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`.