

Введение в программирование на Go

1. [Приступая к работе](#)
2. [Ваша первая программа](#)
3. [Типы](#)
4. [Переменные](#)
5. [Управление потоком](#)
6. [Массивы, срезы, карты](#)
7. [Функции](#)
8. [Указатели](#)
9. [Структуры и интерфейсы](#)
10. [Многопоточность](#)
11. [Пакеты и повторное использование кода](#)
12. [Тестирование](#)
13. [Стандартная библиотека](#)
14. [Дальнейшие шаги](#)

Приступая к работе

Программирование — это искусство, ремесло и наука о написании программ, определяющих то, как компьютер будет работать. Эта книга научит вас писать компьютерные программы с использованием языка программирования, разработанного в компании Google, под названием Go.

Go — язык общего назначения с широкими возможностями и понятным синтаксисом. Благодаря мультиплатформенности, надежной, хорошо документированной стандартной библиотеке и ориентированности на удобные подходы к самой разработке, Go является идеальным языком для первых шагов в программировании.

Процесс разработки приложений на Go (и на большинстве других языков программирования) довольно прост:

- сбор требований,
- поиск решения,
- написание кода, реализующего решения,
- компиляция кода в исполняемый файл,
- запуск и тестирование программы.

Процесс этот итеративный (то есть повторяющийся много раз), и шаги, как правило, совпадают. Но прежде чем мы напишем нашу первую программу на Go, нужно понять несколько основных принципов.

Файлы и директории

Файл представляет собой набор данных, хранящийся в блоке с определенным именем. Современные операционные системы (такие как Windows или Mac OS X) содержат миллионы файлов, содержащих большой объем различной информации — начиная от текстовых документов и заканчивая программами и мультимедиа-файлами.

Файлы определенным образом хранятся в компьютере: все они имеют имя, определенный размер (измеряемый в байтах) и соответствующий тип. Обычно тип файла определяется по его расширению — части имени, которая стоит после последней `.`. Например, файл названный `hello.txt` имеет расширение `txt`, значит содержит текстовую информацию.

Папки (так же называемые директориями) используются для группирования нескольких файлов.

Терминал

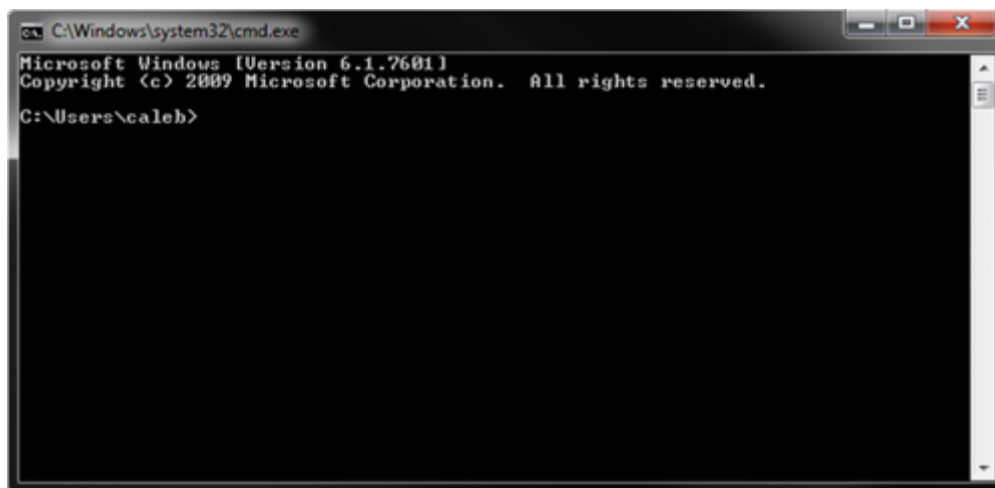
Большая часть взаимодействия с компьютером сейчас осуществляется с помощью графического пользовательского интерфейса (GUI). Мы используем клавиатуру, мышь, сенсорные экраны для взаимодействия с визуальными кнопками и другими отображаемыми элементами.

Но так было не всегда. Перед GUI в ходу был терминал — простой текстовый интерфейс к компьютеру, где вместо работы с кнопками на экране мы вводили команды и получали ответы.

И хотя может показаться, что большая часть компьютерного мира оставила терминал далеко позади как пережиток прошлого, правда в том, что терминал всё еще остаётся фундаментальным пользовательским интерфейсом, используемым большинством языков программирования на большинстве компьютеров. Go не исключение, поэтому прежде чем писать программу на Go, понадобится элементарное понимание того, как работает терминал.

Windows

Чтобы вызвать терминал (командную строку) в Windows, нужно нажать комбинацию клавиш Win+R (удерживая клавишу с логотипом Windows нажмите R), ввести в появившееся окно **cmd.exe** и нажать Enter. Вы должны увидеть черное окно, похожее на то, что ниже:



По умолчанию командная строка запускается из вашей домашней директории (в моём случае это **C:\Users\caleb**). Вы отдаёте команды компьютеру, набирая их в этом окне и нажимая Enter. Попробуйте ввести команду **dir**, которая выводит содержимое текущего каталога на экран. Вы должны увидеть что-то вроде этого:

```
C:\Users\caleb>dir
Volume in drive C has no label.
Volume Serial Number is B2F5-F125
```

Вы можете изменить текущий каталог с помощью команды **cd**. Например, там наверняка есть директория под названием **Desktop**. Вы можете посмотреть её содержимое, набрав **cd Desktop**, а затем **dir**. Чтобы вернуться в домашнюю директорию, используйте специальное имя **..** (две точки): **cd ..**. Одна точка обозначает текущий каталог (известен как рабочая директория), так что **cd .** ничего не сделает. Конечно, существует намного больше команд, которые можно использовать, но этих будет вполне достаточно для начала.

OSX

В OSX терминал можно найти, перейдя в Finder → Applications → Utilities → Terminal. Вы увидите такое окно:



По умолчанию, командная строка запускается из вашей домашней директории (в моём случае это `/Users/caleb`). Вы отдаёте команды компьютеру, набирая их в этом окне и нажимая Enter. Попробуйте ввести команду `ls`, которая выводит содержимое текущего каталога на экран. Вы должны увидеть что-то вроде этого:

```
caleb-mini:~ caleb$ ls
Desktop      Downloads    Movies       Pictures
Documents    Library      Music        Public
```

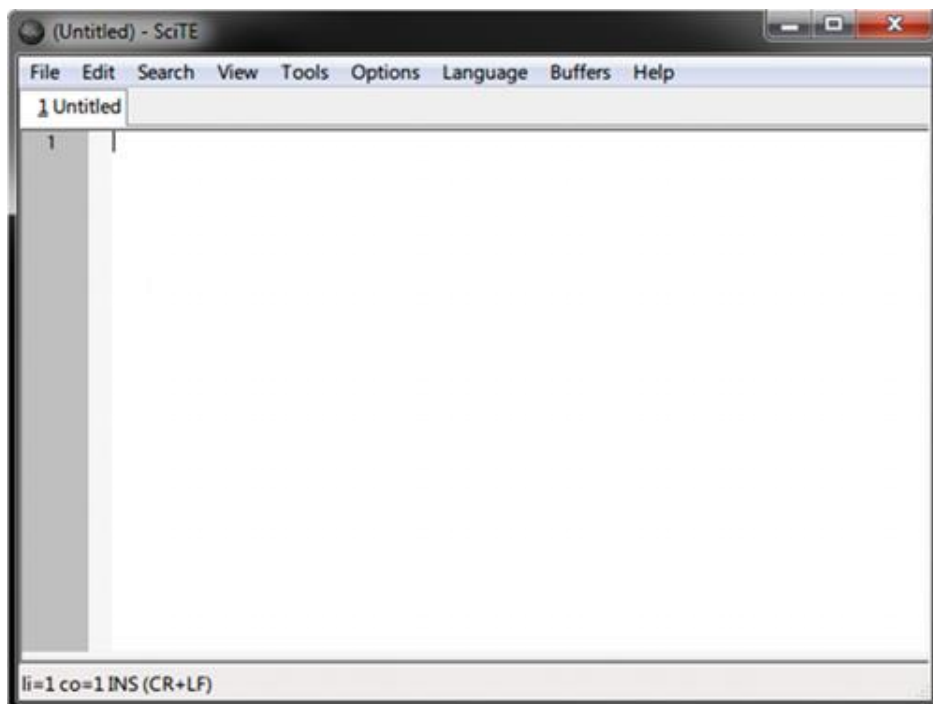
Вы можете изменить текущий каталог с помощью команды `cd`. Например, там наверняка есть директория под названием `Desktop`. Вы можете посмотреть её содержимое набрав `cd Desktop`, а затем `ls`. Чтобы вернуться в домашнюю директорию, используйте специальное имя `..` (две точки): `cd ..`. Одна точка обозначает текущий каталог (известен как рабочая директория), так что `cd .` ничего не сделает. Конечно, существует намного больше команд, которые можно использовать, но этих будет вполне достаточно для начала.

Текстовый редактор

Основным инструментом программиста при разработке программного обеспечения является текстовый редактор. Текстовые редакторы в целом похожи на программы обработки текста (такие как Microsoft Word, OpenOffice, ...), но в отличие от них, там отсутствует какое-либо форматирование (полужирный, курсив и и т.п.), что делает их ориентированными только на работу с простым текстом. Как в OSX, так и в Windows, по умолчанию уже присутствует встроенный текстовый редактор. Но они очень ограничены в возможностях, поэтому я бы порекомендовал что-нибудь получше.

Windows

Например для Windows текстовый редактор SciTe. После запуска вы должны увидеть такое окно:



Текстовый редактор содержит большую белую область для ввода текста. Слева от этой области можно увидеть номера строк. В нижней части окна находится строка состояния, где отображается информация о файле и вашем текущем местоположении в нём (сейчас он говорит, что мы находимся у первого символа первой строки, используется режим вставки текста, а окончания строк обозначаются в Windows-стиле).

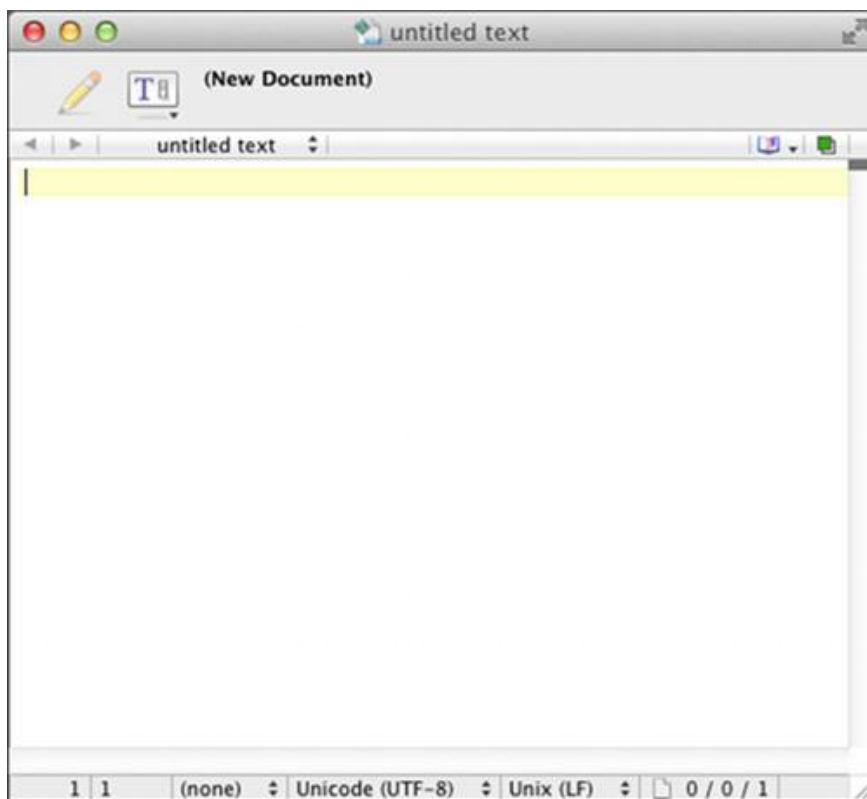
Вы можете открыть файл, выбрав его в диалоге, находящимся в меню File → Open. Файлы могут быть сохранены с помощью меню File → Save или File → Save As.

Так как подобные действия вы будете выполнять достаточно часто, неплохо было бы узнать сочетания клавиш для быстрого доступа к пунктам меню. Вот самые распространённые из них:











- **Ctrl** + **S** — сохранить текущий файл
- **Ctrl** + **X** — вырезать выделенный текст (удалить его, предварительно сохранив в буфере обмена, для возможной вставки позже)
- **Ctrl** + **C** — скопировать выделенный фрагмент текста в буфер обмена
- **Ctrl** + **V** — вставить текст на место текущего положения курсора из буфера обмена
- Используйте клавиши со стрелками для навигации по файлу, **Home** для перехода в начало строки, а **End** для перехода в конец
- Удерживайте **Shift** при использовании клавиш навигации, чтобы выделить фрагмент текста без использования мыши
- **Ctrl** + **F** — открыть диалоговое окно поиска по содержимому файла

OSX

Для OSX установщик поставит редактор Text Wrangler:



Как и Scite на Windows, окно Text Wrangler содержит большую белую область, где вводится текст. Файлы могут быть открыты при помощи File → Open, а сохранены с помощью File → Save или File → Save As. Вот некоторые полезные сочетания клавиш:

-  +  — сохранить текущий файл
-  +  — вырезать выделенный текст (удалить его, предварительно сохранив в буфере обмена, для возможной вставки позже)
-  +  — скопировать выделенный фрагмент текста в буфер обмена
-  +  — вставить текст на место текущего положения курсора из буфера обмена
- Используйте клавиши со стрелками для навигации по файлу
-  +  — открыть диалоговое окно поиска по содержимому файла

Инструментарий Go

Go — компилируемый язык программирования. Это означает, что исходный код (написанный вами код) переводится в язык, понятный компьютеру. Поэтому, прежде чем написать первую программу на Go, нужно разобраться с его компилятором.

Инсталлятор установит Go автоматически. Мы будем использовать первую версию языка. (Больше информации можно найти на <http://golang.org>)
Давайте убедимся, что всё работает.

Откроем терминал и введём там:

```
go version
```

В ответ вы должны увидеть что-то вроде:

```
go version go1.0.2
```

Ваш номер версии может быть немного другим. Если вы получили ошибку, попробуйте перезагрузить компьютер.

Инструментарий Go состоит из нескольких команд и подкоманд. Список всех доступных команд можно увидеть, набрав:

```
go help
```

О том, как их использовать, мы узнаем в следующих главах.

Ваша первая программа

Традиционно, первая программа, с которой начинается изучение любого языка программирования, называется «Hello World» — эта программа просто выводит в консоль строку **Hello World**. Давайте напишем её с помощью Go.

Сначала создадим новую директорию, в которой будем хранить нашу программу. Установщик, о котором говорилось в первой главе, создал в вашей домашней директории каталог **Go**. Теперь создайте директорию под названием **~/Go/src/golang-book/chapter2** (где **~** означает вашу домашнюю директорию). Вы можете сделать это из терминала с помощью следующих команд:

```
mkdir Go/src/golang-book
mkdir Go/src/golang-book/chapter2
```

Используя текстовый редактор, введите следующее:

```
package main

import "fmt"

// this is a comment

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

Убедитесь, что содержимое файла идентично показанному здесь примеру, и сохраните его под именем **main.go** в созданной ранее директории. Затем откройте новое окно терминала и введите:

```
cd Go/src/golang-book/chapter2
go run main.go
```

В окне терминала вы должны увидеть сообщение **Hello World**. Команда **go run** берет указанные файлы (разделенные пробелом), компилирует их в исполняемые файлы, сохраняет во временной директории и запускает. Если вы не увидели **Hello World**, то, вероятно, где-то была допущена ошибка, компилятор подскажет вам, где конкретно. Как и большинство компиляторов, компилятор Go крайне педантичен и не прощает ошибок.

Как читать программу на Go

Теперь давайте рассмотрим программу более детально. Программы на Go читаются сверху вниз, слева направо (как книга). Первая строка гласит:

```
package main
```


Это называется «определение пакета». Любая Go программа начинается с определения имени пакета. Пакеты — это подход Go к организации повторного использования кода. Есть два типа программ на Go: исполняемые файлы и разделяемые библиотеки. Исполняемые являются видом программ, которые можно запустить прямо из терминала (в Windows их имя заканчивается на **.exe**). Библиотеки являются коллекциями кода, который можно использовать из других программ. Детальнее мы будем рассматривать библиотеки чуть позже, а пока просто не забудьте включать эту строку в программы, которые вы пишете.

Дальше следует пустая строка. Компьютер представляет новые строки специальным символом (или несколькими символами). Символы новой строки, пробелы и символы табуляции называются отступами. Go не обращает на них внимания, но мы используем их, чтобы облегчить себе чтение программы. (Вы можете удалить эту строку и убедиться, что программа ведет себя в точности как раньше.)

Дальше следует это:

```
import "fmt"
```

Ключевое слово **import** позволяет подключить сторонние пакеты для использования их функциональности в нашей программе.

Пакет **fmt** (сокращение от format) реализует форматирование для входных и выходных данных. Учитывая то, что мы только что узнали о пакетах, как вы думаете, что будет содержаться в верхней части файлов пакета **fmt**?

Обратите внимание, что **fmt** взят в двойные кавычки. Использование двойных кавычек называется «строковым литералом», который в свою очередь является видом «выражения». Строки в Go представляют собой набор символов (букв, чисел, ...) определенной длины. Строки мы рассмотрим детально в следующей главе, а сейчас главное иметь в виду, что за открывающим символом **"** в конечном итоге должен последовать и закрывающий. Всё, что находится между ними, будет являться строкой (символ **"** сам по себе не является частью строки). Строка, начинающаяся с **//** известна как комментарий. Комментарии игнорируются компилятором Go и служат исключительно пояснениями вам (или тем, кто будет потом читать ваш код). Go поддерживает два вида комментариев: **//** превращает в комментарий весь текст до конца строки и **/* */**, где комментарием является всё, что содержится между символами ***** (включая переносы строк).

Далее можно увидеть объявление функции:

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

Функции являются кирпичиками программы на Go. Они имеют входы, выходы и ряд действий, называемых операторами, расположенных в определенном порядке. Любая функция начинается с ключевого слова **func**, за которым следует имя функции (в нашем случае **main**), список из нуля и более параметров, возвращаемый тип (если есть) и само «тело», заключенное в фигурные скобки. Наша функция не имеет входных параметров, ничего не

возвращает и содержит всего один оператор. Имя `main` является особенным, эта функция будет вызываться сама при запуске программы. Заключительной частью нашей программы является эта строка:

```
fmt.Println("Hello World")
```

Этот оператор содержит три части. Во первых доступ к функции пакета `fmt` под названием `Println` (Print line), затем создается новая строка, содержащая `Hello World` и вызывается функция с этой строкой, в качестве первого и единственного аргумента.

На данный момент вы уже можете быть немного перегружены от количества новой терминологии. Иногда полезно сознательно читать вашу программу вслух. Программу, которую мы только что написали можно прочитать следующим образом:

Создается новая исполняемая программа, которая использует библиотеку `fmt` и содержит функцию `main`. Эта функция не содержит аргументов, ничего не возвращает и делает следующее: использует функцию `Println` из библиотеки `fmt` и вызывает её, используя один аргумент — строку `Hello World`.

Функция `Println` выполняет основную работу в этой программе. Вы можете узнать о ней больше, набрав в терминале команду:

```
godoc fmt Println
```

Среди прочей информации вы должны увидеть это:

```
Println formats using the default formats for its operands and writes to standard output. Spaces are always added between operands and a newline is appended. It returns the number of bytes written and any write error encountered.
```

Go — очень хорошо документированный язык, но эту документацию бывает трудно понять, если вы доселе не были знакомы с другими языками программирования. Тем не менее, команда `godoc` является хорошим местом для начала поиска ответов на возникающие вопросы.

Сейчас документация говорит нам, что вызов `Println` пошлет передаваемые ей данные на стандартный вывод — терминал, вы сейчас работаете в нём. Эта функция является причиной, по которой `Hello World` отображается на экране. В следующей главе вы поймете, каким образом Go хранит и представляет вещи вроде `Hello World` с помощью типов.

Задачи

- Что такое отступ?
- Что такое комментарий? Какие два способа записи комментариев существуют?
- Наша программа начиналась с `package main`. С чего начинаются файлы в пакете `fmt`?
- Мы использовали функцию `Println` из пакета `fmt`. Если бы мы хотели использовать функцию `Exit` из пакета `os`, что бы потребовалось сделать?
- Измените написанную программу так, чтобы вместо `Hello World` она выводила `Hello, my name is` вместе с вашим именем.

Типы

В предыдущей главе мы использовали строковый тип данных, чтобы хранить **Hello World**. Типы данных определяют множество принимаемых значений, описывают, какие операции могут быть применены к ним и определяют, как данные будут храниться. Поскольку типы данных могут быть сложны для понимания, мы попробуем рассмотреть их подробнее, прежде чем разбираться, как они реализованы в Go.

Предположим, у вас есть собака по имени Шарик. Тут «Шарик» — это «Собака», этот тип описывает какой-то набор свойств, присущий всем собакам. Наши рассуждения должны быть примерно следующие: у собак 4 лапы, Шарик — собака, значит, у Шарика 4 лапы. Типы данных в языках программирования работают похожим образом: у всех строк есть длина; **x** — строка, а значит у **x** есть длина.

В математике мы часто говорим о множествах. Например, **R** (множество всех вещественных чисел) или **N** (множество всех натуральных чисел). Каждый элемент этих множеств имеет такие же свойства, как и все прочие элементы этого множества. Например, все натуральные числа ассоциативны - «для всех натуральных чисел a , b и c выполняется: $a + (b + c) = (a + b) + c$ и $a \times (b \times c) = (a \times b) \times c$ »; в этом смысле множества схожи с типами данных в языках программирования тем, что все значения одного типа имеют общие свойства.

Go — это язык программирования со статической типизацией. Это означает, что переменные всегда имеют определенный тип и этот тип нельзя изменить. Статическая типизация, на первый взгляд, может показаться неудобной. Вы потратите кучу времени только на попытки исправить ошибки, не позволяющие программе скомпилироваться. Однако типы дают вам возможность понять, что именно делает программа и помогает избежать распространенных ошибок.

В Go есть несколько встроенных типов данных, с которыми мы сейчас ознакомимся.

Числа

В Go есть несколько различных типов для представления чисел. Вообще, мы разделим числа на два различных класса: целые числа и числа с плавающей точкой.

Целые числа

Целые числа, точно так же, как их математические коллеги, — это числа без вещественной части. В отличие от десятичного представления чисел, которое используем мы, компьютеры используют двоичное представление.

Наша система строится на 10 различных цифрах. Когда мы исчерпываем доступные нам цифры, мы представляем большое число, используя новую цифру 2 (а затем 3, 4, 5, ...) числа следуют одно за другим. Например, число, следующее за 9, это 10, число, следующее за 99, это 100 и так далее. Компьютеры делают то же самое, но они имеют только 2 цифры вместо 10. Поэтому, подсчет выглядит так: 0, 1, 10, 11, 100, 101, 110, 111 и так далее.

Другое отличие между той системой счисления, что используем мы, и той, что использует компьютер - все типы чисел имеют строго определенный размер. У них есть ограниченное количество цифр. Поэтому четырехразрядное число может выглядеть так: 0000, 0001, 0010, 0011, 0100. В конце концов мы можем выйти за лимит, и большинство компьютеров просто вернутся к самому началу (что может стать причиной очень странного поведения программы).

В Go существуют следующие типы целых

чисел: `uint8`, `uint16`, `uint32`, `uint64`, `int8`, `int16`, `int32` и `int64`. 8, 16, 32 и 64 говорит нам, сколько бит использует каждый тип. `uint` означает «unsigned integer» (беззнаковое целое), в то время как `int` означает «signed integer» (знаковое целое). Беззнаковое целое может принимать только положительные значения (или ноль). В дополнение к этому существуют два типа — псевдонима: `byte` (то же самое, что `uint8`) и `rune` (то же самое, что `int32`). Байты — очень распространенная единица измерения в компьютерах (1 байт = 8 бит, 1024 байт = 1 килобайт, 1024 килобайт = 1 мегабайт, ...) и именно поэтому тип `byte` в Go часто используется для определения других типов. Также существует 3 машинно-зависимых целочисленных типа: `uint`, `int` и `uintptr`. Они машинно-зависимы, потому что их размер зависит от архитектуры используемого компьютера.

В общем, если вы работаете с целыми числами — просто используйте тип `int`.

Числа с плавающей точкой

Числа с плавающей точкой — это числа, которые содержат вещественную часть (вещественные числа). (1.234, 123.4, 0.00001234, 12340000) Их представление в компьютере довольно сложно и не особо необходимо для их использования. Так что мы просто должны помнить:

- Числа с плавающей точкой неточны. Бывают случаи, когда число вообще нельзя представить. Например, результатом вычисления `1.01 - 0.99` будет `0.0200000000000000018` - число очень близкое к ожидаемому, но не то же самое.
- Как и целые числа, числа с плавающей точкой имеют определенный размер (32 бита или 64 бита). Использование большего размера увеличивает точность (сколько цифр мы можем использовать для вычисления)
- В дополнение к числам существуют несколько других значений, такие как: «not a number» (не число) (`NaN`, для вещей наподобие `0/0`) а также положительная и отрицательная бесконечность (`+∞` и `-∞`).

В Go есть два вещественных типа: `float32` и `float64` (соответственно, часто называемые вещественными числами с одинарной и двойной точностью). А также два дополнительных типа для представления комплексных чисел (чисел с мнимой частью): `complex64` и `complex128`. Как правило, мы должны придерживаться типа `float64`, когда работаем с числами с плавающей точкой.

Пример

Давайте напишем программу-пример, использующую числа. Во-первых создайте папку «chapter3» с файлом `main.go` внутри со следующим содержимым:

```
package main

import "fmt"

func main() {
    fmt.Println("1 + 1 = ", 1 + 1)
}
```

Если вы запустите программу, то должны увидеть это:

```
$ go run main.go
1 + 1 = 2
```

Заметим, что эта программа очень схожа с программой, которую мы написали в главе 2. Она содержит ту же строку с указанием пакета, ту же строку с импортом, то же определение функции и использует ту же функцию `Println`. В этот раз, вместо печати строки `Hello World`, мы печатаем строку `1 + 1 =` с последующим результатом выражения `1 + 1`. Это выражение состоит из трех частей: числового литерала `1` (который является типом `int`), оператора `+` (который представляет сложение) и другого числового литерала `1`. Давайте попробуем сделать то же самое, используя числа с плавающей точкой:

```
fmt.Println("1 + 1 =", 1.0 + 1.0)
```

Обратите внимание, что мы используем `.0`, чтобы сказать Go, что это число с плавающей точкой, а не целое. При выполнении этой программы результат будет тот же, что и прежде.

В дополнение к сложению, в Go имеется несколько других операций:

Литерал	Пояснение
<code>+</code>	сложение
<code>-</code>	вычитание
<code>*</code>	умножение
<code>/</code>	деление
<code>%</code>	остаток от деления

Строки

Как мы видели в главе 2, строка — это последовательность символов определенной длины, используемая для представления текста. Строки в Go состоят из независимых байтов, обычно по одному на каждый символ (Символы из других языков, таких как Китайский, представляются несколькими байтами)

Строковые литералы могут быть созданы с помощью двойных кавычек `"Hello World"` или с помощью апострофов `'Hello World'`. Различие между ними в том, что строки в двойных кавычках не могут содержать новые строки и они позволяют использовать особые управляющие последовательности символов. Например, `\n` будет заменен символом новой строки и `\t` будет заменен символом табуляции.

Распространенные операции на строках включают в себя нахождение длины строки: `len("Hello World")`, доступ к отдельному символу в строке `"Hello World"[1]`, и конкатенация двух строк вместе: `"Hello " + "World"`. Давайте модифицируем созданную ранее программу, чтобы проверить всё это:

```
package main

import "fmt"

func main() {
    fmt.Println(len("Hello World"))
    fmt.Println("Hello World"[1])
    fmt.Println("Hello " + "World")
}
```

На заметку:

- Пробел тоже считается символом, поэтому длина строки 11 символов, а не 10 и третья строка содержит `"Hello "` вместо `"Hello"`.
- Строки “индексируются” начиная с 0, не с 1. `[1]` даст вам второй элемент, а не первый. Также заметьте, что вы видите `101` вместо `e`, когда выполняете программу. Это происходит из-за того, что символ представляется байтом (помните, байт — это целое число).
Можно думать об индексации так: `"Hello World" 1`. Читайте это так: «Строка Hello World позиция 1», «На 1 позиции строки Hello World» или «Второй символ строки Hello World».
- Конкатенация использует тот же символ, что и сложение. Компилятор Go выясняет, что должно происходить, полагаясь на типы аргументов. Если по обе стороны от `+` находятся строки, компилятор предположит, что вы имели в виду конкатенацию, а не сложение (Ведь сложение для строк бессмысленно).

Логические типы

Булевский тип (названный так в честь Джорджа Буля) — это специальный однобитный целочисленный тип, используемый для представления истинны и лжи. С этим типом используются три логических оператора:

Литерал	Пояснение
&&	и
	или
!	НЕ

Вот пример программы, показывающей их использование:

```
func main() {  
    fmt.Println(true && true)  
    fmt.Println(true && false)  
    fmt.Println(true || true)  
    fmt.Println(true || false)  
    fmt.Println(!true)  
}
```

Запуск этой программы должен вывести:

```
$ go run main.go  
true  
false  
true  
true  
false
```


Используем таблицы истинности, чтобы определить, как эти операторы работают:

Выражение	Значение
true && true	true
true && false	false
false && true	false
false && false	false
Выражение	Значение
true true	true
true false	true
false true	true
false false	false
Выражение	Значение
!true	false
!false	true

Всё это — простейшие типы, включенные в Go и являющиеся основой, с помощью которой строятся все остальные типы.

Задачи

- Как хранятся числа в компьютере?
- Мы знаем, что (в десятичной системе) самое большое число из 1 цифры это 9 и самое большое число из 2 цифр это 99. В бинарной системе самое большое число из двух цифр это 11 (3), самое большое число из трех цифр это 111 (7) и самое большое число из 4 цифр это 1111 (15). Вопрос: какое самое большое число из 8 цифр? (Подсказка: $10^1 - 1 = 9$ и $10^2 - 1 = 99$)
- В зависимости от задачи, вы можете использовать Go как калькулятор. Напишите программу, которая вычисляет `32132 * 42452` и печатает это в терминал. (Используйте оператор `*` для умножения)
- Что такое строка? Как найти её длину?
- Какое значение примет выражение `(true && false) || (false && true) || !(false && false)`?

Переменные

Ранее в этой книге мы имели дело с литеральными значениями (числами, строками, и т.д.), но программы с одними только литералами фактически бесполезны. Для того, чтобы сделать по-настоящему полезные программы, нам нужно узнать о двух важных вещах: переменных и инструкциях, управляющих ходом исполнения. В этой главе будут рассмотрены переменные.

Переменная - это именованное место хранения какого-то типа данных. Давайте изменим программу, которую мы написали в главе 2 так, чтобы там использовались переменные.

```
package main

import "fmt"

func main() {
    var x string = "Hello World"
    fmt.Println(x)
}
```

Обратите внимание, что мы по-прежнему используем строковый литерал из оригинальной программы, но вместо того, чтобы напрямую передать его в функцию **Println**, мы присваиваем его переменной. Переменные в Go создаются с помощью ключевого слова **var**, за которым следует: имя переменной (**x**), тип (**string**) и присваиваемое значение (**Hello World**). Последний шаг не обязателен, поэтому программа может быть переписана так:

```
package main

import "fmt"

func main() {
    var x string
    x = "Hello World"
    fmt.Println(x)
}
```

Переменные в Go похожи на переменные в алгебре, но есть несколько различий:

Во первых, когда мы видим символ **=**, мы по привычке читаем его как «x равен строке Hello World». Нет ничего неверного в том, чтобы читать программу таким образом, но лучше читать это как: «x принимает значение строки Hello World» или «x присваивается строка Hello World». Это различие важно потому, что (как понятно по их названию) переменные могут менять свои значения во время выполнения программы.

Попробуйте сделать следующее:

```
package main

import "fmt"

func main() {
    var x string
    x = "first"
    fmt.Println(x)
    x = "second"
    fmt.Println(x)
}
```

На самом деле, вы можете сделать даже так:

```
var x string
x = "first "
fmt.Println(x)
x = x + "second"
fmt.Println(x)
```

Эта программа будет бессмысленной, если вы будете читать её как теорему из алгебры. Но она обретет смысл если вы будете внимательно читать программу как список команд. Когда вы видим **x = x + "second"**, то должны читать это так: «присвоить конкатенацию значения переменной x и литерала строки переменной x». Операции справа от **=** выполняются первыми и результат присваивается левой части.

Запись **x = x + y** настолько часто встречается в программировании, что в Go есть специальный оператор присваивания **+=**. Мы можем записать **x = x + "second"** как **x += "second"** и результат будет тем же. (Прочие операторы могут быть использованы подобным образом)

Другое отличие между Go и алгеброй в том, что используется другой символ для равенства: **==**. (Два знака равно, один за другим) **==** - это оператор, как и **+** и он возвращает логический тип. Например:

```
var x string = "hello"
var y string = "world"
fmt.Println(x == y)
```

Эта программа напечатает **false**, потому что **hello** отличается от **world**. С другой стороны:

```
var x string = "hello"
var y string = "hello"
fmt.Println(x == y)
```

Напечатает **true**, потому что обе строки одинаковы.

Если мы хотим присвоить значение переменной при её создании, то можем использовать сокращенную запись:

```
x := "Hello World"
```

Обратите внимание на то что **:** стоит перед **=** и на отсутствие типа. Тип в данном случае указывать необязательно, так как компилятор Go способен определить тип по литералу, которым мы инициализируем переменную. (Тут мы присваиваем строку, поэтому **x** будет иметь тип **string**) Компилятор может определить тип и при использовании **var**:

```
var x = "Hello World"
```

И так со всеми типами:

```
x := 5  
fmt.Println(x)
```

В общем, желательно всегда использовать краткий вариант написания.

Как назвать переменную

Правильное именование переменных — важная часть разработки ПО. Имена должны начинаться с буквы и могут содержать буквы, цифры и знак **_** (знак подчеркивания). Компилятору Go, в принципе, всё равно, как вы назовете переменную, но не забудьте, что вам (и может быть кому-то еще) потом это придется читать. Предположим, у нас есть:

```
x := "Max"  
fmt.Println("My dog's name is", x)
```

В этом случае, **x** - это не самое лучшее имя переменной. Лучше было бы так:

```
name := "Max"  
fmt.Println("My dog's name is", name)
```

или даже так:

```
dogsName := "Max"  
fmt.Println("My dog's name is", dogsName)
```

В последнем случае мы использовали специальный способ написания имени переменной, состоящей из нескольких слов, известный как **lower CamelCase** (или **camelBack**). Первая буква первого слова записывается в нижнем регистре, первая буква последующих слов записывается в верхнем регистре - всё остальное в нижнем.

Область видимости

Вернемся к программе, которую мы рассматривали в начале главы:

```
package main

import "fmt"

func main() {
    var x string = "Hello World"
    fmt.Println(x)
}
```

Эту программу можно записать следующим образом:

```
package main

import "fmt"

var x string = "Hello World"

func main() {
    fmt.Println(x)
}
```

Мы вынесли переменные за пределы функции main. Это означает, что теперь другие функции имеют доступ к этой переменной:

```
var x string = "Hello World"

func main() {
    fmt.Println(x)
}

func f() {
    fmt.Println(x)
}
```

Функция **f** имеет доступ к переменной **x**. Теперь предположим, что вместо этого мы написали:

```
func main() {  
    var x string = "Hello World"  
    fmt.Println(x)  
}  
  
func f() {  
    fmt.Println(x)  
}
```

Если вы попытаете выполнить эту программу, то получите ошибку:

```
.\main.go:11: undefined: x
```

Компилятор говорит вам, что переменная **x** внутри функции **f** не существует. Она существует только внутри функции **main**. Места, где может использоваться переменная **x** называется областью видимости переменной. Согласно спецификации: «В Go область видимости ограничена блоками». В основном, это значит, что переменные существуют только внутри текущих фигурных скобок **{ }** (в блоке), включая все вложенные скобки (блоки). Область видимости поначалу может запутать вас, но когда вы увидите больше примеров, то всё станет ясно.

Константы

Go также поддерживает константы. Константы — это переменные, чьи значения не могут быть изменены после инициализации. Они создаются таким же образом, как и переменные, только вместо **var** используется ключевое слово **const**:

```
package main  
  
import "fmt"  
  
func main() {  
    const x string = "Hello World"  
    fmt.Println(x)  
}
```

А вот этот код:

```
const x string = "Hello World"  
x = "Some other string"
```

Вызовет ошибку компиляции:

```
.\main.go:7: cannot assign to x
```

Константы — хороший способ использовать определенные значения в программе, без необходимости писать их каждый раз. Например, константа **Pi** из пакета **math**.

Определение нескольких переменных

В Go существует еще одно сокращение, на случай, если необходимо определить несколько переменных:

```
var (  
    a = 5  
    b = 10  
    c = 15  
)
```

Используя ключевые слово **var** (или **const**), за которым идут круглые скобки с одной переменной в каждой строке.

Пример программы

```
package main  
  
import "fmt"  
  
func main() {  
    fmt.Print("Enter a number: ")  
    var input float64  
    fmt.Scanf("%f", &input)  
  
    output := input * 2  
  
    fmt.Println(output)  
}
```

Тут мы используем другую функцию из пакета **fmt**, чтобы считать пользовательский ввод (**Scanf**). **&input** будет объяснен в следующих главах, и все что нам нужно знать сейчас — что **Scanf** заполняет переменную **input** числом, введенным нами.

Задачи

- Существуют два способа для создания новой переменной. Какие?
- Какое будет значение у **x** после выполнения **x := 5; x += 1**?
- Что такое область видимости и как определяется область видимости переменной в Go?
- В чем отличие **var** от **const**?
- Используя пример программы выше, напишите программу, переводящую температуру из градусов Фаренгейта в градусы Цельсия. (**C = (F - 32) * 5/9**)
- Напишите другую программу, для перевода футов в метры. (1 фут = 0.3048 метр)

Управление потоком

Теперь мы знаем про переменные, самое время написать что-нибудь полезное. Сначала напишем программу, которая по очереди с новой строки выводит числа от 1 до 10. Наши знания достаточно для того, чтобы написать эту программу так:

```
package main

import "fmt"

func main() {
    fmt.Println(1)
    fmt.Println(2)
    fmt.Println(3)
    fmt.Println(4)
    fmt.Println(5)
    fmt.Println(6)
    fmt.Println(7)
    fmt.Println(8)
    fmt.Println(9)
    fmt.Println(10)
}
```

или так:

```
package main
import "fmt"

func main() {
    fmt.Println(`1
2
3
4
5
6
7
8
9
10`)
}
```

Но писать это будет довольно утомительно. Так что нам нужен лучший способ несколько раз повторить определенный набор действий.

For

Оператор **for** даёт возможность повторять список инструкций (блок) определённое количество раз. Давайте перепишем предыдущую программу используя оператор **for**:

```
package main

import "fmt"

func main() {
    i := 1
    for i <= 10 {
        fmt.Println(i)
        i = i + 1
    }
}
```

Сначала создается переменная **i**, хранящая число, которое нужно вывести на экран. Затем, с помощью ключевого слова **for**, создается цикл, указывается условное выражение, которое может быть **true** или **false**, и, наконец, сам блок для выполнения. Цикл **for** работает следующим образом:

- Оценивается (выполняется) условное выражение **i <= 10** («i меньше или равно десяти»). Если оно истинно, выполняются инструкции внутри блока. В противном случае управление переходит следующей после блока строке кода (в нашем случае после цикла ничего нет, поэтому совершается выход из программы).
- После запуска инструкций внутри блока мы возвращаемся в начало цикла и повторяем первый шаг.

Строка **i = i + 1** очень важна, без неё выражение **i <= 10** всегда будет **true**, и выполнение программы никогда не завершится (это называется бесконечным циклом).

Следующий пример показывает выполнение программ точно так же, как это делает компьютер:

- создать переменную **i** со значением 1;
- **i** меньше или равно 10? да;
- вывести **i**;
- присвоить **i** значение **i + 1** (теперь равно 2);
- **i** меньше или равно 10? да;
- вывести **i**;
- присвоить **i** значение **i + 1** (теперь равно 3);
- ...
- присвоить **i** значение **i + 1** (теперь равно 11);
- **i** меньше или равно 10? нет;
- больше нечего делать, выходим.

В других языках программирования существуют разные виды циклов (while, do, until, foreach, ...). У Go вид цикла один, но он может использоваться в разных случаях. Предыдущую программу можно также записать следующим образом:

```
func main() {
    for i := 1; i <= 10; i++ {
        fmt.Println(i)
    }
}
```

Теперь условное значение включает в себя также и две другие инструкции, разделенные точкой с запятой. Сначала инициализируется переменная, затем выполняется условное выражение, и в завершение переменная «инкрементируется». (Добавление 1 к значению переменной является настолько распространённым действием, что для этого существует специальный оператор: **++**. Аналогично вычитание 1 может быть выполнено с помощью **--**.) В следующих главах мы увидим и другие способы использования циклов.

If

Давайте изменим программу так, чтобы вместо простого вывода чисел 1–10 она также указывала, является ли число чётным или нечётным. Вроде этого:

```
1 odd
2 even
3 odd
4 even
5 odd
6 even
7 odd
8 even
9 odd
10 even
```

Для начала нам нужен способ узнать, является ли число чётным или нечётным. Самый простой способ — это разделить число на 2. Если остатка от деления не будет, значит число чётное, иначе — нечётное. Так как же найти остаток от деления на Go? Для этого существует оператор **%**. Например:

- **1 % 2** равно **1**;
- **2 % 2** равно **0**;
- **3 % 2** равно **1**, и так далее.

Далее нам нужен способ, чтобы выполнять действия в зависимости от условия. Для этого мы используем оператор **if**.

```
if i % 2 == 0 {
    // even
} else {
    // odd
}
```

Оператор **if** аналогичен оператору **for** в том, что он выполняет блок в зависимости от условия. Оператор также может иметь необязательную **else** часть. Если условие истинно, выполняется блок, расположенный после условия, иначе же этот блок пропускается, или выполняется блок **else**, если он присутствует. Еще условия могут содержать **else if** часть:

```
if i % 2 == 0 {
    // divisible by 2
} else if i % 3 == 0 {
    // divisible by 3
} else if i % 4 == 0 {
    // divisible by 4
}
```

Условия выполняются сверху вниз, и первое условие, которое окажется истинным, приведет в исполнение связанный с ним блок.

Собрав всё вместе, мы получим:

```
func main() {
    for i := 1; i <= 10; i++ {
        if i % 2 == 0 {
            fmt.Println(i, "even")
        } else {
            fmt.Println(i, "odd")
        }
    }
}
```

Давайте рассмотрим эту программу:

- Создать переменную **i** типа **int** и присвоить ей значение **1**;
- **i** меньше или равно **10**? Да - перейти в блок;
- остаток от **i ÷ 2** равен **0**? Нет - переходим к блоку **else**;
- вывести **i** вместе с **odd**;
- инкрементировать **i** (оператор после условия);
- **i** меньше или равно **10**? Да - перейти в блок;
- остаток от **i ÷ 2** равен **0**? Да - переходим к блоку **if**;
- вывести **i** вместе с **even**;
- ...

Оператор деления с остатком (деление по модулю), который редко можно увидеть за пределами начальной школы, оказывается действительно полезным при программировании. Он будет встречаться везде - от раскрашивания таблиц зеброй до секционирования наборов данных.

Switch

Предположим, мы захотели написать программу, которая печатала бы английские названия для чисел. С использованием того, что мы знали до текущего момента, это могло бы выглядеть примерно так:

```
if i == 0 {
    fmt.Println("Zero")
} else if i == 1 {
    fmt.Println("One")
} else if i == 2 {
    fmt.Println("Two")
} else if i == 3 {
    fmt.Println("Three")
} else if i == 4 {
    fmt.Println("Four")
} else if i == 5 {
    fmt.Println("Five")
}
```

Но эта запись слишком избыточна. Go содержит в себе другой оператор, позволяющий делать такие вещи проще: оператор **switch** (переключатель). С ним программа может выглядеть так:

```
switch i {
case 0: fmt.Println("Zero")
case 1: fmt.Println("One")
case 2: fmt.Println("Two")
case 3: fmt.Println("Three")
case 4: fmt.Println("Four")
case 5: fmt.Println("Five")
default: fmt.Println("Unknown Number")
}
```

Переключатель начинается с ключевого слова **switch**, за которым следует выражение (в нашем случае **i**) и серия возможных значений (**case**). Значение выражения по очереди сравнивается с выражениями, следующими после ключевого слова **case**. Если они оказываются равны, то выполняется действие, описанное после **:**.

Как и условия, обход возможных значений осуществляется сверху вниз, и выбирается первое значение, которое сошлось с выражением. Переключатель также поддерживает действие по умолчанию, которое будет выполнено в случае, если не подошло ни одно из возможных значений (напоминает **else** в операторе **if**).

Таковы основные операторы управления потоком. Дополнительные операторы будут рассмотрены в следующих главах.

Задачи

- Что делает следующий код?

```
i := 10

if i > 10 {
    fmt.Println("Big")
} else {
    fmt.Println("Small")
}
```

- Напишите программу, которая выводит числа от 1 до 100, которые делятся на 3. (3, 6, 9, ...).
- Напишите программу, которая выводит числа от 1 до 100. Но для кратных трём нужно вывести «Fizz» вместо числа, для кратных пяти вывести «Buzz», а для кратных как трём, так и пяти — «FizzBuzz».

Массивы, срезы, карты

В главе 3 мы изучили базовые типы Go. В этой главе мы рассмотрим еще три встроенных типа: массивы, срезы и карты.

Массивы

Массив — это нумерованная последовательность элементов одного типа, с фиксированной длиной. В Go они выглядят так:

```
var x [5]int
```

`x` — это пример массива, состоящего из пяти элементов типа `int`. Запустим следующую программу:

```
package main

import "fmt"

func main() {
    var x [5]int
    x[4] = 100
    fmt.Println(x)
}
```

Вы должны увидеть следующее:

```
[0 0 0 0 100]
```

`x[4] = 100` должно читаться как «присвоить пятому элементу массива `x` значение 100». Может показаться странным то, что `x[4]` является пятым элементом массива, а не четвертым, но, как и строки, массивы нумеруются с нуля. Доступ к элементам массива выглядит так же, как у строк.

Вместо `fmt.Println(x)` мы можем написать `fmt.Println(x[4])` и в результате будет выведено `100`.

Пример программы, использующей массивы:

```
func main() {
    var x [5]float64
    x[0] = 98
    x[1] = 93
    x[2] = 77
    x[3] = 82
    x[4] = 83

    var total float64 = 0
    for i := 0; i < 5; i++ {
        total += x[i]
    }
    fmt.Println(total / 5)
}
```

Эта программы вычисляет среднюю оценку за экзамен. Если вы выполните её, то увидите **86.6**. Давайте рассмотрим её внимательнее:

- сперва мы создаем массив длины 5 и заполняем его;
- затем, мы в цикле считаем общее количество баллов;
- и в конце мы делим общую сумму баллов на количество элементов, чтобы узнать средний балл.

Эта программа работает, но её всё еще можно улучшить. Во-первых, бросается в глаза следующее: **i < 5** и **total / 5**. Если мы изменим количество оценок с 5 на 6, то придется переписывать код в этих двух местах. Будет лучше использовать длину массива:

```
var total float64 = 0
for i := 0; i < len(x); i++ {
    total += x[i]
}
fmt.Println(total / len(x))
```

Напишите этот кусок и запустите программу. Вы должны получить ошибку:

```
$ go run tmp.go
# command-line-arguments
.\tmp.go:19: invalid operation: total / len(x) (mismatched
types float64 and int)
```

Проблема в том, что **len(x)** и **total** имеют разный тип. **total** имеет тип **float64**, а **len(x)** — **int**. Так что, нам надо конвертировать **len(x)** в **float64**:

```
fmt.Println(total / float64(len(x)))
```

Это был пример преобразования типов. В целом, для преобразования типа можно использовать имя типа в качестве функции.

Другая вещь, которую мы можем изменить в нашей программе это цикл:

```
var total float64 = 0
for i, value := range x {
    total += value
}
fmt.Println(total / float64(len(x)))
```

В этом цикле **i** представляет текущую позицию в массиве, а **value** будет тем же самым что и **x[i]**. Мы использовали ключевое слово **range** перед переменной, по которой мы хотим пройти циклом.

Выполнение этой программы вызовет другую ошибку:

```
$ go run tmp.go
# command-line-arguments
.\tmp.go:16: i declared and not used
```

Компилятор Go не позволяет вам создавать переменные, которые никогда не используются в коде. Поскольку мы не используем **i** внутри нашего цикла, то надо изменить код следующим образом:

```
var total float64 = 0
for _, value := range x {
    total += value
}
fmt.Println(total / float64(len(x)))
```

Одиночный символ подчеркивания **_** используется чтобы сказать компилятору, что нам не нужна переменная. (В данном случае нам не нужна переменная итератора)

А еще в Go есть короткая запись для создания массивов:

```
x := [5]float64{ 98, 93, 77, 82, 83 }
```

Указывать тип не обязательно — Go сам может его выяснить, по содержимому массива.

Иногда массивы могут оказаться слишком длинными для записи в одну строку, в этом случае Go позволяет записывать их в несколько строк:

```
x := [5]float64{
    98,
    93,
    77,
    82,
    83,
}
```

Обратите внимание на последнюю **,** после **83**. Она обязательна и позволяет легко удалить элемент из массива просто закомментировав строку:

```
x := [4]float64{
    98,
    93,
    77,
    82,
    // 83,
}
```

Срезы

Срез это часть массива. Как и массивы срезы индексируются и имеют длину. В отличие от массивов, их длину можно изменить. Вот пример среза:

```
var x []float64
```

Единственное отличие объявления среза от объявления массива — отсутствие указания длины в квадратных скобках. В нашем случае **x** будет иметь длину 0. Срез создается встроенной функцией **make**:

```
x := make([]float64, 5)
```

Этот код создаст срез, который связан с массивом типа **float64** и длиной **5**. Срезы всегда связаны с каким-нибудь массивом. Они не могут стать больше чем массив, а вот меньше — пожалуйста. Функция **make** принимает и третий параметр:

```
x := make([]float64, 5, 10)
```

10 — это длина массива, на который указывает срез:



Другой способ создать срез — использовать выражение **[low : high]**:

```
arr := [5]float64{1,2,3,4,5}
x := arr[0:5]
```

low это позиция, с которой будет начинаться срез, а **high** это позиция, где он закончится.

Например: **arr[0:5]** вернет **[1,2,3,4,5]**, **arr[1:4]** вернет **[2,3,4]**.

Для удобства, мы также можем опустить **low**, **high** или и то и другое. **arr[0:]** это то же самое что **arr[0:len(arr)]**, **arr[:5]** то же самое что **arr[0:5]** и **arr[:]** то же самое что **arr[0:len(arr)]**.

Функции срезов

В Go есть две встроенные функции для срезов: **append** и **copy**. Вот пример работы функции **append**:

```
func main() {
    slice1 := []int{1,2,3}
    slice2 := append(slice1, 4, 5)
    fmt.Println(slice1, slice2)
}
```

После выполнения программы **slice1** будет содержать **[1,2,3]**, а **slice2** — **[1,2,3,4,5]**. **append** создает новый срез из уже существующего (первый аргумент) и добавляет к нему все следующие аргументы.

Пример работы **copy**:

```
func main() {
    slice1 := []int{1,2,3}
    slice2 := make([]int, 2)
    copy(slice2, slice1)
    fmt.Println(slice1, slice2)
}
```

После выполнения этой программы **slice1** будет содержать **[1,2,3]**, а **slice2** — **[1,2]**. Содержимое **slice1** копируется в **slice2**, но поскольку в **slice2** есть место только для двух элементов, то только два первых элемента **slice1** будут скопированы.

Карта

Карта (также известна как ассоциативный массив или словарь) — это неупорядоченная коллекция пар вида ключ-значение. Пример:

```
var x map[string]int
```

Карта представляется в связке с ключевым словом **map**, следующим за ним типом ключа в скобках и типом значения после скобок. Читается это следующим образом: «**x** — это карта **string**-ов для **int**-ов».

Подобно массивам и срезам, к элементам карт можно обратиться с помощью скобок. Запустим следующую программу:

```
var x map[string]int
x["key"] = 10
fmt.Println(x)
```

Вы должны увидеть ошибку, похожую на эту:

```
panic: runtime error: assignment to entry in nil map

goroutine 1 [running]:
main.main()
    main.go:7 +0x4d

goroutine 2 [syscall]:
created by runtime.main
    C:/Users/ADMINI~1/AppData/Local/Temp/2/bindit269497170/go/src/pkg/runtime/proc.c:221
exit status 2
```

До этого момента мы имели дело только с ошибками во время компиляции. Сейчас мы видим ошибку исполнения.

Проблема нашей программы в том, что карта должна быть инициализирована перед тем, как будет использована. Надо написать так:

```
x := make(map[string]int)
x["key"] = 10
fmt.Println(x["key"])
```

Если выполнить эту программу, то вы должны увидеть **10**.

Выражение **x["key"] = 10** похоже на то, что использовались при работе с массивами, но ключ тут не число, а строка (потому что в карте указан тип ключа **string**). Мы также можем создать карты с ключом типа **int**:

```
x := make(map[int]int)
x[1] = 10
fmt.Println(x[1])
```

Это выглядит очень похоже на массив, но существует несколько различий. Во-первых длина карты (которую мы можем найти так: **len(x)**) может измениться, когда мы добавим новый элемент в него. В самом начале, при создании длина **0**, после **x[1] = 10** она станет равна **1**. Во-вторых, карта не является последовательностью. В нашем примере у нас есть элемент **x[1]**, в случае массива должен быть и первый элемент **x[0]**, но в картах это не так. Также мы можем удалить элементы из карты используя встроенную функцию **delete**:

```
delete(x, 1)
```

Давайте посмотрим на пример программы, использующей карты:

```
package main

import "fmt"

func main() {
    elements := make(map[string]string)
    elements["H"] = "Hydrogen"
    elements["He"] = "Helium"
    elements["Li"] = "Lithium"
    elements["Be"] = "Beryllium"
    elements["B"] = "Boron"
    elements["C"] = "Carbon"
    elements["N"] = "Nitrogen"
    elements["O"] = "Oxygen"
    elements["F"] = "Fluorine"
    elements["Ne"] = "Neon"

    fmt.Println(elements["Li"])
}
```

В данном примере, `elements` это карта, которое представляет 10 первых химических элементов, индексируемых символами. Это очень частый способ использования карт — в качестве словаря, или таблицы. Предположим, мы пытаемся обратиться к не существующему элементу:

```
fmt.Println(elements["Un"])
```

Если вы выполните это, то ничего не увидите. Технически, карта вернет нулевое значение хранящегося типа (для строк это пустая строка). Несмотря на то, что мы можем проверить нулевое значение с помощью условия (`elements["Un"] == ""`), в Go есть лучший способ сделать это:

```
name, ok := elements["Un"]
fmt.Println(name, ok)
```

Доступ к элементу карты может вернуть два значения вместо одного. Первое значение это результат запроса, второе говорит был ли запрос успешен. В Go часто встречается такой код:

```
if name, ok := elements["Un"]; ok {
    fmt.Println(name, ok)
}
```

Сперва мы пробуем получить значение из карты, а затем, если это удалось, мы выполняем код внутри блока.

Объявления карт можно записывать сокращенно, также как массивы:

```
elements := map[string]string{
    "H": "Hydrogen",
    "He": "Helium",
    "Li": "Lithium",
    "Be": "Beryllium",
    "B": "Boron",
    "C": "Carbon",
    "N": "Nitrogen",
    "O": "Oxygen",
    "F": "Fluorine",
    "Ne": "Neon",
}
```

Карты часто используются для хранения общей информации. Давайте изменим нашу программу так, чтобы вместо имени элемента хранить какую-нибудь дополнительную информацию о нем. Например, его агрегатное состояние:

```
func main() {
    elements := map[string]map[string]string{
        "H": map[string]string{
            "name": "Hydrogen",
            "state": "gas",
        },
        "He": map[string]string{
            "name": "Helium",
            "state": "gas",
        },
        "Li": map[string]string{
            "name": "Lithium",
            "state": "solid",
        },
        "Be": map[string]string{
            "name": "Beryllium",
            "state": "solid",
        },
        "B": map[string]string{
            "name": "Boron",
            "state": "solid",
        },
        "C": map[string]string{
            "name": "Carbon",
            "state": "solid",
        },
        "N": map[string]string{
            "name": "Nitrogen",
            "state": "gas",
        },
    },
}
```

```

    "O": map[string]string{
        "name": "Oxygen",
        "state": "gas",
    },
    "F": map[string]string{
        "name": "Fluorine",
        "state": "gas",
    },
    "Ne": map[string]string{
        "name": "Neon",
        "state": "gas",
    },
}

if el, ok := elements["Li"]; ok {
    fmt.Println(el["name"], el["state"])
}
}

```

Заметим, что тип нашей карты теперь `map[string]map[string]string`. Мы получили карту строк для карты строк. Внешняя карта используется как поиск по символу химического элемента, а внутренняя — для хранения информации об элементе. Не смотря на то, что карты часто используется таким образом, в главе 9 мы узнаем лучший способ хранения данных.

Задачи

- Как обратиться к четвертому элементу массива или среза?
- Чему равна длина среза, созданного таким способом: `make([]int, 3, 9)`?
- Дан массив:

```
x := [6]string{"a","b","c","d","e","f"}
```

что вернет вам `x[2:5]`?

- Напишите программу, которая находит самый наименьший элемент в этом списке:

```
x := []int{
    48,96,86,68,
    57,82,63,70,
    37,34,83,27,
    19,97, 9,17,
}
```

Функции

Функция является независимой частью кода, связывающая один или несколько входных параметров с одним или несколькими выходными параметрами.

Функции (так же известные как процедуры и подпрограммы) можно представить как черный ящик:



До сих пор мы писали программы, используя лишь одну функцию:

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

Но сейчас мы начнем писать программы, содержащие более одной функции.

Ваша вторая функция

Вспомните эту программу из предыдущей главы:

```
func main() {
    xs := []float64{98,93,77,82,83}

    total := 0.0
    for _, v := range xs {
        total += v
    }
    fmt.Println(total / float64(len(xs)))
}
```

Эта программа вычисляет среднее значение ряда чисел. Поиск среднего значения — основная задача и идеальный кандидат для вынесения в отдельную функцию.

Функция **average** должна взять срез из нескольких **float64** и вернуть один **float64**. Напишем перед функцией **main**:

```
func average(xs []float64) float64 {
    panic("Not Implemented")
}
```

Функция начинается с ключевого слова **func**, за которым следует имя функции. Аргументы (входы) определяются так: **имя тип, имя тип,** Наша функция

имеет один параметр (список оценок) под названием **xs**. За параметром следует возвращаемый тип. В совокупности аргументы и возвращаемое значение также известны как сигнатура функции.

Наконец, тело функции, заключенное в фигурные скобки. В теле вызывается встроенная функция **panic**, которая вызывает ошибку выполнения (о ней я расскажу чуть позже в этой главе). Процесс написания функций может быть сложен, поэтому деление этого процесса на несколько частей вместо попытки реализовать всё за один большой шаг — хорошая идея.

Теперь давайте перенесём часть кода из функции **main** в функцию **average**:

```
func average(xs []float64) float64 {
    total := 0.0
    for _, v := range xs {
        total += v
    }
    return total / float64(len(xs))
}
```

Обратите внимание, что мы заменили вызов **fmt.Println** на оператор **return**. Оператор возврата немедленно прервет выполнение функции и вернет значение, указанное после оператора, в функцию, которая вызвала текущую.

Приведем **main** к следующему виду:

```
func main() {
    xs := []float64{98,93,77,82,83}
    fmt.Println(average(xs))
}
```

Запуск этой программы должен дать точно такой же результат, что и раньше. Несколько моментов, которые нужно иметь в виду:

- Имена аргументов не обязательно должны совпадать с именами переменных, при вызове функции. Например, можно сделать так:

```
func main() {
    someOtherName := []float64{98,93,77,82,83}
    fmt.Println(average(someOtherName))
}
```

и программа продолжит работать.

- Функции не имеют доступа к области видимости родительской функции. То есть это не работает:

```
func f() {
    fmt.Println(x)
}
func main() {
    x := 5
    f()
}
```

Как минимум, нужно сделать так:

```
func f(x int) {
    fmt.Println(x)
}
func main() {
    x := 5
    f(x)
}
```

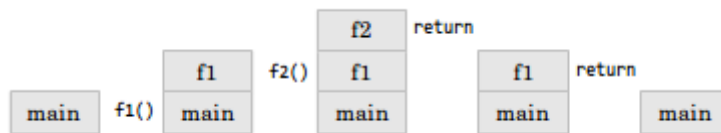
или так:

```
var x int = 5
func f() {
    fmt.Println(x)
}
func main() {
    f()
}
```

- Функции выстраиваются в «стек вызовов». Предположим, у нас есть такая программа:

```
func main() {
    fmt.Println(f1())
}
func f1() int {
    return f2()
}
func f2() int {
    return 1
}
```

Её можно представить следующим образом:



Каждая вызываемая функция помещается в стек вызовов, каждый возврат из функции возвращает нас в последний стек.

- Можно также указать тип возвращаемого значения

```
func f2() (r int) {  
    r = 1  
    return  
}
```

Возврат нескольких значений

Go способен возвращать несколько значений из функции:

```
func f() (int, int) {  
    return 5, 6  
}  
  
func main() {  
    x, y := f()  
}
```

Для этого необходимы три вещи: указать несколько типов возвращаемых значений, разделенных **,**, изменить выражение после **return** так, чтобы оно содержало несколько значений, разделенных **,** и, наконец, изменить конструкцию присвоения так, чтобы она содержала несколько значений в левой стороне перед **:=** или **=**.

Возврат нескольких значений часто используется для возврата ошибки вместе с результатом: (**x, err := f()**) или логическое значение, говорящее об успешном выполнении (**x, ok := f()**).

Переменное число аргументов функции

Существует особая форма записи последнего аргумента в функции Go:

```
func add(args ...int) int {
    total := 0
    for _, v := range args {
        total += v
    }
    return total
}

func main() {
    fmt.Println(add(1,2,3))
}
```

Использование `...` перед типом последнего аргумента означает, что функция может содержать ноль и более таких параметров. В нашем случае мы берем ноль и более `int`. Функцию можно вызывать как и раньше, но при этом ей можно передать любое количество аргументов типа `int`.

Это похоже на реализацию функции `Println`:

```
func Println(a ...interface{}) (n int, err error)
```

Функция `Println` может принимать любое количество аргументов любого типа (тип `interface` мы рассмотрим в главе 9).

Мы также можем передать срез `int`-ов, указав `...` после среза:

```
func main() {
    xs := []int{1,2,3}
    fmt.Println(add(xs...))
}
```

Замыкания

Возможно создавать функции внутри функций:

```
func main() {
    add := func(x, y int) int {
        return x + y
    }
    fmt.Println(add(1,1))
}
```

`add` является локальной переменной типа `func(int, int) int` (функция принимает два аргумента типа `int` и возвращает `int`). При создании локальная функция также получает доступ к локальным переменным (вспомните области видимости из главы 4):

```
func main() {
    x := 0
    increment := func() int {
        x++
        return x
    }
    fmt.Println(increment())
    fmt.Println(increment())
}
```

increment прибавляет **1** к переменной **x**, которая определена в рамках функции **main**. Значение переменной **x** может быть изменено в функции **increment**. Вот почему при первом вызове **increment** на экран выводится **1**, а при втором — **2**.

Функцию, использующую переменные, определенные вне этой функции, называют замыканием. В нашем случае функция **increment** и переменная **x** образуют замыкание.

Один из способов использования замыкания — функция, возвращающая другую функцию, которая при вызове генерирует некую последовательность чисел. Например, следующим образом мы могли бы сгенерировать все четные числа:

```
func makeEvenGenerator() func() uint {
    i := uint(0)
    return func() (ret uint) {
        ret = i
        i += 2
        return
    }
}

func main() {
    nextEven := makeEvenGenerator()
    fmt.Println(nextEven()) // 0
    fmt.Println(nextEven()) // 2
    fmt.Println(nextEven()) // 4
}
```

makeEvenGenerator возвращает функцию, которая генерирует чётные числа. Каждый раз, когда она вызывается, к переменной **i** добавляется **2**, но в отличие от обычных локальных переменных её значение сохраняется между вызовами.

Рекурсия

Наконец, функция может вызывать саму себя. Вот один из способов вычисления факториала числа:

```
func factorial(x uint) uint {
    if x == 0 {
        return 1
    }

    return x * factorial(x-1)
}
```

factorial вызывает саму себя, что делает эту функцию рекурсивной. Для того, чтобы лучше понять, как работает эта функция, давайте пройдемся по **factorial(2)**:

- **x == 0**? Нет. (**x** равен **2**);
- Ищем факториал от **x - 1**;
 - **x == 0**? Нет. (**x** равен **1**)
- Ищем факториал от **0**.
 - **x == 0**? Да, возвращаем **1**.
- Возвращаем **1 * 1**
- Возвращаем **2 * 1**

Замыкание и рекурсивный вызов — сильные техники программирования, формирующие основу парадигмы, известной как функциональное программирование. Большинство людей находят функциональное программирование более сложным для понимания, чем подход на основе циклов, логических операторов, переменных и простых функций.

Отложенный вызов, паника и восстановление

В Go есть специальный оператор **defer**, который позволяет отложить вызов указанной функции до тех пор, пока не завершится текущая. Рассмотрим следующий пример:

```
package main

import "fmt"

func first() {
    fmt.Println("1st")
}

func second() {
    fmt.Println("2nd")
}

func main() {
    defer second()
    first()
}
```


Эта программа выводит **1st**, затем **2nd**. Грубо говоря, **defer** перемещает вызов **second** в конец функции:

```
func main() {  
    first()  
    second()  
}
```

defer часто используется в случаях, когда нужно освободить ресурсы после завершения. Например, открывая файл, необходимо убедиться, что позже он должен быть закрыт. С **defer** это выглядит так:

```
f, _ := os.Open(filename)  
defer f.Close()
```

Такой подход дает нам три преимущества: (1) Вызовы **Close** и **Open** располагаются рядом, что облегчает понимание программы. (2) Если функция содержит несколько операций возврата (например, одна произойдет в блоке **if**, другая в блоке **else**), **Close** будет вызван до выхода из функции. (3) Отложенные функции выполняются, даже если во время выполнения происходит ошибка.

Паника и восстановление

Ранее мы создали функцию, которая вызывает **panic**, чтобы создать ошибку выполнения. Мы можем обрабатывать паники с помощью встроенной функции **recover**. Функция **recover** останавливает панику и возвращает значение, которое было передано функции **panic**. Можно попытаться использовать **recover** следующим образом:

```
package main  
  
import "fmt"  
  
func main() {  
    panic("PANIC")  
    str := recover()  
    fmt.Println(str)  
}
```

Но в данном случае **recover** никогда не будет вызван, поскольку вызов **panic** немедленно останавливает выполнение функции. Вместо этого мы должны использовать его вместе с **defer**:

```
package main

import "fmt"

func main() {
    defer func() {
        str := recover()
        fmt.Println(str)
    }()
    panic("PANIC")
}
```

Паника обычно указывает на ошибку программиста (например, попытку получить доступ к несуществующему индексу массива, забытая и непроинициализированная карта и т.д.) или неожиданное поведение (исключение), которое нельзя обработать (поэтому оно и называется «паника»).

Задачи

- Функция `sum` принимает срез чисел и складывает их вместе. Как бы выглядела сигнатура этой функции?
- Напишите функцию, которая принимает число, делит его пополам и возвращает `true` в случае, если получившееся число чётное, или `false` в случае нечетного результата. Например, `half(1)` должна вернуть `(0, false)`, в то время как `half(2)` вернет `(1, true)`.
- Напишите функцию с переменным числом параметров, которая находит наибольшее число в списке.
- Используя в качестве примера функцию `makeEvenGenerator`, напишите `makeOddGenerator`, генерирующую нечётные числа.
- Последовательность чисел Фибоначчи определяется как `fib(0) = 0, fib(1) = 1, fib(n) = fib(n-1) + fib(n-2)`. Напишите рекурсивную функцию, находящую `fib(n)`.
- Что такое отложенный вызов, паника и восстановление? Как восстановить функцию после паники?

Указатели

Когда мы вызываем функцию с аргументами, аргументы копируются в функцию:

```
func zero(x int) {
    x = 0
}
func main() {
    x := 5
    zero(x)
    fmt.Println(x) // x всё еще равен 5
}
```

В этой программе функция **zero** не изменяет оригинальную переменную **x** из функции **main**. Но что если мы хотим её изменить? Один из способов сделать это — использовать специальный тип данных — указатель:

```
func zero(xPtr *int) {
    *xPtr = 0
}
func main() {
    x := 5
    zero(&x)
    fmt.Println(x) // x is 0
}
```

Указатели указывают (прошу прощения за тавтологию) на участок в памяти, где хранится значение. Используя указатель (***int**) в функции **zero**, мы можем изменить значение оригинальной переменной.

Операторы * и &

В Go указатели представлены через оператор ***** (звёздочка), за которым следует тип хранимого значения. В функции **zero xPtr** является указателем на **int**. ***** также используется для «разыменовывания» указателей. Когда мы пишем ***xPtr = 0**, то читаем это так: «Храним **int** 0 в памяти, на которую указывает **xPtr**». Если вместо этого мы попробуем написать **xPtr = 0**, то получим ошибку компиляции, потому что **xPtr** имеет тип не **int**, а ***int**. Соответственно, ему может быть присвоен только другой ***int**. Также существует оператор **&**, который используется для получения адреса переменной. **&x** вернет ***int** (указатель на **int**) потому что **x** имеет тип **int**. Теперь мы можем изменять оригинальную переменную. **&x** в функции **main** и **xPtr** в функции **zero** указывают на один и тот же участок в памяти.

Оператор new

Другой способ получить указатель — использовать встроенную функцию **new**:

```
func one(xPtr *int) {
    *xPtr = 1
}
func main() {
    xPtr := new(int)
    one(xPtr)
    fmt.Println(*xPtr) // x is 1
}
```

Функция **new** принимает аргументом тип, выделяет для него память и возвращает указатель на эту память.

В некоторых языках программирования есть существенная разница между использованием **new** и **&**, и в них нужно удалять всё, что было создано с помощью **new**. Go не такой - Go хороший. Go — язык с автоматической сборкой мусора. Это означает, что область памяти очищается автоматически, когда на неё не остаётся ссылок.

Указатели редко используются в Go для встроенных типов, но они будут часто фигурировать в следующей главе (они чрезвычайно полезны при работе со структурами).

Задачи

- Как получить адрес переменной?
- Как присвоить значение указателю?
- Как создать новый указатель?
- Какое будет значение у переменной **x** после выполнения программы:

```
func square(x *float64) {  
    *x = *x * *x  
}  
func main() {  
    x := 1.5  
    square(&x)  
}
```

- Напишите программу, которая меняет местами два числа (**x := 1; y := 2; swap(&x, &y)** должно дать **x=2** и **y=1**).

Структуры и интерфейсы

Несмотря на то, что вполне можно писать программы на Go используя только встроенные типы, в какой-то момент это станет очень утомительным занятием. Вот пример — программа, которая взаимодействует с фигурами:

```
package main

import ("fmt"; "math")

func distance(x1, y1, x2, y2 float64) float64 {
    a := x2 - x1
    b := y2 - y1
    return math.Sqrt(a*a + b*b)
}

func rectangleArea(x1, y1, x2, y2 float64) float64 {
    l := distance(x1, y1, x1, y2)
    w := distance(x1, y1, x2, y1)
    return l * w
}

func circleArea(x, y, r float64) float64 {
    return math.Pi * r*r
}

func main() {
    var rx1, ry1 float64 = 0, 0
    var rx2, ry2 float64 = 10, 10
    var cx, cy, cr float64 = 0, 0, 5

    fmt.Println(rectangleArea(rx1, ry1, rx2, ry2))
    fmt.Println(circleArea(cx, cy, cr))
}
```

Отслеживание всех переменных мешает нам понять, что делает программа, и наверняка приведет к ошибкам.

Структуры

С помощью структур эту программу можно сделать гораздо лучше. Структура — это тип, содержащий именованные поля. Например, мы можем представить круг таким образом:

```
type Circle struct {
    x float64
    y float64
    r float64
}
```

Ключевое слово **type** вводит новый тип. За ним следует имя нового типа (**Circle**) и ключевое слово **struct**, которое говорит, что мы определяем структуру и список полей внутри фигурных скобок. Каждое поле имеет имя и тип. Как и с функциями, мы можем объединять поля одного типа:

```
type Circle struct {  
    x, y, r float64  
}
```

Инициализация

Мы можем создать экземпляр нового типа **Circle** несколькими способами:

```
var c Circle
```

Подобно другим типами данных, будет создана локальная переменная типа **Circle**, чьи поля по умолчанию будут равны нулю (0 для **int**, 0.0 для **float**, "" для **string**, **nil** для указателей, ...). Также, для создания экземпляра можно использовать функцию **new**.

```
c := new(Circle)
```

Это выделит память для всех полей, присвоит каждому из них нулевое значение и вернет указатель (***Circle**). Часто, при создании структуры мы хотим присвоить полям структуры какие-нибудь значения. Существует два способа сделать это. Первый способ:

```
c := Circle{x: 0, y: 0, r: 5}
```

Второй способ — мы можем опустить имена полей, если мы знаем порядок в котором они определены:

```
c := Circle{0, 0, 5}
```

Поля

Получить доступ к полям можно с помощью оператора **.** (точка):

```
fmt.Println(c.x, c.y, c.r)  
c.x = 10  
c.y = 5
```


Давайте изменим функцию `circleArea` так, чтобы она использовала структуру `Circle`:

```
func circleArea(c Circle) float64 {  
    return math.Pi * c.r*c.r  
}
```

В функции `main` у нас будет:

```
c := Circle{0, 0, 5}  
fmt.Println(circleArea(c))
```

Очень важно помнить о том, что аргументы в Go всегда копируются. Если мы попытаемся изменить любое поле в функции `circleArea`, оригинальная переменная не изменится. Именно поэтому мы будем писать функции так:

```
func circleArea(c *Circle) float64 {  
    return math.Pi * c.r*c.r  
}
```

И изменим `main`:

```
c := Circle{0, 0, 5}  
fmt.Println(circleArea(&c))
```

Методы

Не смотря на то, что программа стала лучше, мы все еще можем значительно её улучшить, используя метод — функцию особого типа:

```
func (c *Circle) area() float64 {  
    return math.Pi * c.r*c.r  
}
```

Между ключевым словом `func` и именем функции мы добавили «получателя». Получатель похож на параметр — у него есть имя и тип, но объявление функции таким способом позволяет нам вызывать функцию с помощью оператора `.`:

```
fmt.Println(c.area())
```

Это гораздо проще прочесть, нам не нужно использовать оператор `&` (Go автоматически предоставляет доступ к указателю на `Circle` для этого метода), и поскольку эта функция может быть использована только для `Circle` мы можем назвать её просто `area`.

Давайте сделаем то же самое с прямоугольником:

```
type Rectangle struct {
    x1, y1, x2, y2 float64
}
func (r *Rectangle) area() float64 {
    l := distance(r.x1, r.y1, r.x1, r.y2)
    w := distance(r.x1, r.y1, r.x2, r.y1)
    return l * w
}
```

В **main** будет написано:

```
r := Rectangle{0, 0, 10, 10}
fmt.Println(r.area())
```

Встраиваемые типы

Обычно, поля структур представляют отношения принадлежности (включения). Например, у **Circle** (круга) есть **radius** (радиус). Предположим, у нас есть структура **Person** (личность):

```
type Person struct {
    Name string
}
func (p *Person) Talk() {
    fmt.Println("Hi, my name is", p.Name)
}
```

И если мы хотим создать новую структуру **Android**, то можем сделать так:

```
type Android struct {
    Person Person
    Model string
}
```

Это будет работать, но мы можем захотеть создать другое отношение. Сейчас у андроида «есть» личность, можем ли мы описать отношение андроид «является» личностью. Go поддерживает подобные отношения с помощью встраиваемых типов, также называемых анонимными полями. Выглядят они так:

```
type Android struct {
    Person
    Model string
}
```

Мы использовали тип (**Person**) и не написали его имя. Объявленная таким способом структура доступна через имя типа:

```
a := new(Android)
a.Person.Talk()
```

Но мы также можем вызвать любой метод **Person** прямо из **Android**:

```
a := new(Android)
a.Talk()
```

Это отношение работает достаточно интуитивно: личности могут говорить, андроид это личность, значит андроид может говорить.

Интерфейсы

Вы могли заметить, что названия методов для вычисления площади круга и прямоугольника совпадают. Это было сделано не случайно. И в реальной жизни и в программировании отношения могут быть очень похожими. В Go есть способ сделать эти случайные сходства явными с помощью типа называемого интерфейсом. Пример интерфейса для фигуры (**Shape**):

```
type Shape interface {
    area() float64
}
```

Как и структуры, интерфейсы создаются с помощью ключевого слова **type**, за которым следует имя интерфейса и ключевое слово **interface**. Однако, вместо того, чтобы определять поля, мы определяем «множество методов». Множество методов - это список методов, которые будут использоваться для «реализации» интерфейса.

В нашем случае у **Rectangle** и **Circle** есть метод **area**, который возвращает **float64**, получается они оба реализуют интерфейс **Shape**. Само по себе это не очень полезно, но мы можем использовать интерфейсы как аргументы в функциях:

```
func totalArea(shapes ...Shape) float64 {
    var area float64
    for _, s := range shapes {
        area += s.area()
    }
    return area
}
```

Мы будем вызывать эту функцию так:

```
fmt.Println(totalArea(&c, &r))
```

Интерфейсы также могут быть использованы в качестве полей:

```
type MultiShape struct {  
    shapes []Shape  
}
```

Мы можем даже хранить в **MultiShape** данные **Shape**, определив в ней метод **area**:

```
func (m *MultiShape) area() float64 {  
    var area float64  
    for _, s := range m.shapes {  
        area += s.area()  
    }  
    return area  
}
```

Теперь **MultiShape** может содержать **Circle**, **Rectangle** и даже другие **MultiShape**.

Задачи

- Какая разница между методом и функцией?
- В каких случаях могут пригодиться встроенные (скрытые) поля?
- Добавьте новый метод **perimeter** в интерфейс **Shape**, который будет вычислять периметр фигуры. Имплементируйте этот метод для **Circle** и **Rectangle**.

Многопоточность

Очень часто, большие приложения состоят из множества небольших подпрограмм. Например, web-сервер принимает запросы от браузера и отправляет HTML страницы в ответ. Каждый такой запрос выполняется как отдельная небольшая программа.

Такой способ идеально подходит для подобных приложений, так как обеспечивает возможность одновременного запуска множества более мелких компонентов (обработки нескольких запросов одновременно, в случае веб-сервера). Одновременное выполнение более чем одной задачи известно как многопоточность. Go имеет богатую функциональность для работы с многопоточностью, в частности, такие инструменты как горутини и каналы.

Горутини

Горутини — это функция, которая может работать параллельно с другими функциями. Для создания горутини используется ключевое слово **go**, за которым следует вызов функции.

```
package main

import "fmt"

func f(n int) {
    for i := 0; i < 10; i++ {
        fmt.Println(n, ":", i)
    }
}

func main() {
    go f(0)
    var input string
    fmt.Scanln(&input)
}
```

Эта программа состоит из двух горутин. Функция **main**, сама по себе, является горутини. Вторая горутини создаётся, когда мы вызываем **go f(0)**. Обычно, при вызове функции, программа выполнит все конструкции внутри вызываемой функции, а только потом перейдет к, следующей после вызова, строке. С горутини программа немедленно перейдет к следующей строке, не дожидаясь, пока вызываемая функция завершится. Вот почему здесь присутствует вызов **Scanln**, без него программа завершится еще перед тем, как ей удастся вывести числа.

Горутини очень легкие, мы можем создавать их тысячами. Давайте изменим программу так, чтобы она запускала 10 горутин:

```
func main() {
    for i := 0; i < 10; i++ {
        go f(i)
    }
    var input string
    fmt.Scanln(&input)
}
```

При запуске вы наверное заметили, что все горутини выполняются последовательно, а не одновременно, как вы того ожидали. Давайте добавим небольшую задержку функции с помощью функции **time.Sleep** и **rand.Inin**:

```
package main

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

func f(n int) {
    for i := 0; i < 10; i++ {
        fmt.Println(n, ":", i)
        amt := time.Duration(rand.Intn(250))
        time.Sleep(time.Millisecond * amt)
    }
}

func main() {
    for i := 0; i < 10; i++ {
        go f(i)
    }
    var input string
    fmt.Scanln(&input)
}
```

f выводит числа от 0 до 10, ожидая от 0 до 250 мс после каждой операции вывода. Теперь горутини должны выполняться одновременно.

Каналы

Каналы обеспечивают возможность общения нескольких горутин друг с другом, чтобы синхронизировать их выполнение. Вот пример программы с использованием каналов:

```
package main

import (
    "fmt"
    "time"
)

func pinger(c chan string) {
    for i := 0; ; i++ {
        c <- "ping"
    }
}

func printer(c chan string) {
    for {
        msg := <- c
        fmt.Println(msg)
        time.Sleep(time.Second * 1)
    }
}

func main() {
    var c chan string = make(chan string)

    go pinger(c)
    go printer(c)

    var input string
    fmt.Scanln(&input)
}
```

Программа будет постоянно выводить «ping» (нажмите enter, чтобы её остановить). Тип канала представлен ключевым словом **chan**, за которым следует тип, который будет передаваться по каналу (в данном случае мы передаем строки). Оператор **<-** (стрелка влево) используется для отправки и получения сообщений по каналу. Конструкция **c <- "ping"** означает отправку **"ping"**, а **msg := <- c** — его получение и сохранение в переменную **msg**. Строка с **fmt** может быть записана другим способом: **fmt.Println(<-c)**, тогда можно было бы удалить предыдущую строку. Данное использование каналов позволяет синхронизировать две горутин. Когда **pinger** пытается послать сообщение в канал, он ожидает, пока **printer** будет готов получить сообщение. Такое поведение называется блокирующим. Давайте добавим ещё одного отправителя сообщений в программу и посмотрим, что будет. Добавим эту функцию:


```
func ponger(c chan string) {
    for i := 0; ; i++ {
        c <- "pong"
    }
}
```

и изменим функцию **main**:

```
func main() {
    var c chan string = make(chan string)

    go pinger(c)
    go ponger(c)
    go printer(c)

    var input string
    fmt.Scanln(&input)
}
```

Теперь программа будет выводить на экран то **ping**, то **pong** по очереди.

Направление каналов

Мы можем задать направление передачи сообщений в канале, сделав его только отправляющим или принимающим. Например, мы можем изменить функцию **pinger**:

```
func pinger(c chan<- string)
```

и канал **c** будет только отправлять сообщение. Попытка получить сообщение из канала **c** вызовет ошибку компилирования. Также мы можем изменить функцию **printer**:

```
func printer(c <-chan string)
```

Существуют и двунаправленные каналы, которые могут быть переданы в функцию, принимающую только принимающие или отправляющие каналы. Но только отправляющие или принимающие каналы не могут быть переданы в функцию, требующую двунаправленного канала!

Оператор Select

В языке Go есть специальный оператор **select** который работает как **switch**, но для каналов:

```
func main() {
    c1 := make(chan string)
    c2 := make(chan string)

    go func() {
        for {
            c1 <- "from 1"
            time.Sleep(time.Second * 2)
        }
    }()
    go func() {
        for {
            c2 <- "from 2"
            time.Sleep(time.Second * 3)
        }
    }()
    go func() {
        for {
            select {
                case msg1 := <- c1:
                    fmt.Println(msg1)
                case msg2 := <- c2:
                    fmt.Println(msg2)
            }
        }
    }()

    var input string
    fmt.Scanln(&input)
}
```

Эта программа выводит «from 1» каждые 2 секунды и «from 2» каждые 3 секунды. Оператор **select** выбирает первый готовый канал, и получает сообщение из него, или же передает сообщение через него. Когда готовы несколько каналов, получение сообщения происходит из случайно выбранного готового канала. Если же ни один из каналов не готов, оператор блокирует ход программы до тех пор, пока какой-либо из каналов будет готов к отправке или получению.

Обычно **select** используется для таймеров:

```
select {
case msg1 := <- c1:
    fmt.Println("Message 1", msg1)
case msg2 := <- c2:
    fmt.Println("Message 2", msg2)
case <- time.After(time.Second):
    fmt.Println("timeout")
}
```

time After создаёт канал, по которому посылаем метки времени с заданным интервалом. В данном случае мы не заинтересованы в значениях временных меток, поэтому мы не сохраняем его в переменные. Также мы можем задать команды, которые выполняются по умолчанию, используя конструкцию **default**:

```
select {
case msg1 := <- c1:
    fmt.Println("Message 1", msg1)
case msg2 := <- c2:
    fmt.Println("Message 2", msg2)
case <- time.After(time.Second):
    fmt.Println("timeout")
default:
    fmt.Println("nothing ready")
}
```

Выполняемые по умолчанию команды исполняются сразу же, если все каналы заняты.

Буферизированный канал

При инициализации канала можно использовать второй параметр:

```
c := make(chan int, 1)
```

и мы получим буферизированный канал с ёмкостью **1**. Обычно каналы работают синхронно - каждая из сторон ждёт, когда другая сможет получить или передать сообщение. Но буферизованный канал работает асинхронно — получение или отправка сообщения не заставляют стороны останавливаться. Но канал теряет пропускную способность, когда он занят, в данном случае, если мы отправим в канал 1 сообщение, то мы не сможем отправить туда ещё одно до тех пор, пока первое не будет получено.

Задачи

- Как задать направление канала?
- Напишите собственную функцию **Sleep**, используя **time.After**
- Что такое буферизированный канал? Как создать такой канал с ёмкостью в 20 сообщений?

Пакеты и повторное использование кода

Go разработан как язык, который поощряет хорошие инженерные практики. Одной из этих практик, позволяющих создавать высококачественное программное обеспечение, является повторное использование кода, называемое DRY — «Don't Repeat Yourself» — (акроним, в переводе с английского) — «не повторяйтесь!». Как мы уже видели в 7 главе, функции являются первым уровнем повторного использования кода. Но Go поддерживает ещё один механизм для повторного использования кода — пакеты. Почти любая программа, которую мы видели, включает эту строку:

```
import "fmt"
```

fmt — это имя пакета, включающего множество функций, связанных с форматированием строк и выводом на экран. Данный метод распространения кода обусловлен тремя причинами:

- Снижение вероятности дублирования имён функций, что позволяет именам быть простыми и краткими
- Организация кода для упрощения поиска повторно используемых конструкций
- Ускорение компиляции, так как мы должны перекомпилировать только части программы. Не смотря на то, что мы используем пакет **fmt**, мы не должны перекомпилировать его при каждом использовании

Создание пакета

Использовать пакеты имеет смысл, только когда они востребованы отдельной программой. Без неё использовать пакеты невозможно.

Давайте создадим программу, которая будет использовать наш пакет. Создадим директорию в `~/Go/src/golang-book` под названием **chapter11**. В ней создадим файл **main.go** с этим кодом:

```
package main

import "fmt"
import "golang-book/chapter11/math"

func main() {
    xs := []float64{1,2,3,4}
    avg := math.Average(xs)
    fmt.Println(avg)
}
```

А теперь создадим ещё одну директорию внутри **chapter11** под названием **math**. В ней мы создадим файл **math.go** с этим кодом:

```
package math

func Average(xs []float64) float64 {
    total := float64(0)
    for _, x := range xs {
        total += x
    }
    return total / float64(len(xs))
}
```

С помощью терминала в папке **math** запустите команду **go install**. В результате файл **math.go** скомпилируется в объектный файл **~/Go/pkg/os_arch/golang-book/chapter11/math.a** (при этом, **os** может быть **Windows**, а **arch**, например, — **amd64**)

Теперь вернёмся в директорию **chapter11** и выполним **go run main.go**. Программа выведет **2.5** на экран. Подведём итоги:

- **math** является встроенным пакетом, но так как пакеты Go используют иерархические наименования, мы можем перекрыть уже используемое наименование, в данном случае настоящий пакет **math** и будет называться **math**, а наш — **golang-book/chapter11/math**.
- Когда мы импортируем библиотеку, мы используем её полное наименование **import "golang-book/chapter11/math"**, но внутри файла **math.go** мы используем только последнюю часть названия — **package math**.
- Мы используем только краткое имя **math** когда мы обращаемся к функциям в нашем пакете. Если же мы хотим использовать оба пакета, то мы можем использовать псевдоним:

```
import m "golang-book/chapter11/math"

func main() {
    xs := []float64{1,2,3,4}
    avg := m.Average(xs)
    fmt.Println(avg)
}
```

В этом коде **m** — псевдоним.

- Возможно вы заметили, что каждая функция в пакете начинается с заглавной буквы. Любая сущность языка Go, которая начинается с заглавной буквы, означает, что другие пакеты и программы могут использовать эту сущность. Если бы мы назвали нашу функцию **average**, а не **Average**, то наша главная программа не смогла бы обратиться к ней.
- Рекомендуются делать явными только те сущности нашего пакета, которые могут быть использованы другими пакетами, и прятать все остальные

служебные функции, не используемые в других пакетах. Данный подход позволяет производить изменения в скрытых частях пакета без риска нарушить работу других программ, и это облегчает использование нашего пакета

- Имена пакетов совпадают с директориями, в которых они размещены. Данное правило можно обойти, но делать это нежелательно.

Документация к коду

Go позволяет автоматически создавать документацию к пользовательским пакетам так же, как и документировать стандартные пакеты. Запустите эту команду в терминале:

```
godoc golang-book/chapter11/math Average
```

И вы увидите информацию о функции, которую мы только что написали. Мы можем улучшить документацию, добавив комментарий перед функцией:

```
// Найти среднее в массиве чисел.  
func Average(xs []float64) float64 {
```

Если вы запустите **go install**, а потом перезапустите:

```
godoc golang-book/chapter11/math Average
```

то вы увидите наш комментарий — **Найти среднее в массиве чисел**. Также вы можете увидеть документацию на интернет-странице, запустив в терминале команду:

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

и открыв этот адрес в браузере **http://localhost:6060/pkg/**.

Вы увидите документацию по всем пакетам, установленным в системе, в том числе и про наш пакет.

Задачи

- Зачем мы используем пакеты?
- Чем отличаются программные сущности, названные с большой буквы? То есть, чем **Average** отличается от **average**?
- Что такое псевдоним пакета и как его сделать?
- Мы скопировали функцию **Average** из главы 7 в наш новый пакет. Создайте **Min** и **Max** функции для нахождения наименьших и наибольших значений в срезах дробных чисел типа **float64**.
- Напишите документацию к функциям **Min** и **Max** из предыдущей задачи.

Тестирование

Писать программы — не просто. Даже самые лучшие программисты, зачастую, не в состоянии написать программу так, чтобы она работала как положено в любых случаях. Поэтому, важной частью процесса разработки является тестирование. Написание тестов для нашего кода является отличным способом повышения его качества и стабильности.

Go содержит специальную программу, призванную облегчить написание тестов, так что давайте напишем несколько тестов для пакета, который мы создали в предыдущей главе. В папке **chapter11/math** создайте файл под именем **math_test.go**, который будет содержать следующее:

```
package math

import "testing"

func TestAverage(t *testing.T) {
    var v float64
    v = Average([]float64{1,2})
    if v != 1.5 {
        t.Error("Expected 1.5, got ", v)
    }
}
```

Теперь запустим эту команду:

```
go test
```

Вы должны увидеть:

```
$ go test
PASS
ok      golang-book/chapter11/math    0.032s
```

Команда **go test** найдет все тесты для всех файлов в текущей директории и запустит их. Тесты определяются с помощью добавления **Test** к имени функции и принимают один аргумент типа ***testing.T**. В нашем случае, поскольку мы тестируем функцию **Average**, тестирующая функция будет называться **TestAverage**.

После определения тестирующей функции пишется код, который должен использовать тестируемую функцию.

Мы знаем, что среднее от **[1, 2]** будет **1.5**, это и есть то, что мы проверяем.

Возможно, лучшей идеей будет проверить различные комбинации чисел, так что давайте немного изменим тестирующую функцию:

```
package math

import "testing"

type testpair struct {
    values []float64
    average float64
}

var tests = []testpair{
    { []float64{1,2}, 1.5 },
    { []float64{1,1,1,1,1,1}, 1 },
    { []float64{-1,1}, 0 },
}

func TestAverage(t *testing.T) {
    for _, pair := range tests {
        v := Average(pair.values)
        if v != pair.average {
            t.Error(
                "For", pair.values,
                "expected", pair.average,
                "got", v,
            )
        }
    }
}
```

Это очень распространённый способ написания тестов (больше примеров можно найти в исходном коде пакетов, поставляемых с Go). Мы создали **struct**, представляющий входы и выходы для функций. Затем мы создали список из этих структур (пар) и вызвали тестируемую функцию в каждой итерации цикла.

Задачи

- Написать хороший набор тестов не всегда легко, но даже сам процесс их написания, зачастую, может выявить много проблем для первой реализации функции. Например, что произойдет с нашей функцией **Average**, если ей передать пустой список (`[]float64{}`)? Как нужно изменить функцию, чтобы она возвращала **0** в таких случаях?
- Напишите серию тестов для функций **Min** и **Max** из предыдущей главы.

Стандартная библиотека

Вместо того, чтобы каждый раз писать всё с нуля, реальный мир программирования требует от нас умения взаимодействовать с уже существующими библиотеками. В этой главе мы рассмотрим самые часто используемые пакеты, включенные в Go.

Предупреждаю: некоторые библиотеки достаточно очевидны (или были объяснены в предыдущих главах), многие из библиотек, включённых в Go требуют специальных знаний (например: криптография). Объяснение этих технологий выходит за рамки этой книги.

Строки

Go содержит большое количество функций для работы со строками в пакете **string**:

```
package main

import (
    "fmt"
    "strings"
)

func main() {
    fmt.Println(
        // true
        strings.Contains("test", "es"),

        // 2
        strings.Count("test", "t"),

        // true
        strings.HasPrefix("test", "te"),

        // true
        strings.HasSuffix("test", "st"),

        // 1
        strings.Index("test", "e"),

        // "a-b"
        strings.Join([]string{"a", "b"}, "-"),

        // == "aaaaa"
        strings.Repeat("a", 5),

        // "bbaa"
        strings.Replace("aaaa", "a", "b", 2),
    )
}
```

```

// []string{"a","b","c","d","e"}
strings.Split("a-b-c-d-e", "-"),

// "test"
strings.ToLower("TEST"),

// "TEST"
strings.ToUpper("test"),

)
}

```

Иногда нам понадобится работать с бинарными данными. Чтобы преобразовать строку в набор байт (и наоборот), выполните следующие действия:

```

arr := []byte("test")
str := string([]byte{'t','e','s','t'})

```

Ввод / Вывод

Прежде чем мы перейдем к работе с файлами, нужно узнать про пакет **io**. Пакет **io** состоит из нескольких функций, но в основном, это интерфейсы, используемые в других пакетах. Два основных интерфейса — это **Reader** и **Writer**. **Reader** занимается чтением с помощью метода **Read**. **Writer** занимается записью с помощью метода **Write**. Многие функции принимают в качестве аргумента **Reader** или **Writer**. Например, пакет **io** содержит функцию **Copy**, которая копирует данные из **Reader** во **Writer**:

```

func Copy(dst Writer, src Reader) (written int64, err error)

```

Чтобы прочитать или записать **[]byte** или **string**, можно использовать структуру **Buffer** из пакета **bytes**:

```

var buf bytes.Buffer
buf.Write([]byte("test"))

```

Buffer не требует инициализации и поддерживает интерфейсы **Reader** и **Writer**. Вы можете конвертировать его в **[]byte** вызвав **buf.Bytes()**. Если нужно только читать строки, можно так же использовать функцию **strings.NewReader()**, которая более эффективна, чем чтение в буфер.

Файлы и папки

Для открытия файла Go использует функцию **Open** из пакета **os**. Вот пример того, как прочитать файл и вывести его содержимое в консоль:

```
package main

import (
    "fmt"
    "os"
)

func main() {
    file, err := os.Open("test.txt")
    if err != nil {
        // handle the error here
        return
    }
    defer file.Close()

    // get the file size
    stat, err := file.Stat()
    if err != nil {
        return
    }
    // read the file
    bs := make([]byte, stat.Size())
    _, err = file.Read(bs)
    if err != nil {
        return
    }

    str := string(bs)
    fmt.Println(str)
}
```

Мы используем **defer file.Close()** сразу после открытия файла, чтобы быть уверенным, что файл будет закрыт после выполнения функции. Чтение файлов является частым действием, так что вот самый короткий способ сделать это:

```
package main

import (
    "fmt"
    "io/ioutil"
)

func main() {
    bs, err := ioutil.ReadFile("test.txt")
}
```

```
    if err != nil {  
        return  
    }  
    str := string(bs)  
    fmt.Println(str)  
}
```

А вот так мы можем создать файл:

```
package main  
  
import (  
    "os"  
)  
  
func main() {  
    file, err := os.Create("test.txt")  
    if err != nil {  
        // handle the error here  
        return  
    }  
    defer file.Close()  
  
    file.WriteString("test")  
}
```

Чтобы получить содержимое каталога, мы используем тот же **os.Open()**, но передаём ему путь к каталогу вместо имени файла. Затем вызывается функция **Readdir**:

```
package main  
  
import (  
    "fmt"  
    "os"  
)  
  
func main() {  
    dir, err := os.Open(".")  
    if err != nil {  
        return  
    }  
    defer dir.Close()  
  
    fileInfos, err := dir.Readdir(-1)  
    if err != nil {  
        return  
    }  
}
```

```
for _, fi := range fileInfos {  
    fmt.Println(fi.Name())  
}  
}
```

Иногда мы хотим рекурсивно обойти каталоги (прочитать содержимое текущего и всех вложенных каталогов). Это делается просто с помощью функции **Walk**, предоставляемой пакетом **path/filepath**:

```
package main  
  
import (  
    "fmt"  
    "os"  
    "path/filepath"  
)  
  
func main() {  
    filepath.Walk(".", func(path string, info os.FileInfo, err  
error) error {  
        fmt.Println(path)  
        return nil  
    })  
}
```

Функция, передаваемая вторым аргументом, вызывается для каждого файла и каталога в корневом каталоге (в данном случае).

Ошибки

Go имеет встроенный тип для сообщений об ошибках, который мы уже рассматривали (тип **error**). Мы можем создать свои собственные типы сообщений об ошибках используя функцию **New** из пакета **errors**.

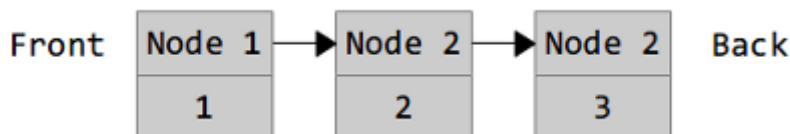
```
package main  
  
import "errors"  
  
func main() {  
    err := errors.New("error message")  
}
```


Контейнеры и сортировки

В дополнение к спискам и картам, Go предоставляет еще несколько видов коллекций, доступных в пакете `container`. В качестве примера рассмотрим `container/list`.

Список

Пакет `container/list` реализует двусвязный список. Структура типа данных связного списка выглядит следующим образом:



Каждый узел списка содержит значение (в нашем случае: 1, 2 или 3) и указатель на следующий узел. Но так как это двусвязный список, узел так же содержит указатель на предыдущий. Такой список может быть создан с помощью следующей программы:

```
package main

import ("fmt" ; "container/list")

func main() {
    var x list.List
    x.PushBack(1)
    x.PushBack(2)
    x.PushBack(3)

    for e := x.Front(); e != nil; e=e.Next() {
        fmt.Println(e.Value.(int))
    }
}
```

Пустым значением `List` (вероятно, опечатка и имелось ввиду `x` — прим. пер.) является пустой список (`*List` создаётся при вызове `list.New`). Значения добавляются в список при помощи `PushBack`. Далее, мы перебираем каждый элемент в списке, получая ссылку на следующий, пока не достигнем `nil`.

Сортировка

Пакет **sort** содержит функции для сортировки произвольных данных. Есть несколько predefined функций (для срезов, целочисленных значений и чисел с плавающей точкой). Вот пример, как отсортировать ваши данные:

```
package main

import ("fmt" ; "sort")

type Person struct {
    Name string
    Age int
}

type ByName []Person

func (this ByName) Len() int {
    return len(this)
}

func (this ByName) Less(i, j int) bool {
    return this[i].Name < this[j].Name
}

func (this ByName) Swap(i, j int) {
    this[i], this[j] = this[j], this[i]
}

func main() {
    kids := []Person{
        {"Jill", 9},
        {"Jack", 10},
    }
    sort.Sort(ByName(kids))
    fmt.Println(kids)
}
```

Хэши и криптография

Функция хэширования принимает набор данных и уменьшает его до фиксированного размера. Хэши используются в программировании повсеместно, начиная от поиска данных, заканчивая быстрым детектированием изменений. Хэш-функции в Go подразделяются на две категории: криптографические и некриптографические.

Некриптографические функции можно найти в пакете **hash**, который включает такие алгоритмы как **adler32**, **crc32**, **crc64** и **fnv**.

Вот пример использования **crc32**:

```
package main

import (
    "fmt"
    "hash/crc32"
)

func main() {
    h := crc32.NewIEEE()
    h.Write([]byte("test"))
    v := h.Sum32()
    fmt.Println(v)
}
```

Объект **crc32** реализует интерфейс **Writer**, так что мы можем просто записать в него набор байт, как и в любой другой **Writer**. После записи мы вызываем **Sum32()**, который вернёт **uint32**. Обычным применением **crc32** является сравнение двух файлов. Если значение **Sum32()** для обоих файлов одинаковы, то, весьма вероятно (не со стопроцентной гарантией), содержимое этих файлов идентично. Если же значения отличаются, значит файлы, безусловно, разные:

```
package main

import (
    "fmt"
    "hash/crc32"
    "io/ioutil"
)

func getHash(filename string) (uint32, error) {
    bs, err := ioutil.ReadFile(filename)
    if err != nil {
        return 0, err
    }
    h := crc32.NewIEEE()
    h.Write(bs)
    return h.Sum32(), nil
}

func main() {
    h1, err := getHash("test1.txt")
    if err != nil {
        return
    }
    h2, err := getHash("test2.txt")
    if err != nil {
```

```

        return
    }
    fmt.Println(h1, h2, h1 == h2)
}

```

Криптографические хэш-функции аналогичны их некриптографическим коллегам, однако у них есть одна особенность: их сложно обратить вспять. Очень сложно определить, что за набор данных содержится в криптографическом хэше, поэтому такие хэши часто используются в системах безопасности.

Одним из криптографических хэш-алгоритмов является SHA-1. Вот как можно его использовать:

```

package main

import (
    "fmt"
    "crypto/sha1"
)

func main() {
    h := sha1.New()
    h.Write([]byte("test"))
    bs := h.Sum([]byte{})
    fmt.Println(bs)
}

```

Этот пример очень похож на пример использования **crc32**, потому что оба они реализуют интерфейс **hash.Hash**. Основное отличие в том, что в то время как **crc32** вычисляет 32-битный хэш, **sha1** вычисляет 160-битный хэш. В Go нет встроенного типа для хранения 160-битного числа, поэтому мы используем вместо него срез размером 20 байт.

Серверы

На Go очень просто создавать сетевые серверы. Сначала давайте взглянем, как создать TCP сервер:

```

package main

import (
    "encoding/gob"
    "fmt"
    "net"
)

func server() {

```

```

// listen on a port
ln, err := net.Listen("tcp", ":9999")
if err != nil {
    fmt.Println(err)
    return
}
for {
    // accept a connection
    c, err := ln.Accept()
    if err != nil {
        fmt.Println(err)
        continue
    }
    // handle the connection
    go handleServerConnection(c)
}
}

func handleServerConnection(c net.Conn) {
    // receive the message
    var msg string
    err := gob.NewDecoder(c).Decode(&msg)
    if err != nil {
        fmt.Println(err)
    } else {
        fmt.Println("Received", msg)
    }

    c.Close()
}

func client() {
    // connect to the server
    c, err := net.Dial("tcp", "127.0.0.1:9999")
    if err != nil {
        fmt.Println(err)
        return
    }

    // send the message
    msg := "Hello World"
    fmt.Println("Sending", msg)
    err = gob.NewEncoder(c).Encode(msg)
    if err != nil {
        fmt.Println(err)
    }

    c.Close()
}

```

```
func main() {
    go server()
    go client()

    var input string
    fmt.Scanln(&input)
}
```

Этот пример использует пакет [encoding/gob](#), который позволяет легко кодировать выходные данные, чтобы другие программы на Go (или конкретно эта программа, в нашем случае) могли их прочитать. Дополнительные способы кодирования доступны в пакете [encoding](#) (например [encoding/json](#)), а также в пакетах сторонних разработчиков (например, можно использовать labix.org/v2/mgo/bson для работы с BSON).

HTTP

HTTP-серверы еще проще в настройке и использовании:

```
package main

import ("net/http" ; "io")

func hello(res http.ResponseWriter, req *http.Request) {
    res.Header().Set(
        "Content-Type",
        "text/html",
    )
    io.WriteString(
        res,
        `<doctype html>
<html>
  <head>
    <title>Hello World</title>
  </head>
  <body>
    Hello World!
  </body>
</html>`,
    )
}

func main() {
    http.HandleFunc("/hello", hello)
    http.ListenAndServe(":9000", nil)
}
```

HandleFunc обрабатывает URL-маршрут (**/hello**) с помощью указанной функции. Мы так же можем обрабатывать статические файлы при помощи **FileServer**:

```
http.Handle(
    "/assets/",
    http.StripPrefix(
        "/assets/",
        http.FileServer(http.Dir("assets")),
    ),
)
```

RPC

Пакеты **net/rpc** (remote procedure call — удаленный вызов процедур) и **net/rpc/jsonrpc** обеспечивают простоту вызова методов по сети (а не только из программы, в которой они используются).

```
package main

import (
    "fmt"
    "net"
    "net/rpc"
)

type Server struct {}
func (this *Server) Negate(i int64, reply *int64) error {
    *reply = -i
    return nil
}

func server() {
    rpc.Register(new(Server))
    ln, err := net.Listen("tcp", ":9999")
    if err != nil {
        fmt.Println(err)
        return
    }
    for {
        c, err := ln.Accept()
        if err != nil {
            continue
        }
        go rpc.ServeConn(c)
    }
}
```

```

func client() {
    c, err := rpc.Dial("tcp", "127.0.0.1:9999")
    if err != nil {
        fmt.Println(err)
        return
    }
    var result int64
    err = c.Call("Server.Negate", int64(999), &result)
    if err != nil {
        fmt.Println(err)
    } else {
        fmt.Println("Server.Negate(999) =", result)
    }
}

func main() {
    go server()
    go client()

    var input string
    fmt.Scanln(&input)
}

```

Эта программа похожа на пример использования TCP-сервера, за исключением того, что теперь мы создали объект, который содержит методы, доступные для вызова, а затем вызвали **Negate** из функции-клиента. Посмотрите документацию по [net/rpc](#) для получения дополнительной информации.

Получение аргументов из командной строки

При вызове команды в консоли, есть возможность передать ей определенные аргументы. Мы видели это на примере вызова команды **go**:

```
go run myfile.go
```

run и **myfile.go** являются аргументами. Мы так же можем передать команде флаги:

```
go run -v myfile.go
```

Пакет **flag** позволяет анализировать аргументы и флаги, переданные нашей программе. Вот пример программы, которая генерирует число от 0 до 6. Но мы можем изменить максимальное значение, передав программе флаг **-max=100**.


```

package main

import ("fmt"; "flag"; "math/rand")

func main() {
    // Define flags
    maxp := flag.Int("max", 6, "the max value")
    // Parse
    flag.Parse()
    // Generate a number between 0 and max
    fmt.Println(rand.Intn(*maxp))
}

```

Любые дополнительные не-флаговые аргументы могут быть получены с помощью **flag.Args()**, которая вернет **[]string**.

Синхронизация примитивов

Предпочтительный способ справиться с параллелизмом и синхронизацией в Go, с помощью горутин и каналов уже описан в главе 10. Однако, Go предоставляет более традиционные способы работать с процедурами в отдельных потоках - в пакетах **sync** и **sync/atomic**.

Мьютексы

Мьютекс (или взаимная блокировка) единовременно блокирует часть кода в одном потоке, а так же используется для защиты общих ресурсов из не-атомарных операций. Вот пример использования мьютекса:

```

package main

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

func main() {
    m := new(sync.Mutex)

    for i := 0; i < 10; i++ {
        go func(i int) {
            m.Lock()
            fmt.Println(i, "start")
            time.Sleep(time.Second)
            fmt.Println(i, "end")
            m.Unlock()
        }(i)
    }
}

```

```
}  
  
var input string  
fmt.Scanln(&input)  
}
```

Когда мьютекс (**m**) заблокирован из одного процесса, любые попытки повторно заблокировать его из других процессов приведут к блокировке самих процессов до тех пор, пока мьютекс не будет разблокирован. Следует проявлять большую осторожность при использовании мьютексов или примитивов синхронизации из пакета **sync/atomic**.

Традиционное многопоточное программирование является достаточно сложным: сделать ошибку просто, а обнаружить её трудно, поскольку она может зависеть от специфичных и редких обстоятельств. Одна из сильных сторон Go в том, что он предоставляет намного более простой и безопасный способ распараллеливания задач, чем потоки и блокировки.

Дальнейшие шаги

Теперь у вас должно быть достаточно знаний, чтобы написать практически любую программу на Go. Но опасно делать выводы о том, что теперь вы стали компетентным программистом. Программирование — это большое мастерство, достаточно простое, если имеются знания. В этой главе я дам вам несколько советов о том, как лучше освоить ремесло программирования.

Учитесь у мастеров

Частью становления хорошего художника или писателя является изучения работ мастеров. Это ничем не отличается от программирования. Один из лучших способов стать квалифицированным программистом — это изучение исходного кода, созданного другими людьми. Go отлично подходит для этой задачи, поскольку исходный код всего проекта находится в свободном доступе.

Например, мы могли бы взглянуть на исходный код библиотеки `io/ioutil` по адресу: <http://golang.org/src/pkg/io/ioutil/ioutil.go>. Читайте код медленно и осознанно. Постарайтесь понять каждую строку и не забывайте про прилагаемые комментарии. Например, в методе `ReadFile` есть комментарий, который гласит:

```
// It's a good but not certain bet that FileInfo
// will tell us exactly how much to read, so
// let's try it but be prepared for the answer
// to be wrong.
```

Этот метод наверняка раньше был проще, чем он есть в данный момент. Это отличный пример того, как программы могут развиваться после тестирования и насколько важно обеспечить комментарием внесённые изменения. Весь исходный код всех пакетов можно найти по адресу: <http://golang.org/src/pkg/>

Делайте что-нибудь

Один из лучших способов оттачивания своих навыков - это практика написания кода. Есть много способов сделать это: вы могли бы поработать над сложными задачками по программированию на таких сайтах, как [Project Euler](#) или попробовать себя в более крупном проекте. Возможно, вы захотите написать веб-сервер или даже написать небольшую игру.

Работайте в команде

Большая часть программных проектов в реальном мире созданы командами программистов. Поэтому умение работать в команде имеет большое значение. Если у вас есть заинтересованный друг или одноклассник — возьмите его и объединитесь в команду для работы над общим проектом. Узнайте, как разделить проект на части и тогда сможете работать над ним в разное время.

Второй вариант заключается в работе над открытым проектом. Найдите какую-нибудь стороннюю библиотеку, напишите новую функциональность (или исправьте ошибки) и отправьте её мейнтейнеру. У Go есть растущее сообщество, которое взаимодействует с помощью [списков рассылки](#).

Это перевод книги [Caleb Doxsey — An introduction to programming in Go](#), распространяемой на условиях [Creative Commons 3.0 Attribution License](#).

Переводчики: Максим Полетаев, Виктор Розаев и другие.

Источник: golang-book.ru

Исходники книги: github.com/zenwalker/golang-book