# Использование языка программирования Go для реализации распределенных алгоритмов

Беляев А. Б.
belyaevab@gmail.com

2018

## Содержание

## 1. Введение

Go – высокоуровневый язык общего назначения. Он разрабатывался Робертом Гризмером, Робом Пайком и Кеном Томпсоном из Google и был анонсирован в 2009 году. В 2012 году состоялся релиз версии Go 1.

Программы Go компилируются в машинный код конкретной архитектуры подобно программам, написанным на C. Также от C Go унаследовал базовые типы данных, комментарии, конструкции управления потоком вычислений, указатели и некоторые детали синтаксиса. От современных высокоуровневых языков Go взял систему пакетов, полноценную работу с функциями как с типом данных, неизменяемые строки в кодировке UTF-8, а также отсутствие необходимости ручного освобождения выделенной памяти. Программы, собранные компилятором Go, включают в себя эффективный, частично параллельный, сборщик мусора, вызывающий паузы в работе основного кода не более чем на 100 (как правило, 10) микросекунд [https://golang.org/doc/go1.9, https://golang.org/doc/go1.8].

В то же время Go старается избегать инструментов, которые могут привести к написанию чрезмерно сложного и ненадежного кода. В частности, в языке отсутствуют неявные преобразования числовых типов, параметры функций по умолчанию, в ООП Go нет конструкторов и деструкторов, концепции наследования, перегрузки операторов (операций) и функций. Вместо наследования предлагается использовать широкие возможности композиции и встраивания. Полиморфизм при этом достигается за счет гибкого механизма проверки пользовательского типа на соответствие интерфейсу, описывающему абстрактный тип данных, без явного объявления об этом соответствии. Также в Go отсутствуют исключительные ситуации, использование которых в условиях асинхронного вызова функций могло приводить к непредсказуемому поведению даже в языках без поддержки многопоточности (например, в JavaScript). Вместо обработки исключений в Go принято вместе с результатом вычисления возвращать статус выполнения функции (булево значение или значение типа `error`), если ее корректное завершение не гарантировано.

Нацеленность Go на минимализм, простоту и стабильность проявляется в его консервативности: изменения в новых версиях касаются в большей степени компилятора, нежели самого языка.

Одной из главных особенностей Go является эффективный и удобный механизм параллелизма, основанный на концепции *взаимодействующих последовательных процессов* (*communicating sequential processes – CSP*), предложенной Хоаром в статье [1] 1978 года, и развитой позднее в книге [2]. В CSP программа представляется как параллельное объединение процессов, не разделяющих общую память и часы. Процессы взаимодействуют и синхронизируются посредством *каналов рандеву* (*rendezvous*, *handshake*), передача сообщений по которым возможна только при готовности другого процесса принять сообщение. CSP служил для формального теоретического описания фундаментальных идей параллельных вычислений. В 80-х

годах прошлого века были первые попытки реализовать предложенную модель в узкоспециализированных языках программирования, не получивших широкого распространения. В разработанном в эпоху многоядерных процессоров языке Go применение описанных идей в сочетании классическими принципами работы с разделяемой памятью [3] дало простой и удобный инструментарий для использования широких возможностей параллелизма, предоставляемых современными компьютерами.

В данном пособии разбирается лишь часть возможностей языка Go, необходимая для создания простых многопоточных программ. Для более подробного знакомства с языком следует обратиться к официальной документации https://golang.org/doc и изданиям, посвященным Go, таким как [4] и [5]. Полное описание стандартной библиотеки Go, а также ссылки на ресурсы с библиотеками сторонних разработчиков доступны по адресу https://golang.org/pkg. Любую дополнительную информацию можно получить на официальном сайте проекта https://golang.org.

## 2. Установка и настройка

Для установки последней версии компилятора Go воспользуйтесь менеджером пакетов вашей операционной системы или скачайте дистрибутив с официального сайта https://golang.org/dl.

После установки следует убедиться, что переменная окружения PATH содержит путь каталога, в который был помещен исполняемый файл `go`.

При установке Go в каталог, отличный от /usr/local/go (c:\Go в Windows), его путь следует сохранить в переменной окружения GOROOT. При использовании стандартных путей GOROOT не требуется.

Единственная переменная окружения, которая должна быть установлена пользователем в обязательном порядке, это GOPATH. Переменная GOPATH содержит путь каталога, в который будут устанавливаться внешние *пакеты* (библиотеки) и относительно которого компилятор будет их искать. Установка внешнего пакета производится командой `go get <пакет>`. Например, команда

```
$ go get golang.org/x/crypto/ssh
```

установит пакет golang.org/x/crypto/ssh в каталог $GOPATH/src/golang.org/x/crypto/ssh.

Для небольших программ, локально использующих общие библиотеки, переменную GOPATH достаточно установить один раз. Для крупных проектов переменная GOPATH, как правило, совпадает с рабочим каталогом проекта. В этом случае при работе над несколькими проектами требуется постоянное переключение переменной GOPATH между путями каталогов проектов. Современные IDE для Go, такие как JetBrains

GoLand (http://www.jetbrains.com/go), позволяют настраивать среду для работы с несколькими GOPATH. Значение GOPATH при этом выбирается автоматически в зависимости от текущего каталога.

## 3. Hello, world!

Напишем первую программу на Go в файле <каталог проекта>/src/helloworld/main.go:

src/helloworld/main.go

```
package main

import "fmt"

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

Для ее компиляции выполним, находясь в <каталоге проекта>, команду:

```
$ go build -o bin/helloworld src/helloworld/main.go
```

В результате будет создан исполняемый файл <каталог проекта>/bin/helloworld:

```
$ ./bin/helloworld
Hello, world!
```

Для быстрого запуска программы можно воспользоваться командой:

```
$ go run src/helloworld/main.go
Hello, world!
```

При этом будет создан временный исполняемый файл.

Обратимся к исходному тексту программы. Код Go организован в виде пакетов, состоящих из одного или нескольких файлов .go в одном каталоге. Каждый файл начинается с объявления *имени пакета*, к которому он относится (в нашем случае – `package main`). В одном каталоге могут содержаться файлы только одного пакета.

Далее следует перечисление имен (точнее, *путей импорта)* используемых в файле пакетов. В нашем случае импортируется пакет `"fmt"`, содержащий функции форматированного вывода. Ниже приведен пример импорта нескольких пакетов:

```
import (
      "bytes"
      "client/api"
      "encoding/json"
      "errors"
      "fmt"
      "github.com/sacOO7/socketcluster-client-go/scclient"
      scclient2 "github.com/sacOO7/socketcluster-client-go-2/scclient"
)
```

Язык Go не определяет смысл понятия *путь импорта*, оставляя это конкретным инструментам. Для компилятора `go` это путь до каталога пакета относительно каталогов $GOPATH/src и $GOROOT/src (<стандартный каталог>/src, если GOROOT не установлена).

Для непосредственного обращения к содержимому пакета в программе используется его имя (из инструкции `package`), которое, *как правило* (для библиотек), совпадает с последней частью пути импорта. В коде выше последние два пакета имеют одинаковое имя. Для исключения коллизий в блоке `import` разрешается давать *псевдонимы* пакетам перед соответствующим путем импорта.

Пакет также является единицей *инкапсуляции* в Go. Вместо введения классов памяти, используемых большинством языков программирования, Go следует простому правилу: *любые объявления уровня пакета (переменные, константы, функции, пользовательские типы) доступны при обращении извне, только если их имена начинаются с прописной буквы*:

```
package example

var PublicVar int
var privateVar int
```

В программе helloworld объявлена функция `main`, которая обращается к функции `Println` пакета `fmt` для вывода сообщения с символом конца строки на конце. Функция `Println` доступна пакету `main` после импорта `fmt`, так как начинается с прописной буквы.

Пакет с названием `main` определяет отдельную программу, то есть исполняемый файл, а не библиотеку. Функция `main` в пакете `main` также имеет особый смысл: программа делает то, что написано в функции `main`.

В заключение рассмотрим некоторые вопросы, связанные с форматированием кода. Как и во многих языках программирования, разделителем операторов в Go является *точка с запятой*. При этом нет необходимости ставить ее в конце каждой строки, так как препроцессор *автоматически заменяет символы конца строки на точки с*

*запятой*, если строку не завершает ряд специальных лексем, таких как *запятая* или *открывающая фигурная скобка*. У такого поведения есть интересные следствия:

- *нельзя переносить фигурную скобку*, открывающую тело функции или блока `if`, `for` и т. п. на новую строку, так как перенос строки будет заменен препроцессором на точку с запятой, что вызовет синтаксическую ошибку;
- в перечислениях, где разделителем является запятая (например, при перечислении аргументов функции), допускается ее наличие после последнего элемента, если перед лексемой, завершающей перечисление, присутствует перенос строки:

```
fmt.Println(
      "Hello,",
      "world!", // <- запятая
)
```

Go занимает жесткую позицию относительно форматирования кода, не допуская появления множества *стилей кодирования*. Программа `gofmt` приводит код в файле к стандартному формату, а команда `go fmt` применяет `gofmt` ко всем файлам в пакете или каталоге.

Другой полезный инструмент – программа `goimports`, приводящая блок `import` файла в соответствие используемым в коде пакетам. `goimports` отсутствует в стандартном дистрибутиве, но ее можно установить с помощью команды:

```
$ go get golang.org/x/tools/cmd/goimports
```

Настройте вашу IDE так, чтобы команды `goimports` и `go fmt` выполнялись после каждого сохранения файла.

# 4. Основы языка

## 4.1. Переменные

Объявление переменной в Go имеет вид:

```
var name type = expression
```

Переменная может быть объявлена на уровне пакета, в теле функции или блока. В объявлении может отсутствовать тип `type` либо инициализирующее выражение `= expression`, но не оба сразу. При отсутствии типа он вычисляется как тип

выражения; при отсутствии выражения переменная инициализируется нулевым значением объявленного типа. В Go не бывает неинициализированных переменных.

Существует конструкция *краткого объявления переменной*:

```
name := expression
```

Ее нельзя использовать на уровне пакета, но в остальных ситуациях она является наиболее употребимой. Кроме того, это единственная форма объявления переменной, разрешенная в блоках инициализации операторов управления потоком вычислений (см. далее).

Ниже показаны все варианты объявления переменных:

```
var a float64 = 0.5
var b = 1                           // var b int = 1
var c string                        // var c string = ""
d := true                           // var d bool = true
var i, j, k int                     // 0, 0, 0
x, y, z := -1, true, "foo"          // int, bool, string
d := false                          // (!) Ошибка: d уже объявлена
```

Из примера видно, что возможны объявление и инициализация сразу нескольких переменных в одном выражении.

В последней строке показано, что повторное объявление переменной в том же блоке запрещено. Для изменения значения (*но не типа*) переменной используется *оператор присваивания* =. В Go разрешено присваивать значение сразу нескольким переменным (*присваивание кортежу*):

```
i, x = x, i // обмен значений переменных
```

Присваивание кортежу можно комбинировать с инициализацией новых переменных:

```
d, b, bar := false, 2, "baz"
```

Приведенная конструкция разрешена, если в левой ее части присутствует хотя бы одна новая переменная.

### 4.2. Фундаментальные типы данных, операции и операторы

В Go используются следующие *фундаментальные типы* данных:

- числовые

```
int int8 int16 int32 int64
uint uint8 uint16 uint32 uint64 uintptr
float32 float64 complex64 complex128
byte rune
```

- строки

```
string
```

- булев тип

```
bool
```

Большинство числовых типов содержат в своем имени количество битов их внутреннего представления. Типы `int` и `uint` могут состоять как из 32, так и из 64 битов – не следует делать никаких предположений относительно их размера. Тип `uintptr` имеет размер, достаточный для представления всех адресов памяти системы. Типы `rune` и `byte` являются *псевдонимами* соответственно типов `int32` и `uint8`. `rune` указывает на то, что значение следует воспринимать как символ Unicode; `byte` подразумевает, что значение является фрагментом неформатированных данных. Работа с этими типами никак не отличается от других числовых.

Список *операций* Go в порядке уменьшения приоритета:

```
// унарные арифметические
+      -      ^

// бинарные арифметические
*      /      %      <<     >>     &      &^
+      -      |      ^

// сравнения
==     !=     <      <=     >      >=

// логические
&&
||
```

Семантика большинства операций такая же, как и в других языках программирования. Исключение составляют унарное *побитовое дополнение* ^ и операция *сброса бита* (*И*

*НЕ*) `&^`. При этом бинарная операция `^` имеет стандартную семантику *побитового исключающего ИЛИ* (*XOR*).

К *целочисленным типам* применимы все арифметические операции и операции сравнения. То же относится к *типам с плавающей точкой* за исключением побитовых операций. Для *комплексных чисел* с плавающей точкой исключены упорядочивающие операции. Разрешена только проверка равенства и неравенства. Над значениями *булевого типа* можно выполнять логические операции, а также проверять равенство и неравенство.

Для *строк* в Go определена операция *конкатенации* `+`, возвращающая *новую* строку, а также допустимы все операции сравнения. Строки *неизменяемы*, поэтому в языке присутствует крайне эффективная операция получения подстроки `s[i:j]`, возвращающая строку, которая использует ту же область памяти, что и исходная строка, и содержит ее *байты* с `i`-го по `j-1`-й. Данную операцию следует применять аккуратно, так как символы используемой в Go кодировки UTF-8 могут содержать *от одного до четырех* байтов.

Все приведенные выше арифметические операции имеют соответствующий *оператор* присваивания, такой как `+=`. К целочисленным переменным применимы операторы *инкремента* `++` и *декремента* `--`. Операторы присваивания, инкремента и декремента в Go *не являются операциями* (не возвращают значения): в Go не используются цепочки присваиваний и арифметические выражения с внутренними инкрементами и декрементами с неочевидной семантикой. Кроме того, операторы инкремента и декремента существуют только в *постфиксной форме*.

### 4.3. Константы

Константа может быть объявлена как на уровне пакета, так и в любом блоке с помощью инструкции:

```
const name type = constExpression
```

*Константное выражение* (`constExpression`) может содержать операции, примененные к другим константам или константным выражениям.

При отсутствии в объявлении типа `type`, будет создана *нетипизированная* константа. Компилятор представляет нетипизированные константы и их арифметику с гораздо большей точностью, чем машинная. Например, константа `math.Pi` хранит значение числа пи с большим числом знаков, чем доступно стандартному типу `float64`.

### 4.4. Пользовательские типы

Объявление `type` определяет новый *именованный тип*:

```
type name baseType
```

Новый тип наследует внутреннее представление *базового типа* (`baseType`) и операции, которые к нему применимы.

В качестве базового типа может выступать любой неименованный или именованный тип. В последнем случае базовый тип используемого именованного типа становится базовым типом нового декларируемого типа. Таким образом, любой именованный тип всегда имеет в качестве базового какой-то неименованный тип.

### 4.5. Приведение типов

В Go запрещены любые бинарные операции над переменными и константами разных типов; также компилятор не осуществляет неявные преобразования типов. *Приведение типов* является обязанностью разработчика программы.

Явное приведение типов `type(expression)` разрешено между числовыми типами (возможно, с потерей информации), а также между именованными типами с одинаковыми или любыми числовыми базовыми типами:

```
var foo int32 = 5
const bar float64 = 0.1

fmt.Println(foo * bar)              // (!) Ошибка: несовпадающие типы

var baz time.Duration = time.Second // type Duration int64

if bar < float64(baz) {             // Приведение типа time.Duration
      fmt.Println("less")           // с базовым int64 к float64
}
```

Описанный запрет не распространяется на нетипизированные константы и числовые литералы, если это не ведет к потере информации:

```
const pi = 3.14

var x float32 = pi
var y complex128 = pi
var z = pi                    // float64
var i int64 = 3
var j = 3                     // int
```

```
var k int32 = 3.14              // (!) Ошибка: потеря информации
```

Обратите внимание, что значения с плавающей точкой без явного указания типа воспринимаются как **float64**, а целочисленные как **int**.

### 4.6. Указатели

*Указатели* содержат адреса переменных в памяти. С их помощью можно косвенно считывать или изменять значения соответствующих переменных, не зная имен последних (если у них вообще есть имена).

Указатели имеют типы вида ***type**, где **type** – любой именованный или неименованный тип. Нулевое значение указателя равно `nil`, независимо от типа. Операция *получения адреса* `&`, примененная к любой переменной или к литералу составного типа (массив, структура), возвращает указатель соответствующего типа на заданную область памяти. Для указателя `p` операция разыменования `*p` позволяет читать и изменять значение, адрес которого содержится в `p`. Примеры работы с указателями:

```
i := 5                              // переменная типа int
p := &i                             // p имеет тип *int и указывает на i
fmt.Println(*p)                     // 5
var q *int                          // новый указатель со значением nil
fmt.Println(q == p, q == nil)       // false, true
q = p
*q++
fmt.Println(i)                      // 6
fmt.Println(q == &i, &i != nil)     // true, true
```

Указатели *одного типа* можно проверять на равенство и неравенство: два указателя равны, если указывают на одну и ту же переменную или оба равны `nil`.

Встроенная функция `new(type)` создает новую переменную типа **type**, не имеющую имени, и возвращает ее адрес. Созданная таким образом переменная ничем не отличается от обычных локальных переменных. В зависимости от дальнейшего использования компилятор может разместить ее как в куче, так и в стеке. Приведенные ниже функции имеют идентичное поведение:

```
func newInt1() *int {
      return new(int)
}

func newInt2() *int {
      var dummy int
```

```
        return &dummy
}
```

В Go нет *адресной арифметики*. Пакет `unsafe` предоставляет инструменты для работы с указателями неопределенного типа и смещениями, но использовать его без крайней необходимости и полного понимания не рекомендуется.

### 4.7. Области видимости и время жизни переменных

Как и в других языках программирования область видимости локальной переменной в Go простирается от места объявления до конца лексического блока, в котором переменная была объявлена. Во вложенных конструкциях переменная может *перекрываться* новыми объявлениями с тем же именем. Область видимости переменных, объявленных на уровне пакета – весь пакет (до и после места ее объявления). Те же правила относятся к функциям и типам. Кроме того, объявленные на уровне пакета имена, начинающиеся с прописной буквы, видны внешним пакетам.

Время жизни переменных уровня пакета равно времени работы всей программы. Время жизни локальных переменных определяется наличием “путей” из указателей и ссылок до занимаемой ею памяти из активных в данный момент функций. При отсутствии таких путей память, занимаемая переменной, освобождается сборщиком мусора. Например, в последнем примере переменная `dummy` не уничтожается после завершения функции, так как возвращаемый функцией указатель на нее продолжает существовать.

В связи с этим область памяти (куча/стек), где будет расположена переменная, не зависит от места и способа создания переменной. Компилятор принимает решение о месте размещения на основании последующего использования переменной в программе.

Как было сказано выше, очищением неиспользуемой памяти занимается сборщик мусора, освобождая разработчика от необходимости следить за цепочками ссылок и указателей. В то же время сборщик мусора никак не касается других ресурсов системы: открытых файлов и сетевых подключений, параллельных потоков, с которыми потеряна связь – контроль за всем этим остается обязанностью автора программы.

### 4.8. Массивы и срезы

*Массивы – составные типы* данных, представляющие собой последовательности фиксированной длины из нуля и более элементов определенного типа. В качестве типа элементов могут выступать любые типы, в том числе другие массивы.

Объявления массива:

```
var foo [3]int
bar := [3]rune{'a', 'b', 'c'}              // литерал массива
baz := [...]string{"one", "two", "three"}  // литерал массива

fmt.Printf("%T %T %T\n", foo, bar, baz) // [3]int [3]int32 [3]string
fmt.Printf("%v %v %v\n", foo, bar, baz) // [0 0 0] [97 98 99] [one two
three]
```

Тип массива представляет собой значение длины в квадратных скобках, за которым следует тип элементов. В качестве длины могут быть использованы целочисленные литералы и константы.

В примере выше показано, как создать массив при помощи литерала. *Литерал* любого *составного типа* в Go представляет собой конструкцию:

```
type{/* содержимое литерала, соответствующее type */}
```

В литерале массива разрешено использовать троеточие `...` вместо длины. Компилятор определит длину по количеству элементов в литерале.

Для доступа к конкретному элементу массива используется операция индексирования:

```
fmt.Printf("%c %c %c\n", bar[0], bar[1], bar[2]) // a b c
```

Индексы массива `a` могут принимать значения от `0` до `len(a)-1`, где `len(a)` – встроенная функция, возвращающая длину массива. При попытке обратиться к индексу за пределами допустимых значений возникает аварийная ситуация.

Массивы являются *сравниваемыми* (разрешены операции `==` и `!=`), если сравниваемыми являются элементы этих массивов.

При присваивании массива переменной того же типа, при передаче его как параметра функции и возвращении как результата происходит поэлементное копирование исходного массива.

Из-за фиксированной длины массивы редко используются в Go напрямую. *Срезы*, которые могут изменять свой размер, являются более гибким инструментом.

Срезы являются *ссылочными типами*, легковесными структурами данных, обеспечивающими доступ к части элементов *базового массива*.

Структура среза:

- *указатель* на первый элемент среза в массиве;
- *емкость* – количество элементов с первого элемента среза до последнего элемента массива;
- *длина* – количество элементов, доступное через срез (не превышает емкости).

Тип среза имеет вид `[]type`, где `type` – тип элементов среза. Нулевым значением среза является `nil`:

```
var slice []string // nil
```

Длина и емкость среза могут быть получены с помощью встроенных функций соответственно `len(slice)` и `cap(slice)`. Новый срез может быть создан при помощи операции среза `s[i:j]`, где $0 \le i \le j \le cap(s)$. Примененная к массиву или другому срезу, данная операция возвращает новый срез, содержащий с `i`-го по `j-1`-й элементы исходной последовательности `s`:

```
slice = baz[1:3]
fmt.Printf("%v %v %v\n", slice, slice[:1], slice[:]) // [two three]
[two] [two three]
```

При отсутствии `i` используется значение `0`, при отсутствии `j` – `len(s)`. При выходе указанных границ за пределы `0` и `cap(s)` возникнет аварийная ситуация.

Срез может быть создан при помощи литерала среза (например, `[]int{1, 2, 3}`), а также возвращен встроенной функцией `make`:

```
make([]type, len)
make([]type, len, cap)
```

В обоих случаях сначала создается базовый массив, к которому применяется операция среза. При использовании второго варианта `make` длина базового массива может превосходить длину среза, предоставляя место для расширения.

Встроенная функция `append` добавляет элементы в срез:

```
slice = append(slice, "four")
slice = append(slice, "five", "six")

fmt.Println(slice) // [two three four five six]
```

Если длина базового массива старого среза недостаточна для добавления новых элементов, будет создан новый массив, длина которого будет составлять удвоенную длину старого среза. Элементы старого среза будут скопированы в новый массив. Далее будет возвращен срез, начинающийся нулевым элементом нового массива и содержащий все старые и новые элементы:

```
fmt.Println(len(slice), cap(slice)) // 5 8
```

Из такого поведения следуют две рекомендации:

- во избежание потери данных результат функции `append` должен быть присвоен переменной, к которой `append` была применена;
- когда заранее известно число элементов, которые будут добавлены в срез, использование второго варианта функции `make` с достаточным значением емкости `cap` исключит затраты на повторное создание массива и копирование.

Допустимо применение функции `append` к нулевому срезу (`nil`).

Срезы *не являются сравниваемыми*. Единственная разрешенная операция – сравнение с `nil`.

При присваивании, передаче среза в функцию и возвращении копируются только его указатель на начало данных, емкость и длина. Никаких операций с базовым массивом при этом не производится.

Байтовые срезы и строки могут быть преобразованы друг в друга.

```
bytes := []byte("string")
str := string(bytes)
fmt.Println(bytes, str) // [115 116 114 105 110 103] string
```

При этом происходит побайтовое копирование данных.

### 4.9. Структуры

Для объединения разнородных данных используются *структуры*. Структура определяет составной тип, который, как правило, сразу используется в качестве базового для нового именованного типа. В примере ниже функция **`NewCharacter`** создает новый экземпляр структурного типа `Character` с помощью *структурного литерала* и возвращает указатель на него. Далее показаны примеры работы с полями структуры через указатель `pHomer` и непосредственно (переменная `homer`).

src/homer/main.go

```go
package main

import "fmt"

var count int

type Character struct {
	id       int
	Name     string
	Children []string
}

func NewCharacter(name string) *Character {
	count++
	return &Character{id: count, Name: name}
}

func main() {
	// Создаем указатель на структуру
	pHomer := NewCharacter("Homer Simpson")

	// Работаем с указателем как с структурой
	pHomer.Children = append(pHomer.Children, "Bartholomew", "Lisa")

	// Копируем данные в новую структуру того же типа
	var homer Character
	homer = *pHomer
	homer.Children = append(homer.Children, "Margaret")

	fmt.Println(homer) // {1 Homer Simpson [Bartholomew Lisa
Margaret]}
}
```

Для полей структуры действует то же правило инкапсуляции, что и для имен уровня пакета: при использовании структуры *вне пакета*, где она определена, разрешен доступ только к тем полям, имя которых начинается с *прописной буквы*. Внутри пакета, в котором определена структура, доступны все ее поля.

Тип, описываемый структурой, является *сравниваемым* (==, !=), если сравниваемы все его элементы. По умолчанию элементы структуры инициализируются нулевыми значениями своих типов.

При присваивании и передаче в функцию (или возврате) происходит поэлементное копирование структуры.

Отдельного внимания заслуживают тип *пустой структуры* `struct{}` и *литерал пустой структуры* `struct{}{}`. Пустые структуры имеют нулевой размер и используются, когда разработчику важен факт наличия информации, а не ее значение, например, при передаче сигналов через каналы.

### 4.10. Функции

Объявление функции состоит из имени, списка параметров (может быть пустым), необязательного списка результатов и тела в фигурных скобках:

```
func name(/*parameters*/) (/*results*/) {
      /*body*/
}
```

В списке параметров `parameters` через запятую перечисляются их имена и типы. Для последовательности однотипных параметров допускается указание их типа один раз в конце последовательности. В Go *отсутствуют* параметры функций *по умолчанию*. Список результатов `results` может содержать как типы результатов, так и их имена. В последнем случае имена являются локальными переменными функции. В случае, когда список результатов состоит только из типа единственного возвращаемого значения, скобки, окружающие список, не нужны.

Примеры объявлений функций:

```
func add(x int, y int) int { return x + y }
func sub(x, y int) (z int) { z = x - y; return }
```

Возможность возвращать несколько значений часто используется для сообщения вызывающей функции о наличии ошибок:

```
func div(x, y int) (int, error) {
      if y == 0 {
            return 0, errors.New("division by zero")
      }

      return x / y, nil
}
```

Результаты такой функции присваиваются кортежу:

```
foo, err := div(bar, baz)
if err != nil {
```

```
	log.Fatal(err) // вывод ошибки и завершение программы
}
```

Функции в Go являются полноценными *значениями*, имеющими *тип*, равный их сигнатуре без имен параметров и имени самой функции. Функции можно присваивать переменным, передавать как аргументы другим функциям и возвращать как результаты:

```
fmt.Printf("%T\n", add) // func(int, int) int
fmt.Printf("%T\n", sub) // func(int, int) int
fmt.Printf("%T\n", div) // func(int, int) (int, error)

var sum func(int, int) int
sum = add
fmt.Println(sum(5, 7)) // 12
```

Новое значение функционального типа может быть создано в любом месте программы при помощи *литерала функции* (тип-сигнатура, за которым следует тело в фигурных скобках):

```
var prod = func(x, y int) int { return x * y }
```

Созданная таким образом функция называется *анонимной*. Анонимные функции могут быть присвоены переменным, как в примере выше, переданы как аргументы другим функциям, возвращены как результаты, а также непосредственно исполнены:

```
fmt.Println(prod(2, 3))                                  // 6
fmt.Println(func(x, y int) int { return x * y }(2, 3)) // 6
```

Возможности анонимных функций существенно расширяются применяемой к ним технологией *замыкания*. Замыкания расширяют область видимости функции переменными из внешних областей видимости, к которым функция обращается:

src/counter/main.go

```
package main

import "fmt"

func getCounter() func() int {
	var c int
	return func() int {
		c++
```

```
            return c
      }
}

func main() {
      counter := getCounter()
      fmt.Println(counter()) // 1
      fmt.Println(counter()) // 2
      fmt.Println(counter()) // 3
      fmt.Println(counter()) // 4
}
```

В приведенном примере функция **getCounter** возвращает анонимную функцию-значение, которая захватывает переменную `c` из родительской области видимости, продлевая время ее жизни. Полученная таким образом функция не только является кодом для многократного использования, но и имеет собственное состояние.

Многие функции содержат несколько точек выхода (**return**). В случаях, когда определенное действие должно быть выполнено в такой точке (закрытие файлового дескриптора, разблокировка мьютекса), может быть применена инструкция *отложенного вызова* функции **defer**. При использовании **defer** *операнды* вызываемой функции вычисляются ***в момент исполнения*** инструкции, а запуск функции откладывается до момента выхода из родительской функции. В следующем примере показано вычисление времени работы функции `job` с применением **defer**:

```
func job() (dur time.Duration) {
      defer func(since time.Time) {
            dur = time.Since(since)
      }(time.Now())

      // do something:
      time.Sleep(10 * time.Second)

      return
}

func main() {
      dur := job()
      fmt.Println(dur) // 10.002967083s
}
```

Отложенная функция выполняется *после* того, как инструкция return сформировала выходное значение dur, и может его изменить как обычную локальную переменную, попавшую в замыкание.

Нулевым значением функциональных типов является `nil`, сравнение с которым разрешено, другие операции сравнения запрещены.

### 4.11. Операторы управления потоком вычислений

Go предоставляет четыре оператора управления потоком: **`if`**, **`switch`**, **`for`** и **`select`**. Последний будет разобран в разделах, посвященных каналам и параллелизму.

Оператор *ветвления* **`if`** имеет вид:

```
if init; condition {
      /* body */
} else {
      /* elseBody */
}
```

Блок `condition` – выражение булева типа; необязательный блок `init` может содержать присваивание или краткое объявление переменной (`:=`). При объявлении переменная или переменные становятся локальными для тел оператора. **`else`** с телом может отсутствовать.

Оператор *выбора* **`switch`**:

```
switch init; expression {
case caseExpression/*, caseExpression, ...*/:
      /* caseBody */
case caseExpression/*, caseExpression, ...*/:
      /* caseBody */
/* ... */
default:
      /* defaultBody */
}
```

Необязательный блок `init` – присваивание или краткое объявление переменной. Блоки `expression` и `caseExpression` – выражения любого *сравниваемого* типа. Оператор выполняет тело *первого* варианта, чье выражение `caseExpression` равно `expression`. По окончании выполнения тела управление переходит к инструкции, следующей за **`switch`** – “проваливания” к следующему телу, как, например, в языке C, не происходит. В случае отсутствия совпадений `expression` и `caseExpression` выполняется тело **`default`**. При отсутствии **`default`** – выход из блока.

Оператор **`switch`** без операнда воспринимается как **`switch`** `true` и может использоваться в качестве множественного ветвления.

Оператор *цикла* `for` существует в следующих вариантах:

```
/* стандартный цикл for */
for init; condition; postAction {
      /* body */
}

/* цикл с условием */
for condition {
      /* body */
}

/* бесконечный цикл */
for {
      /* body */
}
```

Операторы **`break`** и **`continue`** в теле цикла имеют стандартную семантику.

Для типов последовательностей (массивов, срезов) и строк существуют специальные варианты цикла **`for`**:

```
for range sequence {
      /* body */
}

for index := range sequence {
      /* body */
}

for index, value := range sequence {
      /* body */
}
```

При итерировании строк перебираются *символы*, а не байты: значения `value` имеют тип **`rune`**, а число итераций равно количеству символов (не длине строки).

Go не позволяет объявлять переменную без последующего ее использования. Поэтому при отсутствии необходимости в индексе его значение должно быть присвоено специальному *пустому идентификатору* _ (символ подчеркивания):

```
for _, value := range sequence {
      /* body */
```

```
}
```

Обратите внимание, что все перечисленные операторы не содержат скобок в блоках инициализации и условия.

### 4.12. Пример: быстрая сортировка

В качестве примера применения описанных инструментов реализуем алгоритм быстрой сортировки, разработанный Хоаром в 1959 году и опубликованный в [6]. На этой же задаче далее будут рассмотрены основные возможности параллелизма в Go.

Приведенная ниже программа осуществляет побайтовую сортировку файла, имя которого передается как аргумент командной строки. Результат сохраняется в файле с суффиксом `".sorted.txt"`, в терминале отображается длительность сортировки.

src/quick/main.go

```
package main

import (
	"fmt"
	"io/ioutil"
	"log"
	"os"
	"time"
)

func quicksort(a []byte) {
	if len(a) <= 1 {
		return
	}

	i := partition(a)
	quicksort(a[:i]) // from a[0] to a[i - 1]
	quicksort(a[i:]) // from a[i] to a[len(a) - 1]
}

func partition(a []byte) int {
	i := 0
	j := len(a) - 1
	v := a[(j+1)/2]

	for i <= j {
		for a[i] < v {
```

```
                i++
            }
            for a[j] > v {
                j--
            }
            if i >= j {
                break
            }

            a[i], a[j] = a[j], a[i]

            i++
            j--
        }

        return i
}

func main() {
        bytes, err := ioutil.ReadFile(os.Args[1])
        if err != nil {
                log.Fatal(err)
        }

        start := time.Now()
        quicksort(bytes)
        duration := time.Since(start)

        fmt.Printf("duration: %s\n", duration)

        if err := ioutil.WriteFile(os.Args[1]+".sorted.txt", bytes, 0666);
err != nil {
                log.Fatal(err)
        }
}
```

Аргументы командной строки содержатся в срезе `Args` пакета `os`; чтение и запись файлов осуществляют функции пакета `ioutil`; для работы со временем используются функции пакета `time` и тип `time.Duration`, который при выводе в формате строки (`%s`) предстает в удобном для чтения виде. Функция `log.Fatal` выводит текст ошибки в стандартный поток и экстренно завершает программу.

`ioutil.ReadFile` возвращает содержимое файла в виде байтового среза. Его сортировка осуществляется функцией **`quicksort`**, которая вызывает функцию

**partition**, разделяющую срез на две части, а затем рекурсивно применяет себя к каждой из этих частей.

Как и требуется от любой реализации данного алгоритма, сортировка не использует дополнительную память помимо выделенной под исходный массив.

Запустим программу для сортировки текста романа Германа Мелвилла “Моби Дик, или Белый кит”, повторенного 256 раз:

```
$ cat Melville\ Herman.\ Moby\ Dick.txt{,}{,}{,}{,}{,}{,}{,}{,} >
tmp/MD256.txt
$ ./bin/quick tmp/MD256.txt
duration: 13.746921119s
```

На компьютере 2,7 GHz Intel Core i5 8 ГБ 1867 MHz DDR3 сортировка заняла 13.75 секунд. В следующих примерах постараемся улучшить этот результат.

## 5. Параллелизм в Go

Одна из самых сильных сторон Go – поддержка параллелизма “из коробки”. Go предоставляет разработчику два стиля параллельного программирования. Во-первых, язык наследует идеи CSP [1, 2]: независимые *go-подпрограммы* (*goroutines* – *горутины*) взаимодействуют через *каналы*. При этом каждая горутина использует по большей части собственные переменные. Во-вторых, горутины могут быть использованы в рамках более традиционной модели многопоточности с общей памятью [3]. Стандартная библиотека Go предоставляет все необходимые примитивы синхронизации. На практике разработчики совмещают обозначенные стили.

### 5.1. Горутины

Горутина – это независимо выполняемая последовательность инструкций в программе, написанной на Go. При запуске программы ее единственная горутина вызывает функцию **main** (*главная горутина*). Другие горутины порождаются вызовом функций инструкцией **go**:

```
foo()       // синхронный вызов foo() с ожиданием ответа
go foo()    // создание новой горутины и немедленный переход к следующей
инструкции
```

Данные, доступные горутине в программе, определяются обычными правилами для областей видимости функций и замыканий.

Не следует отождествлять горутины с потоками операционной системы. Каждая программа, собранная компилятором Go, включает в себя планировщик, распределяющий *m* горутин программы между *n* потоками операционной системы (*m:n-планирование*). На современных многоядерных процессорах мы можем ожидать *параллельного* выполнения горутин.

Вызовы планировщика Go осуществляются неявно некоторым конструкциями языка и не являются периодическими. Процедура назначения горутины потоку является крайне дешевой по сравнению с *переключением контекста* [3] операционной системой, когда поток назначается ядру процессора. Поэтому при *n*, не превосходящем доступного числа ядер, порождение горутин в количестве *m*, значительно превышающем `n`, является сравнительно безопасным для производительности.

Количество доступных потоков *n* в *m:n-планировании* задается параметром *GOMAXPROCS* планировщика, которое может быть установлено функцией `runtime.GOMAXPROCS` или с помощью переменной окружения GOMAXPROCS. Вызов функции `runtime.GOMAXPROCS` с аргументом, меньшим единицы, возвращает текущее значение:

```
fmt.Printf("GOMAXPROCS: %d\n", runtime.GOMAXPROCS(0)) // GOMAXPROCS: 4
fmt.Printf("NumCPU:     %d\n", runtime.NumCPU())      // NumCPU:     4
```

В последних версиях Go значение GOMAXPROCS по умолчанию равно количеству ядер процессора (функция `runtime.NumCPU`). Некоторые процессоры сообщают операционной системе о большем количестве ядер, чем они физически содержат (*hyper-threading*). Разумеется, Go никак не отличает такие виртуальные ядра от реальных.

Возможностью изменения GOMAXPROCS следует пользоваться с осторожностью: при отсутствии контроля за числом порождаемых горутин увеличение GOMAXPROCS сверх числа доступных ядер поставит выполнение программы в зависимость от переключений контекста операционной системой, что может сказаться на скорости работы.

В качестве примера использования горутин приведем программу из книги [4], вычисляющую 45-е число Фибоначчи. Неэффективный алгоритм обуславливает значительное время вычисления, в процессе которого пользователь может отвлечься просмотром анимации в виде вращающегося отрезка. Анимация выводится отдельной независимой горутиной.

```
// Copyright (c) 2016 Alan A. A. Donovan & Brian W. Kernighan.
// License: https://creativecommons.org/licenses/by-nc-sa/4.0/

// See page 218.
```

```
// Spinner displays an animation while computing the 45th Fibonacci
number.
package main

import (
	"fmt"
	"time"
)

func main() {
	go spinner(100 * time.Millisecond)
	const n = 45
	fibN := fib(n) // slow
	fmt.Printf("\rFibonacci(%d) = %d\n", n, fibN)
}

func spinner(delay time.Duration) {
	for {
		for _, r := range `-\|/` {
			fmt.Printf("\r%c", r)
			time.Sleep(delay)
		}
	}
}

func fib(x int) int {
	if x < 2 {
		return x
	}
	return fib(x-1) + fib(x-2)
}
```

По окончании вычислений функция **main** завершается, что влечет *немедленное прекращение работы* всех горутин. Не считая выхода из функции **main**, нет никакого программного способа остановить из одной горутины другую. При этом горутина самостоятельно прекращает свою работу по завершении своей подпрограммы (функции, вызванной инструкцией **go**).

### 5.2. Каналы

Каналы – ссылочные типы, экземпляры которых используются для взаимодействия горутин. Тип канала записывается как **chan type**, где **type** – тип элементов канала. Нулевым значением канала является **nil**. Экземпляры каналов одного типа равны, если ссылаются на одну и ту же структуру данных канала.

Для создания канала используется встроенная функция `make`:

```
foo := make(chan int)    // make(chan int, 0) - небуферизованный канал
bar := make(chan int, 3) // канал с буфером (очередью) емкостью 3

var baz chan int
baz = bar

fmt.Println(foo == bar, baz == bar) // false true
```

Операции *отправления* и *получения* значений:

```
baz <- 1
baz <- 2
baz <- 3

one := <-baz

fmt.Println(one, <-bar, <-bar) // 1 2 3
```

Операции *коммуникации* через каналы *потокобезопасны (thread-safe)*:

- если горутина начала операцию отправления или получения значения через *буферизованный* канал, другие горутины, пытающиеся осуществить коммуникацию по тому же каналу, будут *заблокированы* до завершения операции первой горутиной;
- операции коммуникации по *небуферизованному* каналу *синхронизируют* две горутины: операция получения будет выполнена строго *после* всех операций, предшествующих отправлению в отправляющей горутине, но *до* завершения самой операции отправления (см. документ “The Go Memory Model”, https://golang.omycat.info/ref/mem).

Оператор *мультиплексирования* `select` случайным образом выбирает *одну* из незаблокированных в данный момент операций коммуникации через каналы и выполняет ее, а следом – соответствующее тело:

```
select {
case <-ch1:
      /* caseBody */
case x := <-ch2:
      /* caseBody */
case ch3 <- y:
      /* caseBody */
```

```
default:
      /* defaultBody */
}
```

Если все операции заблокированы, выполняется тело `default`. При отсутствии `default` выполнение горутины останавливается до разблокировки одной из операций.

Встроенная функция `close` *закрывает* канал. При этом любая попытка отправки значения вызывает аварийную ситуацию, в то время как операция получения *всегда доступна* и возвращает нулевое значение соответствующего типа. Существует *специальный синтаксис* для контроля статуса канала:

```
bar <- 4
four, ok := <-bar
fmt.Println(four, ok) // 4 true

close(bar)

zero, ok := <-bar
fmt.Println(zero, ok) // 0 false
```

При рассмотрении программы, вычисляющей число Фибоначчи, из предыдущего раздела можно допустить ситуацию, при которой отображение результата вычислений будет испорчено горутиной с анимацией, успевшей вывести очередной символ отрезка. Для исключения такого варианта перед выводом результата сообщим горутине `spinner` о необходимости завершения через небуферизованный канал:

```
func main() {
      stopSpinner := make(chan struct{})
      go spinner(100*time.Millisecond, stopSpinner)
      const n = 45
      fibN := fib(n) // slow
      stopSpinner <- struct{}{}
      fmt.Printf("\rFibonacci(%d) = %d\n", n, fibN)
}

func spinner(delay time.Duration, stop chan struct{}) {
      for {
            for _, r := range `-\|/` {
                  select {
                  case <-stop:
                        return
                  default:
                  }
```

```
                fmt.Printf("\r%c", r)
                time.Sleep(delay)
        }
    }
}
```

Главная горутина не выводит результат до тех пор, пока сигнал `struct{}{}` (литерал пустой структуры) не оказывается принятым горутиной **spinner**.

### 5.3. Пример: параллельная быстрая сортировка

В данном разделе приведен пример наивного метода распараллеливания быстрой сортировки. Суть его заключается в том, что после разбиения среза каждая из его частей продолжает сортироваться в отдельной горутине. Очевидно, что с определенной длины среза суммарные затраты времени на порождение новых горутин превысят время однопоточной рекурсивной сортировки. Поэтому срезы, состоящие из 10000 и менее элементов, будут сортироваться одной горутиной. Со значением порога можно экспериментировать для достижения наилучших результатов, но для наших примеров это не играет значительной роли. Изменения коснулись только функции **quicksort**:

```
func quicksort(a []byte) {
    const THRESHOLD = 10000

    var wg sync.WaitGroup

    var psort func([]byte)
    psort = func(a []byte) {
        if len(a) <= 1 {
            return
        }

        if len(a) <= THRESHOLD {
            i := partition(a)
            psort(a[:i]) // from a[0] to a[i - 1]
            psort(a[i:]) // from a[i] to a[len(a) - 1]
            return
        }

        wg.Add(1)
        go func() {
            i := partition(a)
            psort(a[:i]) // from a[0] to a[i - 1]
```

```
                psort(a[i:]) // from a[i] to a[len(a) - 1]
                wg.Done()
            }()
        }

        psort(a)
        wg.Wait()
}
```

Непосредственно сортировка осуществляется функцией `psort`, которая создается в теле **`quicksort`** присваиванием литерала функции соответствующей переменной. Обратите внимание, что переменная `psort` объявлена объявлена *до* присваивания литерала. Это необходимо, чтобы она могла быть захвачена замыканием литерала для внутренних рекурсивных вызовов.

Получившаяся функция для срезов `a` с `len(a) <= THRESHOLD` выполняет обычную рекурсивную сортировку, а для больших срезов порождает новую горутину, выполняющую анонимную сортирующую функцию.

Нельзя допустить выхода из функции **`quicksort`** до полного завершения сортировок всеми запущенными горутинами. Для контроля выполненных работ используется переменная `wg` типа `WaitGroup` из пакета `sync`. `WaitGroup` – это *потокобезопасный счетчик*, значение которого увеличивается *методом* `Add` и уменьшается на единицу методом `Done`. Метод `Wait` блокирует дальнейшее выполнение горутины до обнуления счетчика. (В настоящем пособии не рассматриваются методы типов и другие возможности ООП Go. Для знакомства с ними рекомендуется обратиться к изданиям [4] и [5].)

Переменная `wg` и константа `THRESHOLD` попадают в функцию `psort` и сортирующие горутины посредством механизма замыкания.

Видим ускорение при запуске программы:

```
$ ./bin/pquick tmp/MD256.txt
duration: 6.495119705s
```

Описанному алгоритму присущ взрывной параллелизм: число порождаемых горутин значительно превышает доступное количество ядер процессора. Это практически не сказывается на скорости работы благодаря *m:n-планированию* (см. 5.1). Но если разрешить планировщику использовать большее количество потоков операционной системы, станет заметным эффект от переключений контекста:

```
$ env GOMAXPROCS=256 ./bin/pquick tmp/MD256.txt
duration: 7.022081731s
```

Для повышения устойчивости программы к такому увеличению GOMAXPROCS необходимо алгоритмическое ограничение используемых ею ресурсов.

### 5.4. Семафоры и мьютексы

Традиционно для ограничения использования общих ресурсов используются *семафоры*, предложенные Дейкстрой в начале шестидесятых годов прошлого века [3]. Семафор представляет собой счетчик доступных ресурсов, который может быть уменьшен *атомарной* операцией *P*. При отсутствии свободных ресурсов процесс, вызывающий *P*, блокируется до их появления. Освобождение ресурса осуществляется *атомарной* операцией *V*.

В Go в качестве семафоров могут выступать каналы с величиной буфера, равной количеству ресурсов:

```
sema := make(chan struct{}, 4)      // семафор на 4 ресурса
sema <- struct{}{}                  // P
<-sema                              // V
```

*Мьютексы* – двоичные семафоры, ограничивающие доступ к единственному ресурсу. Для мьютексов в Go предусмотрен специальный тип `sync.Mutex`:

```
var mu sync.Mutex // нулевой (незаблокированный) мьютекс

mu.Lock()   // блокировка
// критическая секция...
mu.Unlock() // разблокировка
```

Другой вариант использования:

```
func () {
      mu.Lock() // блокировка
      defer mu.Unlock() // отложенная разблокировка

      // критическая секция...
}()
```

Если в качестве ресурса выступают общие данные, может быть организован параллельный доступ к ним для *чтения* при исключительном доступе для *записи*. Для этих целей используется мьютекс `sync.RWMutex`. Его методы `RLock` и `RUnlock` обеспечивают разделяемую блокировку ресурса. Метод исключительной блокировки `Lock` срабатывает только на полностью свободном мьютексе. При вызове метода `Lock`

мьютекса, уже захваченного с помощью `RLock` другой горутиной, новые попытки вызвать `RLock` будут заблокированы до начала и завершения критической секции, вызванной методом `Lock`. Из этого следует запрет на использование рекурсивной разделяемой блокировки в одной горутине.

В рамках одной горутины все операции изменения переменных последовательно согласованы, но *нет никакой гарантии*, что изменения общих переменных будут видны разным горутинам в одном и том же порядке при отсутствии явной синхронизации. Это связано с использованием процессорами собственного кэша и порядком его сброса в оперативную память. Операции синхронизации через каналы и мьютексы заставляют процессор фиксировать накопленные записи в оперативной памяти. Поэтому при чтении общих данных, меняющих свое значение в ходе работы программы, необходимо использовать `sync.RWMutex`.

### 5.5. Ограничение параллелизма посредством семафоров на примере быстрой сортировки

Ограничим число порождаемых горутин в программе параллельной быстрой сортировки количеством доступных ядер процессора. Для этого воспользуемся семафором `sema := make(chan struct{}, runtime.NumCPU())` в несколько нетрадиционном виде:

```
func quicksort(a []byte) {
	const THRESHOLD = 10000

	var wg sync.WaitGroup
	sema := make(chan struct{}, runtime.NumCPU())

	var psort func([]byte)
	psort = func(a []byte) {
		if len(a) <= 1 {
			return
		}

		if len(a) <= THRESHOLD {
			i := partition(a)
			psort(a[:i]) // from a[0] to a[i - 1]
			psort(a[i:]) // from a[i] to a[len(a) - 1]
			return
		}

		select {
		case sema <- struct{}{}:
			wg.Add(1)
```

```
                go func() {
                        i := partition(a)
                        psort(a[:i]) // from a[0] to a[i - 1]
                        psort(a[i:]) // from a[i] to a[len(a) - 1]
                        <-sema
                        wg.Done()
                }()
        default:
                i := partition(a)
                psort(a[:i]) // from a[0] to a[i - 1]
                psort(a[i:]) // from a[i] to a[len(a) - 1]
        }
    }

    psort(a)
    wg.Wait()
}
```

Тот факт, что семафор реализован посредством канала, дает нам возможность воспользоваться оператором `select` для проверки состояния семафора без блокировки.

При отсутствии свободных ресурсов горутина продолжает сортировку без порождения новых горутин. Такое поведение уменьшает параллелизм, так как освободившееся ядро процессора не получит работу, пока одна из горутин, попавшая в блок `default` не выполнит свои последовательные инструкции. Вместе с затратами на синхронизацию при выполнении потокобезопасных операций с каналом это несколько снижает скорость программы. В то же время программа устойчива к увеличению GOMAXPROCS:

```
$ ./bin/pquick2 tmp/MD256.txt
duration: 6.619105413s
$ env GOMAXPROCS=256 ./bin/pquick2 tmp/MD256.txt
duration: 6.620002662s
```

### 5.6. Параллельная обработка очереди задач на примере быстрой сортировки

Еще один вариант параллельной быстрой сортировки без порождения лишних горутин – запуск фиксированного количества параллельных обработчиков, берущих задачи из общей очереди `queue`:

```
func quicksort(a []byte) {
```

```
	const THRESHOLD = 10000

	// simple quicksort
	var sort func([]byte)
	sort = func(a []byte) {
		if len(a) <= 1 {
			return
		}

		i := partition(a)
		sort(a[:i]) // from a[0] to a[i - 1]
		sort(a[i:]) // from a[i] to a[len(a) - 1]
	}

	queue := make(chan []byte, len(a)/THRESHOLD+1)
	var wg sync.WaitGroup

	// run workers
	for i := 0; i < runtime.NumCPU(); i++ {
		go func() {
			for {
				a := <-queue

				if len(a) <= THRESHOLD {
					sort(a)
					wg.Done()
					continue
				}

				i := partition(a)

				wg.Add(2)
				queue <- a[:i]
				queue <- a[i:]

				wg.Done()
			}
		}()
	}

	wg.Add(1)

	queue <- a
	wg.Wait()
}
```

Каждый из `runtime.NumCPU()` обработчиков в бесконечном цикле забирает срез из очереди `queue`, применяет к нему функцию **`partition`** и кладет получившиеся части в конец очереди.

Приведенный вариант функции **`quicksort`** корректно выполняет основную задачу, но содержит при этом серьезную *ошибку*: по окончании сортировки с обработчиками будет потеряна связь и они останутся “висеть” на чтении канала `queue` до завершения программы. Для исправления ошибки добавим в функцию **`quicksort`** после инструкции `wg.Add(1)` создание дополнительной горутины, которая по завершении работ будет закрывать канал `queue`:

```
go func() {
	wg.Wait()
	close(queue)
}()
```

А каждый из обработчиков должен прервать свой цикл при обнаружении закрытия канала:

```
a, ok := <-queue
if !ok {
	break
}
```

Полный код программы:

src/pquick3/main.go

```
package main

import (
	"fmt"
	"io/ioutil"
	"log"
	"os"
	"runtime"
	"sync"
	"time"
)

func quicksort(a []byte) {
	const THRESHOLD = 10000

	// simple quicksort
```

```
	var sort func([]byte)
	sort = func(a []byte) {
		if len(a) <= 1 {
			return
		}

		i := partition(a)
		sort(a[:i]) // from a[0] to a[i - 1]
		sort(a[i:]) // from a[i] to a[len(a) - 1]
	}

	queue := make(chan []byte, len(a)/THRESHOLD+1)
	var wg sync.WaitGroup

	// run workers
	for i := 0; i < runtime.NumCPU(); i++ {
		go func() {
			for {
				a, ok := <-queue
				if !ok {
					break
				}

				if len(a) <= THRESHOLD {
					sort(a)
					wg.Done()
					continue
				}

				i := partition(a)

				wg.Add(2)
				queue <- a[:i]
				queue <- a[i:]

				wg.Done()
			}
			fmt.Println("stop worker")
		}()
	}

	wg.Add(1)

	// run stopper
	go func() {
```

```
            wg.Wait()
            close(queue)
      }()

      queue <- a
      wg.Wait()
}

func partition(a []byte) int {
      i := 0
      j := len(a) - 1
      v := a[(j+1)/2]

      for i <= j {
            for a[i] < v {
                  i++
            }
            for a[j] > v {
                  j--
            }
            if i >= j {
                  break
            }

            a[i], a[j] = a[j], a[i]

            i++
            j--
      }

      return i
}

func main() {
      bytes, err := ioutil.ReadFile(os.Args[1])
      if err != nil {
            log.Fatal(err)
      }

      start := time.Now()
      quicksort(bytes)
      duration := time.Since(start)

      fmt.Printf("duration: %s\n", duration)
```

```
      if err := ioutil.WriteFile(os.Args[1]+".sorted.txt", bytes, 0666);
err != nil {
            log.Fatal(err)
      }
}
```

Обратите внимание, что закрытие канала и последующее завершение обработчиков происходит в отдельных горутинах независимо от выхода из функции **quicksort** и, как следствие, не оказывает влияния на подсчет времени сортировки:

```
$ ./bin/pquick3 tmp/MD256.txt
stop worker
stop worker
stop worker
stop worker
duration: 6.774463877s
$ env GOMAXPROCS=256 ./bin/pquick3 tmp/MD256.txt
stop worker
stop worker
stop worker
duration: 6.773008994s
stop worker
```

В данном случае замедление обусловлено синхронизацией при выполнении потокобезопасных операций с общим каналом.

### 5.7. Завершение горутин

В последнем примере обработчики завершались после обнаружения закрытия канала `queue`. Мы имели право на такое решение, так как к моменту закрытия `queue` горутины уже не могли отправлять туда данные. Отправка данных в закрытый канал привела бы к аварийной ситуации.

В следующей программе используется более общий способ информирования горутин о необходимости завершения. Каждый обработчик снабжен специальным каналом `stopChan`, по которому будет доставлен сигнал завершения. В главной горутине эти каналы хранятся в срезе `stopChans`. После запуска обработчиков главная горутина подписывается на сигналы операционной системы: SIGINT (interrupt) и SIGTERM (terminate) – с помощью функции `Notify` пакета `signal` (`"os/signal"`). Эти сигналы будут переданы главной горутине через канал `signals`, на операции чтения которого она блокируется. После прихода SIGINT или SIGTERM будут прождены горутины для отправки `struct{}{}` в каналы из `stopChans`.

src/stop/main.go

```
package main

import (
	"fmt"
	"os"
	"os/signal"
	"runtime"
	"syscall"
	"time"
)

func main() {
	stopChans := make([]chan struct{}, 0, runtime.NumCPU())

	// run workers
	for i := 0; i < runtime.NumCPU(); i++ {
		stopChan := make(chan struct{}, 1)

		go func() {
			for {
				select {
				case <-stopChan:
					fmt.Println("stop goroutine")
					return
				default:
					// do something
					time.Sleep(100 * time.Millisecond)
				}
			}
		}()

		stopChans = append(stopChans, stopChan)
	}

	// waiting for signals
	signals := make(chan os.Signal, 1)
	signal.Notify(signals, syscall.SIGINT, syscall.SIGTERM)
	s := <-signals

	fmt.Printf("%s accepted\n", s)

	// stop goroutines
	for _, ch := range stopChans {
		go func(ch chan struct{}) { ch <- struct{}{} }(ch)
```

```
        }

        time.Sleep(time.Second)
}
```

Запустим программу и нажмем Ctrl+C:

```
$ ./bin/stop
^Cinterrupt accepted
stop goroutine
stop goroutine
stop goroutine
stop goroutine
```

В последнем цикле существует соблазн использовать замыкание для передачи очередного канала `ch` в горутину. Это распространенная *ошибка*:

```
for _, ch := range stopChans {
        go func() { ch <- struct{}{} }() // (!) Ошибка
}
```

Замыкание захватывает *саму переменную*, а не ее текущее значение. Поэтому к моменту начала работы горутин все итерации цикла, скорее всего, уже будут завершены, а переменная `ch` будет иметь значение последнего канала. Как результат, вместо завершения обработчиков мы получим еще несколько “зависших” горутин.

Мы использовали `time.Sleep(time.Second)`, чтобы дать обработчикам время на завершение. В более сложных случаях, когда невозможно оценить необходимое время, следует воспользоваться механизмом `sync.WaitGroup` или синхронизировать главную горутину с обработчиками посредством небуферизованных каналов.

### Список литературы


1. Hoare C. A. R. Communicating Sequential Processes // Comm. ACM 21 (8), 1978. – pp. 666-677.
2. Хоар Ч. Взаимодействующие последовательные процессы: Пер. с англ. – М.: Мир, 1989. – 264 с., ил. ISBN 5-03-001043-2.
3. Таненбаум Э., Вудхал А. Операционные системы. Разработка и реализация. Классика CS. 3-е изд. – СПб.: Питер, 2007. – 703 с.
4. Керниган Б. У., Донован А. А. Язык программирования Go.: Пер. с англ. – М.: Вильямс, 2016. – С. 432. – ISBN 978-5-8459-2051-5.

5. Саммерфильд М. Программирование на Go. Разработка приложений XXI века: пер. с англ.: Киселев А. Н. – М.: ДМК Пресс, 2013. – 580 с.: ил. ISBN 978-5-94074-854-0.
6. Hoare C. A. R. Algorithm 64: Quicksort // Comm. ACM. 4 (7),1961. – p. 321.