

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 С-кода из Go	86
Вызов С-кода из Go в одном файле	86
Вызов из Go С-кода в отдельных файлах	87
С-код	87
Go-код	88
Сочетание кода на Go и С	89
Вызов Go-функций из С-кода	90
Go-пакет	90
С-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 <code>copy()</code>	135
Многомерные срезы	137
Еще один пример использования срезов	137
Сортировка срезов с помощью <code>sort.Slice()</code>	139
Добавление массива к срезу	141
Хеш-таблицы Go	142
Запись в хеш-таблицу со значением <code>nil</code>	144
Когда использовать хеш-таблицы	145
Константы Go	145
Генератор констант <code>iota</code>	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 sudoku.....	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

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 равно `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 создаются 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 ситуациях 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` от-лично скомпилируется — попробуйте сами.

Сейчас еще не время говорить о том, как нарушать правила Go, однако суще-ствует способ обойти такое ограничение. Он показан в следующем Go-коде, кото-рый 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 под-черкивания, 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 за-пустится, 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`.