Первая часть `otherTCPserver.go` выглядит так:

```
package main

import (
    "fmt"
    "net"
    "os"
    "strings"
)
```

Во второй части `otherTCPserver.go` содержится следующий код Go:

```
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
    }

    l, err := net.ListenTCP("tcp", s)
    if err != nil {
        fmt.Println(err)
        return
    }
```

Функция net.ListenTCP() — это эквивалент функции net.Listen() для сетей TCP.

Третий фрагмент otherTCPserver.go:

```
    buffer := make([]byte, 1024)
    conn, err := l.Accept()
    if err != nil {
        fmt.Println(err)
        return
    }
```

Остальной код otherTCPserver.go выглядит так:

```
    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
        }

        fmt.Print("> ", string(buffer[0:n-1]))
        _, err = conn.Write(buffer)
        if err != nil {
            fmt.Println(err)
            return
        }
    }
}
```

Выполнение otherTCPserver.go и использование клиента для взаимодействия с этим сервером приведет к следующим результатам:

```
$ go run otherTCPserver.go 8001
> 1
> 2
> Hello!
> Exiting TCP server!
```

На стороне клиента, которым в этом случае является otherTCPclient.go, мы получим такой результат:

```
$ go run otherTCPclient.go localhost:8001
>> 1
->: 1
>> 2
->: 2
>> Hello!
```

```
->: Hello!
>> ->:
>> STOP
->: TCP client exiting...
```

Наконец, я покажу вам, как узнать имя процесса, который прослушивает данный порт TCP или UDP на машине с UNIX. Итак, чтобы узнать, какой процесс использует TCP-порт с номером 8001, нужно выполнить следующую команду:

```
$ sudo lsof -n -i :8001
COMMAND     PID   USER FD TYPE             DEVICE SIZE/OFF NODE NAME
TCPserver 86775 mtsouk 3u Ipv6 0x98d55014e6c9360f      0t0  TCP *:vcom-tunnel
                                                                (LISTEN)
```

UDP-клиент

Если вы уже знаете, как разрабатывать TCP-клиент, то создание UDP-клиента покажется вам намного проще из-за простоты протокола UDP.

Главное различие между протоколами UDP и TCP состоит в том, что протокол UDP ненадежен. Это также означает, что в целом UDP проще, чем TCP, поскольку UDP не должен сохранять состояние UDP-соединения. Другими словами, работа UDP похожа на самонаводящуюся ракету, что в некоторых случаях идеально подходит.

В этом разделе мы рассмотрим утилиту `UDPclient.go`, код которой разделим на четыре фрагмента. Первая часть `UDPclient.go` выглядит так:

```
package main

import (
    "bufio"
    "fmt"
    "net"
    "os"
    "strings"
)
```

Второй фрагмент `UDPclient.go` выглядит следующим образом:

```
func main() {
    arguments := os.Args
    if len(arguments) == 1 {
        fmt.Println("Please provide a host:port string")
        return
    }
    CONNECT := arguments[1]
```

```
    s, err := net.ResolveUDPAddr("udp4", CONNECT)
    c, err := net.DialUDP("udp4", nil, s)

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

    fmt.Printf("The UDP server is %s\n", c.RemoteAddr().String())
    defer c.Close()
```

Функция `net.ResolveUDPAddr()` возвращает адрес конечной точки UDP, который определяется ее вторым параметром. Первый параметр (`udp4`) указывает на то, что программа поддерживает только протокол IPv4.

Использованная здесь функция `net.DialUDP()` представляет собой аналог `net.Dial()` для сетей UDP.

Третий фрагмент `UDPclient.go` содержит следующий код:

```
    for {
        reader := bufio.NewReader(os.Stdin)
        fmt.Print(">> ")
        text, _ := reader.ReadString('\n')
        data := []byte(text + "\n")
        _, err = c.Write(data)
        if strings.TrimSpace(string(data)) == "STOP" {
            fmt.Println("Exiting UDP client!")
            return
        }
```

Этот код требует от пользователя, чтобы он ввел текст, который затем передается на UDP-сервер. Введенный пользователем текст считывается из стандартного потока ввода UNIX с помощью вызова `bufio.NewReader(os.Stdin)`. Метод `Write(data)` отправляет данные по сетевому соединению UDP.

Остальная часть кода Go выглядит так:

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

        buffer := make([]byte, 1024)
        n, _, err := c.ReadFromUDP(buffer)
        if err != nil {
            fmt.Println(err)
            return
        }
        fmt.Printf("Reply: %s\n", string(buffer[0:n]))
    }
}
```

После того как данные клиента отправлены, нужно дождаться данных, переданных с UDP-сервера, которые считываются с помощью функции ReadFromUDP(). Выполнение UDPclient.go и взаимодействие с утилитой netcat(1), которая действует как UDP-сервер, приведет к следующим результатам:

```
$ go run UDPclient.go localhost:8001
The UDP server is 127.0.0.1:8001
>> Hello!
Reply: Hi there!
>> Have to leave - bye!
Reply: OK.
>> STOP
Exiting UDP client!
```

На стороне UDP-сервера результат будет таким:

```
$ nc -v -u -l 127.0.0.1 8001
Hello!
Hi there!
Have to leave - bye!
OK.
STOP
^C
```

Мы нажали Ctrl+C, чтобы завершить выполнение nc(1), так как у nc(1) нет кода, по которому программа бы завершала работу после того, как она получила на входе строку STOP.

Разработка UDP-сервера

В этом разделе мы разработаем UDP-сервер, который будет возвращать своим UDP-клиентам случайные числа от 1 до 1000. Программа, которую мы рассмотрим, называется UDPserver.go, и мы разделим ее на четыре фрагмента.

Первая часть UDPserver.go выглядит так:

```
package main

import (
    "fmt"
    "math/rand"
    "net"
    "os"
    "strconv"
    "strings"
    "time"
)

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

Второй фрагмент UDPserver.go:

```
func main() {
    arguments := os.Args
    if len(arguments) == 1 {
        fmt.Println("Please provide a port number!")
        return
    }

    PORT := ":" + arguments[1]

    s, err := net.ResolveUDPAddr("udp4", PORT)
    if err != nil {
        fmt.Println(err)
        return
    }
```

В третьем фрагменте UDPserver.go содержится следующий код:

```
    connection, err := net.ListenUDP("udp4", s)
    if err != nil {
        fmt.Println(err)
        return
    }

    defer connection.Close()
    buffer := make([]byte, 1024)
    rand.Seed(time.Now().Unix())
```

Функция net.ListenUDP() — это аналог функции net.ListenTCP() для UDP-сетей.

Остальной код Go программы UDPserver.go выглядит так:

```
    for {
        n, addr, err := connection.ReadFromUDP(buffer)
        fmt.Print("-> ", string(buffer[0:n-1]))

        if strings.TrimSpace(string(buffer[0:n])) == "STOP" {
            fmt.Println("Exiting UDP server!")
            return
        }

        data := []byte(strconv.Itoa(random(1, 1001)))
        fmt.Printf("data: %s\n", string(data))
        _, err = connection.WriteToUDP(data, addr)
        if err != nil {
            fmt.Println(err)
            return
        }
    }
}
```

Функция ReadFromUDP() позволяет считывать данные из UDP-соединения используя буфер, который, как и следовало ожидать, является *байтовым срезом*.

Выполнение UDPserver.go и подключение к нему с помощью UDPclient.go приведет к следующим результатам:

```
$ go run UDPserver.go 8001
-> Hello!
data: 156
-> Another random number please :)
data: 944
-> Leaving...
data: 491
-> STOP
Exiting UDP server!
```

На стороне клиента результат будет таким:

```
$ go run UDPclient.go localhost:8001
The UDP server is 127.0.0.1:8001
>> Hello!
Reply: 156
>> Another random number please :)
Reply: 944
>> Leaving...
Reply: 491
>> STOP
Exiting UDP client!
```

Конкурентный TCP-сервер

В этом разделе вы узнаете, как разработать *конкурентный TCP-сервер* с использованием горутин. Для каждого входящего соединения с TCP-сервером программа будет запускать новую горутину, которая и будет обрабатывать этот запрос. Это позволит серверу принимать больше запросов. Следовательно, конкурентный TCP-сервер сможет обслуживать несколько клиентов одновременно.

Задача нашего конкурентного TCP-сервера — принимать положительное целое число и возвращать натуральное число из последовательности Фибоначчи. Если на входе обнаруживается ошибка, то сервер возвращает -1. Поскольку вычисление чисел последовательности Фибоначчи может быть медленным, мы воспользуемся алгоритмом, который был представлен в главе 11 и применялся в программе benchmarkMe.go. Кроме того, на этот раз я подробнее разъясню работу этого алгоритма.

Программа, которую мы рассмотрим, называется fiboTCP.go. Ее код представлен в пяти частях. Поскольку считается целесообразным передавать номер порта

веб-службы в качестве аргумента командной строки, мы в `fiboTCP.go` тоже будем так делать.

В первой части `fiboTCP.go` содержится следующий код Go:

```
package main

import (
    "bufio"
    "fmt"
    "net"
    "os"
    "strconv"
    "strings"
    "time"
)
```

Во второй части `fiboTCP.go` содержится такой код Go:

```
func f(n int) int {
    fn := make(map[int]int)
    for i := 0; i <= n; i++ {
        var f int
        if i <= 2 {
            f = 1
        } else {
            f = fn[i-1] + fn[i-2]
        }
        fn[i] = f
    }
    return fn[n]
}
```

В этом коде мы видим реализацию функции `f()`, которая генерирует натуральные числа, принадлежащие последовательности Фибоначчи. Поначалу этот алгоритм трудно понять, но он очень эффективен и поэтому быстр.

Прежде всего, в функции `f()` используется хеш-таблица с именем `fn`, что довольно необычно при вычислении чисел последовательности Фибоначчи. В функции `f()` также используется цикл `for`, что также довольно необычно. Наконец, в функции `f()` отсутствует рекурсия, что является главной причиной ее высокой скорости.

Идея алгоритма, реализованного в `f()`, который использует метод *динамического программирования*, состоит в том, что всякий раз, когда вычисляется очередное число Фибоначчи, оно помещается в хеш-таблицу `fn`, чтобы впоследствии не пришлось вычислять его снова. Эта простая идея экономит много времени, особенно когда нужно вычислять большие числа Фибоначчи, потому что тогда не приходится вычислять одно и то же число Фибоначчи несколько раз.

Третий фрагмент кода `fiboTCP.go` выглядит так:

```
func handleConnection(c net.Conn) {
    for {
        netData, err := bufio.NewReader(c).ReadString('\n')
        if err != nil {
            fmt.Println(err)
            os.Exit(100)
        }

        temp := strings.TrimSpace(string(netData))
        if temp == "STOP" {
            break
        }

        fibo := "-1\n"
        n, err := strconv.Atoi(temp)
        if err == nil {
            fibo = strconv.Itoa(f(n)) + "\n"
        }
        c.Write([]byte(string(fibo)))
    }
    time.Sleep(5 * time.Second)
    c.Close()
}
```

Функция `handleConnection()` работает со всеми клиентами конкурентного TCP-сервера.

Четвертая часть `fiboTCP.go`:

```
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()
```

Остальной код Go файла `fiboTCP.go` выглядит так:

```
    for {
        c, err := l.Accept()
        if err != nil {
            fmt.Println(err)
```

```
            return
        }
        go handleConnection(c)
    }
}
```

Конкурентность программы реализуется посредством оператора `go handleConnection(c)`, который запускает новую горутину всякий раз, когда из Интернета или локальной сети приходит новый TCP-клиент. Горутина выполняется конкурентно, что дает серверу возможность обслуживать еще больше клиентов.

Выполнение сервера `fiboTCP.go` и взаимодействие с ним посредством запуска `netcat(1)` и `TCPclient.go` на двух разных терминалах выведет такой результат:

```
$ go run fiboTCP.go 9000
n: 10
fibo: 55
n: 0
fibo: 1
n: -1
fibo: 0
n: 100
fibo: 3736710778780434371
n: 12
fibo: 144
n: 12
fibo: 144
```

На стороне `TCPclient.go` результат будет следующим:

```
$ go run TCPclient.go localhost:9000
>> 12
->: 144
>> a
->: -1
>> STOP
->: TCP client exiting...
```

На стороне `netcat(1)` получим такой результат:

```
$ nc localhost 9000
10
55
0
1
-1
0
100
3736710778780434371
ads
-1
STOP
```

Когда клиент отправляет процессу сервера строку `STOP`, горутина, которая обслуживает этот TCP-клиент, прекращает работу, что приводит к завершению соединения.

Наконец, впечатляет то, что оба клиента обслуживаются одновременно. Это можно проверить с помощью следующей команды:

```
$ netstat -anp TCP | grep 9000
tcp4    0 0 127.0.0.1.9000    127.0.0.1.57309   ESTABLISHED
tcp4    0 0 127.0.0.1.57309   127.0.0.1.9000    ESTABLISHED
tcp4    0 0 127.0.0.1.9000    127.0.0.1.57305   ESTABLISHED
tcp4    0 0 127.0.0.1.57305   127.0.0.1.9000    ESTABLISHED
tcp4    0 0 *.9000            *.*               LISTEN
```

Последняя строка результатов этой команды говорит о том, что существует процесс, который прослушивает порт `9000`, следовательно, мы все равно можем подключиться к порту `9000`. Первые две строки говорят о том, что существует клиент, который использует порт `57309` для связи с процессом сервера. Третья и четвертая строки предыдущего вывода подтверждают, что существует другой клиент, который связывается с сервером, прослушивающим порт `9000`. Этот клиент применяет TCP-порт `57305`.

Удобный конкурентный TCP-сервер

Конкурентный TCP-сервер, который мы создали в предыдущем разделе, работает нормально, однако для практического применения он не подходит. Поэтому в данном подразделе вы узнаете, как преобразовать приложение `keyValue.go` из главы 4 в полнофункциональное конкурентное приложение TCP.

Сейчас мы создадим собственный TCP-протокол для сетевого взаимодействия с хранилищем «ключ — значение». Для каждой функции хранилища «ключ — значение» нам понадобится ключевое слово. Для простоты за каждым ключевым словом будут следовать соответствующие данные. Результатом большинства команд будет сообщение об успехе или неудаче операции.

Разработка собственных протоколов на основе TCP или UDP — непростая задача. При разработке нового протокола необходимы исключительная точность и осторожность.

Утилита, представленная в этом разделе, называется `kvTCP.go`. Разделим ее на шесть частей.

Первая часть `kvTCP.go` выглядит так:

```
package main

import (
```

```
    "bufio"
    "encoding/gob"
    "fmt"
    "net"
    "os"
    "strings"
)

type myElement struct {
    Name     string
    Surname  string
    Id       string
}

const welcome = "Welcome to the Key Value store!\n"

var DATA = make(map[string]myElement)
var DATAFILE = "/tmp/dataFile.gob"
```

Во второй части kvTCP.go содержится следующий код Go:

```
func handleConnection(c net.Conn) {
    c.Write([]byte(welcome))
    for {
        netData, err := bufio.NewReader(c).ReadString('\n')
        if err != nil {
            fmt.Println(err)
            return
        }

        command := strings.TrimSpace(string(netData))
        tokens := strings.Fields(command)
        switch len(tokens) {
        case 0:
            continue
        case 1:
            tokens = append(tokens, "")
            tokens = append(tokens, "")
            tokens = append(tokens, "")
            tokens = append(tokens, "")
        case 2:
            tokens = append(tokens, "")
            tokens = append(tokens, "")
            tokens = append(tokens, "")
        case 3:
            tokens = append(tokens, "")
            tokens = append(tokens, "")
        case 4:
            tokens = append(tokens, "")
        }
```

```
switch tokens[0] {
case "STOP":
    err = save()
    if err != nil {
        fmt.Println(err)
    }
    c.Close()
    return
case "PRINT":
    PRINT(c)
case "DELETE":
    if !DELETE(tokens[1]) {
        netData := "Delete operation failed!\n"
        c.Write([]byte(netData))
    } else {
        netData := "Delete operation successful!\n"
        c.Write([]byte(netData))
    }
case "ADD":
    n := myElement{tokens[2], tokens[3], tokens[4]}
    if !ADD(tokens[1], n) {
        netData := "Add operation failed!\n"
        c.Write([]byte(netData))
    } else {
        netData := "Add operation successful!\n"
        c.Write([]byte(netData))
    }
    err = save()
    if err != nil {
        fmt.Println(err)
    }
case "LOOKUP":
    n := LOOKUP(tokens[1])
    if n != nil {
        netData := fmt.Sprintf("%v\n", *n)
        c.Write([]byte(netData))
    } else {
        netData := "Did not find key!\n"
        c.Write([]byte(netData))
    }
case "CHANGE":
    n := myElement{tokens[2], tokens[3], tokens[4]}
    if !CHANGE(tokens[1], n) {
        netData := "Update operation failed!\n"
        c.Write([]byte(netData))
    } else {
        netData := "Update operation successful!\n"
        c.Write([]byte(netData))
    }
    err = save()
```

```
            if err != nil {
                fmt.Println(err)
            }
        default:
            netData := "Unknown command - please try again!\n"
            c.Write([]byte(netData))
        }
    }
}
```

Функция `handleConnection()` связывается с каждым TCP-клиентом и интерпретирует входные данные этого клиента.

Третий фрагмент `kvTCP.go` содержит следующий код Go:

```
func save() error {
    fmt.Println("Saving", DATAFILE)
    err := os.Remove(DATAFILE)
    if err != nil {
        fmt.Println(err)
    }

    saveTo, err := os.Create(DATAFILE)
    if err != nil {
        fmt.Println("Cannot create", DATAFILE)
        return err
    }
    defer saveTo.Close()

    encoder := gob.NewEncoder(saveTo)
    err = encoder.Encode(DATA)
    if err != nil {
        fmt.Println("Cannot save to", DATAFILE)
        return err
    }
    return nil
}

func load() error {
    fmt.Println("Loading", DATAFILE)
    loadFrom, err := os.Open(DATAFILE)
    defer loadFrom.Close()
    if err != nil {
        fmt.Println("Empty key/value store!")
        return err
    }

    decoder := gob.NewDecoder(loadFrom)
    decoder.Decode(&DATA)
    return nil
}
```

Четвертый фрагмент `kvTCP.go` выглядит так:

```
func ADD(k string, n myElement) bool {
    if k == "" {
        return false
    }

    if LOOKUP(k) == nil {
        DATA[k] = n
        return true
    }
    return false
}

func DELETE(k string) bool {
    if LOOKUP(k) != nil {
        delete(DATA, k)
        return true
    }
    return false
}

func LOOKUP(k string) *myElement {
    _, ok := DATA[k]
    if ok {
        n := DATA[k]
        return &n
    } else {
        return nil
    }
}

func CHANGE(k string, n myElement) bool {
    DATA[k] = n
    return true
}
```

Реализация этих функций такая же, как в `keyValue.go`. Ни одна из них не обращается к TCP-клиенту напрямую.

В пятой части `kvTCP.go` содержится следующий код Go:

```
func PRINT(c net.Conn) {
    for k, d := range DATA {
        netData := fmt.Sprintf("key: %s value: %v\n", k, d)
        c.Write([]byte(netData))
    }
}
```

Функция `PRINT()` передает данные прямо TCP-клиенту, по одной строке.

Остальной код Go программы выглядит так:

```
func main() {
    arguments := os.Args
    if len(arguments) == 1 {
        fmt.Println("Please provide a port number!")
        return
    }

    PORT := ":" + arguments[1]
    l, err := net.Listen("tcp", PORT)
    if err != nil {
        fmt.Println(err)
        return
    }
    defer l.Close()

    err = load()
    if err != nil {
        fmt.Println(err)
    }

    for {
        c, err := l.Accept()
        if err != nil {
            fmt.Println(err)
            os.Exit(100)
        }
        go handleConnection(c)
    }
}
```

Выполнение `kvTCP.go` приведет к таким результатам:

```
$ go run kvTCP.go 9000
Loading /tmp/dataFile.gob
Empty key/value store!
open /tmp/dataFile.gob: no such file or directory
Saving /tmp/dataFile.gob
remove /tmp/dataFile.gob: no such file or directory
Saving /tmp/dataFile.gob
Saving /tmp/dataFile.gob
```

В нашем примере роль TCP-клиента для `kvTCP.go` выполняет утилита `netcat(1)`:

```
$ nc localhost 9000
Welcome to the Key Value store!
PRINT
LOOKUP 1
Did not find key!
```

```
ADD 1 2 3 4
Add operation successful!
LOOKUP 1
{2 3 4}
ADD 4 -1 -2 -3
Add operation successful!
PRINT
key: 1 value: {2 3 4}
key: 4 value: {-1 -2 -3}
STOP
```

`kvTCP.go` — конкурентное приложение, которое применяет горутины и способно обслуживать несколько TCP-клиентов одновременно. Однако все эти TCP-клиенты будут взаимодействовать с одними и теми же данными.

Создание образа Docker для TCP/IP-сервера на Go

В этом разделе вы узнаете, как поместить файл `kvTCP.go` в образ Docker и использовать его оттуда. Это очень удобный способ работы с приложением TCP/IP, поскольку образ Docker легко переносится на другие машины и развертывается в Kubernetes.

Как и следовало ожидать, все начинается с `Dockerfile` с таким содержимым:

```
FROM golang:latest

RUN mkdir /files
COPY kvTCP.go /files
WORKDIR /files

RUN go build -o /files/kvTCP kvTCP.go
ENTRYPOINT ["/files/kvTCP","80"]
```

После этого нужно создать образ Docker:

```
$ docker build -t kvtcp:latest .
Sending build context to Docker daemon 6.656kB
Step 1/6 : FROM golang:latest
    ---> 7ced090ee82e
Step 2/6 : RUN mkdir /files
    ---> Running in bbbbada6271f
Removing intermediate container bbbbada6271f
    ---> 5b0a621eee29
Step 3/6 : COPY kvTCP.go /files
    ---> 4aab441b14c2
Step 4/6 : WORKDIR /files
    ---> Running in 7185606bed2e
```

```
Removing intermediate container 7185606bed2e
    ---> 744e9800fdba
Step 5/6 : RUN go build -o /files/kvTCP kvTCP.go
    ---> Running in f44fcbc8951b
Removing intermediate container f44fcbc8951b
    ---> a8d00c7ead13
Step 6/6 : ENTRYPOINT ["/files/kvTCP","80"]
    ---> Running in ec3227170e09
Removing intermediate container ec3227170e09
    ---> b65ba728849a
Successfully built b65ba728849a
Successfully tagged kvtcp:latest
```

Выполнив команду `docker images`, мы убедимся, что создали желаемый образ Docker:

```
$ docker images
REPOSITORY                   TAG    IMAGE ID       CREATED          SIZE
kvtcp                       latest  b65ba728849a   26 minutes ago   777MB
landoop/kafka-lenses-dev    latest  289093ceee7b   2 days ago      1.37GB
golang                      latest  7ced090ee82e   9 days ago       774MB
```

Затем нужно выполнить этот образ Docker. Следует указать дополнительные параметры, чтобы предоставить номера портов, которые мы хотим сделать доступными извне:

```
$ docker run -d -p 5801:80 kvtcp:latest
af9939992a25cb5ccf405b1b97b9c813fb4cb3f3a2e5b13942db637709c8cee2
```

Последняя команда открывает порт `80` из образа Docker для локального компьютера с помощью порта `5801` локального компьютера.

Теперь мы сможем применить TCP-сервер, расположенный в образе Docker, так:

```
$ nc 127.0.0.1 5801
Welcome to the Key Value store!
```

Обратите внимание, что, если хранить данные вне образа Docker, по окончании работы с ним данные будут потеряны.

Эта возможность Docker позволяет использовать любой номер порта, через который мы хотим получить доступ к TCP-серверу в образе Docker. Но если вы попытаетесь снова назначить порт с номером `80`, то получите следующее сообщение об ошибке:

```
$ docker run -d -p 5801:80 kvtcp:latest
709d44be8668284b101d7dfc253938d13e6797d812821838aa5ab18ea48527ec
docker: Error response from daemon: driver failed programming external
connectivity on endpoint eager_nobel
(fa9d43d3c129734576e824753703d8ac3ff51bcdcdc20e6937b30f3bcfefeff7): Bind
for 0.0.0.0:5801 failed: port is already allocated.
```

В этом сообщении об ошибке говорится, что данный порт уже выделен и поэтому не может быть использован снова. Однако есть возможность обойти это ограничение, которую можно проиллюстрировать с помощью следующей команды:

```
$ docker run -d -p 5801:80 -p 2000:80 kvtcp:latest
5cbb17c5bbf720eaa5ce0f1e11cc73dbe5bef3cc925b7936ae1b97e245d54ae8
```

Здесь в команде запуска Docker создается отображение номера порта `80` из образа Docker на два внешних TCP-порта (`5801` и `2000`). Обратите внимание, что здесь работает только один сервер `kvTCP` и существует только одна копия данных, несмотря на то что эти данные доступны через несколько портов.

Мы можем это проверить с помощью команды `docker ps`:

```
$ docker ps
CONTAINER ID  IMAGE        COMMAND            CREATED STATUS
    PORTS                                        NAMES
5cbb17c5bbf7  kvtcp:latest "/files/kvTCP 80"  3 seconds ago Up 2 seconds
    0.0.0.0:2000->80/tcp, 0.0.0.0:5801->80/tcp   compassionate_morse
```

Дистанционный вызов процедур

Дистанционный вызов процедур (Remote Procedure Call, RPC) — это клиент-серверный механизм для межпроцессного взаимодействия, применяющий протокол TCP/IP. RPC-клиент и RPC-сервер, которые мы разработаем, основаны на пакете, который называется `sharedRPC.go`:

```
package sharedRPC

type MyFloats struct {
    A1, A2 float64
}

type MyInterface interface {
    Multiply(arguments *MyFloats, reply *float64) error
    Power(arguments *MyFloats, reply *float64) error
}
```

Пакет `sharedRPC` определяет интерфейс `MyInterface` и структуру `MyFloats`, которые будут использоваться и клиентом, и сервером. Но реализует этот интерфейс только RPC-сервер.

После этого нам нужно установить пакет `sharedRPC.go`, выполнив следующие команды:

```
$ mkdir -p ~/go/src/sharedRPC
$ cp sharedRPC.go ~/go/src/sharedRPC/
$ go install sharedRPC
```

RPC-клиент

В этом подразделе мы рассмотрим код Go RPC-клиента, который хранится в файле `RPCclient.go`. Разделим его на четыре части.

Первая часть `RPCclient.go` выглядит так:

```
package main

import (
    "fmt"
    "net/rpc"
    "os"
    "sharedRPC"
)
```

Во втором фрагменте `RPCclient.go` содержится следующий код Go:

```
func main() {
    arguments := os.Args
    if len(arguments) == 1 {
        fmt.Println("Please provide a host:port string!")
        return
    }

    CONNECT := arguments[1]
    c, err := rpc.Dial("tcp", CONNECT)
    if err != nil {
        fmt.Println(err)
        return
    }
```

Обратите внимание, что для подключения к RPC-серверу вместо функции `net.Dial()` применяется `rpc.Dial()`, несмотря на то что RCP-сервер использует протокол TCP.

Третий фрагмент `RPCclient.go` выглядит так:

```
args := sharedRPC.MyFloats{16, -0.5}
var reply float64

err = c.Call("MyInterface.Multiply", args, &reply)
if err != nil {
    fmt.Println(err)
    return
}
fmt.Printf("Reply (Multiply): %f\n", reply)
```

С помощью функции `Call()` между RPC-клиентом и RPC-сервером передаются имена функций, их аргументы и результаты вызовов функций, поскольку RPC-клиент ничего не знает о фактической реализации этих функций.

Остальной код Go программы `RPCclient.go` выглядит так:

```
    err = c.Call("MyInterface.Power", args, &reply)
    if err != nil {
        fmt.Println(err)
        return
    }
    fmt.Printf("Reply (Power): %f\n", reply)
}
```

Если запустить `RPCclient.go`, не запуская RPC-сервер, то выведется следующее сообщение об ошибке:

```
$ go run RPCclient.go localhost:1234
dial tcp [::1]:1234: connect: connection refused
```

RPC-сервер

Код RPC-сервера хранится в файле `RPCserver.go`. Мы его рассмотрим, разделив на пять частей.

Первая часть `RPCserver.go` выглядит так:

```
package main

import (
    "fmt"
    "math"
    "net"
    "net/rpc"
    "os"
    "sharedRPC"
)
```

Второй фрагмент `RPCserver.go` содержит следующий код Go:

```
type MyInterface struct{}

func Power(x, y float64) float64 {
    return math.Pow(x, y)
}

func (t *MyInterface) Multiply(arguments *sharedRPC.MyFloats, reply *float64) error {
    *reply = arguments.A1 * arguments.A2
    return nil
}

func (t *MyInterface) Power(arguments *sharedRPC.MyFloats, reply *float64) error {
    *reply = Power(arguments.A1, arguments.A2)
    return nil
}
```

В этом коде Go RPC-сервер реализует требуемый интерфейс, а также вспомогательную функцию `Power()`.

Третий фрагмент `RPCserver.go`:

```
func main() {
    PORT := ":1234"
    arguments := os.Args
    if len(arguments) != 1 {
        PORT = ":" + arguments[1]
    }
```

Четвертая часть `RPCserver.go` содержит следующий код:

```
    myInterface := new(MyInterface)
    rpc.Register(myInterface)
    t, err := net.ResolveTCPAddr("tcp4", PORT)
    if err != nil {
        fmt.Println(err)
        return
    }
    l, err := net.ListenTCP("tcp4", t)
    if err != nil {
        fmt.Println(err)
        return
    }
```

Именно применение функции `rpc.Register()` делает эту программу RPC-сервером. Однако, поскольку RPC-сервер использует протокол TCP, ему все равно необходимо вызывать функции `net.ResolveTCPAddr()` и `net.ListenTCP()`.

Остальная часть кода Go из файла `RPCclient.go` выглядит так:

```
    for {
        c, err := l.Accept()
        if err != nil {
            continue
        }
        fmt.Printf("%s\n", c.RemoteAddr())
        rpc.ServeConn(c)
    }
}
```

Функция `RemoteAddr()` возвращает IP-адрес и номер порта, которые необходимы для связи с RPC-клиентом. Функция `rpc.ServeConn()` обслуживает RPC-клиента.

Выполнение `RPCserver.go` с ожиданием `RPCclient.go` приведет к следующим результатам:

```
$ go run RPCserver.go
127.0.0.1:52289
```

При выполнении RPCclient.go получим такой результат:

```
$ go run RPCclient.go localhost:1234
Reply (Multiply): -8.000000
Reply (Power): 0.250000
```

Низкоуровневое сетевое программирование

Структура http.Transport дает возможность изменять различные низкоуровневые параметры сетевого соединения, однако мы можем написать код Go, который позволит читать необработанные данные сетевых пакетов.

Здесь есть два момента. Во-первых, сетевые пакеты поступают в двоичном формате, так что нам придется не просто принимать все пакеты подряд, а отслеживать определенные типы сетевых пакетов. Проще говоря, при чтении сетевых пакетов необходимо заранее указать один или несколько протоколов, которые поддерживает приложение. Во-вторых, чтобы отправить сетевой пакет, нам придется создать его самостоятельно.

Следующая утилита, которую мы рассмотрим, называется lowLevel.go. Разделим ее на три части. Обратите внимание, что lowLevel.go перехватывает пакеты интернет-протокола управления сообщениями *ICMP* (Internet Control Message Protocol), которые используют протокол IPv4, и выводит их содержимое. Также учтите, что в целях безопасности для доступа к необработанным сетевым данным необходимы привилегии пользователя root.

Первый фрагмент lowLevel.go выглядит так:

```
package main

import (
    "fmt"
    "net"
)
```

Во второй части lowLevel.go содержится следующий код Go:

```
func main() {
    netaddr, err := net.ResolveIPAddr("ip4", "127.0.0.1")
    if err != nil {
        fmt.Println(err)
        return
    }
    conn, err := net.ListenIP("ip4:icmp", netaddr)
    if err != nil {
        fmt.Println(err)
        return
    }
```

Протокол ICMP указан во второй части первого аргумента (`ip4: icmp`) функции `net.ListenIP()`. Более того, часть `ip4` сообщает утилите о том, что следует перехватывать только трафик IPv4.

Остальная часть `lowLevel.go` содержит следующий код Go:

```
    buffer := make([]byte, 1024)
    n, _, err := conn.ReadFrom(buffer)
    if err != nil {
        fmt.Println(err)
        return
    }

    fmt.Printf("% X\n", buffer[0:n])
}
```

Согласно этому коду `lowLevel.go` читает только один сетевой пакет, поскольку здесь нет цикла `for`.

Протокол ICMP используется в утилитах `ping(1)` и `traceroute(1)`, поэтому один из способов создания ICMP-трафика — воспользоваться одним из этих инструментов. При запущенной утилите `lowLevel.go` сетевой трафик на обеих машинах UNIX будет генерироваться с помощью следующих команд:

```
$ ping -c 5 localhost
PING localhost (127.0.0.1): 56 data
64 bytes from 127.0.0.1: icmp_seq=0 ttl=64 time=0.037 ms
64 bytes from 127.0.0.1: icmp_seq=1 ttl=64 time=0.038 ms
64 bytes from 127.0.0.1: icmp_seq=2 ttl=64 time=0.117 ms
64 bytes from 127.0.0.1: icmp_seq=3 ttl=64 time=0.052 ms
64 bytes from 127.0.0.1: icmp_seq=4 ttl=64 time=0.049 ms
--- localhost ping statistics ---
5 packets transmitted, 5 packets received, 0.0% packet loss
round-trip min/avg/max/stddev = 0.037/0.059/0.117/0.030 ms
$ traceroute localhost
traceroute to localhost (127.0.0.1), 64 hops max, 52 byte packets
1 localhost (127.0.0.1) 0.255 ms 0.048 ms 0.067 ms
```

Выполнение `lowLevel.go` на компьютере с macOS Mojave с привилегиями пользователя `root` приведет к таким результатам:

```
$ sudo go run lowLevel.go
03 03 CD DA 00 00 00 00 45 00 34 00 B4 0F 00 00 01 11 00 00 7F 00 00 01 7F
00 00 01 B4 0E 82 9B 00 20 00 00
$ sudo go run lowLevel.go
00 00 0B 3B 20 34 00 00 5A CB 5C 15 00 04 32 A9 08 09 0A 0B 0C 0D 0E 0F 10
11 12 13 14 15 16 17 18 19 1A 1B 1C 1D 1E 1F 20 21 22 23 24 25 26 27 28 29
2A 2B 2C 2D 2E 2F 30 31 32 33 34 35 36 37
```

Первый пример вывода генерируется командой `ping(1)`, а второй — `traceroute(1)`.

Если запустить lowLevel.go на компьютере с Debian Linux, то получим следующий результат:

```
$ uname -a
Linux mail 4.14.12-x86_64-linode92 #1 SMP Fri Jan 5 15:34:44 UTC 2018
x86_64 GNU/Linux
# go run lowLevel.go
08 00 61 DD 3F BA 00 01 9A 5D CB 5A 00 00 00 00 26 DC 0B 00 00 00 00 00 10
11 12 13 14 15 16 17 18 19 1A 1B 1C 1D 1E 1F 20 21 22 23 24 25 26 27 28 29
2A 2B 2C 2D 2E 2F 30 31 32 33 34 35 36 37
# go run lowLevel.go
03 03 BB B8 00 00 00 00 45 00 00 3C CD 8D 00 00 01 11 EE 21 7F 00 00 01 7F
00 00 01 CB 40 82 9A 00 28 FE 3B 40 41 42 43 44 45 46 47 48 49 4A 4B 4C 4D
4E 4F 50 51 52 53 54 55 56 57 58 59 5A 5B 5C 5D 5E 5F
```

Результатом выполнения команды uname(1) является полезная информация об операционной системе Linux. Обратите внимание, что на современных машинах с Linux, чтобы использовать протокол IPv4, нужно выполнить команду ping(1) с флагом -4.

Получение необработанных сетевых данных ICMP

В этом подразделе вы узнаете, как с помощью пакета syscall перехватывать необработанные сетевые данные ICMP и как устанавливать параметры сокета с помощью функции syscall.SetsockoptInt().

Учтите, что передавать необработанные ICMP-данные намного сложнее, поскольку для этого необходимо самостоятельно создавать необработанные сетевые пакеты. Утилита, которую мы рассмотрим, называется syscallNet.go. Разделим ее на четыре части.

Первая часть syscallNet.go выглядит так:

```
package main

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

Второй фрагмент syscallNet.go содержит следующий код Go:

```
func main() {
    fd, err := syscall.Socket(syscall.AF_INET, syscall.SOCK_RAW,
                              syscall.IPPROTO_ICMP)
    if err != nil {
        fmt.Println("Error in syscall.Socket:", err)
        return
    }
```

```
    f := os.NewFile(uintptr(fd), "captureICMP")
    if f == nil {
        fmt.Println("Error in os.NewFile:", err)
        return
    }
```

Параметр `syscall.AF_INET` сообщает функции `syscall.Socket()`, что мы будем работать с протоколом IPv4, а параметр `syscall.SOCK_RAW` — что должен быть сгенерирован необработанный сокет. Последний параметр, `syscall.IPPROTO_ICMP`, говорит `syscall.Socket()`, что нас интересует только ICMP-трафик.

Третья часть `syscallNet.go`:

```
    err = syscall.SetsockoptInt(fd, syscall.SOL_SOCKET, syscall.SO_RCVBUF, 256)
    if err != nil {
        fmt.Println("Error in syscall.Socket:", err)
        return
    }
```

Вызов `syscall.SetsockoptInt()` определяет размер буфера приема сокета – `256`. Параметр `syscall.SOL_SOCKET` указывает на то, что мы работаем на уровне сокетов.

Остальной код Go файла `syscallNet.go` выглядит так:

```
    for {
        buf := make([]byte, 1024)
        numRead, err := f.Read(buf)
        if err != nil {
            fmt.Println(err)
        }
        fmt.Printf("% X\n", buf[:numRead])
    }
}
```

Благодаря циклу `for` утилита `syscallNet.go` будет продолжать перехват сетевых пакетов ICMP, пока вы не прекратите этот процесс вручную.

Выполнение `syscallNet.go` на компьютере с macOS High Sierra приведет к следующим результатам:

```
$ sudo go run syscallNet.go
45 00 40 00 BC B6 00 00 40 01 00 00 7F 00 00 01 7F 00 00 01 00 00 3F 36 71
45 00 00 5A CB 6A 90 00 0B 9F 1A 08 09 0A 0B 0C 0D 0E 0F 10 11 12 13 14 15
16 17 18 19 1A 1B 1C 1D 1E 1F 20 21 22 23 24 25 26 27 28 29 2A 2B 2C 2D 2E
2F 30 31 32 33 34 35 36 37
45 00 40 00 62 FB 00 00 40 01 00 00 7F 00 00 01 7F 00 00 01 00 00 31 EF 71
45 00 01 5A CB 6A 91 00 0B AC 5F 08 09 0A 0B 0C 0D 0E 0F 10 11 12 13 14 15
16 17 18 19 1A 1B 1C 1D 1E 1F 20 21 22 23 24 25 26 27 28 29 2A 2B 2C 2D 2E
2F 30 31 32 33 34 35 36 37
45 00 40 00 9A 5F 00 00 40 01 00 00 7F 00 00 01 7F 00 00 01 00 00 1D D6 71
45 00 02 5A CB 6A 92 00 0B C0 76 08 09 0A 0B 0C 0D 0E 0F 10 11 12 13 14 15
16 17 18 19 1A 1B 1C 1D 1E 1F 20 21 22 23 24 25 26 27 28 29 2A 2B 2C 2D 2E
2F 30 31 32 33 34 35 36 37
```

```
45 00 40 00 6E 0D 00 00 40 01 00 00 7F 00 00 01 7F 00 00 01 00 00 09 CF 71
45 00 03 5A CB 6A 93 00 0B D4 7B 08 09 0A 0B 0C 0D 0E 0F 10 11 12 13 14 15
16 17 18 19 1A 1B 1C 1D 1E 1F 20 21 22 23 24 25 26 27 28 29 2A 2B 2C 2D 2E
2F 30 31 32 33 34 35 36 37
45 00 40 00 3A 07 00 00 40 01 00 00 7F 00 00 01 7F 00 00 01 00 00 FE 9C 71
45 00 04 5A CB 6A 94 00 0B DF AB 08 09 0A 0B 0C 0D 0E 0F 10 11 12 13 14 15
16 17 18 19 1A 1B 1C 1D 1E 1F 20 21 22 23 24 25 26 27 28 29 2A 2B 2C 2D 2E
2F 30 31 32 33 34 35 36 37
45 00 24 00 45 55 00 00 40 01 00 00 7F 00 00 01 7F 00 00 01 03 03 AB 12 00
00 00 00 45 00 34 00 C5 73 00 00 01 11 00 00 7F 00 00 01 7F 00 00 01 C5 72
82 9B 00 20 00 00
45 00 24 00 E8 1E 00 00 40 01 00 00 7F 00 00 01 7F 00 00 01 03 03 AB 10 00
00 00 00 45 00 34 00 C5 74 00 00 01 11 00 00 7F 00 00 01 7F 00 00 01 C5 72
82 9C 00 20 00 00
45 00 24 00 2A 4B 00 00 40 01 00 00 7F 00 00 01 7F 00 00 01 03 03 AB 0E 00
00 00 00 45 00 34 00 C5 75 00 00 01 11 00 00 7F 00 00 01 7F 00 00 01 C5 72
82 9D 00 20 00 00
```

Выполнение `syscallNet.go` на компьютере с Debian Linux приведет к таким результатам:

```
# go run syscallNet.go
45 00 00 54 7F E9 40 00 40 01 BC BD 7F 00 00 01 7F 00 00 01 08 00 6F 07 53
E3 00 01 FA 6A CB 5A 00 00 00 00 AA 7B 06 00 00 00 00 00 10 11 12 13 14 15
16 17 18 19 1A 1B 1C 1D 1E 1F 20 21 22 23 24 25 26 27 28 29 2A 2B 2C 2D 2E
2F 30 31 32 33 34 35 36 37
45 00 00 54 7F EA 00 00 40 01 FC BC 7F 00 00 01 7F 00 00 01 00 00 77 07 53
E3 00 01 FA 6A CB 5A 00 00 00 00 AA 7B 06 00 00 00 00 00 10 11 12 13 14 15
16 17 18 19 1A 1B 1C 1D 1E 1F 20 21 22 23 24 25 26 27 28 29 2A 2B 2C 2D 2E
2F 30 31 32 33 34 35 36 37
45 C0 00 44 68 54 00 00 34 01 8B 8E 86 77 DC 57 6D 4A C1 FD 03 0A 8F 27 00
00 00 00 45 00 00 28 40 4F 40 00 34 06 74 6A 6D 4A C1 FD 86 77 DC 57 B0 B8
DD 96 00 00 00 00 52 F1 AB DA 50 14 00 00 90 9E 00 00
45 00 00 54 80 4E 40 00 40 01 BC 58 7F 00 00 01 7F 00 00 01 08 00 7E 01 53
E3 00 02 FB 6A CB 5A 00 00 00 00 9A 80 06 00 00 00 00 00 10 11 12 13 14 15
16 17 18 19 1A 1B 1C 1D 1E 1F 20 21 22 23 24 25 26 27 28 29 2A 2B 2C 2D 2E
2F 30 31 32 33 34 35 36 37
45 00 00 54 80 4F 00 00 40 01 FC 57 7F 00 00 01 7F 00 00 01 00 00 86 01 53
E3 00 02 FB 6A CB 5A 00 00 00 00 9A 80 06 00 00 00 00 00 10 11 12 13 14 15
16 17 18 19 1A 1B 1C 1D 1E 1F 20 21 22 23 24 25 26 27 28 29 2A 2B 2C 2D 2E
2F 30 31 32 33 34 35 36 37
45 00 00 54 80 9B 40 00 40 01 BC 0B 7F 00 00 01 7F 00 00 01 08 00 93 EC 53
E3 00 03 FC 6A CB 5A 00 00 00 00 83 94 06 00 00 00 00 00 10 11 12 13 14 15
16 17 18 19 1A 1B 1C 1D 1E 1F 20 21 22 23 24 25 26 27 28 29 2A 2B 2C 2D 2E
2F 30 31 32 33 34 35 36 37
45 00 00 54 80 9C 00 00 40 01 FC 0A 7F 00 00 01 7F 00 00 01 00 00 9B EC 53
E3 00 03 FC 6A CB 5A 00 00 00 00 83 94 06 00 00 00 00 00 10 11 12 13 14 15
16 17 18 19 1A 1B 1C 1D 1E 1F 20 21 22 23 24 25 26 27 28 29 2A 2B 2C 2D 2E
2F 30 31 32 33 34 35 36 37
45 C0 00 44 68 55 00 00 34 01 8B 8D 86 77 DC 57 6D 4A C1 FD 03 0A 8F 27 00
```

```
00 00 00 45 00 00 28 40 D1 40 00 34 06 73 E8 6D 4A C1 FD 86 77 DC 57 8E 8E
DD 96 00 00 00 00 6C 6E D3 36 50 14 00 00 71 EF 00 00
45 00 00 54 80 F8 40 00 40 01 BB AE 7F 00 00 01 7F 00 00 01 08 00 F2 E7 53
E3 00 04 FD 6A CB 5A 00 00 00 00 23 98 06 00 00 00 00 00 10 11 12 13 14 15
16 17 18 19 1A 1B 1C 1D 1E 1F 20 21 22 23 24 25 26 27 28 29 2A 2B 2C 2D 2E
2F 30 31 32 33 34 35 36 37
45 00 00 54 80 F9 00 00 40 01 FB AD 7F 00 00 01 7F 00 00 01 00 00 FA E7 53
E3 00 04 FD 6A CB 5A 00 00 00 00 23 98 06 00 00 00 00 00 10 11 12 13 14 15
16 17 18 19 1A 1B 1C 1D 1E 1F 20 21 22 23 24 25 26 27 28 29 2A 2B 2C 2D 2E
2F 30 31 32 33 34 35 36 37
45 00 00 54 82 0D 40 00 40 01 BA 99 7F 00 00 01 7F 00 00 01 08 00 4A 82 53
E3 00 05 FE 6A CB 5A 00 00 00 00 CA FC 06 00 00 00 00 00 10 11 12 13 14 15
16 17 18 19 1A 1B 1C 1D 1E 1F 20 21 22 23 24 25 26 27 28 29 2A 2B 2C 2D 2E
2F 30 31 32 33 34 35 36 37
45 00 00 54 82 0E 00 00 40 01 FA 98 7F 00 00 01 7F 00 00 01 00 00 52 82 53
E3 00 05 FE 6A CB 5A 00 00 00 00 CA FC 06 00 00 00 00 00 10 11 12 13 14 15
16 17 18 19 1A 1B 1C 1D 1E 1F 20 21 22 23 24 25 26 27 28 29 2A 2B 2C 2D 2E
2F 30 31 32 33 34 35 36 37
45 C0 00 44 68 56 00 00 34 01 8B 8C 86 77 DC 57 6D 4A C1 FD 03 0A 8F 27 00
00 00 00 45 00 00 28 41 74 40 00 34 06 73 45 6D 4A C1 FD 86 77 DC 57 2E 9B
DD 96 00 00 00 00 C3 D6 44 57 50 14 00 00 09 5A 00 00
45 C0 00 44 68 57 00 00 34 01 8B 8B 86 77 DC 57 6D 4A C1 FD 03 0A 8F 27 00
00 00 00 45 00 00 28 44 27 40 00 33 06 71 92 6D 4A C1 FD 86 77 DC 57 C5 C2
DD 96 00 00 00 00 CF DD DB BE 50 14 00 00 CE C3 00 00
45 C0 00 58 94 B4 00 00 40 01 E7 2E 7F 00 00 01 7F 00 00 01 03 03 F1 DA 00
00 00 00 45 00 00 3C 85 E1 00 00 01 11 35 CE 7F 00 00 01 7F 00 00 01 95 1E
82 9A 00 28 FE 3B 40 41 42 43 44 45 46 47 48 49 4A 4B 4C 4D 4E 4F 50 51 52
53 54 55 56 57 58 59 5A 5B 5C 5D 5E 5F
45 C0 00 58 94 B5 00 00 40 01 E7 2D 7F 00 00 01 7F 00 00 01 03 03 F9 EA 00
00 00 00 45 00 00 3C 85 E2 00 00 01 11 35 CD 7F 00 00 01 7F 00 00 01 8D 0D
82 9B 00 28 FE 3B 40 41 42 43 44 45 46 47 48 49 4A 4B 4C 4D 4E 4F 50 51 52
53 54 55 56 57 58 59 5A 5B 5C 5D 5E 5F
45 C0 00 58 94 B6 00 00 40 01 E7 2C 7F 00 00 01 7F 00 00 01 03 03 D2 EB 00
00 00 00 45 00 00 3C 85 E3 00 00 01 11 35 CC 7F 00 00 01 7F 00 00 01 B4 0B
82 9C 00 28 FE 3B 40 41 42 43 44 45 46 47 48 49 4A 4B 4C 4D 4E 4F 50 51 52
53 54 55 56 57 58 59 5A 5B 5C 5D 5E 5F
45 C0 00 58 94 B7 00 00 40 01 E7 2B 7F 00 00 01 7F 00 00 01 03 03 D6 AC 00
00 00 00 45 00 00 3C 85 E4 00 00 02 11 34 CB 7F 00 00 01 7F 00 00 01 B0 49
82 9D 00 28 FE 3B 40 41 42 43 44 45 46 47 48 49 4A 4B 4C 4D 4E 4F 50 51 52
53 54 55 56 57 58 59 5A 5B 5C 5D 5E 5F
45 C0 00 58 94 B8 00 00 40 01 E7 2A 7F 00 00 01 7F 00 00 01 03 03 F1 B4 00
00 00 00 45 00 00 3C 85 E5 00 00 02 11 34 CA 7F 00 00 01 7F 00 00 01 95 40
82 9E 00 28 FE 3B 40 41 42 43 44 45 46 47 48 49 4A 4B 4C 4D 4E 4F 50 51 52
53 54 55 56 57 58 59 5A 5B 5C 5D 5E 5F
45 C0 00 58 94 B9 00 00 40 01 E7 29 7F 00 00 01 7F 00 00 01 03 03 CD 43 00
00 00 00 45 00 00 3C 85 E6 00 00 02 11 34 C9 7F 00 00 01 7F 00 00 01 B9 B0
82 9F 00 28 FE 3B 40 41 42 43 44 45 46 47 48 49 4A 4B 4C 4D 4E 4F 50 51 52
53 54 55 56 57 58 59 5A 5B 5C 5D 5E 5F
45 C0 00 58 94 BA 00 00 40 01 E7 28 7F 00 00 01 7F 00 00 01 03 03 9D 8F 00
00 00 00 45 00 00 3C 85 E7 00 00 03 11 33 C8 7F 00 00 01 7F 00 00 01 E9 63
```

```
82 A0 00 28 FE 3B 40 41 42 43 44 45 46 47 48 49 4A 4B 4C 4D 4E 4F 50 51 52
53 54 55 56 57 58 59 5A 5B 5C 5D 5E 5F
45 C0 00 58 94 BB 00 00 40 01 E7 27 7F 00 00 01 7F 00 00 01 03 03 A3 13 00
00 00 00 45 00 00 3C 85 E8 00 00 03 11 33 C7 7F 00 00 01 7F 00 00 01 E3 DE
82 A1 00 28 FE 3B 40 41 42 43 44 45 46 47 48 49 4A 4B 4C 4D 4E 4F 50 51 52
53 54 55 56 57 58 59 5A 5B 5C 5D 5E 5F
45 C0 00 58 94 BC 00 00 40 01 E7 26 7F 00 00 01 7F 00 00 01 03 03 D4 66 00
00 00 00 45 00 00 3C 85 E9 00 00 03 11 33 C6 7F 00 00 01 7F 00 00 01 B2 8A
82 A2 00 28 FE 3B 40 41 42 43 44 45 46 47 48 49 4A 4B 4C 4D 4E 4F 50 51 52
53 54 55 56 57 58 59 5A 5B 5C 5D 5E 5F
45 C0 00 58 94 BD 00 00 40 01 E7 25 7F 00 00 01 7F 00 00 01 03 03 A6 8D 00
00 00 00 45 00 00 3C 85 EA 00 00 04 11 32 C5 7F 00 00 01 7F 00 00 01 E0 62
82 A3 00 28 FE 3B 40 41 42 43 44 45 46 47 48 49 4A 4B 4C 4D 4E 4F 50 51 52
53 54 55 56 57 58 59 5A 5B 5C 5D 5E 5F
45 C0 00 58 94 BE 00 00 40 01 E7 24 7F 00 00 01 7F 00 00 01 03 03 F1 C6 00
00 00 00 45 00 00 3C 85 EB 00 00 04 11 32 C4 7F 00 00 01 7F 00 00 01 95 28
82 A4 00 28 FE 3B 40 41 42 43 44 45 46 47 48 49 4A 4B 4C 4D 4E 4F 50 51 52
53 54 55 56 57 58 59 5A 5B 5C 5D 5E 5F
45 C0 00 58 94 BF 00 00 40 01 E7 23 7F 00 00 01 7F 00 00 01 03 03 A3 FE 00
00 00 00 45 00 00 3C 85 EC 00 00 04 11 32 C3 7F 00 00 01 7F 00 00 01 E2 EF
82 A5 00 28 FE 3B 40 41 42 43 44 45 46 47 48 49 4A 4B 4C 4D 4E 4F 50 51 52
53 54 55 56 57 58 59 5A 5B 5C 5D 5E 5F
45 C0 00 58 94 C0 00 00 40 01 E7 22 7F 00 00 01 7F 00 00 01 03 03 B9 AA 00
00 00 00 45 00 00 3C 85 ED 00 00 05 11 31 C2 7F 00 00 01 7F 00 00 01 CD 42
82 A6 00 28 FE 3B 40 41 42 43 44 45 46 47 48 49 4A 4B 4C 4D 4E 4F 50 51 52
53 54 55 56 57 58 59 5A 5B 5C 5D 5E 5F
45 C0 00 58 94 C1 00 00 40 01 E7 21 7F 00 00 01 7F 00 00 01 03 03 B3 B7 00
00 00 00 45 00 00 3C 85 EE 00 00 05 11 31 C1 7F 00 00 01 7F 00 00 01 D3 34
82 A7 00 28 FE 3B 40 41 42 43 44 45 46 47 48 49 4A 4B 4C 4D 4E 4F 50 51 52
53 54 55 56 57 58 59 5A 5B 5C 5D 5E 5F
45 C0 00 58 94 C2 00 00 40 01 E7 20 7F 00 00 01 7F 00 00 01 03 03 F2 62 00
00 00 00 45 00 00 3C 85 EF 00 00 05 11 31 C0 7F 00 00 01 7F 00 00 01 94 88
82 A8 00 28 FE 3B 40 41 42 43 44 45 46 47 48 49 4A 4B 4C 4D 4E 4F 50 51 52
53 54 55 56 57 58 59 5A 5B 5C 5D 5E 5F
45 C0 00 58 94 C3 00 00 40 01 E7 1F 7F 00 00 01 7F 00 00 01 03 03 DD BE 00
00 00 00 45 00 00 3C 85 F0 00 00 06 11 30 BF 7F 00 00 01 7F 00 00 01 A9 2B
82 A9 00 28 FE 3B 40 41 42 43 44 45 46 47 48 49 4A 4B 4C 4D 4E 4F 50 51 52
53 54 55 56 57 58 59 5A 5B 5C 5D 5E 5F
```

Дополнительные ресурсы

Посмотрите следующие ресурсы:

- посетите страницу документации стандартного Go-пакета `net`, расположенную по адресу https://golang.org/pkg/net/. Это одна из самых больших страниц документации Go;
- чтобы больше узнать о пакете `crypto/tls` из стандартной библиотеки Go, посетите страницу https://golang.org/pkg/crypto/tls/;

- чтобы больше узнать о пакете `crypto/x509`, посетите страницу https://golang.org/pkg/crypto/x509/;
- определение протокола ICMP для IPv4 содержится в RFC 792. Его описание есть на многих ресурсах, включая https://tools.ietf.org/html/rfc792;
- *WebSocket* — это протокол двусторонней связи между клиентом и удаленным хостом. Реализация WebSocket для Go размещена по адресу https://github.com/gorilla/websocket. Чтобы узнать больше о WebSocket, посетите страницу http://www.rfc-editor.org/rfc/rfc6455.txt;
- если вы действительно увлекаетесь сетевым программированием и хотите работать с необработанными TCP-пакетами, то вы найдете интересную и полезную информацию и инструменты в библиотеке `gopacket` по адресу https://github.com/google/gopacket;
- пакет `raw`, расположенный по адресу https://github.com/mdlayher/raw, позволяет читать и записывать данные на уровне драйвера сетевого устройства.

Упражнения

- Разработайте на Go клиент для *протокола передачи файлов* (File Transfer Protocol, FTP).
- Теперь разработайте на Go FTP-сервер. Что сложнее реализовать: FTP-клиент или FTP-сервер? Почему?
- Попробуйте реализовать Go-версию утилиты `nc(1)`. Секрет программирования таких довольно сложных утилит состоит в том, чтобы начать с версии, которая реализует основные функциональные возможности утилиты, и только потом стремиться поддерживать все возможные варианты.
- Измените утилиту `TCPserver.go` таким образом, чтобы она возвращала в одном сетевом пакете дату, а в другом — время.
- Измените утилиту `TCPserver.go` таким образом, чтобы она могла последовательно обслуживать несколько клиентов. Обратите внимание, что это не то же самое, что возможность одновременного обслуживания нескольких запросов. Проще говоря, используйте цикл `for`, чтобы вызов `Accept()` выполнялся несколько раз.
- TCP-серверы, такие как `fiboTCP.go`, имеют тенденцию завершаться при получении определенного сигнала, поэтому добавьте в `fiboTCP.go` код обработки сигналов, как было показано в главе 8.
- Измените код `kvTCP.go` таким образом, чтобы защитить функцию `save()` с помощью `sync.Mutex`. Насколько это необходимо?
- Подключите `https.go` к `TLSserver.go` с помощью любого веб-браузера.

- Попробуйте поместить утилиту `https.go` в образ Docker и использовать ее оттуда.
- Напишите на Go собственный небольшой веб-сервер, применяя вместо функции `http.ListenAndServe()` простую реализацию TCP.

Резюме

В этой главе рассказано о многих интересных возможностях, включая разработку клиентов и серверов UDP и TCP, которые являются приложениями, работающими в компьютерных сетях TCP/IP.

Следующая, последняя, глава посвящена использованию Go для машинного обучения. В нее войдут такие темы, как регрессия, классификация, обнаружение аномалий и нейронные сети.

14 Машинное обучение на Go

В предыдущих двух главах рассмотрены темы, связанные с сетевым программированием, TCP/IP, HTTPS, RPC и пакетом `net`. В этой главе речь пойдет о *машинном обучении* на Go, включая такие темы, как вычисление статистических показателей, классификация, регрессия, кластеризация, обнаружение аномалий, нейронные сети, анализ выбросов и работа с Apache Kafka. Однако, поскольку темы очень объемные и заслуживают отдельной книги, в данной главе они затронуты лишь поверхностно, дано краткое представление о них, показаны некоторые удобные пакеты Go, которые помогут вам в решении этих задач.

Обратите внимание, что в основе каждой технологии машинного обучения лежит своя теория — знание теории, параметров и ограничений технологий, которые вы намерены использовать, имеет важное значение для достижения успеха в работе. Кроме того, время от времени полезно визуализировать получаемые данные, так как это позволяет быстро получить хорошее представление о них.

Если вы действительно интересуетесь машинным обучением на Go, советую начать с книги Machine Learning with Go Дэниела Уайтнека (Daniel Whitenack), изданной Packt Publishing в 2017 году. Если вы хотите изучить теорию, лежащую в основе машинного обучения, то обратитесь к книге An Introduction to Statistical Learning Гэрета Джеймса (Gareth James), Даниелы Уиттен (Daniela Witten), Тревора Хасти (Trevor Hastie) и Роберта Тибширани (Robert Tibshirani), выпущенной издательством Springer в 2013 году, и книге The Elements of Statistical Learning, 2nd Edition Тревора Хасти, Роберта Тибширани и Джерома Фридмана (Jerome Friedman), выпущенной Springer в 2009 году.

В этой главе рассмотрены следующие темы:

- вычисление простых статистических свойств;
- регрессия;
- классификация;

- выявление аномалий;
- кластеризация;
- нейронные сети;
- работа с TensorFlow;
- анализ выбросов;
- работа с Apache Kafka.

Вычисление простых статистических показателей

Статистика — это область математики, которая занимается сбором, анализом, интерпретацией, упорядочением и представлением данных. Делится на две основные области: описательную статистику, которая стремится описать уже существующую группу значений, и логическую статистику, которая пытается предсказать будущие значения на основе информации, присутствующей в текущем наборе значений.

Статистическое обучение — это раздел прикладной статистики, которая связана с машинным обучением. *Машинное обучение* тесно связано с вычислительной статистикой, является областью компьютерных наук, которая пытается строить обучение на основе данных и делать прогнозы без специального программирования.

Статистические модели пытаются интерпретировать данные максимально точно. Однако точность модели может зависеть от внешних факторов, влияющих на данные. Например, можно построить модель прогнозирования погоды, которая может стать совершенно неточной, если рядом случится ураган.

В этом разделе вы научитесь вычислять основные статистические показатели, такие как среднее, минимальное и максимальное значения выборки, *медианное* значение и *дисперсия* выборки. Эти значения позволяют, не вдаваясь в подробности, составить хорошее представление о данных. Однако общие показатели, которые описывают ваш набор данных, могут легко ввести в заблуждение, заставив вас поверить, что вы хорошо знаете свои данные, в то время как на самом деле это не так.

Мы вычислим все эти статистические показатели в программе `stats.go`. Рассмотрим ее, разделив на пять частей. Каждая строка входного файла содержит одно число, так что входной файл читается построчно. Ошибочный ввод будет игнорироваться без каких-либо предупреждающих сообщений.

Обратите внимание, что входные данные будут храниться в срезе, чтобы можно было использовать отдельную функцию для вычисления каждого показателя. Как вы вскоре увидите, значения среза перед обработкой отсортировываются.

Первая часть `stats.go` выглядит так:

```
package main

import (
    "bufio"
    "flag"
    "fmt"
    "io"
    "math"
    "os"
    "sort"
    "strconv"
    "strings"
)

func min(x []float64) float64 {
    return x[0]
}

func max(x []float64) float64 {
    return x[len(x)-1]
}
```

Поскольку срез отсортирован, в нем легко найти минимальное и максимальное значения. Но если элементы среза не являются числовыми, может потребоваться вычислить минимальное и максимальное значения другим способом, предназначенным для ваших данных.

Во второй части `stats.go` содержится следующий код Go:

```
func meanValue(x []float64) float64 {
    sum := float64(0)
    for _, v := range x {
        sum = sum + v
    }
    return sum / float64(len(x))
}
```

Эта функция вычисляет среднее значение числовых данных.

Третья часть `stats.go` выглядит так:

```
func medianValue(x []float64) float64 {
    length := len(x)
    if length%2 == 1 {
        // четные
        return x[(length-1)/2]
```

```
    } else {
        // нечетные
        return (x[length/2] + x[(length/2)-1]) / 2
    }
    return 0
}

func variance(x []float64) float64 {
    mean := meanValue(x)
    sum := float64(0)
    for _, v := range x {
        sum = sum + (v-mean)*(v-mean)
    }
    return sum / float64(len(x))
}
```

Чтобы вычислить медианное значение для набора данных, этот набор данных должен быть отсортирован.

Четвертая часть `stats.go` содержит следующий код:

```
func main() {
    flag.Parse()
    if len(flag.Args()) == 0 {
        fmt.Printf("usage: stats filename\n")
        return
    }

    data := make([]float64, 0)

    file := flag.Args()[0]
    f, err := os.Open(file)
    if err != nil {
        fmt.Println(err)
        return
    }
    defer f.Close()
```

И последняя часть `stats.go`:

```
    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)
            break
        }
        line = strings.TrimRight(line, "\r\n")
        value, err := strconv.ParseFloat(line, 64)
        if err == nil {
```

```
            data = append(data, value)
        }
    }
    sort.Float64s(data)

    fmt.Println("Min:", min(data))
    fmt.Println("Max:", max(data))
    fmt.Println("Mean:", meanValue(data))
    fmt.Println("Median:", medianValue(data))
    fmt.Println("Variance:", variance(data))
    fmt.Println("Standard Deviation:", math.Sqrt(variance(data)))
}
```

Здесь мы начинаем читать входной файл. Есть много способов получить исходные данные — вы всегда можете вернуться к главе 8, чтобы освежить знания о работе с файлами в Go.

Обратите внимание, что, хотя сортировать срез с данными перед их обработкой не обязательно, при некоторых вычислениях это экономит время, поэтому такая сортировка выполняется в функции `main()`.

Последний блок операторов `fmt.Println()` выводит вычисленные статистические показатели на экран.

Выполнение `stats.go` приведет к таким результатам:

```
$ go run stats.go data.txt
Min: -2
Max: 3
Mean: 1.04
Median: 1.2
Variance: 2.8064
Standard Deviation: 1.6752313273097539
```

Содержимое файла `data.txt` будет таким:

```
$ cat data.txt
1.2
-2
1
2.0
not valid
3
```

Как видите, каждая строка содержит одно значение, следовательно, мы работаем с самым простым набором данных (из одного столбца). Манипулирование более сложными данными может немного отличаться, но общая идея остается неизменной.

В программе, представленной в этом разделе, все статистические показатели вычислялись без помощи внешних пакетов Go, предназначенных для статистических вычислений. Большая часть кода Go, который мы рассмотрим дальше, использует существующие пакеты Go, вместо того чтобы реализовывать все с нуля.

Регрессия

Регрессия — это статистический метод вычисления отношений между переменными. В этом разделе реализована *линейная регрессия* — самый популярный и простой из методов регрессии, а также очень хороший способ изучить данные. Обратите внимание, что методы регрессии не являются абсолютно точными, даже если использовать полиномы высшего порядка (нелинейные). Главное в регрессии, как и в большинстве методов машинного обучения, — выбрать не идеальные, а лишь достаточно хорошие метод и модель.

Линейная регрессия

Идея линейной регрессии проста: мы пытаемся построить модель данных, используя уравнение первой степени. Уравнение первой степени можно представить как `y = a x + b`.

Существует много способов, позволяющих построить уравнение первой степени, которое моделирует данные. Все эти методы предназначены для вычисления `a` и `b`.

Реализация линейной регрессии

Код Go, который мы рассмотрим в этом разделе, сохранен в файле `regression.go`. Разделим его на три части. Результат программы — два числа с плавающей точкой, которые соответствуют параметрам `a` и `b` в уравнении первой степени.

В первой части `regression.go` содержится следующий код:

```
package main

import (
    "encoding/csv"
    "flag"
    "fmt"
    "gonum.org/v1/gonum/stat"
    "os"
    "strconv"
)

type xy struct {
    x []float64
    y []float64
}
```

Структура `xy` используется для хранения данных; ее конкретный вид зависит от формата и значений исходных данных.

Вторая часть regression.go выглядит так:

```
func main() {
    flag.Parse()
    if len(flag.Args()) == 0 {
        fmt.Printf("usage: regression filename\n")
        return
    }

    filename := flag.Args()[0]
    file, err := os.Open(filename)
    if err != nil {
        fmt.Println(err)
        return
    }
    defer file.Close()

    r := csv.NewReader(file)

    records, err := r.ReadAll()
    if err != nil {
        fmt.Println(err)
        return
    }
    size := len(records)

    data := xy{
    x: make([]float64, size),
    y: make([]float64, size),
}
```

Последняя часть regression.go:

```
    for i, v := range records {
        if len(v) != 2 {
            fmt.Println("Expected two elements")
            continue
        }

        if s, err := strconv.ParseFloat(v[0], 64); err == nil {
            data.y[i] = s
        }

        if s, err := strconv.ParseFloat(v[1], 64); err == nil {
            data.x[i] = s
        }
    }

    b, a := stat.LinearRegression(data.x, data.y, nil, false)
    fmt.Printf("%.4v x + %.4v\n", a, b)
    fmt.Printf("a = %.4v b = %.4v\n", a, b)
}
```

Содержимое файла исходных данных считывается в переменную `data`. Функция, которая реализует линейную регрессию, называется `stat.LinearRegression()`. Она возвращает два числа, которые являются значениями `b` и `a`, и именно в такой последовательности.

Теперь самое время загрузить пакет `gonum`:

```
$ go get -u gonum.org/v1/gonum/stat
```

Выполнение `regression.go` с исходными данными, хранящимися в файле `reg_data.txt`, приведет к таким результатам:

```
$ go run regression.go reg_data.txt
0.9463 x + -0.3985
a = 0.9463 b = -0.3985
```

Программа возвращает два числа, которые являются значениями `a` и `b` из формулы `y = a x + b`.

Содержимое `reg_data.txt` выглядит так:

```
$ cat reg_data.txt
1,2
3,4.0
2.1,3
4,4.2
5,5.1
-5,-5.1
```

Вывод данных

Теперь настало время вывести результаты и набор данных, чтобы проверить, насколько точны результаты метода линейной регрессии. Для этого мы воспользуемся кодом Go из файла `plotLR.go`, который представлен в четырех частях. Программа `plotLR.go` принимает три аргумента командной строки: значения `a` и `b` из формулы `y = a x + b`, а также имя файла, в котором хранятся точки исходных данных. То обстоятельство, что `plotLR.go` не вычисляет `a` и `b`, позволяет экспериментировать с `a` и `b`, используя собственные значения или значения, вычисленные другой утилитой.

Первая часть `plotLR.go` выглядит так:

```
package main

import (
    "encoding/csv"
    "flag"
    "fmt"
    "gonum.org/v1/plot"
    "gonum.org/v1/plot/plotter"
    "gonum.org/v1/plot/vg"
```

```
    "image/color"
    "os"
    "strconv"
)

type xy struct {
    x []float64
    y []float64
}

func (d xy) Len() int {
    return len(d.x)
}

func (d xy) XY(i int) (x, y float64) {
    x = d.x[i]
    y = d.y[i]
    return
}
```

Методы `Len()` и `XY()` необходимы для построения графика, а пакет `image/color` — для выбора цветов при выводе результатов.

Во второй части `plotLR.go` содержится следующий код:

```
func main() {
    flag.Parse()
    if len(flag.Args()) < 3 {
        fmt.Printf("usage: plotLR filename a b\n")
        return
    }

    filename := flag.Args()[0]
    file, err := os.Open(filename)
    if err != nil {
        fmt.Println(err)
        return
    }
    defer file.Close()

    r := csv.NewReader(file)

    a, err := strconv.ParseFloat(flag.Args()[1], 64)
    if err != nil {
        fmt.Println(a, "not a valid float!")
        return
    }

    b, err := strconv.ParseFloat(flag.Args()[2], 64)
    if err != nil {
        fmt.Println(b, "not a valid float!")
        return
    }
```

```
records, err := r.ReadAll()
if err != nil {
    fmt.Println(err)
    return
}
```

Эта часть программы получает аргументы командной строки и выполняет чтение данных.

Третья часть `plotLR.go`:

```
size := len(records)

data := xy{
    x: make([]float64, size),
    y: make([]float64, size),
}

for i, v := range records {
    if len(v) != 2 {
        fmt.Println("Expected two elements per line!")
        return
    }

    s, err := strconv.ParseFloat(v[0], 64)
    if err == nil {
        data.y[i] = s
    }

    s, err = strconv.ParseFloat(v[1], 64)
    if err == nil {
        data.x[i] = s
    }
}
```

Последняя часть `plotLR.go` выглядит так:

```
line := plotter.NewFunction(func(x float64) float64 { return a*x + b })
line.Color = color.RGBA{B: 255, A: 255}

p, err := plot.New()
if err != nil {
    fmt.Println(err)
    return
}

plotter.DefaultLineStyle.Width = vg.Points(1)
plotter.DefaultGlyphStyle.Radius = vg.Points(2)

scatter, err := plotter.NewScatter(data)
if err != nil {
    fmt.Println(err)
```

```
        return
    }
    scatter.GlyphStyle.Color = color.RGBA{R: 255, B: 128, A: 255}

    p.Add(scatter, line)

    w, err := p.WriterTo(300, 300, "svg")
    if err != nil {
        fmt.Println(err)
        return
    }

    _, err = w.WriteTo(os.Stdout)
    if err != nil {
        fmt.Println(err)
        return
    }
}
```

Функция, график которой мы построим, определяется с помощью метода `plotter.NewFunction()`.

Теперь нам нужно загрузить несколько внешних пакетов, выполнив следующие команды:

```
$ go get -u gonum.org/v1/plot
$ go get -u gonum.org/v1/plot/plotter
$ go get -u gonum.org/v1/plot/vg
```

Выполнение `plotLR.go` приведет к таким результатам:

```
$ go run plotLR.go reg_data.txt
usage: plotLR filename a b
$ go run plotLR.go reg_data.txt 0.9463 -0.3985
<?xml version="1.0"?>
<!-- Generated by SVGo and Plotinum VG -->
<svg width="300pt" height="300pt" viewBox="0 0 300 300"
    xmlns="http://www.w3.org/2000/svg"
    xmlns:xlink="http://www.w3.org/1999/xlink">
<g transform="scale(1, -1) translate(0, -300)">
.
.
.
```

Таким образом, прежде чем выводить вычисленные данные в виде графика, нам нужно сохранить их в файле:

```
$ go run plotLR.go reg_data.txt 0.9463 -0.3985 > output.svg
```

Поскольку выходные данные представлены в формате *Scalable Vector Graphics* (SVG), то, чтобы увидеть график, необходимо загрузить их в браузер. Результаты показаны на рис. 14.1.

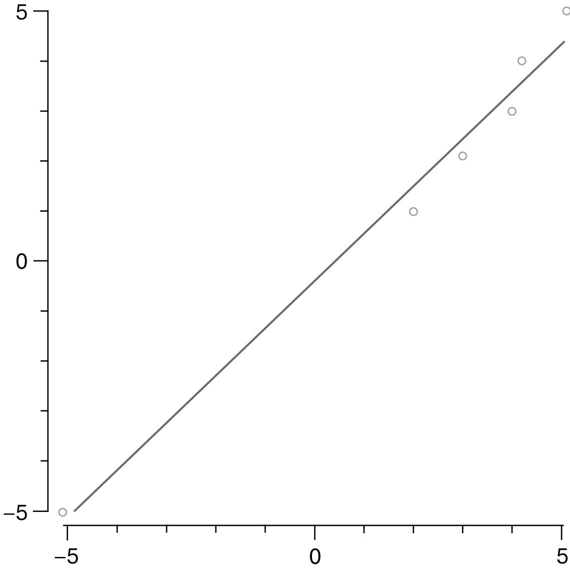


Рис. 14.1. Результат работы программы plotLR.go

На этом графике также видно, насколько точно линейное уравнение способно смоделировать исходные данные.

Классификация

В статистике и машинном обучении *классификация* — это процесс помещения элементов в предопределенные наборы, которые называются категориями. В машинном обучении классификация относится к технологиям *обучения с учителем* (supervised learning), при котором перед работой с реальными данными происходит обучение программы на множестве, которое, как считается, содержит правильно классифицированные данные.

Мы рассмотрим очень популярный и простой в реализации метод классификации — *метод k ближайших соседей* (k-NN). Идея, лежащая в основе метода k-NN, заключается в том, что мы можем классифицировать элементы данных на основе их сходства с другими элементами. Буква *k* в записи k-NN обозначает количество соседей, которые используются при принятии решения. Таким образом, *k* является положительным целым числом, обычно довольно небольшим.

Входными данными для алгоритма *k* ближайших соседей являются обучающие примеры в пространстве признаков. Объекту присваивается категория большинством голосов его соседей, причем объект присваивается классу, который является наиболее распространенным среди его k-NN. Если значение *k* равно 1,

то элемент просто присваивается классу, который является ближайшим соседом, в соответствии с используемой *метрикой расстояния* (distance metric). Метрика расстояния зависит от того, с какими данными мы имеем дело. Например, для работы с комплексными числами потребуется одна метрика расстояния, а для точек в трехмерном пространстве — другая.

Код, который иллюстрирует классификацию в Go, хранится в файле `classify.go`. Рассмотрим его, разделив на три части.

Первая часть `classify.go` выглядит так:

```
package main

import (
    "flag"
    "fmt"
    "strconv"

    "github.com/sjwhitworth/golearn/base"
    "github.com/sjwhitworth/golearn/evaluation"
    "github.com/sjwhitworth/golearn/knn"
)
```

Во второй части `classify.go` содержится следующий код Go:

```
func main() {
    flag.Parse()
    if len(flag.Args()) < 2 {
        fmt.Printf("usage: classify filename k\n")
        return
    }

    dataset := flag.Args()[0]
    rawData, err := base.ParseCSVToInstances(dataset, false)
    if err != nil {
        fmt.Println(err)
        return
    }

    k, err := strconv.Atoi(flag.Args()[1])
    if err != nil {
        fmt.Println(err)
        return
    }
    cls := knn.NewKnnClassifier("euclidean", "linear", k)
```

Метод `knn.NewKnnClassifier()` возвращает новый *классификатор*. Последний параметр функции — это число соседей, которое должен иметь классификатор.

Последняя часть `classify.go` выглядит так:

```
    train, test := base.InstancesTrainTestSplit(rawData, 0.50)
    cls.Fit(train)
```

```
    p, err := cls.Predict(test)
    if err != nil {
        fmt.Println(err)
        return
    }

    confusionMat, err := evaluation.GetConfusionMatrix(test, p)
    if err != nil {
        fmt.Println(err)
        return
    }

    fmt.Println(evaluation.GetSummary(confusionMat))
}
```

Метод `Fit()` сохраняет обучающие данные на тот случай, если вы захотите использовать их позже, а метод `Predict()` возвращает классификацию входных данных на основе обучающих данных с помощью алгоритма k-NN. Наконец, метод `evaluation.GetSummary()` возвращает таблицу точности, отклик модели, истинные положительные, ложные положительные и истинные отрицательные значения каждого класса для заданного значения `ConfusionMatrix`, которое вычислено с помощью функции `evaluation.GetConfusionMatrix()`.

Поскольку base.InstancesTrainTestSplit() не будет все время возвращать одни и те же значения, при каждом выполнении classify.go обучающие и тестовые данные будут разными. Следовательно, вы будете получать разные результаты.

Далее нам нужно установить пакет `github.com/sjwhitworth/golearn`:

```
$ go get -t -u -v github.com/sjwhitworth/golearn
github.com/sjwhitworth/golearn (download)
github.com/sjwhitworth/golearn
$ cd ~/go/src/github.com/sjwhitworth/golearn
$ go get -t -u -v ./...
```

Теперь вернемся в предыдущий каталог, чтобы запустить оттуда программу `classify.go`. Выполнение `classify.go` приведет к таким результатам:

```
$ go run classify.go class_data.txt 2
Reference  Classn True Positives False Positives True Negatives  Precision Recall
F1 Score
--------------  ------------  ---------------  --------------  ---------  ------
--------
Iris-versicolor     25       0        41        1.0000        0.9259        0.9615
Iris-virginica       5       2        61        0.7143        1.0000        0.8333
Iris-setosa         36       0        32        1.0000        1.0000        1.0000
```

```
Overall accuracy: 0.9706
$ go run classify.go class_data.txt 30
Reference  Classn True Positives False Positives True Negatives  Precision Recall
F1 Score
--------------   -------------    --------------    -------------   ---------
------   --------
Iris-versicolor     27       5       36       0.8438       1.0000       0.9153
Iris-virginica       0       0       63       NaN          0.0000       NaN
Iris-setosa         36       0       32       1.0000       1.0000       1.0000
Overall accuracy: 0.9265
```

Содержимое файла `class_data.txt` очень простое и имеет следующий формат:

```
$ head -4 class_data.txt
6.7,3.1,5.6,2.4,Iris-virginica
6.9,3.1,5.1,2.3,Iris-virginica
5.8,2.7,5.1,1.9,Iris-virginica
6.8,3.2,5.9,2.3,Iris-virginica
```

Источник — набор данных **Iris**.

Высокая точность алгоритма объясняется тем, что в файле `class_data.txt` очень мало элементов. Поскольку `golearn` включает в себя несколько примеров данных, я попробовал использовать весь набор данных Iris на своем компьютере с macOS Mojave и получил следующий результат:

```
$ go run classify.go
~/go/src/github.com/sjwhitworth/golearn/examples/datasets/iris.csv 2
Reference  Classn True Positives False Positives True Negatives  Precision Recall
F1 Score
--------------  -------------  --------------  -------------  --------  ------
--------
Iris-setosa          30       0       58       1.0000       1.0000       1.0000
Iris-virginica       28       3       56       0.9032       0.9655       0.9333
Iris-versicolor      26       1       58       0.9630       0.8966       0.9286
Overall accuracy: 0.9545
$ go run classify.go
~/go/src/github.com/sjwhitworth/golearn/examples/datasets/iris.csv 50
Reference  Classn True Positives  False Positives True Negatives  Precision Recall
F1 Score
--------------    -------------    --------------    -------------   ---------
------   --------
Iris-setosa          0       0       58       NaN           0.0000       NaN
Iris-virginica       4       5       54       0.4444        0.1379       0.2105
Iris-versicolor     24      55        4       0.3038        0.8276       0.4444
Overall accuracy: 0.3182
```

Если, работая с набором данных Iris, вы пробуете выбрать другое число соседей, то увидите, что чем больше соседей, тем ниже точность результатов. К сожалению, дальнейшее обсуждение алгоритма k-NN выходит за рамки этой книги.

Кластеризация

Кластеризация — это неконтролируемая (unsupervised, без учителя) версия классификации, при которой разделение данных на категории основано на некотором показателе сходства или расстояния. В этом разделе рассмотрена *кластеризация методом k-средних* — самый известный, а также очень легко реализуемый метод кластеризации. Здесь мы снова воспользуемся внешней библиотекой, которая находится по адресу https://github.com/mash/gokmeans.

Мы разберем кластеризацию в Go на примере утилиты `cluster.go`. Разделим ее на три части. Эта утилита принимает один аргумент командной строки — число кластеров, которые следует создать.

Первая часть `cluster.go` выглядит так:

```
package main

import (
    "flag"
    "fmt"
    "github.com/mash/gokmeans"
    "strconv"
)

var observations []gokmeans.Node = []gokmeans.Node{
    gokmeans.Node{4},
    gokmeans.Node{5},
    gokmeans.Node{6},
    gokmeans.Node{8},
    gokmeans.Node{10},
    gokmeans.Node{12},
    gokmeans.Node{15},
    gokmeans.Node{0},
    gokmeans.Node{-1},
}
```

Здесь ради простоты данные включены прямо в программу. Однако ничто не запрещает вам читать их из одного или нескольких внешних файлов.

Во второй части `cluster.go` содержится следующий код Go:

```
func main() {
    flag.Parse()
    if len(flag.Args()) == 0 {
        fmt.Printf("usage: cluster k\n")
        return
    }

    k, err := strconv.Atoi(flag.Args()[0])
    if err != nil {
```

```
        fmt.Println(err)
        return
    }
```

Последняя часть cluster.go:

```
    if success, centroids := gokmeans.Train(observations, k, 50); success {
        fmt.Println("The centroids are the following:")
        for _, centroid := range centroids {
            fmt.Println(centroid)
        }
        fmt.Println("The clusters are the following:")
        for _, observation := range observations {
            index := gokmeans.Nearest(observation, centroids)
            fmt.Println(observation, "belongs in cluster", index+1, ".")
        }
    }
}
```

Теперь нам нужно скачать внешнюю библиотеку Go, выполнив следующую команду:

```
$ go get -v -u github.com/mash/gokmeans
github.com/mash/gokmeans (download)
github.com/mash/gokmeans
```

Если мы хотим получить только один кластер, то выполнение cluster.go приведет к следующим результатам:

```
$ go run cluster.go 1
The centroids are the following:
[6.555555555555555]
The clusters are the following:
[4] belongs in cluster 1 .
[5] belongs in cluster 1 .
[6] belongs in cluster 1 .
[8] belongs in cluster 1 .
[10] belongs in cluster 1 .
[12] belongs in cluster 1 .
[15] belongs in cluster 1 .
[0] belongs in cluster 1 .
[-1] belongs in cluster 1 .
```

Поскольку кластер всего один, то все элементы будут принадлежать этому кластеру, и выходные данные cluster.go выглядят соответствующим образом.

Если изменить значение k, то результат будет таким:

```
$ go run cluster.go 5
The centroids are the following:
[5]
[-0.5]
```

```
[13.5]
[10]
[8]
The clusters are the following:
[4] belongs in cluster 1 .
[5] belongs in cluster 1 .
[6] belongs in cluster 1 .
[8] belongs in cluster 5 .
[10] belongs in cluster 4 .
[12] belongs in cluster 3 .
[15] belongs in cluster 3 .
[0] belongs in cluster 2 .
[-1] belongs in cluster 2 .
$ go run cluster.go 8
The centroids are the following:
[0]
[4.5]
[-1]
[9]
[-1]
[6]
[12]
[15]
The clusters are the following:
[4] belongs in cluster 2 .
[5] belongs in cluster 2 .
[6] belongs in cluster 6 .
[8] belongs in cluster 4 .
[10] belongs in cluster 4 .
[12] belongs in cluster 7 .
[15] belongs in cluster 8 .
[0] belongs in cluster 1 .
[-1] belongs in cluster 3 .
```

Выявление аномалий

Методы выявления аномалий стремятся определить вероятность того, что данное множество содержит элементы с аномальным поведением, которые могут иметь необычные значения или закономерности.

Утилита, которую мы разработаем в этом разделе, называется `anomaly.go`. Разделим ее на три части. В этой утилите реализовано вероятностное выявление аномалий с помощью пакета *Anomalyzer*. Вычислим вероятность того, что данный набор числовых значений содержит элементы с аномальным поведением.

Первая часть `anomaly.go` выглядит так:

```
package main

import (
    "flag"
```

```
    "fmt"
    "math/rand"
    "strconv"
    "time"
    "github.com/lytics/anomalyzer"
)

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

Вторая часть `anomaly.go` выглядит так:

```
func main() {
    flag.Parse()
    if len(flag.Args()) == 0 {
        fmt.Printf("usage: anomaly MAX\n")
        return
    }

    MAX, err := strconv.Atoi(flag.Args()[0])
    if err != nil {
        fmt.Println(err)
        return
    }

    conf := &anomalyzer.AnomalyzerConf{
        Sensitivity:  0.1,
        UpperBound:   5,
        LowerBound:   anomalyzer.NA,
        ActiveSize:   1,
        Nseasons:     4,
        Methods:      []string{"diff", "fence", "magnitude", "ks"},
    }
```

Пакет `anomalyzer` поддерживает алгоритмические тесты `cdf`, `diff`, `high rank`, `low rank`, `magnitude`, `fence` и `bootstrap ks`. Если вы решите использовать некоторые или все из них в качестве значения `Methods`, то `anomalyzer` их выполнит. Поскольку каждый из этих тестов возвращает вероятность аномального поведения, пакет `anomalyzer` будет вычислять средневзвешенное значение, которое покажет, есть ли в данном множестве элементы с аномальным поведением.

Поле `Sensitivity` предназначено для теста `magnitude` и по умолчанию равно `0.1`. Поля `UpperBound` и `LowerBound` используются в тесте `fence`. А вот поле `ActiveSize` является обязательным, и его значение должно быть не менее `1`. Наконец, поле `NSeasons`, если оно не определено, по умолчанию равно `4`.

В третьей части `anomaly.go` содержится следующий код Go:

```
    data := []float64{}
    SEED := time.Now().Unix()
    rand.Seed(SEED)
```

```
    for i := 0; i < MAX; i++ {
        data = append(data, float64(random(0, MAX)))
    }
    fmt.Println("data:", data)
```

Код anomaly.go самостоятельно генерирует случайные данные. Количество элементов задается в виде аргумента командной строки программы.

Последняя часть anomaly.go выглядит так:

```
    anom, _ := anomalyzer.NewAnomalyzer(conf, data)
    prob := anom.Push(8.0)
    fmt.Println("Anomalous Probability:", prob)
}
```

Теперь нам нужно загрузить внешний пакет Go:

```
$ go get -v -u github.com/lytics/anomalyzer
github.com/lytics/anomalyzer (download)
github.com/drewlanenga/govector (download)
github.com/drewlanenga/govector
github.com/lytics/anomalyzer
```

Выполнение anomaly.go приведет к таким результатам:

```
$ go run anomaly.go 20
data: [18 3 2 19 2 16 5 15 3 14 2 9 11 10 2 17 17 14 19 1]
Anomalous Probability: 0.8612730015082957
$ go run anomaly.go 20
data: [17 8 19 10 0 14 12 7 7 13 2 5 18 1 15 4 0 14 13 9]
Anomalous Probability: 0.7885470085470085
$ go run anomaly.go 100
data: [85 5 64 32 69 55 0 67 11 96 75 92 25 54 2 49 58 6 16 38 55 11 93 90
90 47 66 97 37 61 85 92 15 45 33 43 61 44 73 18 10 86 17 15 67 28 26 7 25
76 79 51 9 32 70 99 9 39 6 25 10 57 50 84 20 67 42 89 0 1 8 96 49 6 20 33
57 18 48 84 53 98 51 84 41 97 69 62 11 44 21 13 90 25 52 85 48 27 90 20]
Anomalous Probability: 0.8977395577395577
```

Нейронные сети

Нейронные сети стремятся работать подобно человеческому мозгу, учатся решать задачи на основе предоставленных им примеров. Нейронная сеть делится на слои. В самой маленькой нейронной сети должно быть как минимум два слоя: входной и выходной. На этапе обучения данные проходят через все слои нейронной сети. Реальные выходные значения обучающих данных используются для корректировки рассчитанных выходных значений обучающих данных, чтобы следующая итерация была более точной.

Утилита neural.go, которую мы разработаем в этом разделе, будет реализовывать простейшую нейронную сеть. Мы разделим ее на четыре части.

Первая часть neural.go выглядит так:

```
package main

import (
    "fmt"
    "math/rand"
    "time"

    "github.com/goml/gobrain"
)
```

Новая строка в списке import дает инструменту gofmt команду сортировать имена пакетов по блокам, разделенным пустыми строками. Во второй части neural.go содержится следующий код Go:

```
func main() {
    seed := time.Now().Unix()
    rand.Seed(seed)
    patterns := [][][]float64{
        {{0, 0, 0, 0}, {0}},
        {{0, 1, 0, 1}, {1}},
        {{1, 0, 1, 0}, {1}},
        {{1, 1, 1, 1}, {1}},
    }
```

Срез patterns содержит обучающие данные, которые будут использованы позже. Функция rand.Seed() создает и инициализирует генератор случайных чисел, который автоматически используется пакетом github.com/goml/gobrain.

Третья часть neural.go выглядит так:

```
    ff := &gobrain.FeedForward{}
    ff.Init(4, 2, 1)
    ff.Train(patterns, 1000, 0.6, 0.4, false)
```

Здесь инициализируется нейронная сеть. Первый параметр метода Init() – это количество входов сети, второй – количество скрытых узлов, а третий – количество выходов. Измерение и данные среза patterns должны соответствовать значениям, использованным в Init(), и наоборот.

Последняя часть neural.go выглядит так:

```
    in := []float64{1, 1, 0, 1}
    out := ff.Update(in)
    fmt.Println(out)

    in = []float64{0, 0, 0, 0}
    out = ff.Update(in)
    fmt.Println(out)
}
```

Здесь выполняется два теста. Для первого теста используются входные данные `{1, 1, 0, 1}`, а для второго — `{0, 0, 0, 0}`.

Теперь вы уже знаете, какой пакет Go следует загрузить с помощью команды `go get`, поэтому скачайте его, прежде чем пытаться запустить `neural.go`.

Выполнение `neural.go` приведет к таким результатам:

```
$ go run neural.go
[0.9918648920317314]
[0.02826477691747802]
```

Первое значение приближается к 1, тогда как второе — к 0.

Поскольку здесь присутствует элемент случайности, результаты нескольких запусков `neural.go` будут незначительно различаться:

```
$ go run neural.go
[0.9920127780655835]
[0.028029429851140687]
go run neural.go
[0.9913803776914417]
[0.028875009295811015]
```

Анализ выбросов

Анализ выбросов — это поиск значений, которые выглядят так, как будто они не имеют отношения к остальным значениям. Проще говоря, выбросы — это экстремальные значения, которые сильно отличаются от остальных наблюдений. Вот хорошая книга по анализу выбросов: *Outlier Analysis, 2nd Edition*, автор Хару С. Аггарвал (Charu C. Aggarwal), издательство Springer, 2017 год.

Мы рассмотрим метод анализа выбросов, основанный на стандартном отклонении. Этот метод реализован в программе `outlier.go`, которую мы разделим на четыре части. Позже вы ближе познакомитесь с этим методом.

Первая часть `outlier.go` выглядит так:

```
package main

import (
    "bufio"
    "flag"
    "fmt"
    "io"
    "math"
    "os"
    "sort"
    "strconv"
    "strings"
)
```

Метод, который будет реализован, требует только пакетов стандартной библиотеки Go.

Во второй части `outlier.go` содержится следующий код Go:

```
func variance(x []float64) float64 {
    mean := meanValue(x)
    sum := float64(0)
    for _, v := range x {
        sum = sum + (v-mean)*(v-mean)
    }
    return sum / float64(len(x))
}

func meanValue(x []float64) float64 {
    sum := float64(0)
    for _, v := range x {
        sum = sum + v
    }
    return sum / float64(len(x))
}
```

Эти две функции уже использовались ранее в программе `stats.go`. Если вы постоянно применяете одни и те же функции, удобно создать одну или несколько библиотек Go и сгруппировать в них код Go.

В третьей части `outlier.go` содержится следующий код Go:

```
func outliers(x []float64, limit float64) []float64 {
    deviation := math.Sqrt(variance(x))
    mean := meanValue(x)
    anomaly := deviation * limit
    lower_limit := mean - anomaly
    upper_limit := mean + anomaly
    fmt.Println(lower_limit, upper_limit)

    y := make([]float64, 0)
    for _, val := range x {
        if val < lower_limit || val > upper_limit {
            y = append(y, val)
        }
    }
    return y
}
```

В этой функции реализована вся логика программы. Она рассчитывает стандартное отклонение и среднее значение выборки, чтобы затем вычислить верхний и нижний пределы. Все, что выходит за эти пределы, будет считаться выбросом.

Последняя часть `outlier.go` выглядит так:

```
func main() {
    flag.Parse()
```

```
    if len(flag.Args()) != 2 {
        fmt.Printf("usage: stats filename limit\n")
        return
    }
    file := flag.Args()[0]

    f, err := os.Open(file)
    if err != nil {
        fmt.Println(err)
        return
    }
    defer f.Close()

    limit, err := strconv.ParseFloat(flag.Args()[1], 64)
    if err != nil {
        fmt.Println(err)
        return
    }

    data := make([]float64, 0)
    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)
            break
        }
        line = strings.TrimRight(line, "\r\n")
        value, err := strconv.ParseFloat(line, 64)
        if err == nil {
            data = append(data, value)
        }
    }

    sort.Float64s(data)
    out := outliers(data, limit)
    fmt.Println(out)
}
```

Функция `main()` принимает аргументы командной строки и считывает данные из входного файла, после чего вызывает `outliers()`.

Выполнение `outlier.go` приведет к такому результату:

```
$ go run outlier.go data.txt 2
-94.21189713007178 95.36745268562734
[-100 100]
$ go run outlier.go data.txt 5
-236.3964094918461 237.55196504740167
```

```
[]
$ go run outlier.go data.txt 0.02
-0.3701189713007176 1.5256745268562737
[-100 -10 -2 2 3 10 100]
```

Если уменьшить значение переменной `limit`, то получим больше выбросов из набора данных.

Однако это зависит от задачи, которую мы пытаемся решить.

В файле `data.txt` содержатся следующие данные:

```
$ cat data.txt
1.2
-2
1
2.0
not valid
3
10
100
-10
-100
```

Работа с TensorFlow

TensorFlow — довольно известная платформа с открытым исходным кодом, предназначенная для машинного обучения. Чтобы использовать TensorFlow в Go, вам сначала нужно скачать следующий пакет Go:

```
$ go get github.com/tensorflow/tensorflow/tensorflow/go
```

Однако для работы этой команды необходимо предварительно установить C-интерфейс для TensorFlow. На компьютере с macOS Mojave это можно сделать так:

```
$ brew install tensorflow
```

При попытке установить пакет Go для TensorFlow, если C-интерфейс еще не установлен, выведется следующее сообщение об ошибке:

```
$ go get github.com/tensorflow/tensorflow/tensorflow/go
# github.com/tensorflow/tensorflow/tensorflow/go
ld: library not found for -ltensorflow
clang: error: linker command failed with exit code 1 (use -v to see invocation)
```

Поскольку TensorFlow — довольно сложный пакет, не лишним будет проверить правильность его установки с помощью такой команды:

```
$ go test github.com/tensorflow/tensorflow/tensorflow/go
ok github.com/tensorflow/tensorflow/tensorflow/go 0.109s
```

Кроме тестов Go, также можно выполнить следующую программу Go, которая выведет версию используемого Go-пакета TensorFlow:

```
package main

import (
    tf "github.com/tensorflow/tensorflow/tensorflow/go"
    "github.com/tensorflow/tensorflow/tensorflow/go/op"
    "fmt"
)

func main() {
    s := op.NewScope()
    c := op.Const(s, "Using TensorFlow version: " + tf.Version())
    graph, err := s.Finalize()

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

    sess, err := tf.NewSession(graph, nil)
    if err != nil {
        fmt.Println(err)
        return
    }

    output, err := sess.Run(nil, []tf.Output{c}, nil)
    if err != nil {
        fmt.Println(err)
        return
    }

    fmt.Println(output[0].Value())
}
```

Если сохранить эту программу как `tfVersion.go` и выполнить ее, то получим следующий результат:

```
$ go run tfVersion.go
2019-06-10 22:30:12.880532: I
tensorflow/core/platform/cpu_feature_guard.cc:141] Your CPU supports instructions
that this TensorFlow binary was not compiled to use: AVX2 FMA Using TensorFlow
version: 1.13.1
```

Первое сообщение — это предупреждение, сгенерированное Go-пакетом TensorFlow, которое в данном случае можно проигнорировать.

Теперь мы готовы продолжить, и я покажу вам настоящую программу, на примере которой вы увидите, как работает TensorFlow. Код этой программы хранится в файле `tFlow.go`. Разделим его на четыре части. Эта программа выполняет сло-

жение и умножение двух чисел, которые она принимает в качестве аргументов командной строки.

Первая часть `tFlow.go` выглядит так:

```
package main

import (
    "fmt"
    "os"
    "strconv"

    tf "github.com/tensorflow/tensorflow/tensorflow/go"
    "github.com/tensorflow/tensorflow/tensorflow/go/op"
)
```

Во второй части `tFlow.go` содержится следующий код:

```
func Add(sum_arg1, sum_arg2 int8) (interface{}, error) {
    sum_scope := op.NewScope()
    input1 := op.Placeholder(sum_scope.SubScope("a1"), tf.Int8)
    input2 := op.Placeholder(sum_scope.SubScope("a2"), tf.Int8)
    sum_result_node := op.Add(sum_scope, input1, input2)

    graph, err := sum_scope.Finalize()
    if err != nil {
        fmt.Println(err)
        return 0, err
    }

    a1, err := tf.NewTensor(sum_arg1)
    if err != nil {
        fmt.Println(err)
        return 0, err
    }

    a2, err := tf.NewTensor(sum_arg2)
    if err != nil {
        fmt.Println(err)
        return 0, err
    }

    session, err := tf.NewSession(graph, nil)
    if err != nil {
        fmt.Println(err)
        return 0, err
    }
    defer session.Close()

    sum, err := session.Run(
        map[tf.Output]*tf.Tensor{
            input1: a1,
```

```
            input2: a2,
        },
        []tf.Output{sum_result_node}, nil)

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

    return sum[0].Value(), nil
}
```

Функция `Add()` складывает два числа типа `int8`. Для простого сложения двух чисел здесь слишком много кода, что показывает, что у TensorFlow есть определенный стиль работы. Так происходит главным образом потому, что TensorFlow является расширенной средой со многими возможностями.

Третья часть `tFlow.go` выглядит так:

```
func Multiply(sum_arg1, sum_arg2 int8) (interface{}, error) {
    sum_scope := op.NewScope()
    input1 := op.Placeholder(sum_scope.SubScope("x1"), tf.Int8)
    input2 := op.Placeholder(sum_scope.SubScope("x2"), tf.Int8)

    sum_result_node := op.Mul(sum_scope, input1, input2)
    graph, err := sum_scope.Finalize()
    if err != nil {
        fmt.Println(err)
        return 0, err
    }

    x1, err := tf.NewTensor(sum_arg1)
    if err != nil {
        fmt.Println(err)
        return 0, err
    }

    x2, err := tf.NewTensor(sum_arg2)
    if err != nil {
        fmt.Println(err)
        return 0, err
    }

    session, err := tf.NewSession(graph, nil)
    if err != nil {
        fmt.Println(err)
        return 0, err
    }
    defer session.Close()

    sum, err := session.Run(
        map[tf.Output]*tf.Tensor{
```

```
            input1: x1,
            input2: x2,
        },
        []tf.Output{sum_result_node}, nil)

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

    return sum[0].Value(), nil
}
```

Функция `Multiply()` выполняет умножение двух чисел типа `int8` и возвращает результат.

Последняя часть `tFlow.go` выглядит так:

```
func main() {
    if len(os.Args) != 3 {
        fmt.Println("Need two integer parameters!")
        return
    }

    t1, err := strconv.Atoi(os.Args[1])
    if err != nil {
        fmt.Println(err)
        return
    }
    n1 := int8(t1)

    t2, err := strconv.Atoi(os.Args[2])
    if err != nil {
        fmt.Println(err)
        return
    }
    n2 := int8(t2)

    res, err := Add(n1, n2)
    if err != nil {
        fmt.Println(err)
    } else {
        fmt.Println("Add:", res)
    }

    res, err = Multiply(n1, n2)
    if err != nil {
        fmt.Println(err)
    } else {
        fmt.Println("Multiply:", res)
    }
}
```

Выполнение `tFlow.go` выдаст такой результат:

```
$ go run tFlow.go 1 20
2019-06-14 18:46:52.115676: I
tensorflow/core/platform/cpu_feature_guard.cc:141] Your CPU supports
instructions that this TensorFlow binary was not compiled to use: AVX2 FMA
Add: 21
Multiply: 20
$ go run tFlow.go -2 20
2019-06-14 18:47:23.104918: I
tensorflow/core/platform/cpu_feature_guard.cc:141] Your CPU supports
instructions that this TensorFlow binary was not compiled to use: AVX2 FMA
Add: 18
Multiply: -40
```

Предупреждение, которое появляется на выходе программы, говорит о том, что TensorFlow работает на вашем компьютере не так быстро, как мог бы. По сути, вам сообщают, что для решения проблемы нужно скомпилировать TensorFlow с нуля.

Поговорим о Kafka

В этом разделе вы узнаете, как писать и читать записи в формате JSON с помощью *Kafka*. Эта платформа получила название в честь писателя Франца Кафки. Платформа Kafka написана на *Scala* и *Java*. Первоначально она разрабатывалась компанией LinkedIn, а в 2011 году была передана в дар Apache Software Foundation. На структуру Kafka повлияли журналы транзакций.

Главным преимуществом Kafka является то, что ее можно использовать для быстрого сохранения большого количества данных. Это может заинтересовать тех, кому приходится работать с огромными объемами данных в реальном времени. Недостаток этой платформы — для поддержания высокой скорости обработки данные предназначены только для чтения и сохраняются примитивным способом.

На примере программы `writeKafka.go` разберем, как записывать данные в Kafka. В терминологии Kafka `writeKafka.go` называется *производителем*. Мы рассмотрим эту утилиту, разделив ее на три части.

Первая часть `writeKafka.go` выглядит так:

```
package main

import (
    "context"
    "encoding/json"
    "fmt"
```

```
    "github.com/segmentio/kafka-go"
    "math/rand"
    "os"
    "strconv"
    "time"
)
```

Go-драйвер Kafka требует использования внешнего пакета, который нужно загрузить самостоятельно.

Во второй части `writeKafka.go` содержится следующий код Go:

```
type Record struct {
    Name   string   `json:"name"`
    Random int      `json:"random"`
}

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

func main() {
    MIN := 0
    MAX := 0
    TOTAL := 0
    topic := ""
    if len(os.Args) > 4 {
        MIN, _ = strconv.Atoi(os.Args[1])
        MAX, _ = strconv.Atoi(os.Args[2])
        TOTAL, _ = strconv.Atoi(os.Args[3])
        topic = os.Args[4]
    } else {
        fmt.Println("Usage:", os.Args[0], "MIX MAX TOTAL TOPIC")
        return
    }
```

Структура `Record` используется для хранения данных, которые будут записаны в соответствующий топик Kafka.

Третья часть `writeKafka.go`:

```
    partition := 0
    conn, err := kafka.DialLeader(context.Background(), "tcp",
        "localhost:9092", topic, partition)
    if err != nil {
        fmt.Printf("%s\n", err)
        return
    }

    rand.Seed(time.Now().Unix())
```

В этой части программы мы определяем адрес сервера Kafka (`localhost: 9092`).

Последняя часть `writeKafka.go` выглядит так:

```
    for i := 0; i < TOTAL; i++ {
        myrand := random(MIN, MAX)
        temp := Record{strconv.Itoa(i), myrand}
        recordJSON, _ := json.Marshal(temp)

        conn.SetWriteDeadline(time.Now().Add(1 * time.Second))
        conn.WriteMessages(
            kafka.Message{Value: []byte(recordJSON)},
        )

        if i%50 == 0 {
            fmt.Print(".")
        }
        time.Sleep(10 * time.Millisecond)
    }

    fmt.Println()
    conn.Close()
}
```

Теперь мы рассмотрим программу Go, которая называется `readKafka.go`. На ее примере я покажу, как читать данные из Kafka. Программы, считывающие данные из Kafka, в терминологии Kafka называются *потребителями*. Мы разделим код Go программы `readKafka.go` на четыре части.

Первая часть `readKafka.go` выглядит так:

```
package main

import (
    "context"
    "encoding/json"
    "fmt"
    "github.com/segmentio/kafka-go"
    "os"
)
```

Утилите `readKafka.go` также нужен внешний Go-пакет.

Вторая часть `readKafka.go` выглядит так:

```
type Record struct {
    Name    string   `json:"name"`
    Random  int      `json:"random"`
}

func main() {
    if len(os.Args) < 2 {
        fmt.Println("Need a Kafka topic name.")
```

```
        return
    }

    partition := 0
    topic := os.Args[1]
    fmt.Println("Kafka topic:", topic)
```

В этой части программы определяется структура Go, в которой будут храниться записи Kafka. Также здесь обрабатываются аргументы командной строки программы.

В третьей части `readKafka.go` содержится следующий код Go:

```
    r := kafka.NewReader(kafka.ReaderConfig{
        Brokers:   []string{"localhost:9092"},
        Topic:     topic,
        Partition: partition,
        MinBytes:  10e3,
        MaxBytes:  10e6,
    })
    r.SetOffset(0)
```

В структуре `kafka.NewReader()` содержатся данные, необходимые для подключения к процессу сервера Kafka и теме Kafka.

Последний фрагмент кода `readKafka.go` выглядит так:

```
    for {
        m, err := r.ReadMessage(context.Background())
        if err != nil {
            break
        }
        fmt.Printf("message at offset %d: %s = %s\n", m.Offset,
                   string(m.Key), string(m.Value))
        temp := Record{}
        err = json.Unmarshal(m.Value, &temp)
        if err != nil {
            fmt.Println(err)
        }
        fmt.Printf("%T\n", temp)
    }

    r.Close()
}
```

Этот цикл `for` считывает данные из заданного топика Kafka и выводит их на экран. Обратите внимание, что программа постоянно ожидает данных и никогда не завершает работу.

В данном примере мы, чтобы выполнить обе программы — `readKafka.go` и `writeKafka.go`, — воспользуемся образом Docker с Kafka. Поэтому прежде всего

нам требуется выполнить следующую команду, чтобы загрузить нужный образ Kafka, если он еще не установлен на нашем локальном компьютере:

```
$ docker pull landoop/fast-data-dev:latest
```

Результаты выполнения команды `docker images` позволят нам убедиться, что на компьютере есть нужный образ Kafka Docker. Затем следует выполнить этот образ и запустить его как *контейнер*:

```
$ docker run --rm --name=kafka-box -it -p 2181:2181 -p 3030:3030 -p
8081:8081 -p 8082:8082 -p 8083:8083 -p 9092:9092 -p 9581:9581 -p 9582:9582
-p 9583:9583 -p 9584:9584 -e ADV_HOST=127.0.0.1 landoop/fast-data-
dev:latest
Setting advertised host to 127.0.0.1.
Starting services.
This is Landoop's fast-data-dev. Kafka 2.0.1-L0 (Landoop's Kafka
Distribution).
You may visit http://127.0.0.1:3030 in about a minute.
.
.
.
```

Для того чтобы все эти утилиты работали, необходимо скачать Go-пакет, который позволяет обмениваться данными с Kafka.

Это можно сделать следующим образом:

```
$ go get -u github.com/segmentio/kafka-go
```

Теперь мы готовы использовать образ Docker Kafka и запустить обе утилиты Go.

Выполнение `writeKafka.go` приведет к таким результатам:

```
$ go run readKafka.go my_topic | head
Kafka topic: my_topic
message at offset 0:  = {"name":"0","random":134}
main.Record
message at offset 1:  = {"name":"1","random":27}
main.Record
message at offset 2:  = {"name":"2","random":168}
main.Record
message at offset 3:  = {"name":"3","random":317}
main.Record
message at offset 4:  = {"name":"4","random":455}
signal: broken pipe
```

Дополнительные ресурсы

Советую посмотреть следующие ресурсы:

- чтобы больше узнать о Kafka, посетите страницу https://kafka.apache.org/;
- чтобы больше узнать о TensorFlow, пройдите на страницу https://www.tensorflow.org/;

- *Lenses* — отличный продукт для работы с платформой Kafka и ее записями. Подробнее о Lenses читайте на сайте https://lenses.io/;
- страницу документации Go-пакета TensorFlow вы найдете по адресу https://godoc.org/github.com/tensorflow/tensorflow/tensorflow/go;
- чтобы больше узнать о наборе данных Iris, посетите страницу https://archive.ics.uci.edu/ml/datasets/iris;
- на сайте https://machinebox.io/ вы найдете программное обеспечение для машинного обучения, рассчитанного на тех, у кого нет специальной подготовки. Оно бесплатно предоставляется для разработчиков и функционирует из образов Docker;
- страницу документации Go-пакета Anomalyzer вы найдете по адресу https://github.com/lytics/anomalyzer.

Упражнения

- Разработайте собственный производитель Kafka, который бы делал записи в топик Kafka в формате JSON с тремя полями.
- Есть такое очень интересное статистическое свойство — *ковариация*. Найдите его формулу и реализуйте его в Go.
- Измените код `stats.go` таким образом, чтобы работать только с целочисленными значениями.
- Измените код `cluster.go` таким образом, чтобы получать данные из внешнего файла, имя которого будет передаваться программе в качестве аргумента командной строки.
- Измените код `outlier.go` таким образом, чтобы разделить входные данные на два среза и работать с каждым из этих срезов отдельно.
- Измените код `outlier.go` таким образом, чтобы принимать верхний и нижний пределы от пользователя, а не вычислять их.
- Если вам трудно использовать TensorFlow, попробуйте `tfgo`, который вы найдете по адресу https://github.com/galeone/tfgo.

Резюме

В этой главе рассмотрено несколько интересных тем, связанных с машинным обучением, включая регрессию, обнаружение аномалий, классификацию, кластеризацию и выбросы. Go — отличный, надежный язык программирования для применения в области машинного обучения. Смело используйте его в своих проектах!

Что дальше?

С философской точки зрения ни одна книга по программированию не является идеальной, так что и эта книга не исключение. Я раскрыл не все темы, касающиеся Go? Конечно же! Почему? Потому что тем всегда больше, чем можно уместить в одной книге. Если бы я попытался охватить их все, то эта книга никогда не была бы готова к печати. Ситуация в чем-то аналогична спецификациям программы: в нее всегда можно добавлять новые интересные функции, но если в какой-то момент не заморозить спецификации, то программа всегда будет находиться в состоянии разработки.

Главное, что, прочитав эту книгу, вы будете готовы учиться самостоятельно. Это самое большое преимущество любой хорошей книги по программированию. Основная цель издания — помочь вам научиться программировать на Go и получить некоторый опыт. Однако ничто не заменит самостоятельных упражнений и частых ошибок, потому что единственный способ изучить язык программирования — постоянно разрабатывать нетривиальные вещи. Теперь вы готовы писать собственное программное обеспечение на Go и изучать что-то новое.

Я хочу поздравить вас и поблагодарить за выбор этой книги. Надеюсь, что она была для вас полезной и вы будете и дальше пользоваться ею в качестве справочного пособия. Go — отличный язык программирования, и я уверен, вы не пожалеете о том, что изучили его. Для меня это завершение еще одной книги о Go, но для вас только начало пути!

Soli Deo Gloria — только Богу слава.

Михалис Цукалос

Golang для профи: работа с сетью, многопоточность, структуры данных и машинное обучение с Go

Перевела с английского *Е. Сандицкая*

Заведующая редакцией	*Ю. Сергиенко*
Руководитель проекта	*С. Давид*
Ведущий редактор	*Н. Гринчик*
Научный редактор	*Р. Пресняков*
Литературный редактор	*А. Дубейко*
Художественный редактор	*В. Мостипан*
Корректор	*А. Лавровская, Е. Павлович*
Верстка	*Г. Блинов*

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: 05.2020. 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 27.04.20. Формат 70×100/16. Бумага офсетная. Усл. п. л. 58,050. Тираж 600. Заказ 0000.

Отпечатано в ОАО «Первая Образцовая типография». Филиал «Чеховский Печатный Двор».
142300, Московская область, г. Чехов, ул. Полиграфистов, 1.
Сайт: www.chpk.ru. E-mail: marketing@chpk.ru
Факс: 8(496) 726-54-10, телефон: (495) 988-63-87

Пол Тронкон, Карл Олбинг

BASH И КИБЕРБЕЗОПАСНОСТЬ: АТАКА, ЗАЩИТА И АНАЛИЗ ИЗ КОМАНДНОЙ СТРОКИ LINUX

Командная строка может стать идеальным инструментом для обеспечения кибербезопасности. Невероятная гибкость и абсолютная доступность превращают стандартный интерфейс командной строки (CLI) в фундаментальное решение, если у вас есть соответствующий опыт. Авторы Пол Тронкон и Карл Олбинг рассказывают об инструментах и хитростях командной строки, помогающих собирать данные при упреждающей защите, анализировать логи и отслеживать состояние сетей. Пентестеры узнают, как проводить атаки, используя колоссальный функционал, встроенный практически в любую версию Linux.

Мехди Меджуи, Эрик Уайлд, Ронни Митра, Майк Амундсен

НЕПРЕРЫВНОЕ РАЗВИТИЕ API. ПРАВИЛЬНЫЕ РЕШЕНИЯ В ИЗМЕНЧИВОМ ТЕХНОЛОГИЧЕСКОМ ЛАНДШАФТЕ

Для реализации API необходимо провести большую работу. Чрезмерное планирование может стать пустой тратой сил, а его недостаток приводит к катастрофическим последствиям. В этой книге вы получите решения, которые позволят вам распределить необходимые ресурсы и достичь требуемого уровня эффективности за оптимальное время. Как соблюсти баланс гибкости и производительности, сохранив надежность и простоту настройки? Четыре эксперта из Академии API объясняют разработчикам ПО, руководителям продуктов и проектов, как максимально увеличить ценность их API, управляя интерфейсами как продуктами с непрерывным жизненным циклом.