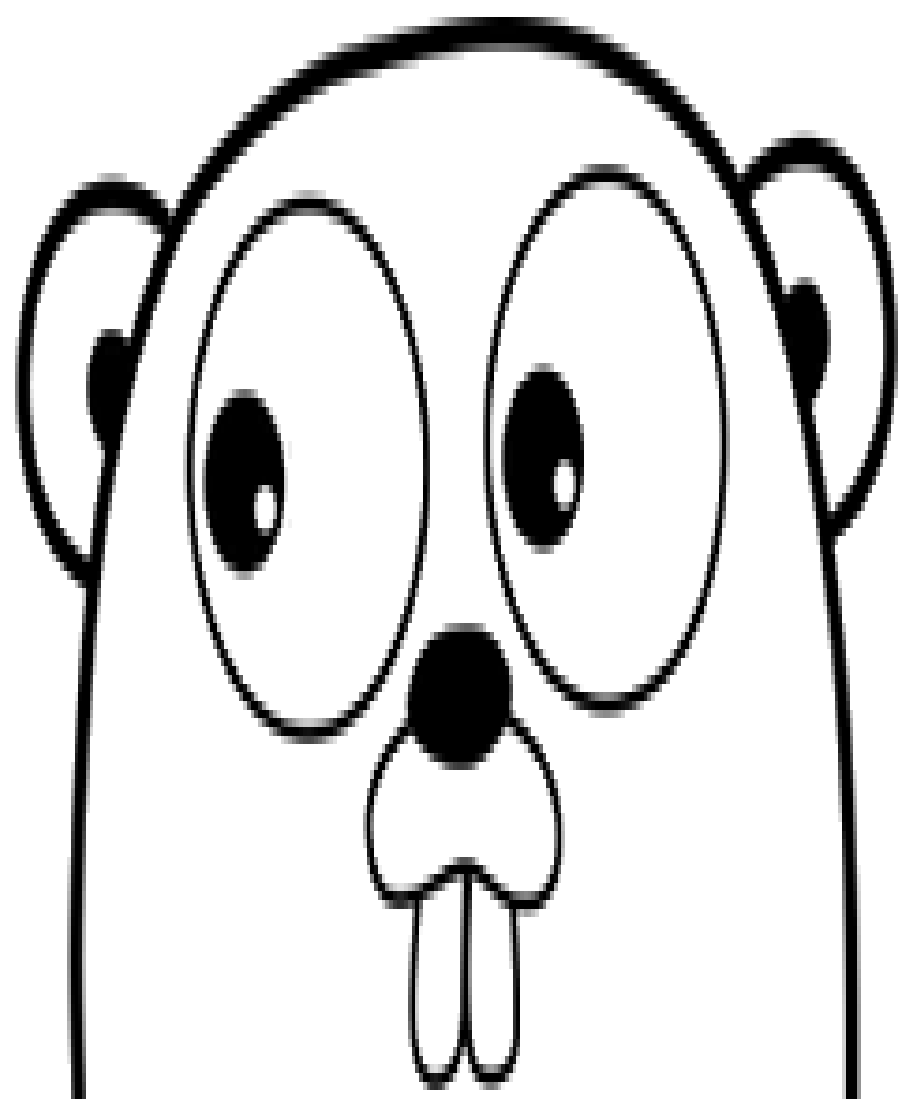
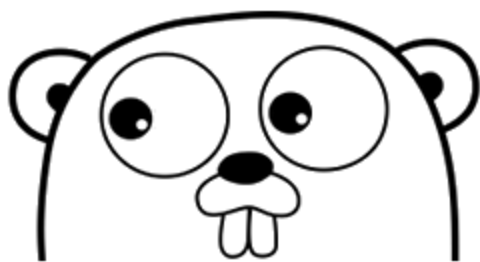




The Little Go Book

by Karl Seguin



The Little Go Book

by Karl Seguin

The Little Go Book

Karl Seguin

Об этой книге

Лицензия

Маленькая книга о Go распространяется под лицензией Attribution-NonCommercial-ShareAlike 4.0 International license. Вы не должны платить за эту книгу.

Вы можете свободно копировать, распространять, изменять или публиковать книгу. Однако, я прошу чтобы вы всегда приписывали авторство книги мне, Карлу Сегину, и не использовали её в коммерческих целях.

Вы можете прочесть полный текст лицензии здесь:

<http://creativecommons.org/licenses/by-nc-sa/4.0/>

Последняя версия

Последний исходный код этой книги доступен тут:

<http://github.com/karlseguin/the-little-go-book>

Введение

У меня всегда было чувство на грани любви и ненависти, когда дело доходило до изучения новых языков. С одной стороны, языки имеют настолько фундаментальное значение для того, что мы делаем, что даже мелкое различие может оказать ощутимое значение. Этот *вау* эффект когда какой-то клик может оказать настолько сильное влияние на то, как вы программируете, что может предопределить ваши ожидания от других языков. С другой стороны, понимание языка является поэтапным процессом. Изучение новых ключевых слов, системы типов, стиля кодирования, а так же новых библиотек, сообществ и парадигм это много работы, которую, казалось бы сложно оправдать. По сравнению со всем остальным, что мы могли бы изучить, новые языки часто кажутся плохой инвестицией нашего времени.

Тем не менее, мы *должны* двигаться вперед. Мы *должны* делать постепенные шаги, потому что, опять же, языки являются основой того, что мы делаем. И хотя изменения являются поэтапными, они, как правило, имеют широкую область применения и влияют на продуктивность, читабельность, быстродействие, тестируемость, управление зависимостями, обработку ошибок, документацию, профилирование, сообщества, стандартные библиотеки, и так далее.

У нас может возникнуть важный вопрос: **почему Go?** Для меня, существуют две весомые причины. Первая в том, что это относительно простой язык с относительно простой стандартной библиотекой. Во многих отношениях инкрементная природа Go является упрощением некоторых сложностей, которые были добавлены к другим языкам за последнюю пару десятилетий. Другая причина в том, что для многих разработчиков он будет дополнять их существующий арсенал.

Go был разработан в качестве системного языка (например, для операционных систем, драйверов устройств) и направлен на C и C++ разработчиков. По данным команды Go, и что верно для меня,

разработчики приложений, не системные разработчики, являются основными пользователями Go. Почему? Я не могу авторитетно говорить за системных разработчиков, но те из нас, кто создает веб-сайты, сервисы, приложения для рабочего стола, и т. п. частично приходят к необходимости наличия класса систем, которые находятся где-то между системами низкого уровня и приложениями высокого уровня.

Это может быть обмен сообщениями, кеширование, тяжелые вычисления для анализа данных, утилиты командной строки, логирование или мониторинг. Я не знаю как это назвать, но в течении моей карьеры, когда растет сложность системы и параллелизм часто измеряется в десятках тысяч потоков, появляется быстро растущая потребность в пользовательских инфраструктурных системах. Вы *можете* построить систему с помощью Ruby или Python или чего-то еще (и многие это делают), но для таких типов систем выгодней использовать более жесткую систему типов и высокую производительность. Кроме того, вы можете использовать Go для разработки веб-сайтов (и многие это делают), но я по прежнему считаю что Node или Ruby с большим отрывом превосходят его для таких систем.

Есть и другие области применения, в которых Go превосходит другие языки. Например, не нужно устанавливать никаких дополнительных зависимостей при запуске скомпилированной программы на Go. Вам не нужно переживать, есть ли у ваших пользователей Ruby или установлена ли JVM, и если да, то какой версии. По этой причине Go становится все более популярным как язык для приложений командной строки и другим разновидностям утилит, которые вам необходимо распространять (например сборщик файлов логов).

Становится ясно, что изучение Go будет эффективным использованием вашего времени. Вам не нужно будет тратить много часов на изучение или хотя бы освоение Go, и в конечном итоге у вас останется что-то полезное от ваших усилий.

Примечание автора

Я колебался писать эту книгу по нескольким причинам. Первая в том, что Go имеет солидную документацию, в частности [Effective Go](#).

Вторая в моем дискомфорте при написании книги о языке. Когда я написал маленькую книгу о MongoDB (The Little MongoDB Book), можно было с уверенностью сказать, что большинство читателей поняли основы реляционных баз данных и моделирования. С маленькой книгой о Redis (The Little Redis Book) вы могли получить представление о хранилищах ключ-значение.

Когда я думаю об абзацах и главах, которые впереди, я знаю, что буду не в состоянии сделать те же предположения. Сколько времени займет разговор об интерфейсах, зная, что для некоторых это будет новая концепция, а другим будет достаточно сказать, что *в Go есть интерфейсы*? В конечном счете, меня утешает то, что вы можете дать мне знать, если какие-то части слишком поверхностные или другие слишком подробные. Считайте это ценой за книгу.

Приступая к работе

Если вы хотите немного поиграться с Go, вы должны посетить [Go Playground](#) которая позволяет запускать код на сайте без необходимости что-то устанавливать. Это также наиболее распространенный способ делиться кодом Go когда вам нужна помощь на форуме [Go's discussion forum](#) или на таких сайтах, как StackOverflow.

Установка Go проста. Вы можете установить его из исходного кода, но я предлагаю вам использовать уже скомпилированные бинарные файлы. Когда вы [перейдете на страницу загрузки](#), вы увидите установочные файлы для различных платформ.

Давайте откажемся от них и научимся устанавливать Go самостоятельно. Как вы увидите, это не трудно.

Кроме простых примеров, Go спроектирован для работы с вашим кодом внутри рабочего пространства. Рабочее пространство это папка которая состоит из подпапок `bin`, `pkg` и `src`. Вы могли бы заставить Go следовать вашему стилю - не нужно.

Как правило, я располагаю свои проекты внутри папки `~/code`. Например, `~/code/blog` содержит мой блог. Для Go мое рабочее пространство это `~/code/go` и мой работающий на Go блог будет в `~/code/go/src/blog`. Так как это долго печатать, я использую символическую ссылку, чтобы сделать его доступным по адресу `~/code/blog`:

```
ln -s ~/code/go/src/blog ~/code/blog
```

Короче говоря, создайте папку `go` с подпапкой `src` куда вы будете складывать свои проекты.

OSX / Linux

Скачайте файл с расширением `tar.gz` для вашей платформы. Для OSX, то, что нужно, находится в файле `go#.#.#.darwin-amd64-osx10.8.tar.gz`, где `#.#.#` это последняя версия Go.

Извлеките содержимое файла в `/usr/local` с помощью команды `tar -C /usr/local -xzf go#.#.#.darwin-amd64-osx10.8.tar.gz`.

Установите две переменные среды:

1. `GOPATH` указывает на папку с вашим рабочим пространством, для меня, это `$HOME/code/go`.
2. Вам нужно добавить путь к исполняемым файлам Go в переменную `PATH`.

Вы можете установить их с помощью команд:

```
echo 'export GOPATH=$HOME/code/go' >> $HOME/.profile
echo 'export PATH=$PATH:/usr/local/go/bin' >> $HOME/.profile
```

Затем нужно активировать эти переменные. Вы можете закрыть и открыть заново ваш интерпретатор команд или выполнить `source $HOME/.profile`.

Наберите `go version` и вы должны увидеть примерно следующее: `go version go1.3.3 darwin/amd64`.

Windows

Скачайте zip-файл с последней версией. Если у вас 64-битная система, вам нужен `go#.#.#.windows-amd64.zip`, где `#.#.#` последняя версия Go.

Распакуйте его в какую-нибудь папку. `c:\Go` будет хорошим выбором.

Установите две переменные среды:

1. `GOPATH` указывает на папку с вашим рабочим пространством. Это может быть что-то вроде `c:\users\goku\work\go`.

2. Добавьте `c:\Go\bin` в переменную `PATH`.

Переменные среды могут быть установлены с помощью кнопки `Environment Variables` на вкладке `Advanced` в панели управления `System`. Некоторые версии `Windows` отображают эту панель в пункте `Advanced System Settings` внутри панели `System`.

Откройте окно командной строки и наберите `go version`. Вы увидите что-то подобное: `go version go1.3.3 windows/amd64`.

Глава 1 - Основы

Go компилируемый язык со статической типизацией. Он имеет C-подобный синтаксис и сборку мусора. Что это означает?

Компиляция

Компиляция это процесс перевода исходного кода, который написали вы, в язык более низкого уровня – либо ассемблер (как в случае с Go), либо в какой-то другой промежуточный язык (как в случае с Java или C#).

С компилируемыми языками бывает неприятно работать потому, что компиляция может быть медленной. Сложно совершать быстрые итерации, если вы тратите минуты или часы в ожидании компиляции вашего кода. Скорость компиляции является одной из основных целей Go. Это хорошие новости для людей, которые работают над большими проектами, а так же для тех, кто привык быстро получать обратную связь при работе с интерпретируемыми языками.

Компилируемые языки, как правило, работают быстрее, и их исполняемые файлы могут быть выполнены без установки дополнительных зависимостей (по крайней мере это верно для таких языков, как C, C++ и Go, которые компилируются непосредственно в ассемблер).

Статическая типизация

Статическая типизация означает, что все переменные должны быть определенного типа (int, string, bool, []byte, и т. д.). Это достигается либо путем указания типа при объявлении переменной, либо, во многих случаях, тип определяет компилятор (мы скоро увидим примеры).

Можно много говорить о статической типизации, но я считаю, что её лучше понять глядя на код. Если вы использовали языки с динамической типизацией, вам она может показаться громоздкой. Вы не ошибаетесь, но есть и преимущества, особенно когда она используется в сочетании с компиляцией. Эти две особенности нередко объединяются. Когда есть одна, обычно присутствует и другая, но это не строгое правило. Со строгой системой типов, компилятор может обнаруживать проблемы, которые выходят за рамки синтаксических ошибок, а также проводить дополнительную оптимизацию.

С-подобный синтаксис

Говоря о том, что язык имеет С-подобный синтаксис, имеется ввиду то, что если вы уже использовали любой другой С-подобный язык как С, С++, Java, JavaScript и С#, то Go вам покажется похожим – по крайней мере на первый взгляд. Например, это значит, что `&&` используется как логическое И, `==` применяется для сравнения, `{` и `}` обозначает начало и конец области видимости, а индексы массивов начинаются с 0.

С-подобный синтаксис также имеет тенденцию к постановке точки с запятой в окончании строк и использованию скобок вокруг условий. В Go нет ни того, ни другого, хотя скобки все еще используются для разделения приоритета. Например, оператор `if` выглядит так:

```
if name == "Leto" {  
    print("the spice must flow")  
}
```

Но в более сложных случаях скобки все еще полезны:

```
if (name == "Goku" && power > 9000) || (name == "gohan" &&  
power < 4000) {  
    print("super Saiyan")  
}
```

Помимо этого, Go гораздо ближе к C, чем к C# или Java - не только с точки зрения синтаксиса, но и с точки зрения назначения. Это отражается в лаконичности и простоте языка, которая, надеюсь, будет очевидной, как только вы начнете изучать его.

Сборка мусора

Некоторые переменные при создании имеют легко определяемую жизнь. Локальная переменная функции, например, исчезает при выходе из функции. В других случаях это не так очевидно, по крайней мере для компилятора. Например, жизнь переменной, которая была возвращена функцией или на которую ссылаются другие переменные и объекты, бывает сложно определить. Без сборки мусора задачей разработчика являлась очистка памяти от переменных там, где по их мнению они не нужны. Как? В C, вы выполняли `free(str);` для переменной.

Языки со сборщиками мусора (такие, как Ruby, Python, Java, JavaScript, C#, Go) способны отслеживать и освобождать переменные, которые больше не используются. Сборка мусора добавляет свои накладные расходы, но также устраняет ряд разрушительных ошибок.

Запуск Go кода

Давайте начнем наше путешествие созданием простой программы и научимся её компилировать и выполнять. Откройте ваш любимый текстовый редактор и наберите следующий код:

```
package main

func main() {
    println("it's over 9000!")
}
```

Сохраните файл с именем `main.go`. Сейчас, вы можете сохранить его где угодно; не обязательно использовать рабочее пространство Go для

тривиальных примеров.

Затем откройте оболочку/командную строку и перейдите в папку, в которую вы сохранили файл. Мне для этого нужно было набрать `cd ~/code`.

Наконец, запустите программу, введя:

```
go run main.go
```

Если все работает, вы должны увидеть *it's over 9000!*.

Но подождите, что насчет этапа с компиляцией? `go run` это удобная команда, которая компилирует *и* запускает ваш код. Она использует временную директорию для сборки программы, выполняет её и затем очищает. Вы можете увидеть расположение временной папки выполнив:

```
go run --work main.go
```

Чтобы явно скомпилировать код, используйте `go build`:

```
go build main.go
```

Эта команда создаст исполняемый файл `main` который вы сможете запустить. В Linux / OSX не забудьте, что перед именем выполняемого файла нужно набрать точку и слэш: `./main`.

Во время разработки, вы можете использовать `go run` или `go build`. Однако в случае развертывания кода, вам нужно переносить бинарный файл, полученный с помощью `go build` и выполнять его.

Main

Надеюсь код, который мы только что выполнили, был понятен. Мы создали функцию и напечатали строку с помощью встроенной функции `Println`. Команда `go run` знала что выполнять потому, что у нее не было выбора? Нет, в Go точкой входа в программу является функция с именем `main` в пакете `main`.

Мы поговорим о пакетах в позже. Сейчас, когда вы сфокусированы на понимании основ Go, мы всегда будем писать наш код в пакете `main`.

Если хотите, вы можете отредактировать код и изменить имя пакета. Запустите код с помощью команды `go run` и вы получите ошибку. Затем измените имя обратно на `main`, но используйте другое имя функции. Вы должны увидеть другое сообщение об ошибке. Попробуйте сделать тоже самое используя команду `go build`. Обратите внимание на то, что код компилируется, но в нем нет точки входа чтобы запустить его. Это совершенно нормально когда вы, к примеру, создаете библиотеку.

Импорты

Go имеет ряд встроенных функций, таких как `Println`, которые могут быть использованы без упоминания. Вы не сможете зайти далеко без использования стандартной библиотеки Go и библиотек других разработчиков. В Go ключевое слово `import` используется для объявления пакета, который будет использован кодом в файле.

Давайте изменим нашу программу:

```
package main

import (
    "fmt"
    "os"
)

func main() {
    if len(os.Args) != 2 {
        os.Exit(1)
    }
    fmt.Println("It's over ", os.Args[1])
}
```

И запустим её с помощью команды:

```
go run main.go 9000
```

Здесь мы используем два стандартных пакета Go: `fmt` и `os`. Мы также можем увидеть другую встроенную функцию `len`. `len` возвращает длину строки, или число значений в словаре, или, как мы видим здесь, число элементов в массиве. Если вам интересно, почему мы ожидаем два аргумента, это потому, что первый аргумент имеет индекс 0 — это всегда путь к текущему исполняемому файлу. (Изменить программу чтобы вывести его и убедитесь сами.)

Вы наверное заметили префикс перед именем функции, совпадающий с именем пакета, например `fmt.Println`. Это отличается от многих других языков. Мы узнаем больше о пакетах в следующих главах. Сейчас, знание того, как импортировать и использовать пакеты, уже хороший старт.

Go строго относится к импорту пакетов. Программа не будет скомпилирована, если вы импортируете пакет и не используете его. Попробуйте выполнить:

```
package main

import (
    "fmt"
    "os"
)

func main() {
}
```

Вы должны получить две ошибки о том, что `fmt` и `os` были импортированы и не использованы. Раздражает ли это? Безусловно. Со временем вы привыкните к этому (хотя это всё равно будет раздражать). Go строг в этом плане потому, что неиспользуемые пакеты замедляют компиляцию; правда у большинства из нас не возникает проблем с этим.

Еще одна вещь, которую хотелось бы отметить, это то, что стандартная библиотека Go хорошо документирована. Вы можете взглянуть на <http://golang.org/pkg/fmt/#Println> чтобы узнать больше о функции

`println`, которую мы использовали. Можно кликнуть на заголовок и увидеть её исходный код. Также, промотайте вверх страницы, чтобы узнать больше о возможностях форматирования Go.

Если у вас нет доступа к Интернету, можно посмотреть документацию локально, выполнив команду:

```
godoc -http=:6060
```

И ввести в браузере адрес `http://localhost:6060`

Переменные и определения

Было бы неплохо начать и закончить обзор переменных просто сказав: *вы можете объявить переменную и задать её значение с помощью `x = 4`*. К сожалению в Go все сложнее. Начнем наш разговор глядя на простые примеры. Затем, в следующей главе, познакомимся с ними более подробно рассматривая создание и использование структур. Тем не менее, полное понимание, возможно, займет некоторое время.

Вы можете подумать *Ого! Что тут может быть сложного?* Давайте посмотрим на несколько примеров.

Наиболее явный способ использования переменных в Go также наиболее подробный:

```
package main

import (
    "fmt"
)

func main() {
    var power int
    power = 9000
    fmt.Printf("It's over %d\n", power)
}
```

Здесь мы определяем переменную `power` типа `int`. По умолчанию, Go присваивает переменным нулевые значения. Для целых чисел это 0, для булевых `false`, для строк `""` и так далее. Затем, мы задаем значение 9000 для переменной `power`. Две строки кода можно объединить в одну:

```
var power int = 9000
```

Все еще много печатать. В Go есть короткий оператор объявления переменных `:=`, с которым можно объявить тип так:

```
power := 9000
```

Это удобно и работает и точно также с функциями:

```
func main() {  
    power := getPower()  
}  
  
func getPower() int {  
    return 9001  
}
```

Важно помнить, что `:=` используется для объявления переменной, а так же задания ей значения. Почему? Потому, что переменную нельзя объявить дважды (по крайней мере в той же области видимости). Если вы попытаетесь выполнить следующее, вы получите ошибку.

```
func main() {  
    power := 9000  
    fmt.Printf("It's over %d\n", power)  
  
    // COMPILER ERROR:  
    // no new variables on left side of :=  
    power := 9001  
    fmt.Printf("It's also over %d\n", power)  
}
```

Компиляция завершится с сообщением *нет новых переменных слева от :=*. Это значит, что когда переменная объявляется в первый раз, используется `:=`, но при последующих присваиваниях используется оператор `=`. В этом есть большой смысл, но может быть трудным для запоминания когда переключаться между этими операторами.

Если вы читали сообщение об ошибке внимательно, вы могли заметить, что слово *переменные* во множественном числе. Это потому, что Go позволяет присваивать несколько значений переменным (использованием `=` или `:=`):

```
func main() {  
    name, power := "Goku", 9000  
    fmt.Printf("%s's power is over %d\n", name, power)  
}
```

До тех пор пока переменная является новой, можно использовать `:=`. Рассмотрим пример:

```
func main() {  
    power := 1000  
    fmt.Printf("default power is %d\n", power)  
  
    name, power := "Goku", 9000  
    fmt.Printf("%s's power is over %d\n", name, power)  
}
```

Хотя `power` используется дважды с помощью `:=`, компилятор не будет жаловаться когда мы будем использовать эту переменную второй раз, он видит что другая переменная `name` новая и разрешает использовать `:=`. Однако, вы не можете изменить тип переменной `power`. Она была объявлена (косвенным образом) как целое число и может принимать только целочисленные значения.

Теперь, последнее что нужно знать, это то, что Go, как и в случае с импортами, не позволяет иметь в программе неиспользуемые переменные. Например,

```
func main() {  
    name, power := "Goku", 1000  
    fmt.Printf("default power is %d\n", power)  
}
```

не будет скомпилировано потому, что `name` была объявлена, но не используется. Как и неиспользуемые импорты это будет причинять одни расстройства, но в целом, я думаю, это улучшает чистоту кода и его читаемость.

Еще многое предстоит узнать об объявлениях и присваиваниях. А пока запомните, что когда используется `var ИМЯ ТИП переменная` объявляется с нулевым значением, *ИМЯ := ЗНАЧЕНИЕ* значение присваивается одновременно с объявлением переменной, а *ИМЯ = ЗНАЧЕНИЕ* когда присваивается значение уже объявленной переменной.

Объявление функций

Настало время рассказать о том, что функции могут возвращать несколько значений. Возьмем три функции: одна не возвращает значение, другая возвращает одно значение, и третья возвращает два значения.

```
func log(message string) {  
}  
  
func add(a int, b int) int {  
}  
  
func power(name string) (int, bool) {  
}
```

Последнюю можно использовать так:

```
value, exists := power("goku")  
if exists == false {
```

```
// handle this error case  
}
```

Иногда нужно только одно из возвращаемых значений. В этом случае другое значение присваивают переменной `_`:

```
_, exists := power("goku")  
if exists == false {  
    // handle this error case  
}
```

Это больше, чем просто договорённость. `_` – пустой идентификатор, его особенность в том, что возвращаемое значение в действительности не присваивается. Это позволяет вам использовать `_` снова и снова независимо от возвращаемого типа переменной.

И наконец еще кое-что, с чем вы наверняка столкнетесь при объявлении функций. Если параметры имеют одинаковый тип, можно использовать короткий синтаксис:

```
func add(a, b int) int {  
  
}
```

Вы будете часто использовать возможность возвращения нескольких значений. И так же часто использовать идентификатор `_` для их игнорирования. Множественные возвращаемые значения и чуть более короткий способ указания параметров не являются обязательными правилами. Тем не менее, рано или поздно вы столкнетесь с ними, поэтому важно знать о них.

Перед тем, как продолжить

Мы рассмотрели несколько небольших отдельных кусочков, кажущихся разрозненными на данном этапе. Мы будем постепенно создавать примеры побольше, в надежде, что куски сойдутся вместе.

Если вы раньше работали с динамическим языком, сложности, связанные с типами и объявлениями, могут показаться шагом назад. Я соглашусь с вами. Для некоторых систем динамические языки являются более продуктивными.

Если вы работали со статически типизированным языком, вы, вероятно, почувствуете себя комфортно с Go. Определяемые типы и множественные возвращаемые значения выглядят изящно (хотя, конечно, они есть не только в Go). Надеюсь, когда мы узнаем больше, вы оцените чистый и лаконичный синтаксис.

Глава 2 - Структуры

Go не является объектно-ориентированным языком (ОО-языком), таким как C++, Java, Ruby или C#. В нем нет объектов, нет наследования и многих других понятий, свойственных ОО-языкам, полиморфизма или перегрузки.

В Go есть структуры, которые могут быть связаны с методами. В Go также есть простая, но эффективная форма композиции. В целом, это приводит к более простому коду, но бывают случаи когда вам будет не хватать некоторых возможностей ООП (стоит отметить, что *композиция вместо наследования* старый лозунг и Go первый язык, который я использовал, занимающий твердую позицию по этому вопросу).

Хотя Go и не использует ООП, вы заметите много похожего между определением структуры и использованием классов. В качестве простого примера возьмем структуру Saiyan:

```
type Saiyan struct {  
    Name string  
    Power int  
}
```

Мы скоро увидим, как добавить метод к этой структуре, как вы добавляли бы методы к классам. Но перед тем, как это сделать, мы вернемся назад к объявлениям.

Объявления и инициализация

Когда мы впервые рассматривали переменные и их объявление, мы видели только встроенные типы данных, такие как целые числа и строки. Сейчас, когда мы говорим о структурах, мы должны дополнить этот разговор, включив в него указатели.

Самый простой способ создать значение нашей структуры выглядит так:

```
goku := Saiyan{
    Name: "Goku",
    Power: 9000,
}
```

Примечание: Запятая в конце каждой строки внутри структуры является обязательной, включая последнюю строку. Без нее, компилятор выдаст ошибку. Вы оцените это соглашение по требованию замыкающей запятой если вы раньше использовали какой-то язык или формат, в котором этого требования не было.

Не обязательно задавать все значения или даже какое-то одно. Обе записи являются корректными:

```
goku := Saiyan{}
```

// или

```
goku := Saiyan{Name: "Goku"}
goku.Power = 9000
```

Точно также, как и с переменными, необъявленные поля содержат нулевые значения.

Кроме того, можно опустить имена полей и указать сразу значения в порядке их следования при объявлении структуры (хотя для верности, лучше использовать этот способ только для структур с несколькими полями):

```
goku := Saiyan{"Goku", 9000}
```

Все из приведенных выше примеров объявляют переменную `goku` и присваивают ей значение.

Иногда нам не нужна переменная, которая напрямую связана со своим значением, а нужна переменная, которая хранит указатель на это значение. Указатель это адрес в памяти. Это то место, где можно найти фактическое значение. Это степень косвенности. Грубо говоря, между значением и указателем та же разница, что и между домом и его адресом.

Почему стоит использовать указатель вместо фактического значения? Это объясняется тем, что Go передает аргументы в функции как копии. Зная этот факт, что выведет следующий код?

```
func main() {  
    goku := Saiyan{"Goku", 9000}  
    Super(goku)  
    fmt.Println(goku.Power)  
}  
  
func Super(s Saiyan) {  
    s.Power += 10000  
}
```

Ответ 9000, а не 19000. Почему? Потому, что Super изменяет копию оригинального значения goku и таким образом изменения, сделанные в Super не отражаются на переданном значении. Для того, чтобы код работал как ожидается, нужно передать указатель на значение:

```
func main() {  
    goku := &Saiyan{"Goku", 9000}  
    Super(goku)  
    fmt.Println(goku.Power)  
}  
  
func Super(s *Saiyan) {  
    s.Power += 10000  
}
```

Мы сделали два изменения. Первое в том, что мы использовали оператор & для получения адреса. Затем мы изменили входной параметр в функции Super. Ожидалось значения типа Saiyan но теперь

стал тип `*Saiyan`, где `*X` означает *указатель на значение типа X*. Очевидно, что существует связь между типами `Saiyan` и `*Saiyan`, но это все равно два разных типа.

Отметим, что мы все еще передаем копию значения переменной `goku` в `Super`, но теперь значением `goku` является адрес. И это копия того же адреса, что хранится в оригинальной переменной. Думайте об этом как о копии пути в ресторан. То, что у вас есть – это копия, но она ведет к тому же ресторану, что и оригинал.

Мы можем проверить, что это копия, изменив указатель (хотя это вероятно не то, что вы хотели бы сделать):

```
func main() {
    goku := &Saiyan{"Goku", 9000}
    Super(goku)
    fmt.Println(goku.Power)
}

func Super(s *Saiyan) {
    s = &Saiyan{"Gohan", 1000}
}
```

Код выше снова выведет 9000. Так ведут себя многие языки, включая Ruby, Python, Java и C#. Go и в некоторой степени C#, просто делают этот факт очевидным.

Очевидно, что копирование указателя будет стоить дешевле с точки зрения ресурсов, чем копирование сложной структуры. В 64 битной системе указатель занимает 64 бита памяти. Если мы имеем структуру со множеством полей, то создание копии будет дорогой операцией. Смысл указателей в том, что они дают общий доступ к значениям. Хотим ли мы чтобы `Super` изменил копию `goku` или вместо этого изменил общее значение `goku`?

Все это не означает, что вам всегда нужно использовать указатели. В конце этой главы, после того, как мы увидим немного больше

операций со структурами, мы пересмотрим вопрос указатели-против-значения.

Функции в структурах

Мы можем ассоциировать метод со структурой:

```
type Saiyan struct {  
    Name string  
    Power int  
}  
  
func (s *Saiyan) Super() {  
    s.Power += 10000  
}
```

В коде выше мы говорим, что тип `*Saiyan` это **получатель** метода `Super`. Мы можем вызвать `Super` так:

```
goku := &Saiyan{"Goku", 9001}  
goku.Super()  
fmt.Println(goku.Power) // will print 19001
```

Конструкторы

Структуры не имеют конструкторов. Вместо этого, вы создаёте функцию, которая возвращает экземпляр нужного типа (как фабрика):

```
func NewSaiyan(name string, power int) *Saiyan {  
    return &Saiyan{  
        Name: name,  
        Power: power,  
    }  
}
```

Этот шаблон направляет многих разработчиков на неверный путь. С одной стороны – это лишь небольшое изменение синтаксиса, с другой – это позволяет чувствовать себя немного менее разобренным.

Наша фабрика не должна возвращать указатель; это абсолютно справедливо:

```
func NewSaiyan(name string, power int) Saiyan {  
    return Saiyan{  
        Name: name,  
        Power: power,  
    }  
}
```

New

Несмотря на отсутствие конструкторов, в Go есть встроенная функция `new`, которая используется для выделения памяти, требуемой каким-то типом данных. Результат от `new(X)` будет такой же, как и от `&X{}`:

```
goku := new(Saiyan)  
// тоже самое  
goku := &Saiyan{}
```

Какой метод использовать, решать вам, но многие люди предпочитают второй, так как он позволяет сразу инициализировать поля, что более удобно для чтения:

```
goku := new(Saiyan)  
goku.name = "goku"  
goku.power = 9001  
  
// против  
  
goku := &Saiyan {  
    name: "goku",  
    power: 9000,  
}
```

Какой бы способ вы не выбрали, если вы будете использовать шаблон с фабрикой выше, вы сможете отградить остальной код от деталей инициализации.

Поля структур

В примерах, которые мы видели ранее, структура `Saiyan` имела два поля `Name` и `Power` типа `string` и `int` соответственно. Поля могут быть любого типа, включая другие структуры и типы, которые мы еще не рассматривали, такие как: массивы, карты, интерфейсы и функции.

Например, мы могли бы расширить определение `Saiyan`:

```
type Saiyan struct {
    Name string
    Power int
    Father *Saiyan
}
```

и инициализировали бы так:

```
gohan := &Saiyan{
    Name: "Gohan",
    Power: 1000,
    Father: &Saiyan {
        Name: "Goku",
        Power: 9001,
        Father: nil,
    },
}
```

Композиция

Go поддерживает композицию, которая является включением одной структуры в другую. В некоторых языках это называется трейт (trait) или примесь (mixin). Языки, которые не имеют явной поддержки механизма композиции всегда могут пойти долгим путем. В Java:

```

public class Person {
    private String name;

    public String getName() {
        return this.name;
    }
}

public class Saiyan {
    // говорим, что Saiyan включает Person
    private Person person;

    // ме переадресуем вызов классу Person
    public String getName() {
        return this.person.getName();
    }
    ...
}

```

Это довольно утомительно. Каждый метод класса `Person` нужно продублировать в классе `Saiyan`. Go избегает этого занудства:

```

type Person struct {
    Name string
}

func (p *Person) Introduce() {
    fmt.Printf("Hi, I'm %s\n", p.Name)
}

type Saiyan struct {
    *Person
    Power int
}

// и для использования этого:
goku := &Saiyan{
    Person: &Person{"Goku"},
    Power: 9001,
}
goku.Introduce()

```

Структура `Saiyan` имеет поле типа `*Person`. Так как мы не дали явного имени полю, мы получаем косвенный доступ к полям и методам составного типа. Однако, компилятор Go дал имя этому полю, что прекрасно видно:

```
goku := &Saiyan{
    Person: &Person{"Goku"},
}
fmt.Println(goku.Name)
fmt.Println(goku.Person.Name)
```

Оба метода выведут "Goku".

Композиция лучше наследования? Многие люди считают, что это более надежный способ делиться кодом. При использовании наследования, ваш класс тесно связан с суперклассом и в конечном итоге вы сфокусированы на иерархии, а не на поведении.

Перегрузка

Перегрузка не является специфичной операцией для структур, она стоит адресации. Проще говоря, Go не поддерживает перегрузку. По этой причине вы увидите (и напишете) множество функций вроде `Load`, `LoadById`, `LoadByName` и так далее.

Тем не менее, поскольку неявная композиция на самом деле это трюк компилятора, мы можем "переписать" функции композитного типа. Например, наша структура `Saiyan` может иметь собственную функцию `Introduce`:

```
func (s *Saiyan) Introduce() {
    fmt.Printf("Hi, I'm %s. Ya!\n", s.Name)
}
```

Композитная функция всегда доступна через `s.Person.Introduce()`.

Указатели против значений

Когда вы пишете код на Go, вполне естественно задать себе вопрос: *должен ли я использовать значение или указатель на это значение?* Есть две хорошие новости. Первая – ответ не зависит от следующих элементов, перечисленных ниже:

- Присваивание локальной переменной
- Поле в структуре
- Возвращение значения из функции
- Параметры функции
- Получатель метода

Вторая – если вы не уверены, используйте указатель.

Как мы уже видели, передача значения является хорошим способом сделать данные неизменяемыми (изменения, совершенные в функции не влияют на исходное значение). Иногда это тот результат которого вы хотите, но чаще это не так.

Даже если вы не собираетесь изменять данные, учитывайте стоимость создания копий больших структур. И наоборот, возможно, у вас есть маленькие структуры:

```
type Point struct {  
    X int  
    Y int  
}
```

В таких случаях стоимость копирования структуры будет смещена в пользу прямого доступа к `x` и `y` непосредственно без какой-либо косвенности.

Опять же, это всё тонкие случаи. Если вы не производите итерацию по тысячам или десяткам тысяч таких указателей, вы, возможно, не заметите разницу.

Перед тем как продолжить

С практической точки зрения, эта глава является введением в структуры, о том, как сделать экземпляр структуры получающий функцию и добавляет указатели в ваши знания о системе типов Go. В следующих главах мы будем опираться на то, что уже знаем о структурах и о том как они работают.

Глава 3 - Карты, Массивы и Срезы

Ранее мы уже видели несколько простых структур. Настало время познакомиться с массивами, срезами и картами.

Массивы

Если вы уже знакомы с Python, Ruby, Perl, JavaScript или PHP (и т. д.), при программировании вы скорее всего использовали *динамические массивы*. Это массивы, которые способны изменять свой размер в зависимости от хранимых в них данных. В Go, и как во многих других языках, массивы фиксированы. При объявлении массива необходимо указать его размер, после чего изменить его нельзя:

```
var scores [10]int
scores[0] = 339
```

Массив выше может хранить до 10 очков, используя индексы от `scores[0]` до `scores[9]`. При попытке обращения к индексам, не входящим в этот диапазон, произойдет ошибка на этапе компиляции или выполнении программы.

Мы можем инициализировать массив вместе со значениями:

```
scores := [4]int{9001, 9333, 212, 33}
```

Можно использовать `len` для получения размера массива. `range` используется для итерации по нему:

```
for index, value := range scores {  
  
}
```

Массивы эффективны в использовании, но жестко заданы. Часто мы не знаем заранее число используемых элементов. В таких случаях применяются срезы.

Срезы

В Go вы редко, даже почти никогда, не будете использовать массивы напрямую. Вместо них вы будете использовать срезы. Срез – это легковесная структура, которая представляет собой часть массива. Есть несколько способов создать срез, и мы позже рассмотрим их подробнее. Первый способ является слегка измененным способом объявления массива:

```
scores := []int{1, 4, 293, 4, 9}
```

В отличие от декларирования массива, срез объявлен без указания длины в квадратных скобках. Для того, чтобы понять их различия, давайте рассмотрим другой способ создания среза с использованием `make`:

```
scores := make([]int, 10)
```

Мы используем `make` вместо `new` потому, что при создании среза происходит немного больше, чем просто выделение памяти (что делает `new`). В частности, мы должны выделить память для массива, а также инициализировать срез. В приведенном выше примере мы создаем срез длиной 10 и вместимостью 10. Длина – это размер среза. Вместимость – это размер лежащего в его основе массива. При использовании `make` мы можем указать эти два параметра отдельно:

```
scores := make([]int, 0, 10)
```

Эта инструкция создает срез с длиной 0 и вместимостью 10. (Если вы были внимательны, вы могли заметить, что `make` и `len` *были* перегружены. Go — это такой язык, в котором, к разочарованию некоторых, используются возможности, недоступные разработчикам).

Для лучшего понимания взаимосвязи длины и вместимости, рассмотрим несколько примеров:

```
func main() {  
    scores := make([]int, 0, 10)  
    scores[5] = 9033  
    fmt.Println(scores)  
}
```

Наш первый пример не работает. Почему? Потому, что срез имеет длину 0. Да, в его основе лежит массив, содержащий 10 элементов, но нам нужно явно расширить срез для получения доступа к этим элементам. Один из способов расширить срез — это `append`:

```
func main() {  
    scores := make([]int, 0, 10)  
    scores = append(scores, 5)  
    fmt.Println(scores) // выведет [5]  
}
```

Но такой способ изменит смысл оригинального кода. Добавление элемента к срезу длиной 0 является установкой первого значения. По определённым причинам наш нерабочий код требует установки элемента по индексу 5. Чтобы это сделать, мы должны пере-срезать наш срез:

```
func main() {  
    scores := make([]int, 0, 10)  
    scores = scores[0:6]  
}
```

```
scores[5] = 9033
fmt.Println(scores)
}
```

Как сильно мы можем изменить размер среза? До размера его вместимости, в нашем случае это 10.

Вы можете подумать *на самом деле это не решает проблему фиксированной длины массивов*. Оказывается, что `append` это что-то особенное. Если основной массив заполнен, создается больший массив и все значения копируются в него (также работают динамические массивы в PHP, Python, Ruby, JavaScript, ...). Поэтому пример выше использует `append`, мы должны повторно присвоить значение, которое было возвращено `append` переменной `scores`: `append` может создать новое значение, если в исходном не хватает места.

Если я скажу вам, что Go увеличивает массивы в два раза, вы сможете догадаться, что выведет данный код?

```
func main() {
    scores := make([]int, 0, 5)
    c := cap(scores)
    fmt.Println(c)

    for i := 0; i < 25; i++ {
        scores = append(scores, i)

        // если вместимость изменена,
        // Go увеличивает массив, чтобы приспособиться к новым
        // данным
        if cap(scores) != c {
            c = cap(scores)
            fmt.Println(c)
        }
    }
}
```

Изначальная вместимость переменной `scores` это 5. Для того, чтобы вместить 20 значений, она должна быть расширена 3 раза до вместимости в 10, 20 и наконец 40.

И как последний пример, рассмотрим:

```
func main() {  
    scores := make([]int, 5)  
    scores = append(scores, 9332)  
    fmt.Println(scores)  
}
```

Здесь вывод будет [0, 0, 0, 0, 0, 9332]. Возможно, вы думали что получится [9332, 0, 0, 0, 0]? Для человека это выглядит логично. Но для компилятора, вы говорите: добавить значение к срезу, который уже содержит 5 значений.

В итоге, есть четыре способа инициализировать срез:

```
names := []string{"leto", "jessica", "paul"}  
checks := make([]bool, 10)  
var names []string  
scores := make([]int, 0, 20)
```

Когда какой использовать? Первый не требует особых объяснений. Его можно использовать когда вы заранее знаете значения массива.

Второй полезен когда вам нужно записывать значения по определенным индексам среза. Например:

```
func extractPowers(saiyans []*Saiyans) []int {  
    powers := make([]int, len(saiyans))  
    for index, saiyan := range saiyans {  
        powers[index] = saiyan.Power  
    }  
    return powers  
}
```

Третий случай – это пустой срез. Используется в сочетании с `append`, когда число элементов заранее неизвестно.

Последний способ позволяет задать изначальную вместимость; полезен когда у вас есть общее представление о том, сколько элементов вам нужно.

Даже если вы знаете размер, можно использовать `append`. Это момент по большей части зависит от ваших предпочтений:

```
func extractPowers(saiyans []*Saiyans) []int {
    powers := make([]int, 0, len(saiyans))
    for _, saiyan := range saiyans {
        powers = append(powers, saiyan.Power)
    }
    return powers
}
```

Срезы в роли оберток массивов представляют собой мощный концепт. Во многих языках существует понятие нарезки массива. И в JavaScript и в Ruby массивы имеют метод `slice`. Вы можете получить срез в Ruby используя `[START..END]` или в Python с помощью `[START:END]`. Однако в этих языках срезы в действительности являются новыми массивами со скопированными в них значениями. Если мы возьмем Ruby, что выведет следующий код?

```
scores = [1,2,3,4,5]
slice = scores[2..4]
slice[0] = 999
puts scores
```

Ответ: `[1, 2, 3, 4, 5]`. Потому, что `slice` совершенно новый массив с копией значений. Теперь рассмотрим эквивалент в Go:

```
scores := []int{1,2,3,4,5}
slice := scores[2:4]
slice[0] = 999
fmt.Println(scores)
```

Результат: `[1, 2, 999, 4, 5]`.

Это изменяет принцип кодирования. Например несколько функций принимают номер позиции в качестве параметра. В JavaScript, если вам нужен символ в строке (да, срезы работают со строками тоже!) идущий после пятого, вам нужно написать:

```
haystack = "the spice must flow";
console.log(haystack.indexOf(" ", 5));
```

В Go мы используем срезы:

```
strings.Index(haystack[5:], " ")
```

В примере выше мы видим, что `[x:]` — это сокращение, которое означает *от X до конца*, а `[:x]` это короткая запись, означающая *от начала до X*. В отличие от других языков, Go здесь не поддерживает отрицательные индексы. Если мы хотим получить все значения среза, кроме последнего, нам нужно выполнить:

```
scores := []int{1,2,3,4,5}
scores = scores[:len(scores)-1]
```

С помощью этого способа мы можем реализовать эффективный способ удаления значения из несортированного среза:

```
func main() {
    scores := []int{1,2,3,4,5}
    scores = removeAtIndex(scores, 2)
    fmt.Println(scores)
}

func removeAtIndex(source []int, index int) []int {
    lastIndex := len(source) - 1
    //меняем последнее значение и значение, которое хотим
    //удалить, местами
    source[index], source[lastIndex] = source[lastIndex],
    source[index]
```

```
    return source[:lastIndex]
}
```

Наконец, когда мы уже достаточно знаем о срезах, давайте взглянем ещё на одну часто используемую функцию: `copy`. `copy` одна из тех функций, которая показывает как срезы влияют на способ кодирования. Обычно метод, который копирует значения из одного массива в другой имеет 5 параметров: `source`, `sourceStart`, `count`, `destination` и `destinationStart`. При работе со срезами нам нужны только два:

```
import (
    "fmt"
    "math/rand"
    "sort"
)

func main() {
    scores := make([]int, 100)
    for i := 0; i < 100; i++ {
        scores[i] = int(rand.Int31n(1000))
    }
    sort.Ints(scores)

    worst := make([]int, 5)
    copy(worst, scores[:5])
    fmt.Println(worst)
}
```

Немного поиграйте с кодом выше. Попробуйте различные вариации. Посмотрите, что произойдет, если вы измените копирование на `copy(worst[2:4], scores[:5])`, или посмотрите, что будет если вы попытаетесь скопировать больше, чем 5 значений в `worst`?

Карты

Карты в Go — это то, что в других языках называют хеш-таблицами или словарями. Они работают так, как и ожидается: вы определяете ключ и значение, можете получать, устанавливать и удалять значения.

Карты, как и срезы, создаются с помощью функции `make`. Давайте взглянем на пример:

```
func main() {
    lookup := make(map[string]int)
    lookup["goku"] = 9001
    power, exists := lookup["vegeta"]

    // prints 0, false
    // 0 это значение по умолчанию для типа integer
    fmt.Println(power, exists)
}
```

Для получения количества ключей используйте `len`. Для удаления значения по определенному ключу вызывайте `delete`:

```
// returns 1
total := len(lookup)

// ничего не возвращает, можно указывать несуществующий ключ
delete(lookup, "goku")
```

Карты увеличиваются динамически. Однако вы можете указать второй аргумент в `make` для установки начального значения:

```
lookup := make(map[string]int, 100)
```

Если вы имеете какое-то представление о том, сколько ключей вам понадобится в карте, указание начального размера может помочь с производительностью.

Когда вам нужна карта в роли поля структуры, вы указываете её так:

```
type Saiyan struct {
    Name string
    Friends map[string]*Saiyan
}
```

Один из способов инициализации:

```
goku := &Saiyan{
    Name: "Goku",
    Friends: make(map[string]*Saiyan),
}
goku.Friends["krillin"] = ... //загрузить или создать Krillin
```

Существует еще один способ объявления и инициализации значений в Go. Как и `make`, этот подход является специфичным для карт и массивов. Вы можете объявить карту как составной литерал:

```
lookup := map[string]int{
    "goku": 9001,
    "gohan": 2044,
}
```

Итерация по карте производится с помощью цикла `for` в комбинации с ключевым словом `range`:

```
for key, value := range lookup {
    ...
}
```

Итерация по карте происходит не по порядку. Каждая итерация будет возвращать пару ключа и значения в случайном порядке.

Указатели против значений

Мы закончили главу 2 вопросом о том, следует ли присваивать и передавать указатели или значения. Сейчас вернемся к нему говоря уже о массивах и картах. Какой способ стоит использовать?

```
a := make([]Saiyan, 10)
//или
b := make([]*Saiyan, 10)
```

Многие разработчики считают, что при передаче или возвращении в функцию будет более эффективной. Тем не менее то, что передается/возвращается является копией среза, который в свою очередь является ссылкой. Таким образом в отношении передачи/возврата самого среза нет никакой разницы.

Разница будет видна когда вы изменяете значение среза или карты. В этом случае логика такая же как и в конце главы 2. Так что решение о том, объявлять ли массив указателей или массив значений, принимается исходя из того, что вы будете делать со значениями, а не с самими картами или массивами.

Перед тем как продолжить

Массивы и карты в Go работают так же, как и в других языках. При использовании динамических массивов существуют небольшие изменения, но `append` должен избавить вас от большинства проблем. Если мы хотим выйти за пределы поверхности массивов, мы используем срезы. Срезы – мощные конструкции, оказывающие большое влияние на чистоту вашего кода.

Мы не рассмотрели несколько крайних случаев, но вам скорее всего не придется вникать в них так глубоко. Хотя если это и понадобится, надеюсь основы, полученные здесь, помогут вам самостоятельно разобраться в том, что происходит.

Глава 4 - Организация кода и интерфейсы

Настало время поговорить о том, как организовывать ваш код.

Пакеты

Для того, чтобы хранить сложные системы и библиотеки организованно, нам нужно научиться пользоваться пакетами. В Go имена пакетов следуют структуре директорий в вашем рабочем пространстве. Если мы создаём систему для покупок, вероятно мы начнем с пакета по имени "shopping" и сохраним его исходные файлы в папке `$GOPATH/src/shopping/`.

Однако не хочется хранить все подряд в одной папке. Например, возможно мы захотим вынести логику работы с базой данных в другую директорию. Для этого, мы создадим поддиректорию в `$GOPATH/src/shopping/db`. Имя пакета для файлов в этом подкаталоге будет просто `db`, но для получения доступа к ним из других пакетов, включая `shopping`, нам нужно импортировать `shopping/db`.

Другими словами, когда вы указываете пакет с помощью ключевого слова `package`, вы используете одно название, а не полную иерархию (например "shopping" или "db"). Когда вы импортируете пакет, необходимо указывать полный путь.

Давайте попробуем. Внутри папки `src` рабочего пространства Go (которое мы задали в разделе «Приступая к работе» во [введении](#)), создадим новую директорию с именем `shopping` и поддиректорию, которую назовём `db`.

Внутри `shopping/db`, создайте файл с именем `db.go` и напишите следующий код:

```

package db

type Item struct {
    Price float64
}

func LoadItem(id int) *Item {
    return &Item{
        Price: 9.001,
    }
}

```

Заметьте, что имя пакета совпадает с именем директории. Кроме того, очевидно, что на самом деле мы не подключаемся к базе данных. Мы просто будем использовать этот пример, чтобы показать, как организовывать код.

Теперь создайте файл с именем `pricecheck.go` внутри папки `shopping`. С таким содержанием:

```

package shopping

import (
    "shopping/db"
)

func PriceCheck(itemId int) (float64, bool) {
    item := db.LoadItem(itemId)
    if item == nil {
        return 0, false
    }
    return item.Price, true
}

```

Этот код наталкивает на мысль, что импорт `shopping/db` почему-то так задан специально, ведь мы уже находимся в пакете/папке `shopping`. В действительности, вы импортируете `$GOPATH/src/shopping/db`, это означает, что вы так же легко можете импортировать `test/db`, если у вас есть пакет с именем `db` внутри папки `src/test` вашего рабочего пространства.

Если вы создаёте пакет, вам больше ничего не нужно знать, кроме того, что вы уже узнали. Для того, чтобы создать исполняемый файл, вам ещё нужен `main`. Я предпочитаю делать его в папке `main` внутри `shopping` с помощью файла с именем `main.go` следующего содержания:

```
package main

import (
    "shopping"
    "fmt"
)

func main() {
    fmt.Println(shopping.PriceCheck(4343))
}
```

Вы можете запустить ваш код перейдя в проект `shopping` и набрав:

```
go run main/main.go
```

Циклические импорты

Когда вы начнете писать более сложные системы, вы обязательно столкнётесь с циклическими импортами. Это происходит когда пакет А импортирует пакет Б, а пакет Б импортирует пакет А (непосредственно или косвенно через другой пакет). Это то, что компилятор делать не позволяет.

Давайте изменим структуру нашей системы покупок так, чтобы вызвать ошибку.

Переместите определение `Item` из `shopping/db/db.go` в `shopping/pricecheck.go`. Файл `pricecheck.go` должен выглядеть так:

```
package shopping

import (
    "shopping/db"
```

```

)

type Item struct {
    Price float64
}

func PriceCheck(itemId int) (float64, bool) {
    item := db.LoadItem(itemId)
    if item == nil {
        return 0, false
    }
    return item.Price, true
}

```

Если вы попытаетесь запустить код, вы получите несколько ошибок в `db/db.go` о том, что `Item` не был определён. Это логично. `Item` больше не существует в пакете `db`; он был перемещен в пакет `shopping`. Нам нужно изменить файл `shopping/db/db.go` так:

```

package db

import (
    "shopping"
)

func LoadItem(id int) *shopping.Item {
    return &shopping.Item{
        Price: 9.001,
    }
}

```

Теперь, когда вы запустите код, вы получите ошибку: *import cycle not allowed*. Мы исправим её с помощью добавления нового пакета, который будет содержать общие структуры. Директория в итоге будет выглядеть так:

```

$GOPATH/src
- shopping
  pricecheck.go
  db
  db.go
- models

```

```
    item.go
- main
  main.go
```

pricecheck.go всё ещё импортирует shopping/db, но db.go теперь импортирует shopping/models вместо shopping, тем самым прерывая цикл. Так как мы переместили общую структуру Item в shopping/models/item.go, нам нужно изменить в shopping/db/db.go ссылку на структуру Item из пакета models:

```
package db

import (
    "shopping/models"
)

func LoadItem(id int) *models.Item {
    return &models.Item{
        Price: 9.001,
    }
}
```

Вам часто будет необходимо получать доступ не только к одному пакету models, но, возможно, к другим похожим по смыслу папкам вроде utilities и т. д. Важное правило использования общих пакетов заключается в том, что они не должны импортировать ничего из пакета shopping или его вложенных пакетов. Вскоре мы рассмотрим интерфейсы, которые помогут нам разрешить такие зависимости.

Видимость

Go использует простое правило определения видимости типов данных и функций извне пакета. Если имя функции или типа начинается с большой буквы, то они видимы. Если с маленькой, то нет.

Это правило распространяется и на поля структур. Если поле структуры начинается с маленькой буквы, то доступ к нему получает только тот код, который находится в пределах того же пакета.

Например, если наш файл `items.go` содержит функцию, которая выглядит так:

```
func NewItem() *Item {  
    // ...  
}
```

То она может быть вызвана с помощью `models.NewItem()`. Но если функцию назвать `newItem`, получить доступ к ней из другого пакета не получится.

Попробуйте изменить имя другой функции, структуры или поля из пакета `shopping`. Например, если вы измените имя поля `Price` структуры `Item` на `price`, то вы получите ошибку.

Управление пакетами

Мы использовали команду `go` для запуска и сборки с помощью `run` и `build`, но у нее есть еще команда `get`, которая используется для получения библиотек сторонних разработчиков. `go get` поддерживает различные протоколы, но для этого примера мы получим библиотеку с сайта GitHub. Для этого вы должны установить `git` на свой компьютер.

Когда `git` установлен, из командной строки выполните:

```
go get github.com/mattn/go-sqlite3
```

`go get` забирает удалённые файлы и сохраняет их в вашем рабочем пространстве. Проверьте свою папку `$GOPATH/src`. В дополнении к проекту `shopping`, который создали мы, вы увидите папку `github.com`. Внутри находится папка `mattn`, а в ней папка `go-sqlite3`.

Мы только что говорили о том, как импортировать пакеты, которые лежат в вашем рабочем пространстве. Для использования нового пакета `go-sqlite3` мы импортируем его так:

```
import (  
    "github.com/mattn/go-sqlite3"  
)
```

Я знаю, это выглядит как URL адрес, но на самом деле происходит импорт пакета `go-sqlite3`, который должен находиться в папке `$GOPATH/src/github.com/mattn/go-sqlite3`.

Управление зависимостями

Команда `go get` имеет пару дополнительных хитростей. Если мы выполним `go get` внутри проекта, она просмотрит все файлы на наличие библиотек третьих лиц в блоках `import` и скачает их. В некотором смысле, весь наш исходный код является чем-то вроде `Gemfile` или `package.json`.

Если вы выполните `go get -u`, произойдёт обновление пакетов (или определённого пакета при выполнении `go get -u ПОЛНОЕ_ИМЯ_ПАКЕТА`).

В конечном счете работа `go get` вам может показаться неадекватной. По той причине, что нет возможности указать конкретную ревизию, всегда будет использована последняя версия из `master/head/trunk/default`. Это может стать проблемой, если у вас два проекта и вам необходимы две разные версии одной библиотеки.

Чтобы избавиться от этой проблемы вы можете использовать утилиты управления зависимостями от сторонних разработчиков. Все они достаточно молоды, но две из них выглядят перспективными – [goop](#) и [godep](#). Полный список доступен на [go-wiki](#).

Интерфейсы

Интерфейс – это такой тип, который определяет описание, но не содержит реализации:

```
type Logger interface {  
    Log(message string)  
}
```

Вам может показаться непонятным то, для каких целей их можно применить. Интерфейсы помогают разделять ваш код в зависимости от реализации. Например, мы можете иметь несколько разных типов, используемых для логирования:

```
type SqlLogger struct { ... }  
type ConsoleLogger struct { ... }  
type FileLogger struct { ... }
```

При программировании с помощью интерфейсов вместо их конкретных реализаций, вы можете легко изменять (и тестировать) разные варианты без внесения изменений в ваш код.

Как можно использовать интерфейс? Так же как и любой другой тип, он может быть полем в структуре:

```
type Server struct {  
    logger Logger  
}
```

или параметром функции (или возвращаемым значением):

```
func process(logger Logger) {  
    logger.Log("hello!")  
}
```

В таких языках как C# или Java мы должны явно указывать, что класс реализует интерфейс:

```
public class ConsoleLogger : Logger {  
    public void Log(message string) {  
        Console.WriteLine(message)  
    }  
}
```

```
}  
}
```

В Go это происходит неявно. Если ваша структура имеет функцию с именем `Log` принимающую параметр `string` и не возвращающую значений, то она может быть использована как `Logger`. Это избавляет от подробного описания при использовании интерфейсов:

```
type ConsoleLogger struct {}  
func (l ConsoleLogger) Log(message string) {  
    fmt.Println(message)  
}
```

Также существует тенденция к использованию небольших и целенаправленных интерфейсов. Интерфейсов полно в стандартной библиотеке. Пакет `io` содержит целую кучу реализаций `io.Reader`, `io.Writer`, и `io.Closer`. Если вы пишете функцию, которая ожидает параметр с методом, который вы назвали `Close()`, то в качестве типа параметра лучше использовать `io.Closer` вместо вашего конкретного типа.

Интерфейсы могут использовать композицию. И интерфейсы сами по себе могут состоять из интерфейсов. Например, интерфейс `io.ReadCloser` состоит из `io.Reader` и `io.Closer`.

Наконец, интерфейсы обычно используются для предотвращения циклических импортов. Так как они не содержат реализации, они не будут иметь лишних зависимостей.

Перед тем как продолжить

В конечном счете то, как структурировать ваш Go код в рабочем пространстве, вы поймёте после того, как напишете пару нетривиальных проектов. Главное запомнить, что существует тесная связь между именами пакетов и структурой директорий (не только в самом проекте, но и во всём рабочем пространстве).

Путь Go в определении областей видимости очень прост и эффективен. Он просто согласован. Несколько вещей мы не рассмотрели, таких как константы и глобальные переменные, но будьте уверены, их видимость определяется по тому же правилу.

И, наконец, если вы новичок в использовании интерфейсов, вам может понадобиться некоторое время, чтобы получить какое-то представление о них. Тем не менее, когда вы впервые увидите, что функция ожидает какой-то параметр вроде `io.Reader`, мысленно поблагодарите её автора за то, что он не требует передавать ему больше чем, нужно.

Глава 5 - Лакомые кусочки

В этой главе мы поговорим о возможностях Go, которые не вписываются в остальные разделы.

Обработка ошибок

Предпочтительным способом обработки ошибок в Go является возвращение значений вместо исключений. Взглянем на функцию `strconv.Atoi`, которая принимает строку и пытается конвертировать её в целое число:

```
package main

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

func main() {
    if len(os.Args) != 2 {
        os.Exit(1)
    }

    n, err := strconv.Atoi(os.Args[1])
    if err != nil {
        fmt.Println("не является числом")
    } else {
        fmt.Println(n)
    }
}
```

Вы можете создать свой тип ошибок. Единственное требование, которое необходимо выполнить, это реализовать встроенный интерфейс `error`:

```
type error interface {  
    Error() string  
}
```

Также мы можем создать свою ошибку с помощью импорта пакета `errors` и вызова функции `New`:

```
import (  
    "errors"  
)  
  
func process(count int) error {  
    if count < 1 {  
        return errors.New("Invalid count")  
    }  
    ...  
    return nil  
}
```

Это общепринятый способ использования переменных ошибок в стандартной библиотеке Go. Например, в пакете `io` есть переменная `EOF`, которая определяется так:

```
var EOF = errors.New("EOF")
```

Это переменная пакета (определённая вне функции), которая имеет публичный доступ (имя начинается с большой буквы). Различные функции могут возвращать эту ошибку во время чтения из файла или `STDIN`. Если в вашем контексте она имеет смысл, вам тоже нужно её использовать. Так можно обработать чтение только одного файла:

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

```
func main() {
    var input int
    _, err := fmt.Scan(&input)
    if err == io.EOF {
        fmt.Println("no more input!")
    }
}
```

И последнее замечание, в Go есть функции `panic` и `recover`. `panic` похожа на выброс исключения, а `recover` на `catch`. Они редко используются.

Defer

Хотя в Go и есть сборщик мусора, некоторые ресурсы требуют, чтобы мы явно освобождали их. Например, нам нужно вызывать `Close()` для закрытия файла, после того, как работа с ним окончена. Такой код всегда опасен. С одной стороны, когда мы пишем функцию, легко забыть вызвать `Close`, для того, что мы открыли на 10 строк выше. С другой, функция может иметь несколько мест с возвращением результата, и нужно вызывать закрытие ресурса в каждом из них. Решением в Go является ключевое слово `defer`:

```
package main

import (
    "fmt"
    "os"
)

func main() {
    file, err := os.Open("a_file_to_read")
    if err != nil {
        fmt.Println(err)
        return
    }
    defer file.Close()
    // read the file
}
```

Если вы попытаетесь выполнить этот код, вы, вероятно, получите ошибку (файл не существует). Смысл в том, чтобы показать как работает `defer`. Неважно, где находится ваш `defer`, он всё равно будет выполнен после того, как метод вернет результат. Это позволяет вам освобождать ресурсы прямо там же, где вы их инициализировали и спасает от дублирования кода в случае нескольких `return`.

go fmt

Большинство программ, написанных на Go, используют одинаковые правила форматирования. Символ табуляции используется для отступа, а скобка ставится на той же строке, что и инструкция.

Я знаю, у вас есть свой собственный стиль и вы придерживаетесь его. Я следовал ему долгое время, но я рад, что в конечном итоге сдался. Главная причина этому была в команде `go fmt`. Она проста в использовании и не вызывает споров (по поводу личных предпочтений).

Когда вы находитесь внутри проекта, вы можете применить правила форматирования для него и всех под-проектов с помощью:

```
go fmt ./...
```

Попробуйте. Она делает больше, чем просто расставляет отступы. Она выравнивает объявления полей, а так же сортирует ваши импорты в алфавитном порядке.

Инициализация в условии

Go поддерживает немного модифицированные условные блоки. Инициализация значения в них имеет приоритет при вычислении условия:

```
if x := 10; count > x {  
    ...  
}
```

Это довольно простой пример. В реальном коде будет что-то такое:

```
if err := process(); err != nil {  
    return err  
}
```

Интересно, что в то время, как эти значения недоступны вне инструкции `if`, они доступны внутри `else if` или `else`.

Пустой интерфейс и преобразования

В многих объектно-ориентированных языках существует базовый класс, часто называемый `object`, который является супер классом для всех остальных классов. В Go нет наследования и нет такого супер класса. Что у него есть, так это пустой интерфейс без методов: `interface{}`. Так как каждый тип реализует все 0 методов этого интерфейса, то можно сказать, что в неявном виде каждый тип реализует пустой интерфейс.

Если бы нам было нужно, мы бы могли написать функцию `add` со следующей сигнатурой:

```
func add(a interface{}, b interface{}) interface{} {  
    ...  
}
```

Для преобразования переменной в определенный тип, используйте `.` (ТИП):

```
return a.(int) + b.(int)
```

Также вы можете использовать такой `switch`:

```
switch a.(type) {  
    case int:
```

```

    fmt.Printf("a теперь int и равно %d\n", a)
case bool, string:
    // ...
default:
    // ...
}

```

Вы встретите и, возможно, будете использовать пустой интерфейс чаще, чем может показаться с первого взгляда. Хотя стоит признать, что он не способствует чистоте кода. Преобразование значений туда и обратно – это и некрасиво и опасно. Но иногда в языках со статической типизацией это единственный выбор.

Строки и массивы байтов

Строки и массивы байтов тесно связаны. Мы можем легко конвертировать одно в другое:

```

stra := "the spice must flow"
byts := []byte(stra)
strb := string(byts)

```

На самом деле, такой способ преобразования, является также общим для всех типов. Некоторые функции ожидают явно `int32`, или `int64`, или их беззнаковые эквиваленты. Используется это так:

```

int64(count)

```

Тем не менее, иметь дело с байтами и строками вы, вероятно, будете часто. Заметьте, что когда вы используете `[]byte(x)` или `string(x)`, вы создаёте копию данных. Это необходимо, так как строки в Go неизменяемы.

Строки состоят из *рун* (в Go, *тип rune* является псевдонимом для `int32`), которые являются частями Юникода. Если взять длину строки, мы можете получить не то, чего ожидали. Следующий код выведет 3:

```
fmt.Println(len("椒"))
```

Когда вы производите итерацию по строке с использованием `range`, вы получаете руны, а не байты. Конечно, когда вы переводите строку в `[]byte`, вы получаете корректные данные.

Тип функция

Функция – это тип первого порядка:

```
type Add func(a int, b int) int
```

который можно использовать как угодно – как поле структуры, как параметр, как возвращаемое значение.

```
package main

import (
    "fmt"
)

type Add func(a int, b int) int

func main() {
    fmt.Println(process(func(a int, b int) int{
        return a + b
    })))
}

func process(adder Add) int {
    return adder(1, 2)
}
```

Использование таких функций может помочь отделить код от реализации, как и в случае с интерфейсами.

Перед тем как продолжить

Мы рассмотрели различные аспекты программирования с Go. В частности, мы увидели, как происходит обработка ошибок и как освобождаются ресурсы на примере открытых файлов. Многие люди не любят подход Go к обработке ошибок. Она кажется шагом назад. Иногда, я согласен. Тем не менее, я считаю, что такой код легче отслеживать. `defer` является необычным, но практичным подходом к управлению ресурсами. На самом деле он не привязан конкретно к ресурсам. Вы можете использовать `defer` для любых целей, таких как логирование после завершения функции.

Конечно мы рассмотрели не все вкусности, которые предлагает Go. Но вы должны чувствовать себя достаточно комфортно при понимании того, с чем вы столкнётесь.

Глава 6: Конкурентность

Go часто описывают, как дружелюбный к параллельному программированию язык. Причина этому заключается в предоставлении простого синтаксиса для двух мощных механизмов: горутин и каналов.

Горутины

Горутины похожи на потоки, но они управляются самим Go, а не операционной системой. Код, который запускается как горутина, может работать одновременно с другим кодом. Давайте посмотрим на пример:

```
package main

import (
    "fmt"
    "time"
)

func main() {
    fmt.Println("старт")
    go process()
    time.Sleep(time.Millisecond * 10) // это плохо, не делайте так!
    fmt.Println("готово")
}

func process() {
    fmt.Println("обработка")
}
```

Здесь происходит несколько интересных вещей, но самое главное то, как мы запускаем горутину. Мы просто используем ключевое слово `go`, а затем пишем функцию, которую хотим выполнить. Если нужно выполнить немного кода, мы можем воспользоваться анонимной

функцией. Хотя стоит отметить, что анонимные функции используются не только вместе с горутинами.

```
go func() {  
    fmt.Println("обработка")  
}()
```

Горутинны просты в создании и не несут много накладных расходов. Несколько горутин будут работать по похожему на потоки операционной системы принципу. Это часто называют М:N поточной моделью, потому, что мы имеем М потоков приложения (горутин) запущенных в N потоков операционной системы. В результате этого горутинны дают определённую долю оверхеда (несколько килобайт) по сравнению с потоками ОС. Современное железо позволяет запустить порядка миллиона горутин.

Кроме того, вся внутренняя сложность скрыта. Мы просто говорим, что *этот код должен выполняться параллельно* и позволяем Go позаботиться обо всём остальном.

Если мы вернёмся к примеру, то увидим, что выполняется `Sleep` на несколько миллисекунд. Это необходимо потому, что основной процесс завершается быстрее, чем успеет выполниться горутинна (процесс не ждёт, пока все горутинны выполнят свою работу перед выходом). Для того, чтобы решить эту проблему, нам нужно скоординировать наш код.

Синхронизация

Создание горутин тривиально и они дёшево стоят, поэтому мы можем запустить их много. Однако конкурентный код требует наличия согласованности. Для решения этой проблемы Go предоставляет каналы. Перед тем, как их рассмотреть, я думаю важным будет немного объяснить основы конкурентного программирования.

Написание параллельного кода требует особого внимания к тому, как и где вы будете читать и записывать значения. В некотором роде, это как программирование без сборщика мусора – оно требует, чтобы вы взглянули на данные под другим углом и всегда были бдительны перед возможной опасностью. Рассмотрим:

```
package main

import (
    "fmt"
    "time"
)

var counter = 0

func main() {
    for i := 0; i < 2; i++ {
        go incr()
    }
    time.Sleep(time.Millisecond * 10)
}

func incr() {
    counter++
    fmt.Println(counter)
}
```

Как вы считаете, что выведет этот код?

Если вы думаете, что результат будет 1, 2, вы одновременно правы и нет. Это правда, что, если вы запустите этот код, вы скорее всего получите такой результат. Однако реальность такова, что поведение такого кода неопределённо. Почему? Потому, что мы потенциально имеем несколько (в данном случае две) горютины, пишущие одну и ту же переменную `counter` в одно и то же время. Или, что хуже, одна горютина может читать `counter` в то же время, когда другая записывает в него значение.

Это действительно опасно? Определённо да. `counter++` может выглядеть, как простая строка кода, но в действительности она преобразуется в множество инструкций ассемблера, конкретная

реализация которого зависит от платформы, на которой запущен код. Это правда, что в этом примере скорее всего всё будет работать хорошо. Тем не менее, возможен случай, когда обе горутины обратятся к значению `counter`, когда оно будет равно 0 и вы получите 1, 1 на выходе. В более худшем варианте это приведет к системному сбою или доступу к произвольным данным и инкрементированию их!

Единственная параллельная вещь, которую вы можете делать безопасно с переменной – это читать её значение. Вы можете иметь столько читателей, сколько вам угодно, но запись необходимо синхронизировать. Есть несколько разных способов сделать это, в том числе некоторые действительно атомарные операции, которые опираются на специальные инструкции процессора. Тем не менее, наиболее распространённым подходом является использование мьютекса:

```
package main

import (
    "fmt"
    "time"
    "sync"
)

var (
    counter = 0
    lock sync.Mutex
)

func main() {
    for i := 0; i < 2; i++ {
        go incr()
    }
    time.Sleep(time.Millisecond * 10)
}

func incr() {
    lock.Lock()
    defer lock.Unlock()
    counter++
    fmt.Println(counter)
}
```

Мьютекс обеспечивает последовательный доступ с заблокированному коду. Причина по которой мы определяем нашу блокировку как `lock sync.Mutex` в том, что по умолчанию значение `sync.Mutex` разблокировано.

Выглядит достаточно просто? Но простота в этом примере обманчива. В нем есть целый класс ошибок, которые могут возникать во время параллельного программирования. Прежде всего, не всегда очевидно то, что код должен быть защищен. Хотя может быть и заманчиво использование грубых блокировок (которые охватывают почти весь код), это в первую очередь подрывает саму суть параллельного программирования. Как правило нужны небольшие блокировки, иначе мы превратим десятиполосное шоссе в однополосную дорогу.

Другая проблема в том, что делать со взаимными блокировками. С одной блокировкой проблем нет, но когда вы используете две или больше в одном коде, возникает опасная ситуация, когда горутина А имеет блокировку А и ей необходим доступ к блокировке Б, которую держит горутина Б потому, что ей нужен доступ к блокировке А.

В действительности взаимная блокировка *может* произойти и с одной, если вы забыли освободить её. Это не так опасно, как множественные блокировки (потому, что их *действительно* сложно определить), но вы можете увидеть как это происходит. Попробуйте выполнить:

```
package main

import (
    "time"
    "sync"
)

var (
    lock sync.Mutex
)

func main() {
    go func() { lock.Lock() }()
    time.Sleep(time.Millisecond * 10)
```

```
    lock.Lock()
}
```

При параллельном программировании происходит больше вещей, чем мы здесь видим. Поскольку мы имеем множество единовременных читателей, существует один общий мьютекс чтения-записи. Он запускает две блокирующие функции: одна для блокировки чтения и другая для блокировки записи. В Go, `sync.RWMutex` — это просто замок. В дополнении к методам `Lock` и `Unlock` обычного `sync.Mutex`, он предоставляет еще и методы `RLock` и `RUnlock`, где `R` означает *Read* (чтение). В то время, как мьютексы чтения-записи широко используются, они добавляют забот разработчикам. Мы должны обращать внимание не только на то, когда обращаться к данным, но и на то как.

Кроме того, часть параллельного программирования связана не только с обеспечением сериализации доступа между различными частями кода, но и с координацией работы множества горутин. Например, засыпание на 10 миллисекунд не является элегантным решением. Что будет, если выполнение горутин займёт больше, чем 10 миллисекунд? А что, если меньше, и мы просто зря потратим ресурсы в ожидании? Что если бы вместо простого ожидания горутин мы могли бы сказать: *эй, у меня есть новые данные для твоей обработки?*

Все эти вещи выполнимы и без каналов. Конечно для простых случаев я считаю, что вы **должны** использовать такие примитивы как `sync.Mutex` и `sync.RWMutex`, но как мы увидим в следующем разделе, каналы помогают сделать код параллельного программирования более чистым и защищенным от ошибок.

Каналы

Настоящим испытанием в конкурентном программировании является совместное использование данных. Если ваша горутина не предоставляет никаких данных, вам не нужно заботиться о её синхронизации. Хотя это и подходит не для всех систем. По факту, многие системы создаются с целью разделения данных между

несколькими запросами. Кэширование в памяти или база данных будут хорошими примерами таких систем.

Каналы помогают сделать конкурентное программирование более разумным, выделяя совместные данные из общей картины. Канал – это труба для взаимодействия между горутинами, которая используется для передачи данных. Другими словами, горутина, которая имеет данные, может передать их в другую горутину с помощью канала. В результате, в любой момент времени только одна горутина имеет доступ к данным.

Канал, как и всё остальное, имеет тип. Это тип данных, передаваемых через канал. Например, чтобы создать канал для передачи целых чисел, мы делаем:

```
c := make(chan int)
```

Типом этого канала является `chan int`. Для передачи этого канала в функцию используем такое определение:

```
func worker(c chan int) { ... }
```

Каналы поддерживают две операции: приём и отправка. Мы отправляем в канал выполняя:

```
CHANNEL <- DATA
```

и получаем из него

```
VAR := <-CHANNEL
```

Стрелка указывает направление потока данных. Когда происходит отправка, данные передаются в канал. При получении данные извлекаются из канала.

Последняя вещь, которую необходимо знать перед тем, как посмотреть на первый пример, это то, что прием и отправка данных в и из канала

являются блокирующими операциями. Это значит, что во время получения данных из канала выполнение горютины останавливается пока данные не доступны. Аналогично когда мы отправляем в канал, выполнение не продолжается пока данные не получены.

Рассмотрим систему, в которой входящие данные нам необходимо обработать в отдельных горютинах. Это обычное требование. Если мы делаем тяжелую обработку в горютине, которая принимает входящие данные, мы рискуем тем, что клиент может отключиться по тайм-ауту. Сначала мы напишем обработчик. Он может быть простой функцией, но я сделаю его частью структуры, так как до этого мы еще не видели горютин используемых таким образом:

```
type Worker struct {  
    id int  
}  
  
func (w Worker) process(c chan int) {  
    for {  
        data := <-c  
        fmt.Printf("обработчик %d получил %d\n", w.id, data)  
    }  
}
```

Наш обработчик прост. Он ждет пока данные не станут доступны, затем "обрабатывает" их. Он послушно делает это в бесконечном цикле ожидая данных для обработки.

Для начала, запустим несколько обработчиков:

```
c := make(chan int)  
for i := 0; i < 4; i++ {  
    worker := Worker{id: i}  
    go worker.process(c)  
}
```

А затем дадим им немного работы:

```

for {
    c <- rand.Int()
    time.Sleep(time.Millisecond * 50)
}

```

Полный код для запуска:

```

package main

import (
    "fmt"
    "time"
    "math/rand"
)

func main() {
    c := make(chan int)
    for i := 0; i < 5; i++ {
        worker := &Worker{id: i}
        go worker.process(c)
    }

    for {
        c <- rand.Int()
        time.Sleep(time.Millisecond * 50)
    }
}

type Worker struct {
    id int
}

func (w *Worker) process(c chan int) {
    for {
        data := <-c
        fmt.Printf("обработчик %d получил %d\n", w.id, data)
    }
}

```

Мы не знаем какой именно обработчик какие данные получает. Но мы знаем то, что Go гарантирует, что данные, отправляемые в канал, будут приняты только одним получателем.

Обратите внимание на то, что единственное общее состояние, которое мы можем безопасно получать и отправлять одновременно – это канал. Каналы предоставляют весь код, необходимый для синхронизации, и позволяют убедиться в том, что только одна горутина имеет доступ к определенному участку данных.

Буферизированные каналы

А что произойдёт с кодом выше, если он получит больше данных, чем сможет обработать? Вы можете симулировать такую ситуацию изменив обработчик, добавив задержку после получения данных:

```
for {
    data := <-c
    fmt.Printf("обработчик %d получил %d\n", w.id, data)
    time.Sleep(time.Millisecond * 500)
}
```

Произойдёт то, что наш код, принимающий пользовательские данные (которые мы симулировали с помощью генератора случайных чисел) перестанет их отправлять в каналы потому, что не будет доступных получателей.

В тех случаях, когда вам необходимо гарантировать обработку данных, вы, возможно, захотите заблокировать их получение на некоторое время. В других случаях, вы можете пойти на уменьшение гарантий обработки. Существуют несколько популярных методов это сделать. Первый – это буфер данных. Если нет доступных обработчиков, нам нужно временно сохранить данные в какого-либо рода очередь. Каналы имеют встроенную поддержку буферизации. Когда мы создаём канал с помощью `make`, мы можем передать длину:

```
c := make(chan int, 100)
```

Вы можете сделать это изменение, но вы увидите, что обработка всё ещё прерывается. Буферизация каналов не увеличивает их ёмкость, она

просто обеспечивает очередь для ожидания обработки и хороший способ справляться с резкими скачками. В нашем примере мы постоянно отправляем больше данных, чем обработчики могут принять.

Тем не менее, мы можем посмотреть на то, что представляет собой буферизированный канал и, фактически, буферизация, посмотрев на длину канала `len`:

```
for {
    c <- rand.Int()
    fmt.Println(len(c))
    time.Sleep(time.Millisecond * 50)
}
```

Вы можете увидеть, как она растёт и растёт, пока не заполнится, и с этого момента отправка в канал снова будет заблокирована.

Select

Даже при использовании буферизации со временем наступает момент, когда нам нужно начинать отбрасывать входящие сообщения. Мы не можем бесконечно занимать память, в надежде, что обработчик освободится. Для таких целей в Go используется `select`.

Синтаксически `select` похож на `switch`. С его помощью, мы можем описать действия, выполняемые при недоступности канала для отправки. Сначала, давайте уберём буферизацию канала, чтобы увидеть работу `select` более наглядно:

```
c := make(chan int)
```

Затем, мы изменим наш цикл `for`:

```
for {
    select {
```

```

case c <- rand.Int():
    //опциональный код здесь
default:
    //тут можно ничего не писать, чтобы данные молча
отбрасывались
    fmt.Println("выброшено")
}
time.Sleep(time.Millisecond * 50)
}

```

Мы посылаем 20 сообщений в секунду, в то время как обработчики могут принять только 10, поэтому половина будет отброшена.

Это первое из того, что можно сделать используя `select`. Главным его назначением является управление множеством каналов. Получая несколько каналов, `select` блокируется до тех пор, пока один из них не освободится. Если доступных каналов нет, будет выполнен необязательный блок `default`. Если доступно несколько каналов, канал будет выбран случайным образом.

Трудно придумать простой пример для демонстрации такого поведения, поскольку это довольно продвинутая особенность. Возможно следующая секция сможет помочь показать это.

Тайм-Аут

Мы рассмотрели буферные сообщения и простое отбрасывание их. Другой популярный метод – это тайм-аут. Мы будем блокировать выполнение на какое-то время, но не навсегда. Это очень легко реализовать в Go. Правда синтаксис может оттолкнуть, но это такой аккуратный и полезный способ, что я не мог его опустить.

Для блокировки на максимально возможное время мы можем использовать функцию `time.After`. Давайте посмотрим на пример и попробуем разобраться в этой магии. Для этого изменим отправку данных так:

```

for {
    select {

```

```

case c <- rand.Int():
case <-time.After(time.Millisecond * 100):
    fmt.Println("тайм-аут", )
}
time.Sleep(time.Millisecond * 50)
}

```

time.After возвращает канал, так что мы можем использовать select для его выбора. Канал будет завершен после истечения указанного времени. Вот и всё. Больше никакой магии нет. Если вам интересно, реализация after может выглядеть как-то так:

```

func after(d time.Duration) chan bool {
    c := make(chan bool)
    go func() {
        time.Sleep(d)
        c <- true
    }()
    return c
}

```

Возвращаясь к нашему select, есть пара вещей, с которыми можно поиграть. Первая, что если мы вернём обратно блок default? Можете угадать? Попробуйте. Если вы не уверены, что произойдёт, вспомните, что default выполняется немедленно, если ни один из каналов не доступен.

Также time.After это канал типа chan time.Time. В приведённом выше примере мы просто не использовали отправленное в канал значение. Если хотите, вы можете получить его:

```

case t := <-time.After(time.Millisecond * 100):
    fmt.Println("тайм-аут после ", t)

```

Обратите особое внимание на наш select. Заметьте, что мы отправляем в c, но получаем из time.After. select работает независимо от того, откуда мы получаем и что мы отправляем, в любой комбинации каналов:

- Первый доступный канал будет выбран.
- Если доступно несколько каналов, будет выбран случайный.
- Если нет доступных каналов, будет выполнен блок `default`.
- Если блока `default` нет, выполнение `select` блокируется.

Наконец, часто можно увидеть `select` внутри `for`:

```
for {
    select {
        case data := <-c:
            fmt.Printf("обработчик %d получил %d\n", w.id, data)
        case <-time.After(time.Millisecond * 10):
            fmt.Println("Перерыв")
            time.Sleep(time.Second)
    }
}
```

Перед тем как продолжить

Если вы новичок в мире конкурентного программирования, оно может показаться вам довольно тягостным. Оно принципиально требует значительно больше внимания и забот. Двигайтесь дальше и будет легче.

Горутины являются эффективной абстракцией всего того, что необходимо для запуска конкурентного кода. Каналы помогают устранить различные серьезные ошибки, возникающие во время обмена данными или удаления общих данных. Они не просто избавляют от ошибок, но они изменяют то, как вы обращаетесь с конкурентным программированием. Вы начинаете думать о согласованности по отношению к передаче сообщений больше, чем об опасных участках кода.

Я уже говорил, что до сих пор использую различные примитивы для синхронизации из пакетов `sync` и `sync/atomic`. Я призываю вас сначала сделать упор на каналы, но когда вы встречаете простой случай, в котором необходима кратковременная блокировка, подумайте об использовании мьютекса или мьютекса чтения-записи.

Заключение

Недавно я услышал о том, что Go — это *скучный* язык. Скучный потому, что его легко изучить, легко на нём писать и, что самое главное, легко читать. Возможно, я оказал вам медвежью услугу. Мы *потратили* три главы на разговоры о типах и о том, как объявить переменную.

Если у вас уже был опыт работы со статически типизированным языком, многое из того, что вы видели, в лучшем случае освежило ваши знания. То, что Go делает видимыми указатели и то, что срезы являются просто тонкими обёртками вокруг массивов, скорее всего, не удивительно для опытных Java или C# разработчиков.

Если вы в основном использовали динамические языки, вы могли почувствовать небольшую разницу. Её нужно понять. Не на последнем месте стоит также синтаксис объявления и инициализации. Несмотря на то, что я фанат Go, я считаю, что весь успех в достижении простоты. И все сводится к простым правилам (например то, что вы можете объявить переменную только один раз и `:=` делает это) и фундаментальным понятиям (то, что `new(X)` или `&X{}` только выделяет память, но срезы, карты и каналы требуют дополнительных действий при инициализации и поэтому нужен `make`).

Помимо этого, Go предоставляет простой, но эффективный способ организации кода. Интерфейсы, основанная на возвращении обработка ошибок, `defer` для управления ресурсами и простой способ достижения композиции.

Последним, но важным, является встроенная поддержка конкурентности. Горутины довольно эффективны и просты (просты в использовании по крайней мере). Это хорошая абстракция. Каналы немного сложнее. Я всегда считал, что понимание базовых вещей важно перед использованием высокоуровневых обёрток. Я думаю полезно узнать о конкурентном программировании без каналов. Всё же

каналы реализованы таким образом, что они совсем не похожи на простую абстракцию. Они почти как самостоятельные фундаментальные блоки. Я говорю это потому, что они изменяют стиль написания и понимания конкурентного программирования. Учитывая то, каким сложным конкурентное программирование может быть, реализация определённо хороша.