

Конспект

Основы. Резюме

Шаг 1 · Резюме · Stepik

Базовые конструкции

Go предоставляет базовые типы для чисел, строк и логических значений.

Переменные объявляют через `var` или сразу инициализируют через `:=`. Константы — через `const`.

```
var n int
s := "apple"
const lang = "go"
```

Из циклов поддерживается только `for`. Поддерживаются `break` и `continue`.

```
for i <= 3 { ... }
for i := 1; i <= 3; i++ { ... }
for { ... }
```

Поддерживаются условные конструкции `if-else` и `switch`.

```
if num < 0 { ... }
if num := 9; num < 0 { ... }

switch i {
case 1: ...
case 2: ...
default: ...
}
```

Шаг 2 · Резюме · Stepik

Массивы, срезы и карты

Массив — нумерованная последовательность элементов. Длина массива известна заранее и зафиксирована.

```
arr := [5]int{1, 2, 3, 4, 5}
```

Срез — ссылка на массив изменяемой длины. Новые элементы добавляются через `append()`. Поддерживается конструкция `slice[from:to]`

```
s := make([]string, 3)
s = append(s, "a")
s = append(s, "b", "c")
sl1 := s[1:]
```

Строку можно преобразовать в срез байт или рун (unicode-символов) и обратно.

```
str := "ro!"
bytes := []byte(str)
str = string(bytes)
runes := []rune(str)
str = string(runes)
```

Карта (map) — это неупорядоченный набор пар «ключ-значение». Создают через `make()`, удаляют элементы через `delete()`.

```
m := make(map[string]int)
m["key"] = 7
m["other"] = 13
delete(m, "other")
val, ok := m["key"]
```

Для обхода массивов, срезов и карт используют цикл `for range`. Аналогично обходят строку по рунам.

```
for idx, val := range slice { ... }
for key, val := range mp { ... }
for idx, char := range str { ... }
```

Шаг 3 · Резюме · Stepik

Функции и указатели

Функция принимает и возвращает параметры. Может принимать произвольное количество, а возвращать несколько. Поддерживаются анонимные функции, их можно использовать как любые другие значения.

```
func fn(in1, in2 int, in3 bool) (int, string) { ... }
func sum(nums ...int) int { ... }
fn := func(s string) int { ... }
```

Указатель содержит адрес памяти, который ссылается на конкретное значение. Тип `*T` — указатель на значение типа `T`. Если указатель не инициализирован, он равен `nil`. Для перехода

от значения к указателю используют оператор `&`, для перехода от указателя к значению — оператор `*`.

```
var iptr *int
i := 42
iptr = &i
i = *iptr
```

Шаг 4 · Резюме · Stepik

Структуры и методы

Структура группирует поля в единую запись. Структуры могут включать другие структуры.

```
type person struct {
    firstName string
    lastName  string
}
```

```
type book struct {
    title string
    author *person
}
```

Собственные типы можно создавать на основе других типов, как встроенных, так и нет. На типах можно определять методы.

```
type customID string

type validator func(customID) bool

type inn customID
func (id inn) isValid() bool { ... }
```

Вместо наследования используют композицию — новое поведение собирают из кирпичиков существующего. Часто для этого применяют встраивание.

```
type counter struct { ... }
func (c *counter) increment() { ... }

type usage struct {
    service string
    counter
}
```

Шаг 5 · Резюме · Stepik

Интерфейсы

Интерфейс — это набор сигнатур методов. Интерфейсы используют, чтобы не зависеть от конкретной реализации, а полагаться только на объявленный контракт. Тип `T` реализует интерфейс `I`, если у него есть все методы, объявленные в `I`.

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

type rect struct { ... }
func (r rect) area() float64 { ... }

type circle struct { ... }
func (c circle) area() float64 { ... }

func measure(g geometry) { ... }
```

Переменная с типом `interface{}` может содержать любое значение. Чтобы его извлечь, используют приведение типов.

```
var ival interface{} = "hello"
str, ok := ival.(string)
```

Шаг 6 · Резюме · Stepik

Ошибки

В Go нет исключений и блока try-catch-finally. Вместо этого функции явно возвращают ошибку — значение интерфейсного типа `error`. Чтобы реализовать интерфейс, достаточно реализовать метод `Error()`.

```
func sqrt(x float64) (float64, error) { ... }

type lookupError struct { ... }
func (e *lookupError) Error() string { ... }
```

Чтобы освободить занятые ресурсы, используют `defer`.

```
func main() {
    f, err := createFile("/tmp/defer.txt")
    if err != nil {
        log.Fatal("Error creating file: ", err)
    }
}
```

```
}  
defer closeFile(f)  
}
```

Необработанные ошибки вызывают панику. Вызвать ее вручную можно функцией `panic()`, а отловить — функцией `recover()` внутри `defer`.

```
func main() {  
    defer func() {  
        if err := recover(); err != nil {  
            fmt.Println("ERROR:", err)  
        }  
    }()  
    panic(errors.New("oops"))  
}
```

Шаг 7 · Резюме · Stepik

Дополнительное чтение

[Go Slices: usage and internals](#)

[Effective Error Handling in Golang](#)

Основы. 1. Базовые конструкции

Шаг 1 · Базовые конструкции · Stepik

Каждый учебник «для начинающих» считает своим долгом посвятить 200 страниц самой скучной части языка. Мы обойдемся одним уроком.

Всем привет

Наша первая программа печатает в консоль сообщение «всем привет»:

```
package main  
  
import "fmt"  
  
func main() {  
    fmt.Println("hello everyone")  
}
```

Чтобы запустить программу, [установите Go](#), сохраните код в `hello.go` и запустите из консоли с помощью `go run`. Если пока неохота возиться с установкой — используйте [песочницу](#).

```
$ go run hello.go
hello everyone
```

Чтобы скомпилировать исходный код в бинарник, воспользуемся `go build`:

```
$ go build hello.go
$ ls
hello  hello.go
```

Запустим скомпилированный бинарник:

```
$ ./hello
hello everyone
```

Мы научились писать и запускать программы на Go, расходимся. Шучу, только начинаем ツ

Шаг 2 · Базовые конструкции · Stepik

Значения

В Go есть целые и дробные числа, строки и логические значения. Вот несколько примеров.

Строки можно складывать через `+`:

```
fmt.Println("go" + "lang")
// go lang
```

Целые и дробные числа:

```
fmt.Println("1+1 =", 1+1)
// 1+1 = 2

fmt.Println("7.0/3.0 =", 7.0/3.0)
// 7.0/3.0 = 2.3333333333333335
```

Логические значения и операторы:

```
fmt.Println(true && false)
// false

fmt.Println(true || false)
// true

fmt.Println(!true)
// false
```

[песочница](#)

Шаг 3 · Базовые конструкции · Stepik

Продолжительность в минутах

Напишите программу, которая считает количество минут во временном отрезке.

```
1h30m = 90 min
300s = 5 min
10m = 10 min
```

Используйте для этого [time.Duration.Minutes\(\)](#) из стандартной библиотеки. Вообще, по ходу курса мы часто будем использовать стандартную библиотеку Go. У нее понятная [документация с примерами](#) — заглядывайте, если что-то непонятно.

Sample Input:

```
1h30m
```

Sample Output:

```
1h30m = 90 min
```

Напишите программу. Тестируется через stdin → stdout

Отличное решение!

```
package main
import (
    "fmt"
    "time"
)
func main() {
    // считываем временной отрезок из os.Stdin
    // гарантируется, что значение корректное
    // не меняйте этот блок
    var s string
    fmt.Scan(&s)
    d, _ := time.ParseDuration(s)

    // выведите исходное значение
    // и количество минут в нем
    // в формате "исходное = X min"
    // используйте метод .Minutes() объекта d
    fmt.Println(s + " =", d.Minutes(), "min")
}
```

Шаг 4 · Базовые конструкции · Stepik

Переменные

Go статически типизирован. Переменные явно объявляются, типы заранее известны компилятору.

`var` объявляет переменную, `=` инициализирует конкретным значением:

```
var b bool = true
fmt.Println(b)
// true

var s string = "hello"
fmt.Println(s)
// hello

var i int = 42
fmt.Println(i)
// 42

var f float64 = 12.34
fmt.Println(f)
// 12.34
```

Можно объявить сразу несколько:

```
var one, two int = 1, 2
fmt.Println(one, two)
// 1 2
```

Если инициализировать переменную при объявлении, тип можно и не указывать. Go сам догадается:

```
var sunny = true
fmt.Printf("%v is %T\n", sunny, sunny)
// true is bool
```

`fmt.Printf()` подставляет переменные в строку по шаблону. `%v` — формат по умолчанию, `%T` — тип переменной.

Если не инициализировать переменную при объявлении, она получит *нулевое значение*. У каждого типа оно свое. У `int`, например, `0`:

```
var num int
fmt.Println(num)
```



```
// 0
```

Оператор `:=` объявляет и инициализирует переменную. `var` и `тип` можно не указывать:

```
food := "apple" // var food string = "apple"
fmt.Println(food)
// apple
```

[песочница](#)

```
package main

import (
    "fmt"
    "math"
)

func main() {
    // объявите переменные x1, y1, x2, y2 типа float64
    var x1, y1, x2, y2 float64

    // считываем числа из os.Stdin
    // гарантируется, что значения корректные
    // не меняйте этот блок
    fmt.Scan(&x1, &y1, &x2, &y2)

    // рассчитайте d по формуле эвклидова расстояния
    // используйте math.Pow(x, 2) для возведения в квадрат
    // используйте math.Sqrt(x) для извлечения корня
    d := math.Sqrt(math.Pow(x1 - x2, 2) + math.Pow(y1 - y2, 2))

    // выводим результат в os.Stdout
    fmt.Println(d)
}
```

Шаг 5 · Базовые конструкции · Stepik

Константы

Go поддерживает *константы* для символов, строк, логических и числовых значений.

`const` объявляет константу:

```
const s string = "constant"
fmt.Println(s)
```

```
// constant

const n = 500000000
fmt.Println(n)
// 500000000

const ch = 'a'
fmt.Println(ch)
// 97
// это числовой код латинского символа 'a'
```

Если указать выражение — Go рассчитает и сохранит результат. В выражении можно использовать объявленные ранее константы:

```
const d = 3e20 / n
fmt.Println(d)
// 6e+11
```

Тип числовой константы определяется из контекста. Например, когда есть явное приведение типа:

```
fmt.Println(int64(d))
// 60000000000000
```

Или функция, которая требует значение определенного типа (`math.Sin()` ожидает `float64`):

```
fmt.Println(math.Sin(n))
// -0.28470407323754404
```

[песочница](#)

Шаг 6 · Базовые конструкции · Stepik

Расстояние между точками

Напишите программу, которая рассчитает евклидово расстояние между точками на плоскости `(x1, y1)` и `(x2, y2)`:

$$d = \sqrt{(x1 - x2)^2 + (y1 - y2)^2}$$

Как устроены задачи

Это и все следующие задания модуля проверяются так:

- на вход программы подается строка через стандартный ввод (`os.Stdin`);
- программа обрабатывает и печатает результат в стандартный вывод (`os.Stdout`);
- напечатанный результат сравнивается с ожидаемым.

«Степик» делает это автоматически, когда вы нажимаете на «Запустить код» (программа отработает на тех данных, что вы укажете) или на «Отправить» (программа отработает на тестовых данных, которые я подготовил).

Пользоваться редактором «Степика» не всегда удобно. Если вы решите запускать программу локально, передать данные ей на вход можно через `echo` и пайп. Например, если исходный файл называется `distance.go`:

```
$ echo "1 1 4 5" | go run distance.go
5
```

Sample Input:

```
1 1 4 5
```

Sample Output:

```
5
```

Шаг 7 · Базовые конструкции · Stepik

Циклы

`for` — единственная конструкция для циклов в Go. Вот несколько примеров.

С одним условием, аналог `while` в питоне:

```
i := 1
for i <= 3 {
    fmt.Println(i)
    i = i + 1
}
// 1
// 2
// 3
```

Классический `for` из трех частей (инициализация, условие, шаг):

```
for j := 7; j <= 9; j++ {
    fmt.Println(j)
}
// 7
// 8
// 9
```

Бесконечный цикл, выполняется до `break` для выхода из цикла или `return` для выхода из функции:

```
for {
    fmt.Println("loop")
    break
}
// loop
```

`continue` переходит к следующей итерации цикла:

```
for n := 0; n <= 5; n++ {
    if n%2 == 0 {
        continue
    }
    fmt.Println(n)
}
// 1
// 3
// 5
```

[песочница](#)

```
package main

import (
    "fmt"
    "math"
)

func main() {
    // объявите переменные x1, y1, x2, y2 типа float64
    var x1, y1, x2, y2 float64

    // считываем числа из os.Stdin
    // гарантируется, что значения корректные
    // не меняйте этот блок
    fmt.Scan(&x1, &y1, &x2, &y2)

    // рассчитайте d по формуле эвклидова расстояния
    // используйте math.Pow(x, 2) для возведения в квадрат
    // используйте math.Sqrt(x) для извлечения корня
    d := math.Sqrt(math.Pow(x1 - x2, 2) + math.Pow(y1 - y2, 2))
}
```

```
// выводим результат в os.Stdout
fmt.Println(d)
}
```

Шаг 8 · Базовые конструкции · Stepik

Повтор строки

Программа принимает на вход строку `source` и число `times`. Требуется склеить `source` саму с собой `times` раз и вернуть результат:

- `source = x`, `times = 3` → `xxx`
- `source = omm`, `times = 2` → `ommomm`

Sample Input:

a 5

Sample Output:

aaaaa

```
package main

import (
    "fmt"
)

func main() {
    var source string
    var times int
    // гарантируется, что значения корректные
    fmt.Scan(&source, &times)

    // возьмите строку `source` и повторите ее `times` раз
    // запишите результат в `result`
    var result string = ""
    for i := 0; i < times; i++ {
        result += source
    }

    fmt.Println(result)
}
```

Шаг 9 · Базовые конструкции · Stepik

If/else

Конструкция `if-else` в Go ведет себя без особых сюрпризов.

Вокруг условия не нужны круглые скобки, но фигурные скобки для веток обязательны:

```
if 7%2 == 0 {  
    fmt.Println("7 is even")  
} else {  
    fmt.Println("7 is odd")  
}  
// 7 is odd
```

Можно использовать `if` без `else`:

```
if 8%4 == 0 {  
    fmt.Println("8 is divisible by 4")  
}  
// 8 is divisible by 4
```

Единственный нюанс: перед условием может идти выражение. Объявленные в нем переменные доступны во всех ветках:

```
if num := 9; num < 0 {  
    fmt.Println(num, "is negative")  
} else if num < 10 {  
    fmt.Println(num, "has 1 digit")  
} else {  
    fmt.Println(num, "has multiple digits")  
}  
// 9 has 1 digit
```

[песочница](#)

Шаг 10 · Базовые конструкции · Stepik

Switch

`switch` описывает условие с множеством веток.

В отличие от многих других языков, не нужно указывать `break`. Go выполняет только подходящую ветку и не проваливается в следующую:

```
i := 2  
fmt.Print("Write ", i, " as ")  
switch i {
```

```

case 1:
    fmt.Println("one")
case 2:
    fmt.Println("two")
case 3:
    fmt.Println("three")
}
// Write 2 as two

```

Чтобы провалиться в следующую ветку, есть специальное ключевое слово `fallthrough`. Его можно использовать только в качестве последней инструкции в блоке:

```

i := 2
fmt.Print("Write ", i, " as ")
switch i {
case 1:
    fmt.Println("one")
case 2:
    fmt.Println("two")
    fallthrough
case 3:
    fmt.Println("Bye-bye")
}
// Write 2 as two
// Bye-bye

```

В одной ветке можно указать несколько выражений. Ветка `default` сработает, если остальные не подошли:

```

switch time.Now().Weekday() {
case time.Saturday, time.Sunday:
    fmt.Println("It's the weekend")
default:
    fmt.Println("It's a weekday")
}
// It's the weekend

```

Выражения в ветках не обязательно должны быть константами. `switch` может работать как `if`:

```

t := time.Now()
switch {
case t.Hour() < 12:
    fmt.Println("It's before noon")
default:
    fmt.Println("It's after noon")
}

```

```
}  
// It's before noon
```

Дополнительное чтение

Сам я предпочитаю не закапываться в теорию, а изучать ее параллельно с практикой. Но знаю, что есть люди, которые готовы две недели читать документацию, прежде чем напишут строчку кода. Специально для них (а также для тех, кто прошел курс и готов углубиться в теорию) в каждом уроке есть раздел «дополнительное чтение» со ссылками на спецификацию языка и другие материалы.

[Lexical elements](#): комментарии, идентификаторы, ключевые слова, операторы и пунктуация.

Значения: [integer](#) • [floating-point](#) • [imaginary](#) • [rune](#) • [string](#)

[Constants](#) • [constant declarations](#) • [constant expressions](#)

[Variables](#) • [variable declarations](#) • [short declarations](#)

Типы: [boolean](#) • [numeric](#) • [string](#)

[Operators](#): [arithmetic](#) • [comparison](#) • [logical](#) • [order of evaluation](#)

Инструкции: [IncDec](#) • [assignments](#) • [if-else](#) • [switch](#) • [for](#) • [break](#) • [continue](#) • [goto](#) • [fallthrough](#)

[Zero value](#)

[Blocks](#)

[песочница](#)

Шаг 11 · Базовые конструкции · Stepik

Язык по коду

Напишите программу, которая определяет название языка по его коду. Правила:

- `en` → English
- `fr` → French
- `ru` или `rus` → Russian
- иначе → Unknown

Sample Input:

```
en
```

Sample Output:

English

```
package main

import (
    "fmt"
)

func main() {
    var code string
    fmt.Scan(&code)

    // определите полное название языка по его коду
    // и запишите его в переменную `lang`
    var lang string = "Unknown"
    switch code {
        case "en":
            lang = "English"
        case "fr":
            lang = "French"
        case "ru", "rus":
            lang = "Russian"
    }

    fmt.Println(lang)
}
```

Основы. 2. Массивы и карты

Шаг 1 · Массивы и карты · Stepik

Массивы

Массив в Go — это нумерованная последовательность элементов. Длина массива известна заранее и зафиксирована.

Массив `a` содержит 5 целых чисел. Тип и количество элементов — часть определения массива. По умолчанию элементы массива принимают нулевое значение, в данном случае — `0`:

```
var a [5]int
fmt.Println("empty:", a)
// empty: [0 0 0 0 0]
```

Обращение к элементу массива — через квадратные скобки:

```
a[4] = 100
fmt.Println("set:", a)
// set: [0 0 0 0 100]

fmt.Println("get:", a[4])
// get: 100
```

Встроенная функция `len()` возвращает количество элементов:

```
fmt.Println("len:", len(a))
// len: 5
```

Можно инициализировать массив при объявлении:

```
b := [5]int{1, 2, 3, 4, 5}
fmt.Println("init:", b)
// init: [1 2 3 4 5]
```

Массивы одномерные, но их можно комбинировать, чтобы получить нужную размерность:

```
var twoD [2][3]int
for i := 0; i < 2; i++ {
    for j := 0; j < 3; j++ {
        twoD[i][j] = i + j
    }
}
fmt.Println("2d:", twoD)
// 2d: [[0 1 2] [1 2 3]]
```

[песочница](#)

Шаг 2 · Массивы и карты · Stepik

Срезы

Срез (slice) — ключевая структура данных в Go. Это массив изменяемой длины, как *list* в питоне или *Array* в js. Обычно в программах на Go оперируют именно срезами, «чистые» массивы встречаются намного реже.

Срез определяется только типом элементов, но не их количеством. Чтобы создать срез ненулевой длины, используют встроенную функцию `make()`. Здесь мы создаем срез из трех пустых строк:

```
s := make([]string, 3)
fmt.Printf("empty: %#v\n", s)
// empty: []string{"", "", ""}
```

Шаблон `%#v` возвращает «внутреннее представление» значения, примерно как `repr()` в питоне.

С элементами среза можно работать точно так же, как с элементами массива:

```
s[0] = "a"
s[1] = "b"
s[2] = "c"

fmt.Println("set:", s)
// set: [a b c]

fmt.Println("get:", s[2])
// get: c
```

`len()` возвращает длину среза:

```
fmt.Println("len:", len(s))
// len: 3
```

В отличие от массива, в срез можно добавлять новые элементы через встроенную функцию `append()`. Функция возвращает новый срез:

```
fmt.Println("src:", s)
// src: [a b c]

s = append(s, "d")
s = append(s, "e", "f")
fmt.Println("upd:", s)
// upd: [a b c d e f]
```

Всегда используйте значение, которое возвращает `append()`. Вот так делать не стоит:

```
append(s, "d")
fmt.Println("upd:", s)
```

Дело в том, что срез сам по себе не хранит данные, это ссылка на конкретный массив. Если в массиве нет места для нового элемента, `append()` создаст новый массив побольше, скопирует в него старые элементы, добавит новый элемент и вернет ссылку на новый массив. Если эту ссылку проигнорировать, новый срез вы потеряете.

Срез можно скопировать через встроенную функцию `copy()`. Здесь создаем пустой срез `dst` такой же длины, как `s`, и копируем в него элементы `s`:

```
dst := make([]string, len(s))
copy(dst, s)
```

```
fmt.Println("copy:", dst)
// copy: [a b c d e f]
```

Срезы поддерживают... срезы (отсюда их название). Выражение `slice[from:to]` вернет срез от элемента с индексом `from` включительно до элемента с индексом `to` не включительно:

```
sl1 := s[2:5]
fmt.Println("sl1:", sl1)
// sl1: [c d e]
```

Этот срез включает все элементы, кроме `s[5]`:

```
sl2 := s[:5]
fmt.Println("sl2:", sl2)
// sl2: [a b c d e]
```

А этот срез включает элементы от `s[2]` и до конца:

```
sl3 := s[2:]
fmt.Println("sl3:", sl3)
// sl3: [c d e f]
```

Срез можно инициализировать при объявлении:

```
t := []string{"g", "h", "i"}
fmt.Println("init:", t)
// init: [g h i]
```

[песочница](#)

Шаг 3 · Массивы и карты · Stepik

Срезы и строки

Строку можно преобразовать в срез байт и обратно:

```
str := "ro!"

bytes := []byte(str)

fmt.Println(bytes)
// [208 179 208 190 33]

fmt.Println(str == string(bytes))
// true
```

Строку можно преобразовать в срез unicode-символов (Go называет их *рунами*). Одна руна может занимать несколько байт (что и произошло с рунами `г` и `о`):

```
runes := []rune(str)

fmt.Println(runes)
// [1075 1086 33]

fmt.Println(str == string(runes))
// true
```

[песочница](#)

Шаг 4 · Массивы и карты · Stepik

Укорот байтовой строки

Напишите программу, которая укорачивает строку до указанной длины и добавляет в конце многоточие:

- `text = Eyjafjallajokull`, `width = 6` → `Eyjafj...`

Если строка не превышает указанной длины, менять ее не следует:

- `text = hello`, `width = 6` → `hello`

Гарантируется, что в исходной строке `text` используются только однобайтовые символы без пробелов, а длина `width` строго больше 0.

Sample Input:

```
Eyjafjallajokull 6
```

Sample Output:

```
Eyjafj...
```

```
package main

import (
    "fmt"
)

func main() {
    var text string
    var width int
    fmt.Scanf("%s %d", &text, &width)
```

```

// Возьмите первые `width` байт строки `text`,
// допишите в конце `...` и сохраните результат
// в переменную `res`
tmp := make([]byte, len([]byte(text)))
copy(tmp, []byte(text))

var res string
if width < len(text) {
    res = string(tmp[:width]) + "..."
} else {
    res = string(tmp)
}

fmt.Println(res)
}

```

что-то с описанием срезов перемудрили, тут же просто

```

res := text
if len(text) > width
    res = text[:width] + "..."
}

```

Шаг 5 · Массивы и карты · Stepik

Карты

Карта (map), так же известная как словарь (dict), хеш-таблица (hash table) или ассоциативный массив (associative array) — это неупорядоченный набор пар «ключ-значение».

Чтобы создать пустую карту, используют `make()`:

```
m := make(map[string]int)
```

Задать пары «ключ-значение»:

```

m["key"] = 7
m["other"] = 13

```

Вывести содержимое карты:

```

fmt.Println("map:", m)
// map: map[key:7 other:13]

```

Получить значение по ключу:

```
val := m["key"]
fmt.Println("val:", val)
// val: 7
```

`len()` возвращает количество записей (пар «ключ-значение») в карте:

```
fmt.Println("len:", len(m))
// len: 2
```

`delete()` удаляет запись по ключу:

```
delete(m, "other")
fmt.Println("map:", m)
// map: map[key:7]
```

Обращение к записи по ключу возвращает необязательное второе значение: признак, есть такой ключ в карте или нет. Обращение по несуществующему ключу не приведет к ошибке, но вернет этот признак со значением `false`:

```
_, ok := m["other"]
fmt.Println("has other:", ok)
// has other: false
```

Пустой идентификатор `_` указывает, что нам не интересно само значение по ключу, важен только признак «есть/нет» (`ok`).

Карту можно инициализировать при объявлении:

```
n := map[string]int{"foo": 1, "bar": 2}
fmt.Println("map:", n)
// map: map[bar:2 foo:1]
```

[песочница](#)

Шаг 6 · Массивы и карты · Stepik

Счетчик цифр

Напишите программу, которая считает, сколько раз каждая цифра встречается в числе.

Гарантируется, что на вход подаются только положительные целые числа, не выходящие за диапазон `int`.

Sample Input:

```
12823
```

Sample Output:

1:1 2:2 3:1 8:1

```
package main

import (
    "fmt"
    "sort"
)

func main() {
    var number int
    fmt.Scanf("%d", &number)

    // Посчитайте, сколько раз каждая цифра встречается
    // в числе `number`. Запишите результат в карту `counter`,
    // где ключом является цифра числа, а значением -
    // сколько раз она встречается
    counter := make(map[int]int)
    rest := number / 10
    for rest != 0 {
        counter[number%10]++
        number = rest
        rest = number / 10
    }
    counter[number%10]++

    printCounter(counter)
}

// 

// | не меняйте код ниже этой строки |
//



// printCounter печатает карту в формате
// key1:val1 key2:val2 ... keyN:valN
func printCounter(counter map[int]int) {
    digits := make([]int, 0)
    for d := range counter {
        digits = append(digits, d)
    }
    sort.Ints(digits)
    for idx, digit := range digits {
        fmt.Printf("%d:%d", digit, counter[digit])
    }
}
```



```

        if idx < len(digits)-1 {
            fmt.Print(" ")
        }
    }
    fmt.Print("\n")
}

```

варианты

```

counter := make(map[int]int)
for i:= 1; i <= number; i *= 10 {
    counter[number / i % 10]++
}

```

```

counter := make(map[int]int)
x := number
for x != 0 {
    digit := x % 10
    counter[digit]++
    x = x / 10
}

```

Шаг 7 · Массивы и карты · Stepik

Обход коллекции

`range` обходит элементы коллекций. Посмотрим, как использовать его на срезах, картах и строках.

Просуммировать элементы среза (или массива):

```

nums := []int{2, 3, 4}
sum := 0
for _, num := range nums {
    sum += num
}
fmt.Println("sum:", sum)
// sum: 9

```

`range` на массивах и срезах возвращает индекс и значение для каждого элемента. В примере выше мы не использовали индекс, поэтому заглушили его пустым идентификатором `_`. Но иногда индекс может и пригодиться:

```

for idx, num := range nums {
    if num == 3 {

```

```
        fmt.Println("index:", idx)
    }
}
// index: 1
```

`range` на карте итерирует по записям:

```
m := map[string]string{"a": "apple", "b": "banana"}
for key, val := range m {
    fmt.Printf("%s -> %s\n", key, val)
}
// a -> apple
// b -> banana
```

Или только по ключам:

```
for key := range m {
    fmt.Println("key:", key)
}
// key: a
// key: b
```

`range` на строках итерирует по unicode-символам (рунам). Первое значение — порядковый номер байта, с которого начинается руна (руна может занимать несколько байт). Второе значение — числовой код самой руны:

```
for idx, char := range "oro" {
    fmt.Println(idx, char, string(char))
}
// 0 1086 o
// 2 1075 r
// 4 1086 o
```

Дополнительное чтение

[Slice](#) • [Map](#)

[Index expressions](#) • [Slice expressions](#)

[Length and capacity](#)

[Making slices, maps and channels](#)

[Appending and copying slices](#)

[Deletion of map elements](#)

[Go Slices: usage and internals](#)

[Strings, bytes, runes and characters in Go](#)

[The mechanics of 'append'](#)

[Go maps in action](#)

[песочница](#)

Шаг 8 · Массивы и карты · Stepik

Аббревиатура

Напишите программу, которая принимает на вход фразу и составляет аббревиатуру по первым буквам слов:

- Today I learned → TIL
- Высшее учебное заведение → ВУЗ
- Кот обладает талантом → КОТ

Если слово начинается не с буквы, игнорируйте его:

- Ар 2 Ди #2 → АД

Разделителями слов считаются только пробельные символы. Дефис, дробь и прочие можно не учитывать:

- Анна-Мария Волхонская → АВ

Sample Input:

Today I learned

Sample Output:

TIL

```
package main

import (
    "bufio"
    "fmt"
    "os"
    "strings"
    "unicode"
)
```

```

func main() {
    phrase := readString()

    // 1. Разбейте фразу на слова, используя `strings.Fields()`
    // 2. Возьмите первую букву каждого слова и приведите
    //    ее к верхнему регистру через `unicode.ToUpper()`
    // 3. Если слово начинается не с буквы, игнорируйте его
    //    проверяйте через `unicode.IsLetter()`
    // 4. Составьте слово из получившихся букв и запишите его
    //    в переменную `abbr`
    var abbr []rune

    for _, token := range strings.Fields(phrase) {
        for j, ch := range token {
            if j == 0 && unicode.IsLetter(ch) {
                abbr = append(abbr, unicode.ToUpper(ch))
            }
        }
    }

    fmt.Println(string(abbr))
}

// 

//     не меняйте код ниже этой строки
//



// readString читает строку из `os.Stdin` и возвращает ее
func readString() string {
    rdr := bufio.NewReader(os.Stdin)
    str, _ := rdr.ReadString('\n')
    return str
}

```

варианты

```

// разбейте фразу на слова, используя `strings.Fields()`
words := strings.Fields(phrase)

// будущая аббревиатура
var abbr []rune
for _, word := range words {
    // возьмите первую букву каждого слова

```

```
letter := []rune(word)[0]

// если слово начинается не с буквы, игнорируйте его
if !unicode.IsLetter(letter) {
    continue
}

// приведите первую букву к верхнему регистру
// составьте слово из получившихся букв
abbr = append(abbr, unicode.ToUpper(letter))
}
```

```
abbr := []rune("")
for _, word := range strings.Fields(readString()) {
    letter, _ := utf8.DecodeRuneInString(word)//[]rune(word)[0]
    if unicode.IsLetter(letter){
        abbr = append(abbr, unicode.ToUpper(letter))
    }
}
```

Основы. 3. Функции и указатели

Шаг 1 · Функции и указатели · Stepik

Функции

Функция — центральная конструкция языка. Вот несколько примеров.

Эта функция принимает два целых числа и возвращает их сумму:

```
func sum(a int, b int) int {
    return a + b
}
```

Результат возвращается явно, через `return`.

Если у нескольких идущих подряд параметров один и тот же тип, их можно «схлопнуть» и указать тип только для последнего:

```
func sumAbc(a, b, c int) int {
    return a + b + c
}
```

Функция может возвращать несколько значений. В этом примере — частное и остаток от деления одного числа на другое:

```
func divide(divisible, divisor int) (int, int) {
    quotient := divisible / divisor
    remainder := divisible % divisor
    return quotient, remainder
}
```

Результат вызова можно сразу разложить по переменным:

```
q, r := divide(10, 3)
fmt.Println("10 / 3 =", q)
// 10 / 3 = 3
fmt.Println("10 % 3 =", r)
// 10 % 3 = 1
```

Или проигнорировать одно из значений с помощью пустого идентификатора `_`:

```
_, r = divide(42, 2)
if r == 0 {
    fmt.Println("42 is divisible by 2")
}
// 42 is divisible by 2
```

[песочница](#)

Шаг 2 · Функции и указатели · Stepik

Переменное количество аргументов

Функция может принимать произвольное количество аргументов в «хвосте» (как **args* в питоне или *...args* в js). Например, так ведет себя `fmt.Println()`.

Эта функция суммирует целые числа:

```
func sum(nums ...int) {
    fmt.Print(nums, " -> ")
    total := 0
    for _, num := range nums {
        total += num
    }
    fmt.Println(total)
}
```

Можно передавать индивидуальные аргументы, как у обычной функции:

```
sum(1, 2)
// [1 2] -> 3
```

```
sum(1, 2, 3)
// [1 2 3] -> 6
```

А можно передать срез, преобразовав его в список аргументов с помощью `...`:

```
nums := []int{1, 2, 3, 4}
sum(nums...)
// [1 2 3 4] -> 10
```

[песочница](#)

Шаг 3 · Функции и указатели · Stepik

Анонимные функции

Go поддерживает *анонимные функции*. Работают они как обычные, но не имеют названия (как лямбды в питоне или стрелочные функции в js).

Чаще всего анонимные функции используют, чтобы *вернуть из функции другую функцию*. В примере ниже `intSeq()` возвращает функцию-генератор, которая при каждом вызове выдает очередное значение счетчика `i`. Генератор использует переменную, определенную во внешней функции — то есть образует *замыкание* (closure):

```
func intSeq() func() int {
    i := 0
    return func() int {
        i++
        return i
    }
}
```

Результат вызова `intSeq()` — функцию-генератор — мы записываем в переменную `nextInt`. У `nextInt` собственное значение счетчика `i`, которое увеличивается при каждом вызове:

```
nextInt := intSeq()

fmt.Println(nextInt())
// 1
fmt.Println(nextInt())
// 2
fmt.Println(nextInt())
// 3
```

Если создать еще один генератор — он будет обладать собственным счетчиком `i`:

```
newInts := intSeq()
fmt.Println(newInts())
// 1
```

Иногда анонимную функцию передают как *аргумент другой функции*. Пример из пакета `sort` :

```
func Search(n int, f func(int) bool) int
```

`Search()` находит наименьшее `i` из диапазона `[0, n)`, для которого функция-предикат `f(i)` вернет `true`. В качестве предиката удобно использовать анонимную функцию:

```
a := []int{1, 2, 4, 8, 16, 32, 64, 128}
x := 53

// ближайший сверху к `x` элемент среза `a`
closest := sort.Search(len(a), func(i int) bool { return a[i] >= x })

fmt.Println(a[closest], "is the closest to", x)
// 64 is the closest to 53
```

[песочница](#)

Шаг 4 · Функции и указатели · Stepik

Фильтр для коллекции

Напишите функцию `filter()`, которая фильтрует срез целых чисел с помощью функции-предиката и возвращает отфильтрованный срез. Функция-предикат вызывается для каждого элемента исходного среза. Если она возвращает `true`, элемент попадает в отфильтрованный срез. Если возвращает `false` — не попадает.

Считайте исходный срез из стандартного ввода с помощью готовой функции `readInput()`. Затем выполните на нем `filter()`. В качестве предиката используйте функцию, которая возвращает `true` только для четных чисел. Напечатайте отфильтрованный срез.

Гарантируется, что на вход подаются только целые числа.

Sample Input:

```
1 2 3 4 5 6
```

Sample Output:

```
[2 4 6]
```



```

package main

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

func filter(predicate func(int) bool, iterable []int) []int {
    // отфильтруйте `iterable` с помощью `predicate`
    // и верните отфильтрованный срез
    var res []int
    for _, num := range iterable {
        if predicate(num) {
            res = append(res, num)
        }
    }
    return res
}

func main() {
    src := readInput()
    // отфильтруйте `src` так, чтобы остались только четные числа
    // и запишите результат в `res`
    res := filter(func (i int) bool { return i%2 == 0 }, src)
    fmt.Println(res)
}

// 

//     не меняйте код ниже этой строки
//



// readInput считывает целые числа из `os.Stdin`
// и возвращает в виде среза
// разделителем чисел считается пробел
func readInput() []int {
    var nums []int
    scanner := bufio.NewScanner(os.Stdin)
    scanner.Split(bufio.ScanWords)
    for scanner.Scan() {
        num, _ := strconv.Atoi(scanner.Text())
        nums = append(nums, num)
    }
}

```

```
}  
    return nums  
}
```

Шаг 5 · Функции и указатели · Stepik

Указатели

Указатель (pointer) содержит адрес памяти, который ссылается на конкретное значение.

Тип `*T` — указатель на значение типа `T`. Если указатель не инициализирован, он равен `nil` (аналог `None` в питоне и `null` в js).

```
var iptr *int
```

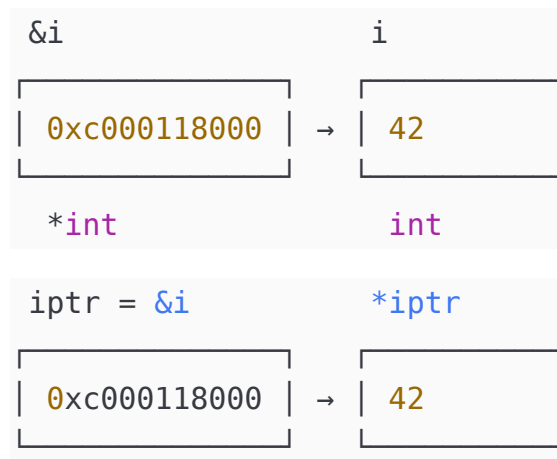
Оператор `&` возвращает указатель на значение:

```
i := 42  
iptr = &i  
  
fmt.Println(iptr)  
// 0xc000118000
```

Оператор `*` обращается к значению, на которое ссылается указатель. Оно доступно как для чтения, так и для записи:

```
fmt.Println(*iptr)  
// прочитает значение `i` через указатель `iptr`  
  
*iptr = 21  
// установить значение `i` через указатель `iptr`
```

Вот схема непростых отношений между значением и указателем на него:



Остаток урока мы посвятим указателям. Если они вам «не зашли», можно перейти к следующему уроку и вернуться сюда, когда будет подходящий настрой.

Шаг 6 · Функции и указатели · Stepik

Указатели в параметрах функции

Указатели в параметрах функции позволяют изменять переданные значения. Вот как это работает.

Функция `zeroval()` принимает параметр типа `int` — конкретное число. При вызове Go передает не оригинальное число `i`, а его копию — `ival`:

```
func zeroval(ival int) {
    ival = 0
}

i := 42
fmt.Println("initial:", i)
// initial: 42

zeroval(i)
fmt.Println("zeroval:", i)
// zeroval: 42
```

`zeroval()` занулила копию оригинального числа `ival`, так что само `i` не изменилось.

Функция `zeroptr()` принимает параметр типа `*int` — указатель на число. При вызове Go передает адрес в памяти, по которому находится оригинальное число `i` — указатель `iptr`. Функция изменяет значение `i` по указателю через оператор `*`:

```
func zeroptr(iptr *int) {
    *iptr = 0
}

i := 42
fmt.Println("initial:", i)
// initial: 42

zeroptr(&i)
fmt.Println("zeroptr:", i)
// zeroptr: 0
```

Благодаря оператору `&`, в функцию передано не значение `i`, а указатель на него. `zeroptr()` изменила оригинальное значение `i` через переданный указатель.

Шаг 7 · Функции и указатели · Stepik

Указатель или значение?

Базовый принцип такой:

- Если функция только читает переменную, но не изменяет — передавайте значение.
- Если функция изменяет значение — передавайте указатель.

Функция `math.Max()` возвращает максимальное из двух чисел:

```
a := 5
b := 3
max := math.Max(a, b)
```

`a` и `b` не изменяются, поэтому функция принимает обычные значения.

Функция `fmt.Scanf()` считывает значения из стандартного ввода и записывает их в переданные переменные. Поэтому принимает указатели:

```
var a, b int
fmt.Scanf("%d-%d", &a, &b)
```

Правило работает для скалярных значений (логических, чисел, строк) и массивов. Со срезами и картами другая история.

Шаг 8 · Функции и указатели · Stepik

Указатель или значение? Срезы и карты

Функция стандартной библиотеки `sort.Ints()` сортирует срез:

```
func Ints(nums []int)
```

Обратите внимание — функция ничего не возвращает, она изменяет элементы оригинального среза. Но почему тогда `nums` передан как значение (`[]int`), а не как указатель (`*[]int`)?

Дело в том, что срез сам по себе не содержит данные массива. Срез — это легковесная структура данных, одно из полей которой — указатель на конкретный массив. Поэтому `nums` внутри функции — это копия, но не всего массива, а этой легковесной структуры с указателем на массив. Обращаясь к элементу среза, функция переходит по указателю и модифицирует оригинальный элемент массива.

slice

array



Если функция изменяет отдельные элементы среза, передавайте его как значение:

```
func sortSlice(nums []int) {  
    sort.Ints(nums)  
}
```

```
nums := []int{5, 1, 3, 9}  
sortSlice(nums)  
fmt.Println(nums)  
// [1 3 5 9]
```

Этот подход не сработает, если изменить сам срез (добавить или удалить элементы):

```
func appendByVal(nums []int, i int) {  
    nums = append(nums, i)  
}
```

```
nums := []int{42}  
appendByVal(nums, 43)  
fmt.Println(nums)  
// ожидание: [42 43]  
// реальность: [42]
```

`nums` внутри функции — это копия оригинального среза. Изменив `nums` через `append()`, функция поменяла копию, а оригинал не изменился.

Чтобы изменить срез в целом, можно использовать указатель:

```
func appendByPtr(nums *[]int, i int) {  
    *nums = append(*nums, i)  
}
```

```
nums := []int{42}  
appendByPtr(&nums, 43)  
fmt.Println(nums)  
// [42 43]
```

Но в обычных функциях такой подход нечасто встретишь. Лучше вернуть новый срез, чем переопределять старый по указателю:

```
func appendAndReturn(nums []int, i int) []int {
    nums = append(nums, i)
    return nums
}

nums := []int{42}
nums = appendAndReturn(nums, 43)
fmt.Println(nums)
// [42 43]
```

Итого:

- если функция не меняет срез — передавать значение;
- если функция меняет отдельные элементы, но не сам срез — передавать значение;
- если функция меняет сам срез — передавать значение и возвращать новое значение.

Вариант «передавать указатель на срез» остается для *методов*, о которых мы поговорим на следующем уроке.

Для карт принцип такой же.

Дополнительное чтение

[Functions](#) • [function declarations](#) • [function literals](#)

[Calls](#) • [variadic](#) • [return statements](#)

[Pointers](#) • [address operators](#)

[песочница](#)

Шаг 9 · Функции и указатели · Stepik

Перетасовка

Напишите функцию `shuffle()`, которая тасует элементы среза в случайном порядке. Функция должна обрабатывать in-place, то есть менять содержимое переданного среза, а не создавать новый срез. Чтобы перетасовать элементы, используйте функцию [rand.Shuffle\(\)](#):

```
func Shuffle(n int, swap func(i, j int))
```

Гарантируется, что на вход подаются только целые числа.

Sample Input:

1 2 3 4 5 6

Sample Output:

[2 4 5 6 1 3]

```
package main

import (
    "bufio"
    "fmt"
    "math/rand"
    "os"
    "strconv"
)

func shuffle(nums []int) {
    // Функция `Seed()` инициализирует генератор случайных чисел
    // здесь мы используем константу `42`, чтобы программу
    // можно было проверить тестами.
    // В реальных задачах не используйте константы!
    // Используйте, например, время в наносекундах:
    // rand.Seed(time.Now().UnixNano())
    rand.Seed(42)
    rand.Shuffle(len(nums), func(i, j int) {
        nums[i], nums[j] = nums[j], nums[i]
    })
}

// | не меняйте код ниже этой строки |
// |

func main() {
    nums := readInput()
    shuffle(nums)
    fmt.Println(nums)
}

// readInput считывает целые числа из `os.Stdin`
// и возвращает в виде среза
// разделителем чисел считается пробел
func readInput() []int {
    var nums []int
    scanner := bufio.NewScanner(os.Stdin)
    scanner.Split(bufio.ScanWords)
```

```
    for scanner.Scan() {
        num, _ := strconv.Atoi(scanner.Text())
        nums = append(nums, num)
    }
    return nums
}
```

Основы. 4. Структуры и методы

Шаг 1 · Структуры и методы · Stepik

Структуры

Структура (struct) группирует поля в единую запись. В Go нет классов и объектов, так что структура — наиболее близкий аналог объекта в питоне и js.

Объявим тип `person` на основе структуры с полями `name` и `age`:

```
type person struct {
    name string
    age  int
}
```

Так создается новая структура типа `person`:

```
bob := person{"Bob", 20}
fmt.Println(bob)
// {Bob 20}
```

Можно явно указать названия полей:

```
alice := person{name: "Alice", age: 30}
fmt.Println(alice)
// {Alice 30}
```

Если не указать поле, оно получит нулевое значение:

```
fred := person{name: "Fred"}
fmt.Println(fred)
// {Fred 0}
```

Оператор `&` возвращает указатель на структуру:

```
annptr := &person{name: "Ann", age: 40}
fmt.Println(annptr)
// &{Ann 40}
```


В Go иногда создают новые структуры через функцию-конструктор с префиксом `new`:

```
func newPerson(name string) *person {  
    p := person{name: name}  
    p.age = 42  
    return &p  
}
```

Функция возвращает указатель на локальную переменную — это нормально. Go распознает такие ситуации, и выделяет память под структуру в куче (heap) вместо стека (stack), так что структура продолжит существовать после выхода из функции.

```
john := newPerson("John")  
fmt.Println(john)  
// &{John 42}
```

Если функция-конструктор возвращает саму структуру, а не указатель — принято использовать префикс `make` вместо `new`:

```
func makePerson(name string) person {  
    p := person{name: name}  
    p.age = 42  
    return p  
}
```

Доступ к полям структуры — через точку:

```
sean := person{name: "Sean", age: 50}  
fmt.Println(sean.name)  
// Sean
```

Чтобы получить доступ к полям структуры через указатель, не обязательно разыменовывать его через `*`. Эти два варианта эквивалентны:

```
sven := &person{name: "Sven", age: 50}  
fmt.Println((*sven).age)  
fmt.Println(sven.age)  
// 50
```

Поля структуры можно изменять:

```
sven.age = 51  
fmt.Println(sven.age)  
// 51
```

[песочница](#)

Шаг 2 · Структуры и методы · Stepik

Составные структуры

Структуры могут включать другие структуры:

```
type person struct {  
    firstName string  
    lastName  string  
}
```

```
type book struct {  
    title string  
    author person  
}
```

```
b := book{  
    title: "The Majik Gopher",  
    author: person{  
        firstName: "Christopher",  
        lastName:  "Swanson",  
    },  
}  
fmt.Println(b)  
// {The Majik Gopher {Christopher Swanson}}
```

Если вложенная структура не представляет самостоятельной ценности, можно даже не объявлять отдельный тип:

```
type user struct {  
    name string  
    karma struct {  
        value int  
        title string  
    }  
}
```

```
u := user{  
    name: "Chris",  
    karma: struct {  
        value int  
        title string  
    }{  
        value: 100,  
        title: "^-^",  
    },  
}
```

```
    },  
}  
fmt.Printf("%+v\n", u)  
// {name:Chris karma:{value:100 title:^-^}}
```

Благодаря шаблону `%+v`, `Printf()` печатает структуру вместе с названиями полей.

Поле структуры может ссылаться на другую структуру:

```
type comment struct {  
    text    string  
    author  *user  
}
```

```
chris := user{  
    name: "Chris",  
}  
c := comment{  
    text:  "Gophers are awesome!",  
    author: &chris,  
}  
fmt.Printf("%+v\n", c)  
// {text:Gophers are awesome! author:0xc0000981e0}
```

[песочница](#)

Шаг 3 · Структуры и методы · Stepik

Методы

Go позволяет определять *методы* на типах.

Метод отличается от обычной функции специальным параметром — *получателем*. В определении метода получатель указывается сразу после ключевого слова `func`. В данном случае — получатель типа `rect`:

```
type rect struct {  
    width, height int  
}  
  
func (r rect) area() int {  
    return r.width * r.height  
}
```

Метод вызывается для получателя через точку, как в питоне или js:

```
r := rect{width: 10, height: 5}
fmt.Println("rect area:", r.area())
// rect area: 50
```

Получателем может быть не значение заданного типа, а указатель на это значение:

```
type circle struct {
    radius int
}

func (c *circle) area() float64 {
    return math.Pi * math.Pow(float64(c.radius), 2)
}
```

```
cptr := &circle{radius: 5}
fmt.Println("circle area:", cptr.area())
// circle area: 78.54
```

При вызове метода Go автоматически преобразует значение получателя в указатель или указатель в значение, как того требует определение метода. Любой из перечисленных вариантов будет работать:

```
rpтр := &r
r.area()
rpтр.area()

c := *cptr
c.area()
cptr.area()
```

Считается хорошим тоном во всех методах использовать или только значение, или только указатель, но не смешивать одно с другим. Обычно используют указатель: так Go не приходится копировать всю структуру, а метод может ее изменить.

```
// Если метод принимает получателя как значение, изменить его не получится
func (r rect) scale(factor int) {
    r.width *= factor
    r.height *= factor
}

fmt.Println("rect before scaling:", r)
// rect before scaling: {10 5}

r.scale(2)
```

```
fmt.Println("rect after scaling:", r)
// rect after scaling: {10 5}

// Если метод принимает получателя как указатель, его можно изменить
func (c *circle) scale(factor int) {
    c.radius *= factor
}

fmt.Println("circle before scaling:", c)
// circle before scaling: {5}

c.scale(2)

fmt.Println("circle after scaling:", c)
// circle after scaling: {10}
```

[песочница](#)

Шаг 4 · Структуры и методы · Stepik

Определяемые типы

На предыдущем шаге мы создали структурный тип и методы для него. Но новый тип не обязательно создавать на основе структуры — можно использовать любые базовые типы.

Создадим тип «ИНН» на основе строки:

```
type inn string
```

Тип `inn` (он называется *определяемым типом*, defined type) получил свойства базового типа `string`. Добавим ему новое поведение с помощью метода:

```
func (id inn) isValid() bool {
    if len(id) != 12 {
        return false
    }
    for _, char := range id {
        if !unicode.IsDigit(char) {
            return false
        }
    }
    return true
}
```

```

inn1 := inn("111201284667")
fmt.Println("inn", inn1, "is valid:", inn1.isValid())
// inn 111201284667 is valid: true

inn2 := inn("ohmyinn12345")
fmt.Println("inn", inn2, "is valid:", inn2.isValid())
// inn ohmyinn12345 is valid: false

```

Это чем-то похоже на наследование, но механизм более примитивный. Если создать новый определяемый тип на основе `inn` — он унаследует структуру и свойства `inn`, но не методы:

```

type otherid inn

other := otherid("111201284667")
fmt.Println("other inn", other, "is valid:", other.isValid())
// ОШИБКА: other.isValid undefined

```

[песочница](#)

Метод-значение

Есть тип «счетчик»:

```

type counter struct {
    value uint
}

func (c *counter) increment() {
    c.value++
}

```

Можно создать значение `c` типа `counter` и вызвать метод `c.increment`:

```

c := new(counter)

c.increment()
c.increment()
c.increment()

fmt.Println(c.value)
// 3

```

А можно вызвать метод `c.increment` как функцию:

```

c := new(counter)
inc := c.increment

```

```
inc()
inc()
inc()

fmt.Println(c.value)
// 3
```

Здесь функция `inc` работает как замыкание — она имеет доступ к внутренним полям `c`, хотя снаружи они не видны.

Такая штука называется *метод-значение* (method value). Используется редко. Но может пригодиться, чтобы разрешить клиенту вызывать метод структуры, не давая при этом доступ к ее полям.

[песочница](#)

Метод-выражение

Можно сделать еще более причудливый финт. Использовать метод вообще без привязки к конкретному значению, как обычную функцию:

```
inc := (*counter).increment

first := new(counter)
inc(first)
inc(first)
inc(first)

second := new(counter)
inc(second)

fmt.Println(first.value)
// 3
fmt.Println(second.value)
// 1
```

Здесь функция `inc` принимает первым аргументом получателя метода — значение `x` типа `*counter` — и дальше работает как метод, увеличивая значение `x.value`.

Такая штука называется *метод-выражение* (method expression). На практике встречается еще реже, чем метод-значение.

[песочница](#)

Дополнительное чтение

[Struct](#)

[Properties of types and values](#) • [type declarations](#) • [conversions](#)

[Method declarations](#) • [method expressions](#) • [method values](#)

[Pointers vs. Values](#)

Шаг 5 · Структуры и методы · Stepik

Футбольный турнир

Четыре команды (A, B, C и D) сыграли футбольный турнир. Каждая команда сыграла с каждой по одному разу, так что всего прошло 6 матчей. Рассчитайте, сколько очков получила каждая команда.

Победа приносит 3 очка, ничья — 1, поражение — 0.

Результаты игр подаются на вход программы строкой вида:

```
ABW DCD DAW CBL BDL ACW
```

Каждая тройка букв означает одну игру. Внутри тройки первая буква — название первой команды, вторая буква — название второй, третья — код результата (W — первая выиграла, L — первая проиграла, D — ничья).

Код, который считывает и парсит результаты игр, уже написан. Ваша задача — реализовать структуры данных и логику подсчета результата.

Sample Input:

```
ABW DCD DAW CBL BDL ACW
```

Sample Output:

```
A6 B3 C1 D7
```

```
package main
```

```
import (  
    "bufio"  
    "fmt"  
    "io"  
    "log"  
    "os"
```



```

    "sort"
    "strings"
)

// result представляет результат матча
type result byte

// возможные результаты матча
const (
    win  result = 'W'
    draw result = 'D'
    loss result = 'L'
)

// team представляет команду
type team byte

// match представляет матч
// содержит три поля:
// - first (первая команда)
// - second (вторая команда)
// - result (результат матча)
// например, строке BAW соответствует
// first=B, second=A, result=W
type match struct {
    first team
    second team
    result result
}

// rating представляет турнирный рейтинг команд -
// количество набранных очков по каждой команде
type rating map[team]int

// tournament представляет турнир
type tournament []match

// calcRating считает и возвращает рейтинг турнира
func (trn *tournament) calcRating() rating {
    rt := make(rating)
    for _, match := range *trn {
        switch match.result {
            case win: {

```

```

        rt[match.first] += 3
        rt[match.second] += 0 // чтоб инициализировать 0 все
команды турнира
    }
    case loss: {
        rt[match.second] += 3
        rt[match.first] += 0
    }
    case draw: {
        rt[match.first]++
        rt[match.second]++
    }
}
}
return rt
}

//
// | не меняйте код ниже этой строки |
//

// код, который парсит результаты игр, уже реализован
// код, который печатает рейтинг, уже реализован
// ваша задача - реализовать недостающие структуры и методы выше
func main() {
    src := readString()
    trn := parseTournament(src)
    rt := trn.calcRating()
    rt.print()
}

// readString считывает и возвращает строку из os.Stdin
func readString() string {
    rdr := bufio.NewReader(os.Stdin)
    str, err := rdr.ReadString('\n')
    if err != nil && err != io.EOF {
        log.Fatal(err)
    }
    return str
}

// parseTournament парсит турнир из исходной строки
func parseTournament(s string) tournament {

```

```

descriptions := strings.Split(s, " ")
trn := tournament{}
for _, descr := range descriptions {
    m := parseMatch(descr)
    trn.addMatch(m)
}
return trn
}

// parseMatch парсит матч из фрагмента исходной строки
func parseMatch(s string) match {
    return match{
        first:  team(s[0]),
        second: team(s[1]),
        result: result(s[2]),
    }
}

// addMatch добавляет матч к турниру
func (t *tournament) addMatch(m match) {
    *t = append(*t, m)
}

// print печатает результаты турнира
// в формате Aw Bx Cy Dz
func (r *rating) print() {
    var parts []string
    for team, score := range *r {
        part := fmt.Sprintf("%c%d", team, score)
        parts = append(parts, part)
    }
    sort.Strings(parts)
    fmt.Println(strings.Join(parts, " "))
}

```

Шаг 6 · Структуры и методы · Stepik

Композиция

В Go нет наследования. Вместо него активно используется композиция — когда новое поведение собирают из кирпичиков существующего.

Есть тип «счетчик»:

```
type counter struct {  
    value uint  
}
```

Его можно увеличивать на единицу:

```
func (c *counter) increment() {  
    c.value++  
}
```

Или на указанное число:

```
func (c *counter) incrementDelta(delta uint) {  
    c.value += delta  
}
```

Мы хотим замерять использование сервисов. Чтобы не дублировать существующие функции, добавим счетчик в тип «использование сервиса»:

```
type usage struct {  
    service string  
    counter counter  
}  
  
func makeUsage(service string) usage {  
    return usage{service, counter{}}  
}
```

Будем мерить использование сервиса, увеличивая его счетчик:

```
usage := makeUsage("find")  
usage.counter.increment()  
usage.counter.increment()  
usage.counter.increment()  
fmt.Printf("%s usage: %d\n", usage.service, usage.counter.value)  
// find usage: 3
```

Для типа «просмотры страницы» тоже добавим счетчик:

```
type pageviews struct {  
    url *url.URL  
    counter counter  
}  
  
func makePageviews(uri string) pageviews {  
    u, err := url.Parse(uri)
```

```

    if err != nil {
        log.Fatal(err)
    }
    return pageviews{u, counter{}}
}

```

И будем мерить просмотры:

```

pv := makePageviews("/doc/find")
pv.counter.incrementDelta(100)
fmt.Printf("%s views: %d\n", pv.url, pv.counter.value)
// /doc/find views: 100

```

[песочница](#)

Шаг 7 · Структуры и методы · Stepik

Встраивание

Все хорошо, но несколько неудобно было писать `usage.counter.increment()` на предыдущем шаге. По-хорошему, `usage` расширяет `counter` — отношение между ними больше похоже на наследование, чем на композицию. В Go в таких случаях используют *встраивание* (embedding). Посмотрим, как оно работает.

Есть тип «счетчик», такой же, как на предыдущем шаге:

```

type counter struct {
    value uint
}
func (c *counter) increment() {
    c.value++
}
func (c *counter) incrementDelta(delta uint) {
    c.value += delta
}

```

Мы хотим замерять использование сервисов. *Встроим* счетчик в тип «использование сервиса»:

```

type usage struct {
    service string
    counter
}

func makeUsage(service string) usage {

```

```
    return usage{service, counter{}}
}
```

Благодаря встраиванию, поля и методы счетчика доступны прямо на `usage`, без обращения к полю `counter`:

```
usage := makeUsage("find")
usage.increment()
usage.increment()
usage.increment()
fmt.Printf("%s usage: %d\n", usage.service, usage.value)
// find usage: 3
```

Аналогично с типом «просмотры страниц»:

```
type pageviews struct {
    url *url.URL
    counter
}

func makePageviews(uri string) pageviews {
    u, err := url.Parse(uri)
    if err != nil {
        log.Fatal(err)
    }
    return pageviews{u, counter{}}
}
```

Поля и методы счетчика доступны прямо на `pageviews`:

```
pv := makePageviews("/doc/find")
pv.incrementDelta(100)
fmt.Printf("%s views: %d\n", pv.url, pv.value)
// /doc/find views: 100
```

[песочница](#)

Шаг 8 · Структуры и методы · Stepik

Валидатор пароля

Напишите программу, которая проверяет корректность пароля. Корректным считается пароль, который удовлетворяет любому из условий:

- содержит буквы и цифры;

- длина не менее 10 символов.

Следуйте указаниям по тексту программы. Не меняйте сигнатуры функций, определение типа `password` и переменную `validator` в `main()`.

Гарантируется, что на вход программы подается строка без пробелов.

Sample Input:

```
helloworld
```

Sample Output:

```
true
```

```
package main

import (
    "fmt"
    "unicode"
    "unicode/utf8"
)

// validator проверяет строку на соответствие некоторому условию
// и возвращает результат проверки
type validator func(s string) bool

// digits возвращает true, если s содержит хотя бы одну цифру
// согласно unicode.IsDigit(), иначе false
func digits(s string) bool {
    for _, ch := range s {
        if unicode.IsDigit(ch) {
            return true
        }
    }
    return false
}

// letters возвращает true, если s содержит хотя бы одну букву
// согласно unicode.IsLetter(), иначе false
func letters(s string) bool {
    for _, ch := range s {
        if unicode.IsLetter(ch) {
            return true
        }
    }
}
```

```

    }
    return false
}

// minlen возвращает валидатор, который проверяет, что длина
// строки согласно utf8.RuneCountInString() - не меньше указанной
func minlen(length int) validator {
    l := length
    return func(s string) bool {
        if utf8.RuneCountInString(s) >= l {
            return true
        }
        return false
    }
}

// and возвращает валидатор, который проверяет, что все
// переданные ему валидаторы вернули true
func and(funcs ...validator) validator {
    return func(s string) bool {
        for _, valid := range funcs {
            if !valid(s) {
                return false
            }
        }
        return true
    }
}

// or возвращает валидатор, который проверяет, что хотя бы один
// переданный ему валидатор вернул true
func or(funcs ...validator) validator {
    return func(s string) bool {
        for _, valid := range funcs {
            if valid(s) {
                return true
            }
        }
        return false
    }
}

// password содержит строку со значением пароля и валидатор

```



```

type password struct {
    value string
    validator
}

// isValid() проверяет, что пароль корректный, согласно
// заданному для пароля валидатору
func (p *password) isValid() bool {
    if p.validator(p.value) {
        return true
    } else {
        return false
    }
}

//
// | не меняйте код ниже этой строки |
//

func main() {
    var s string
    fmt.Scan(&s)
    // валидатор, который проверяет, что пароль содержит буквы и цифры,
    // либо его длина не менее 10 символов
    validator := or(and(digits, letters), or(minlen(10)))
    p := password{s, validator}
    fmt.Println(p.isValid())
}

```

Основы. 5. Интерфейсы

Шаг 1 · Интерфейсы · Stepik

Интерфейсы

Интерфейс в Go — это набор сигнатур методов (то есть список методов без реализации).

Интерфейс геометрической фигуры:

```

type geometry interface {
    area() float64
    perim() float64
}

```

Реализуем интерфейс в типе «прямоугольник». Реализовать интерфейс = реализовать его методы. Действует «утиный» принцип, как в питоне: если у типа есть перечисленные в интерфейсе методы — значит, он реализовал интерфейс. Явно указывать, что `rect` реализует `geometry`, не требуется:

```
type rect struct {
    width, height float64
}

func (r rect) area() float64 {
    return r.width * r.height
}

func (r rect) perim() float64 {
    return 2*r.width + 2*r.height
}
```

Аналогично для типа «круг»:

```
type circle struct {
    radius float64
}

func (c circle) area() float64 {
    return math.Pi * c.radius * c.radius
}

func (c circle) perim() float64 {
    return 2 * math.Pi * c.radius
}
```

Если у переменной интерфейсный тип, она поддерживает все методы, заданные на интерфейсе. Благодаря этому функция `measure()` работает с любой фигурой, реализующей интерфейс `geometry`:

```
func measure(g geometry) {
    fmt.Printf("%T: %+v\n", g, g)
    fmt.Println("area:", g.area())
    fmt.Println("perimeter:", g.perim())
}
```

Раз типы `circle` и `rect` реализуют интерфейс `geometry`, мы можем передать их экземпляры в функцию `measure()`:

```
r := rect{width: 3, height: 4}
c := circle{radius: 5}

measure(r)
// main.rect: {width:3 height:4}
// area: 12
// perimeter: 14

measure(c)
// main.circle: {radius:5}
// area: 78.53981633974483
// perimeter: 31.41592653589793
```

[песочница](#)

Шаг 2 · Интерфейсы · Stepik

Встраивание интерфейса

Иногда при композиции хочется дать доступ к поведению, но скрыть структуру. В этом поможет *встраивание интерфейса* (interface embedding).

Есть тип «счетчик»:

```
type counter struct {
    val uint
}
func (c *counter) increment() {
    c.val++
}
func (c *counter) value() uint {
    return c.val
}
```

Мы хотим встраивать счетчик в другие типы, но не давать прямой доступ к полю `val` — чтобы менять значение счетчика можно было только через методы.

Определим интерфейс счетчика:

```
type Counter interface {
    increment()
    value() uint
}
```

И вместо конкретного типа `counter` встроим интерфейс `Counter`, который он реализует:

```
type usage struct {
    service string
    Counter
}
```

В конструкторе будем создавать конкретное значение типа `counter`, но потребителям об этом знать не обязательно:

```
func newUsage(service string) *usage {
    return &usage{service, &counter{}}
}
```

Поскольку мы встроили интерфейс, прямого доступа к `counter.val` больше нет. Можно использовать только методы интерфейса:

```
usage := newUsage("find")
usage.increment()
usage.increment()
usage.increment()
fmt.Printf("%s usage: %d\n", usage.service, usage.value())
// find usage: 3
```

[песочница](#)

Шаг 3 · Интерфейсы · Stepik

Пустой интерфейс

Если у интерфейса нет методов, его называют *пустым* (empty):

```
interface{}
```

Пустой интерфейс может содержать значение любого типа (ведь у каждого типа есть как минимум 0 методов). Пустые интерфейсы используют, если тип значения заранее не известен. Например, функция из пакета `fmt`:

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

`fmt.Println()` умеет печатать что угодно, поэтому принимает значения типа `interface{}`.

Приведение типа

Приведение типа (type assertion) извлекает конкретное значение из переменной интерфейсного типа:

```
var ival interface{} = "hello"
str := ival.(string)
```

```
fmt.Println(str)
// hello
```

Если тип конкретного значения отличается от указанного, произойдет ошибка:

```
flo := ival.(float64)
// ошибка
```

Чтобы проверить тип конкретного значения, используют опциональный флаг, который сигнализирует — правильный тип или нет:

```
str, ok := ival.(string)
fmt.Println(str, ok)
// hello true

flo, ok = ival.(float64)
fmt.Println(flo, ok)
// 0 false
```

Переключатель типа

Приведение типа можно использовать вместе со `switch`. Такая конструкция называется *переключателем типа* (type switch):

```
switch ival.(type) {
case string:
    fmt.Println("It's a string")
case float64:
    fmt.Println("It's a float")
default:
    fmt.Println("It's a mystery")
}
// It's a string
```

[песочница](#)

Шаг 4 · Интерфейсы · Stepik

Универсальный итератор

У Go удобный синтаксис итерирования по срезам:

```
for idx, elem := range slice {
    // do stuff
}
```

Но бывает, что есть не чистый срез, а другой объект, который включает последовательность элементов:

- файл (последовательность — строки);
- таблица базы данных (последовательность — записи);
- API поиска (последовательность — страницы с результатами).

Удобно было бы работать с такими последовательностями единообразно, через *итератор*. Вот так:

```
func iterate(it iterator) {  
    for it.next() {  
        curr := it.val()  
        fmt.Println(curr)  
    }  
}
```

Определите интерфейс `iterator`, который содержит необходимые методы. Реализовывать интерфейс в конкретном типе не надо — этим займемся в следующем задании.

Sample Input:

```
1 2 3 4 5
```

Sample Output:

```
1  
2  
3  
4  
5
```

без понятия почему так, типы выставлены по логу ошибок в проверке

```
package main  
  
import (  
    "fmt"  
)  
  
// element - интерфейс элемента последовательности  
// (пустой, потому что элемент может быть любым).  
type element interface{}  
  
// iterator - интерфейс, который умеет
```

```
// поэлементно перебирать последовательность
type iterator interface {
    // определите методы итератора
    // чтобы понять сигнатуры методов - посмотрите,
    // как они используются в функции iterate() ниже
    next() bool
    val() element
}

// iterate обходит последовательность
// и печатает каждый элемент
func iterate(it iterator) {
    for it.next() {
        curr := it.val()
        fmt.Println(curr)
    }
}

// в этом задании функция main() определена "за кадром",
// не добавляйте ее
```

Дополнительное чтение

[Interfaces](#)

[Type assertions](#)

[Interfaces and other types](#)

[Embedding](#)

Шаг 5 · Интерфейсы · Stepik

Максимальный элемент последовательности

Определите интерфейс универсального итератора (`iterator`), который можно использовать в функции выбора максимального элемента (`max`). Реализуйте интерфейс для итератора по срезу целых чисел.

Подробности — по коду задания.

Если не понимаете, как подступиться к задаче, вот [подсказка](#).

Sample Input:

```
1 4 5 2 3
```

Sample Output:

5

[на основе](#)

```
package main

import (
    "bufio"
    "fmt"
    "log"
    "os"
    "strconv"
)

// element - интерфейс элемента последовательности
type element interface{}

// weightFunc - функция, которая возвращает вес элемента
type weightFunc func(element) int

// iterator - интерфейс, который умеет
// поэлементно перебирать последовательность
type iterator interface {
    // чтобы понять сигнатуры методов - посмотрите,
    // как они используются в функции max() ниже
    next() bool
    val() element
}

// intIterator - итератор по целым числам
// (реализует интерфейс iterator)
type intIterator struct {
    // поля структуры
    index int
    ints []int
}

// методы intIterator, которые реализуют интерфейс iterator
func (it *intIterator) next() bool {
    if it.index < len(it.ints) {
        return true
    }
}
```



```

    return false
}

func (it * intIterator) val() element {
    if it.next() {
        elem := it.ints[it.index]      // вариант [it.index++] в Go не
        работает
        it.index++
        return elem
    }
    return nil
}

// конструктор intIterator
func newIntIterator(src []int) *intIterator {
    return &intIterator{
        index: 0,
        ints: src,
    }
}

// 

не меняйте код ниже этой строки


//

// main находит максимальное число из переданных на вход программы.
func main() {
    nums := readInput()
    it := newIntIterator(nums)
    weight := func(el element) int {
        return el.(int)
    }
    m := max(it, weight)
    fmt.Println(m)
}

// max возвращает максимальный элемент в последовательности.
// Для сравнения элементов используется вес, который возвращает
// функция weight.
func max(it iterator, weight weightFunc) element {
    var maxEl element
    for it.next() {
        curr := it.val()

```

```

        if maxEl == nil || weight(curr) > weight(maxEl) {
            maxEl = curr
        }
    }
    return maxEl
}

// readInput считывает последовательность целых чисел из os.Stdin.
func readInput() []int {
    var nums []int
    scanner := bufio.NewScanner(os.Stdin)
    scanner.Split(bufio.ScanWords)
    for scanner.Scan() {
        num, err := strconv.Atoi(scanner.Text())
        if err != nil {
            log.Fatal(err)
        }
        nums = append(nums, num)
    }
    return nums
}

```

Основы. 6. Ошибки

Шаг 1 · Ошибки · Stepik

Ошибки

В Go нет исключений и блока try-catch, как в питоне или js. Вместо этого функции явно возвращают ошибку отдельным значением. Благодаря этому ошибки невозможно проигнорировать, а разработчики продумывают поведение программы в случае проблем.

Ошибки принято возвращать последним значением с интерфейсным типом `error`:

```

func sqrt(x float64) (float64, error) {
    if x < 0 {
        return 0, errors.New("expect x >= 0")
    }
    // `nil` в качестве ошибки указывает, что ошибок не было.
    return math.Sqrt(x), nil
}

```

Проверим работу `sqrt()` на положительном и отрицательном значениях. Обратите внимание, как мы получаем результат и проверяем ошибку внутри условия `if` — это стандартная практика в

Go.

```
for _, x := range []float64{49, -49} {
    if res, err := sqrt(x); err != nil {
        fmt.Printf("sqrt(%v) failed: %v\n", x, err)
    } else {
        fmt.Printf("sqrt(%v) = %v\n", x, res)
    }
}
// sqrt(49) = 7
// sqrt(-49) failed: expect x >= 0
```

[песочница](#)

Шаг 2 · Ошибки · Stepik

Собственный тип ошибки

Чтобы создать собственный тип ошибки, достаточно реализовать метод `Error()`.

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

Создадим ошибку, которая описывает проблему поиска `substr` в строке `src`:

```
type lookupError struct {
    src    string
    substr string
}

func (e *lookupError) Error() string {
    return fmt.Sprintf("%s' not found in '%s'", e.substr, e.src)
}
```

Напишем функцию `indexOf()`, которая возвращает индекс вхождения подстроки `substr` в строку `src`. Если вхождения нет, возвращает ошибку типа `lookupError`:

```
func indexOf(src string, substr string) (int, error) {
    idx := strings.Index(src, substr)
    if idx == -1 {
        // Создаем и возвращаем ошибку типа `lookupError`.
        return -1, &lookupError{src, substr}
    }
}
```

```
    return idx, nil
}
```

Проверим работу `indexOf()` для разных подстрок.

```
src := "go is awesome"
for _, substr := range []string{"go", "js"} {
    if res, err := indexOf(src, substr); err != nil {
        fmt.Printf("indexOf(%#v, %#v) failed: %v\n", src, substr, err)
    } else {
        fmt.Printf("indexOf(%#v, %#v) = %v\n", src, substr, res)
    }
}
// indexOf("go is awesome", "go") = 0
// indexOf("go is awesome", "js") failed: 'js' not found in 'go is awesome'
```

Поскольку `indexOf()` возвращает общий тип `error`, чтобы получить доступ к конкретному объекту ошибки, придется использовать приведение типа:

```
_, err := indexOf(src, "js")
if err, ok := err.(*lookupError); ok {
    fmt.Println("err.src:", err.src)
    fmt.Println("err.substr:", err.substr)
}
// err.src: go is awesome
// err.substr: js
```

[песочница](#)

Но можно также в сигнатуре метода `indexOf()` вместо `error` возвращать указатель на `lookupError`

```
func indexOf(src string, substr string) (int, *lookupError)
```

Шаг 3 · Ошибки · Stepik

Счет в банке

Напишите логику работы с лицевым счетом в банке. У счета есть баланс (сумма на счете) и овердрафт (на какую сумму можно уйти в минус). К счету последовательно применяются транзакции списания и пополнения. Транзакции изменяют баланс счета, овердрафт не меняется.

Со счета нельзя списать больше, чем (баланс + овердрафт). Зачислить или списать нулевую сумму тоже нельзя.

Например:

- начальное состояние счета: баланс = 100, овердрафт = 10
- транзакции: 10, -50, 20
- результат: баланс = 80, овердрафт = 10

Или:

- начальное состояние счета: баланс = 100, овердрафт = 10
- транзакции: -100, -10, -10
- результат: недостаточно средств на счете

Гарантируется, что сумма на счете, овердрафт и размер транзакции не выйдут за пределы типа `int`, как по отдельности, так и все вместе. Гарантируется, что овердрафт больше либо равен 0.

Sample Input:

```
{100 10} [10 -50 20]
```

Sample Output:

```
{80 10}
```

```
package main

import (
    "bufio"
    "errors"
    "fmt"
    "io"
    "log"
    "os"
    "strings"
)

// errInsufficientFunds сигнализирует,
// что на счете недостаточно денег,
// чтобы выполнить списание
var errInsufficientFunds error = errors.New("insufficient funds")

// errInvalidAmount сигнализирует,
// что указана некорректная сумма транзакции
var errInvalidAmount error = errors.New("invalid transaction amount")

// account представляет счет
type account struct {
    balance  int
```

```

    overdraft int
}

// deposit зачисляет деньги на счет.
func (acc *account) deposit(amount int) error {
    if amount != 0 {
        acc.balance += amount
        return nil
    } else {
        return errInvalidAmount
    }
}

// withdraw списывает деньги со счета.
func (acc *account) withdraw(amount int) error {
    if amount != 0 {
        if newbalance := acc.balance - amount; newbalance >= -acc.overdraft
        {
            acc.balance = newbalance
            return nil
        } else {
            return errInsufficientFunds
        }
    } else {
        return errInvalidAmount
    }
}

// 
// | не меняйте код ниже этой строки |
// 

type test struct {
    acc  account
    trans []int
}

var tests = map[string]test{
    "{100 10} [10 -50 20]": {account{100, 10}, []int{10, -50, 20}},
    "{30 0} [-20 -10]": {account{30, 0}, []int{-20, -10}},
    "{30 0}, [-20 -10 -10]": {account{30, 0}, []int{-20, -10, -10}},
    "{30 0}, [-100]": {account{30, 0}, []int{-100}},
    "{0 0}, [10 20 30]": {account{0, 0}, []int{10, 20, 30}},
}

```

```

    "{0 0}, [10 -10 20 -20]": {account{0, 0}, []int{10, -10, 20, -20}},
    "{20 10}, [-20 -10]":      {account{20, 10}, []int{-20, -10}},
    "{20 10}, [-20 -10 -10]": {account{20, 10}, []int{-20, -10, -10}},
    "{0 100}, [-20 -10]":      {account{0, 100}, []int{-20, -10}},
    "{0 30}, [-20 -10]":       {account{0, 30}, []int{-20, -10}},
    "{0 30}, [-20 -10 -10]":   {account{0, 30}, []int{-20, -10, -10}},
    "{70 30}, [-100 100]":      {account{70, 30}, []int{-100, 100}},
    "{100 10}, [10 0 20]":      {account{100, 10}, []int{10, 0, 20}},
}

```

```

func main() {
    var err error
    name, err := readString()
    if err != nil {
        log.Fatal(err)
    }
    testCase, ok := tests[name]
    if !ok {
        log.Fatalf("Test case '%v' not found", name)
    }
    for _, t := range testCase.trans {
        if t >= 0 {
            err = testCase.acc.deposit(t)
        } else {
            err = testCase.acc.withdraw(-t)
        }
        if err != nil {
            fmt.Println(err)
            break
        }
    }
    if err == nil {
        fmt.Println(testCase.acc)
    }
}

```

```

// readString считывает и возвращает строку из os.Stdin
func readString() (string, error) {
    rdr := bufio.NewReader(os.Stdin)
    str, err := rdr.ReadString('\n')
    if err != nil && err != io.EOF {
        return "", err
    }
}

```

```
    return strings.TrimSpace(str), nil
}
```

Шаг 4 · Ошибки · Stepik

Defer

Defer позволяет выполнить код сразу после того, как функция завершилась. Обычно его используют, чтобы освободить ресурсы, выделенные внутри функции (открытые файлы, соединения и тому подобное). В питоне в таких случаях применяют контекстные менеджеры, а в js конструкцию try-finally.

Допустим, мы хотим создать файл, записать в него что-то и закрыть. Вот как поможет `defer`:

```
func main() {
    f, err := createFile("/tmp/defer.txt")
    if err != nil {
        log.Fatal("Error creating file: ", err)
    }
    defer closeFile(f)
    if err := writeFile(f); err != nil {
        log.Fatal("Error writing to file: ", err)
    }
}
```

После того как файл открыт, мы с помощью `defer` указываем, что необходимо вызвать отложенную функцию `closeFile()` в самом конце `main()`, после завершения блока `if` с `writeFile()`. Причем отложенная функция отработает в любом случае — даже если во время записи в файл произошла ошибка.

```
func createFile(name string) (*os.File, error) {
    fmt.Println("Creating file...")
    f, err := os.Create(name)
    if err != nil {
        return nil, err
    }
    return f, nil
}
```

```
func writeFile(f *os.File) error {
    fmt.Println("Writing to file...")
    if _, err := fmt.Fprintln(f, "data"); err != nil {
        return err
    }
}
```



```
    return nil
}
```

Внутри отложенной функции тоже важно проверять ошибки:

```
func closeFile(f *os.File) {
    fmt.Println("Closing file...")
    err := f.Close()
    if err != nil {
        log.Fatal("Error closing file: ", err)
    }
}
```

[песочница](#)

Шаг 5 · Ошибки · Stepik

Panic

Если во время выполнения программы происходит неисправимая ошибка, срабатывает *паника* (panic). Это аналог исключения в питоне или js.

Допустим, мы написали функцию, которая возвращает символ строки по индексу, но забыли проверить, что индекс попадает в границы:

```
func getChar(str string, idx int) byte {
    return str[idx]
}
```

Если вызвать `getChar()` с некорректным индексом — сработает паника:

```
c := getChar("hello", 10)
// panic: runtime error: index out of range [10] with length 5
```

Панику можно вызвать и вручную с помощью одноименной встроенной функции:

```
panic("oops")
```

Так редко делают — в Go принято возвращать ошибку из функции, а не паниковать.

[песочница](#)

Шаг 6 · Ошибки · Stepik

Recover

Раз есть непредвиденные ошибки (паника), должен быть и способ их поймать. В Go для этого используется встроенная функция `recover()`. Посмотрим, как она работает.

Мы все так же забыли проверить, что индекс попадает в границы:

```
func getChar(str string, idx int) byte {  
    return str[idx]  
}
```

Но зная свою забывчивость, решили отловить любые непредвиденные ошибки:

```
func protect(fn func()) {  
    defer func() {  
        if err := recover(); err != nil {  
            fmt.Println("ERROR:", err)  
        } else {  
            fmt.Println("Everything went smoothly!")  
        }  
    }()  
    fn()  
}
```

`protect()` первым делом объявляет анонимную отложенную функцию, которая сработает после того, как будет выполнена `fn()`. Если срабатывает паника, вызывается отложенная функция. Внутри нее `recover()` возвращает ошибку, которая вызвала панику. Если паники не было, отложенная функция тоже вызывается, но `recover()` внутри возвращает `nil`.

Здесь сработает паника:

```
protect(func() {  
    c := getChar("hello", 10)  
    fmt.Println("hello[10] = ", c)  
})  
// ERROR: runtime error: index out of range [10] with length 5
```

А здесь функция отработает без ошибок:

```
protect(func() {  
    c := getChar("hello", 4)  
    fmt.Println("hello[4] =", c)  
})  
// hello[4] = 111  
// Everything went smoothly!
```

Возможно, вы заметили, что ручной вызов `panic()` в сочетании с `defer()` и `recover()` можно использовать, чтобы эмулировать конструкцию try-catch. В Go так не принято. Всегда старайтесь

явно возвращать ошибки из функции, а на вызывающей стороне проверять их.

[песочница](#)

Шаг 7 · Ошибки · Stepik

От паники к ошибкам

Я написал программу, которая считывает исходную строку вида:

```
balance/overdraft t1 t2 t3 ... tn
```

И превращает ее в счет типа `account` и список транзакций типа `[]int`.

К сожалению, я поленился явно обработать ошибки, положившись на отлов паники через `defer` и `recover()`. Исправьте программу, заменив их на явный возврат и обработку ошибок.

В получившейся программе не должно быть конструкций `defer` и `recover()`. Если в исходной строке есть ошибка — программа должна вывести только ошибку. Иначе — вывести счет и транзакции.

Например:

- на входе `80/10 10 -20 30`
- на выходе `-> {80 10} [10 -20 30]`

Или:

- на входе `80/10 10 z 30`
- на выходе `-> strconv.Atoi: parsing "z": invalid syntax`

Если в исходной строке несколько ошибок, программа должна вывести только первую.

Не меняйте текст ошибок, используйте те же строки, что принимали вызовы `panic()` в исходной программе.

Гарантируется, что общий формат исходной строки соблюдается (то есть значения разделены пробелами, овердрафт отделен от баланса дробью, и тому подобное), но могут быть ошибки в отдельных значениях (как в примере выше).

Программа не должна завершаться паникой или `os.Exit(1)` ни при каких обстоятельствах.

Sample Input:

```
80/10 10 -20 30
```

Sample Output:

```
-> {80 10} [10 -20 30]
```

```
package main
```

```
import (  
    "bufio"  
    // раскомментируйте строчку ниже,  
    // чтобы создавать ошибки через errors.New()  
    "errors"  
    "fmt"  
    "os"  
    "strconv"  
    "strings"  
)
```

```
// account представляет счет
```

```
type account struct {  
    balance    int  
    overdraft int  
}
```

```
func main() {  
    var acc account  
    var trans []int  
    var err error  
    //defer func() {  
    //    fmt.Print("-> ")  
    //    err := recover()  
    //    if err != nil {  
    //        fmt.Println(err)  
    //        return  
    //    }  
    //    fmt.Println(acc, trans)  
    //}()  
    acc, trans, err = parseInput()  
    fmt.Print("-> ")  
    if err != nil {  
        fmt.Println(err)  
        //return  
    } else { fmt.Println(acc, trans) }  
}
```

```
// parseInput считывает счет и список транзакций из os.Stdin.
```

```

func parseInput() (account, []int, error) {
    accSrc, transSrc := readInput()
    acc, err := parseAccount(accSrc)
    if err != nil { return account{0, 0}, nil, err }
    trans, err := parseTransactions(transSrc)
    if err != nil { return account{0, 0}, nil, err }
    return acc, trans, err
}

// readInput возвращает строку, которая описывает счет
// и срез строк, который описывает список транзакций.
// эту функцию можно не менять
func readInput() (string, []string) {
    scanner := bufio.NewScanner(os.Stdin)
    scanner.Split(bufio.ScanWords)
    scanner.Scan()
    accSrc := scanner.Text()
    var transSrc []string
    for scanner.Scan() {
        transSrc = append(transSrc, scanner.Text())
    }
    return accSrc, transSrc
}

// parseAccount парсит счет из строки
// в формате balance/overdraft.
func parseAccount(src string) (account, error) {
    var err error
    parts := strings.Split(src, "/")
    balance, err := strconv.Atoi(parts[0])
    if err != nil { return account{0, 0}, err }
    overdraft, err := strconv.Atoi(parts[1])
    if err != nil { return account{0, 0}, err }

    //var err error
    if overdraft < 0 {
        //panic("expect overdraft >= 0")
        err = errors.New("expect overdraft >= 0")
        if err != nil { return account{0, 0}, err }
    }
    if balance < -overdraft {
        //panic("balance cannot exceed overdraft")
        err = errors.New("balance cannot exceed overdraft")
    }
}

```

```

        if err != nil { return account{0, 0}, err }
    }
    return account{balance, overdraft}, err
}

// parseTransactions парсит список транзакций из строки
// в формате [t1 t2 t3 ... tn].
func parseTransactions(src []string) ([]int, error) {
    trans := make([]int, len(src))
    var err error
    for idx, s := range src {
        t, err := strconv.Atoi(s)
        if err != nil { return nil, err }
        trans[idx] = t
    }
    return trans, err
}

```

Обертывание ошибок

Допустим, есть функция, которая извлекает значение по ключу из карты и возвращает ошибку, если ключ не найден:

```

var errNotFound error = errors.New("not found")

func getValue(m map[string]string, key string) (string, error) {
    val, ok := m[key]
    if !ok {
        return "", errNotFound
    }
    return val, nil
}

```

И есть структура `languages` с информацией о языках. Она возвращает описание языка по названию:

```

type languages map[string]string

func (l languages) describe(lang string) (string, error) {
    descr, err := getValue(l, lang)
    if err != nil {
        return "", err
    }
    return descr, nil
}

```

```

var langs languages = languages{
    "go":      "is awesome",
    "python":  "is everywhere",
    "php":     "just is",
}

func main() {
    descr, err := langs.describe("java")
    if err != nil {
        fmt.Println(err)
        return
    }
    fmt.Println(descr)
}

```

Внутри у `languages` карта, а метод `languages.describe()` использует `getValue()`, чтобы получить информацию о языке по названию. Получив ошибку, метод транслирует ее клиенту. В примере для `"java"` напечатается такой результат:

```
not found
```

Формально все верно. Но `errNotFound` — низкоуровневая ошибка общего назначения. Она ничего не говорит о проблеме с поиском языка. Клиент хотел бы больше информации.

Можно создать новую ошибку в `describe()` с помощью `fmt.Errorf()`:

```

func (l languages) describe(lang string) (string, error) {
    descr, err := getValue(l, lang)
    if err != nil {
        return "", fmt.Errorf("error describing %s: unknown language", lang)
    }
    return descr, nil
}

```

```
error describing java: unknown language
```

Но создав новую ошибку, мы потеряли информацию о первоначальной `errNotFound`. Вдруг клиенту она важна?

Можно обернуть (*wrap*) исходную ошибку в новую с помощью `fmt.Errorf()` и спецификатора `%w`:

```

func (l languages) describe(lang string) (string, error) {
    descr, err := getValue(l, lang)
    if err != nil {

```

```

        return "", fmt.Errorf("error describing %s: %w", lang, err)
    }
    return descr, nil
}

```

Теперь метод возвращает ошибку-матрешку: снаружи у нее информативная `error describing...`, а внутри исходная `errNotFound`. В сложных программах таких «обертываний» может быть много, пока ошибка поднимается от самых нижних слоев кода к уровню API или UI.

Обертывание собственных ошибок

Если вместо `fmt.Errorf()` мы захотим использовать собственный тип ошибки — дело усложнится. Допустим, хотим записывать в ошибку название языка отдельным полем:

```

type languageErr struct {
    lang string
}

func (le languageErr) Error() string {
    return fmt.Sprintf("%s language error", le.lang)
}

```

Чтобы `languageErr` могла выступать оберткой для других ошибок, придется сделать еще две вещи:

1. Добавить отдельное поле для внутренней ошибки (ее будем оборачивать).
2. Добавить метод `Unwrap()`, который возвращает внутреннюю ошибку («разворачивает»).

```

type languageErr struct {
    lang string
    err  error
}

func (le languageErr) Error() string {
    return fmt.Sprintf("%s language error: %v", le.lang, le.err)
}

func (le languageErr) Unwrap() error {
    return le.err
}

```

Теперь можно создать новую `languageErr` как обертку над исходной ошибкой в методе `describe()`:

```

func (l languages) describe(lang string) (string, error) {
    descr, err := getValue(l, lang)

```



```

    if err != nil {
        return "", languageErr{lang, err}
    }
    return descr, nil
}

```

Сама по себе слоеная ошибка — только половина дела. Вторая половина — научиться клиенту с ней работать. Сейчас посмотрим, как.

errors.Is()

Получив ошибку-матрешку, клиент может проверить, есть ли на каком-то слое интересующая его проблема. Для этого используют функцию `errors.Is()`:

```

func (l languages) describe(lang string) (string, error) {
    descr, err := getValue(l, lang)
    if err != nil {
        return "", languageErr{lang, err}
    }
    return descr, nil
}

// ...

func main() {
    descr, err := langs.describe("java")
    if errors.Is(err, errNotFound) {
        fmt.Println("this is an errNotFound error")
        // do something about it...
    }
    if err != nil {
        fmt.Println(err)
        return
    }
    fmt.Println(descr)
}

```

```

this is an errNotFound error
java language error: not found

```

Неважно, сколько в ошибке слоев. Если на каком-то из них встретилось значение `errNotFound` — `errors.Is()` вернет `true`.

[песочница](#)

errors.As()

Раньше мы использовали приведение типа, чтобы получить доступ к ошибке конкретного типа вместо абстрактного `error`:

```
descr, err := langs.describe("java")
if langErr, ok := err.(languageErr); ok {
    fmt.Println("Language error:", langErr.lang)
}
```

Language error: java

Но это работает только для ошибки самого верхнего уровня. До ошибки из середины «матрешки» через приведение типа не добраться. А вот через `errors.As()` — можно:

```
descr, err := langs.describe("java")
// обернем еще раз, чтобы languageErr
// оказалась внутри матрешки
err = fmt.Errorf("wrap once more: %w", err)

var langErr languageErr
if errors.As(err, &langErr) {
    fmt.Println("Language error:", langErr.lang)
}
```

Language error: java

wrap once more: java language error: not found

`errors.As()` проверяет каждый слой ошибки, и если видит там искомый тип `languageErr` — заполняет значение `langErr` по переданному указателю, и возвращает `true`. Если искомого типа нет — возвращает `false`.

[песочница](#)

Итого по обертыванию:

- Простой способ создать новую ошибку на основе существующей — `fmt.Errorf()` и спецификатор `%w`
- Если нужен собственный тип ошибки, придется добавить в него поле типа `error` и метод `Unwrap()`
- `errors.Is()` проверяет конкретную ошибку на каждом слое.
- `errors.As()` заполняет ошибку конкретного типа, если он встречается на одном из слоев.

Дополнительное чтение

Спецификация: [errors](#) • [defer](#) • [panic / recover](#)

[Defer, Panic, and Recover](#)

[Effective Go: Errors](#)

[Errors are values](#)

[Working with Errors in Go](#)

Закрепим задачей.

Игра без права на ошибку

В совсем маленьких программах обертывать ошибки обычно не требуется. Поэтому сразу скажу, что задача эта непривычно большая. Заодно попрактикуетесь в работе с унаследованным кодом на Go. Поскольку это финальная задача модуля — можно считать ее толстым боссом ٩

Мы будем писать игру, в которой любая ошибка приводит к трагическому финалу. Сначала пройдем по коду, а затем я расскажу, что требуется сделать.

Команда, объект, шаг

В нашей игре участник выполняет некоторые команды над объектами игрового мира. Поэтому есть тип `command` и список доступных команд:

```
// label - уникальное наименование
type label string

// command - команда, которую можно выполнять в игре
type command label

// список доступных команд
var (
    eat  = command("eat")
    take = command("take")
    talk = command("talk to")
)
```

Есть объекты игрового мира — тип `thing`. Каждый объект знает, какие команды для него доступны:

```
// thing - объект, который существует в игре
type thing struct {
    name    label
    actions map[command]string
}
```

```
// supports() возвращает true, если объект
// поддерживает команду action
func (t thing) supports(action command) bool {
    _, ok := t.actions[action]
    return ok
}
```

Список объектов:

```
// полный список объектов в игре
var (
    apple = thing{"apple", map[command]string{
        eat:  "mmm, delicious!",
        take: "you have an apple now",
    }}
    bob = thing{"bob", map[command]string{
        talk: "Bob says hello",
    }}
    coin = thing{"coin", map[command]string{
        take: "you have a coin now",
    }}
    mirror = thing{"mirror", map[command]string{
        take: "you have a mirror now",
        talk: "mirror does not answer",
    }}
    mushroom = thing{"mushroom", map[command]string{
        eat:  "tastes funny",
        take: "you have a mushroom now",
    }}
)
```

Есть тип `step` — шаг игры. Он объединяет команду и объект, над которым необходимо выполнить действие:

```
// step описывает шаг игры: сочетание команды и объекта
type step struct {
    cmd command
    obj thing
}

// isValid() возвращает true, если объект
// совместим с командой
func (s step) isValid() bool {
```

```
    return s.obj.supports(s.cmd)
}
```

Игрок

Есть игрок — тип `player`:

```
// player - игрок
type player struct {
    // количество съеденного
    nEaten int
    // количество диалогов
    nDialogs int
    // инвентарь
    inventory []thing
}
```

Игрок умеет проверить, есть ли предмет в инвентаре:

```
// has() возвращает true, если у игрока
// в инвентаре есть предмет obj
func (p *player) has(obj thing) bool {
    for _, got := range p.inventory {
        if got.name == obj.name {
            return true
        }
    }
    return false
}
```

И выполнить действие над указанным объектом:

```
// do() выполняет команду cmd над объектом obj
// от имени игрока
func (p *player) do(cmd command, obj thing) error {
    // действуем в соответствии с командой
    switch cmd {
    case eat:
        if p.nEaten > 1 {
            return errors.New("you don't want to eat anymore")
        }
        p.nEaten++
    case take:
        if p.has(obj) {
            return fmt.Errorf("you already have a %s", obj)
        }
    }
}
```

```

    }
    p.inventory = append(p.inventory, obj)
case talk:
    if p.nDialogs > 0 {
        return errors.New("you don't want to talk anymore")
    }
    p.nDialogs++
}
return nil
}

```

Как видите, тут кое-что может пойти не так:

- игрок уже ел или разговаривал, и больше не хочет,
- предмет уже есть в инвентаре.

На каждую такую ситуацию метод `do()` возвращает ошибку.

Игра

Наконец, есть тип `game` — сама игра:

```

// game описывает игру
type game struct {
    // игрок
    player *player
    // объекты игрового мира
    things map[label]int
    // количество успешно выполненных шагов
    nSteps int
}

```

Исходно в игре есть некоторое количество предметов каждого типа (поле `things`). Игра умеет проверить, остались ли они или уже закончились:

```

// has() проверяет, остались ли в игровом мире указанные предметы
func (g *game) has(obj thing) bool {
    count := g.things[obj.name]
    return count > 0
}

```

Игра может выполнить шаг:

```

// execute() выполняет шаг step
func (g *game) execute(st step) error {
    // проверяем совместимость команды и объекта

```

```

if !st.isValid() {
    return fmt.Errorf("cannot %s", st)
}

// когда игрок берет или съедает предмет,
// тот пропадает из игрового мира
if st.cmd == take || st.cmd == eat {
    if !g.has(st.obj) {
        return fmt.Errorf("there are no %ss left", st.obj)
    }
    g.things[st.obj.name]--
}

// выполняем команду от имени игрока
if err := g.player.do(st.cmd, st.obj); err != nil {
    return err
}

g.nSteps++
return nil
}

```

В методе `execute()` последовательность действий такая:

1. Игра проверяет, что шаг корректный.
2. Игра проверяет, остались ли предметы, с которыми требуется выполнить действие.
3. Игра делегирует выполнение шага игроку.

Если что-то пошло не так — метод возвращает ошибку.

Новая игра выглядит так:

```

// newGame() создает новую игру
func newGame() *game {
    p := newPlayer()
    things := map[label]int{
        apple.name: 2,
        coin.name: 3,
        mirror.name: 1,
        mushroom.name: 1,
    }
    return &game{p, things, 0}
}

```

Пример игры с конкретной последовательностью шагов:

```
func main() {
    gm := newGame()
    steps := []step{
        {eat, apple},
        {talk, bob},
        {take, coin},
        {eat, mushroom},
    }

    for _, st := range steps {
        if err := tryStep(gm, st); err != nil {
            fmt.Println(err)
            os.Exit(1)
        }
    }
    fmt.Println("You win!")
}

// tryStep() выполняет шаг игры и печатает результат
func tryStep(gm *game, st step) error {
    // ...
}
```

Здесь шаги приводят к успеху:

```
trying to eat apple... OK
trying to talk to bob... OK
trying to take coin... OK
trying to eat mushroom... OK
You win!
```

А здесь нет:

```
func main() {
    gm := newGame()
    steps := []step{
        {talk, bob},
        {talk, bob},
    }
    // ...
}
```



```
trying to talk to bob... OK
trying to talk to bob... FAIL
you don't want to talk anymore
exit status 1
```

Задание

Как видно по коду, ошибки создаются через `errors.New()` и `fmt.Errorf()`. Хочется больше структуры, поэтому добавьте отдельный тип для каждого вида ошибок:

```
// invalidStepError - ошибка, которая возникает,
// когда команда шага не совместима с объектом
type invalidStepError

// notEnoughObjectsError - ошибка, которая возникает,
// когда в игре закончились объекты определенного типа
type notEnoughObjectsError

// commandLimitExceededError - ошибка, которая возникает,
// когда игрок превысил лимит на выполнение команды
type commandLimitExceededError

// objectLimitExceededError - ошибка, которая возникает,
// когда игрок превысил лимит на количество объектов
// определенного типа в инвентаре
type objectLimitExceededError
```

Метод `player.do()` должен возвращать либо одну из этих ошибок, либо `nil` (если ошибок не было).

Кроме того, создайте ошибку верхнего уровня:

```
// gameOverError - ошибка, которая произошла в игре
type gameOverError struct {
    // количество шагов, успешно выполненных
    // до того, как произошла ошибка
    nSteps int
    // ...
}
```

Метод `game.execute()` должен возвращать либо ошибку типа `gameOverError`, либо `nil` (если ошибок не было).

- Если метод получил ошибку от `player.do()` — пусть обернет ее в `gameOverError`.

- Если ошибка произошла в самом `game.execute()` — пусть создаст ошибку подходящего типа, а затем обернет ее в `gameOverError`.

И последнее. Создайте функцию `giveAdvice()`, которая дает игроку совет, как избежать случившейся ошибки в будущем:

```
func giveAdvice(err error) string {  
    // ...  
}
```

Правила работы `giveAdvice()`:

- Если команда не совместима с объектом, возвращает `things like 'COMMAND OBJECT' are impossible`, где `COMMAND` — название команды, а `OBJECT` — название объекта.
- Если в игре закончились объекты определенного типа, возвращает `be careful with scarce OBJECTs`, где `OBJECT` — название объекта.
- Если игрок слишком много ел, возвращает `eat less`. Если игрок слишком много говорил, возвращает `talk to less`.
- Если игрок превысил лимит на количество объектов определенного типа в инвентаре, возвращает `don't be greedy, LIMIT OBJECT is enough`, где `LIMIT` — значение лимита, а `OBJECT` — название объекта.

Например:

```
things like 'eat bob' are impossible  
be careful with scarce mirrors  
don't be greedy, 1 apple is enough
```

Итого

1. Создайте отдельные типы ошибок и возвращайте ошибки этих типов в подходящих случаях.
2. Создайте тип `gameOverError` и возвращайте ошибку этого типа из `game.execute()`.
3. Создайте функцию `giveAdvice()`, которая возвращает совет на основе ошибки.

Полный код программы — в песочнице.

Важно: в качестве решения отправляйте не весь код, а только фрагмент, отмеченный комментариями «начало решения» и «конец решения».

[песочница](#)

Sample Input:

Sample Output:

PASS

Чистый код. Резюме

Шаг 1 · Резюме · Stepik

Пакеты и модули

Пакет в Go — это набор исходных файлов, которые находятся в одном каталоге и компилируются вместе. При этом функции, типы, переменные и константы, определенные в одном файле пакета, доступны всем файлам из этого пакета.

Модуль в Go — это набор логически связанных пакетов, которые поставляются вместе.

`go mod init` инициализирует модуль и создает файл с описанием модуля `go.mod`:

```
$ go mod init github.com/gothanks/match
$ cat go.mod
module github.com/gothanks/match

go 1.16
```

`go build` собирает модуль в исполняемый файл:

```
$ go build
$ ./match -p is 'go is awesome'
go is awesome
```

`go get` скачивает и подключает внешние зависимости, а также обновляет patch- и minor- версии:

```
$ go get github.com/sahilm/fuzzy
$ go get -u=patch github.com/sahilm/fuzzy
$ go get -u github.com/sahilm/fuzzy
```

Пакеты импортируются по полному имени с идентификатором модуля, а в коде доступны по локальному имени:

```
package main

import (
    "github.com/gothanks/match/glob"
)

func main() {
    // ...
    isMatch, err := glob.Match(pattern, src)
```

```
// ...  
}
```

Шаг 2 · Резюме · Stepik

Тесты

Тест — это функция с префиксом `Test`, которая принимает указатель на `testing.T`:

```
func TestIntMin(t *testing.T) {  
    got := IntMin(2, -2)  
    want := -2  
    if got != want {  
        t.Errorf("got %d; want %d", got, want)  
    }  
}
```

`t.Run()` запускает вложенные тесты:

```
func TestIntMin(t *testing.T) {  
    var tests = []struct {...}  
    for _, test := range tests {  
        name := ...  
        t.Run(name, func(t *testing.T) {  
            got := IntMin(test.a, test.b)  
            if got != test.want {  
                t.Errorf("got %d, want %d", got, test.want)  
            }  
        })  
    }  
}
```

Для подготовки и завершения тестов используют функцию `TestMain()`:

```
func TestMain(m *testing.M) {  
    // setup  
  
    // run tests  
    m.Run()  
  
    // tear down  
}
```

Внешние вызовы тестируют через *заглушки*. Если есть внешний ресурс `T` с методом `T.m()`, то создают интерфейс `I` с методом `I.m()` и заглушку `MT` с методом `MT.m()`. После чего в

основном коде используют интерфейс `I` с реализацией `T`, а в тестах — интерфейс `I` с реализацией `MT`.

Тесты пишут в отдельном файле с суффиксом `_test`.

`go test` запускает тесты:

```
$ go test
$ go test -v
$ go test -run=TestIntMin
```

`go test -cover` замеряет тестовое покрытие:

```
$ go test -cover
$ go test -coverprofile=cover.prof
$ go tool cover -html=cover.prof
```

Шаг 3 · Резюме · Stepik

Бенчмарки

Бенчмарки замеряют время выполнения и использование памяти. Они похожи на обычные тесты, только начинаются со слова `Benchmark` вместо `Test`, и принимают параметр `testing.B` вместо `testing.T`:

```
func BenchmarkMatchContains(b *testing.B) {
    for n := 0; n < b.N; n++ {
        MatchContains(pattern, src)
    }
}
```

`go test -bench` запускает бенчмарки:

```
$ go test -bench=.
$ go test -bench=BenchmarkFields
$ go test -run=! -bench=.
$ go test -bench=. -benchmem
```

Regexp-10-8	541161	2268 ns/op	1503 B/op
15 allocs/op			
Regexp-100-8	50229	23702 ns/op	22686 B/op
115 allocs/op			
Regexp-1000-8	4534	261560 ns/op	240035 B/op
1050 allocs/op			
Regexp-10000-8	270	4572228 ns/op	2497343 B/op
10302 allocs/op			

Результаты бенчмарка показывают:

- количество прогонов;
- среднее время выполнения одного прогона;
- среднее количество байт, которые пришлось выделить за один прогон;
- сколько в среднем раз приходилось выделять память за один прогон.

Шаг 4 · Резюме · Stepik

Профайлер

Профилирование помогает найти «узкие места» в коде — функции, которые потребляют больше всего процессорного времени или памяти.

Как профилировать:

1. Написать код.
2. Написать бенчмарк на код.
3. Прогнать бенчмарк с включенным профайлером.
4. По собранному профилю определить проблемные участки кода и оптимизировать их.
5. Повторить.

`go test -bench` с опциями `cpuprofile` или `memprofile` выполняет бенчмарк с включенным профайлером:

```
$ GOGC=off go test -bench=. -cpuprofile=cpu.prof
$ GOGC=off go test -bench=. -memprofile=mem.prof
```

`go tool pprof` запускает утилиту просмотра профиля:

```
$ go tool pprof cpu.prof
$ go tool pprof mem.prof
```

Три полезных команды внутри `pprof`:

- `topN` — какие функции потребляют больше всего процессора или памяти (без учета дочерних);
- `topN -cum` — какие функции потребляют больше всего процессора или памяти (с учетом дочерних);
- `peek` — разбивка использования процессора или памяти по дочерним функциям.

```
(pprof) top10
(pprof) top10 -cum
(pprof) peek words.UniqWords
```

Чистый код. 1 .Пакеты и модули

Шаг 1 · Пакеты и модули · Stepik

Если вы пришли из мира питона или JS, то привыкли, что *модуль* — это конкретный файл с исходным кодом, а *пакет* — набор логически связанных модулей, которые поставляются вместе. К сожалению, в Go все не так.

Пакет в Go — это набор исходных файлов, которые находятся в одном каталоге и компилируются вместе. При этом функции, типы, переменные и константы, определенные в одном файле пакета, доступны всем файлам из этого пакета. Таким образом, привычной области видимости «на уровне файла» не существует.

Модуль в Go — это набор логически связанных пакетов, которые поставляются вместе. Наверно, более правильный по смыслу термин был бы «проект», но авторы решили назвать его «модулем». Модуль хранится в репозитории исходного кода (гитхаб, гитлаб, или что вам ближе). Как правило, один репозиторий = один модуль.

Разберемся на примере.

Напишем утилиту `match`, которая проверяет строку на соответствие шаблону:

```
$ ./match -p is 'go is awesome'
go is awesome

$ ./match -p was 'go is awesome'
(empty)
```

Шаг 2 · Пакеты и модули · Stepik

Новый модуль

Модуль инициализируется командой `go mod init`:

```
$ mkdir match
$ cd match
$ go mod init github.com/gothanks/match
go: creating new go.mod: module github.com/gothanks/match
$ touch main.go
```

`github.com/gothanks/match` — это глобально уникальный идентификатор модуля и одновременно URL, по которому он должен быть доступен (за минусом префикса `https://`).

Если будете повторять шаги урока локально, репозиторий на гитхабе можно не создавать. Пока никто не использует ваш модуль, размещать его публично не требуется.

Наш модуль:

```
match/  
├─ go.mod  
└─ main.go
```

`go.mod` описывает модуль: идентификатор, минимальная версия Go, зависимости (их пока нет):

```
$ cat go.mod  
module github.com/gothanks/match  
  
go 1.16
```

Наш модуль содержит пакет с единственным файлом `main.go`. Реализуем в нем логику программы:

```
// match tool checks a string against a pattern.  
// If it matches - prints the string, otherwise prints nothing.  
package main  
  
import (  
    "errors"  
    "flag"  
    "fmt"  
    "os"  
    "strings"  
)  
  
func main() {  
    pattern, src, err := readInput()  
    if err != nil {  
        fail(err)  
    }  
    isMatch := match(pattern, src)  
    if !isMatch {  
        os.Exit(0)  
    }  
    fmt.Println(src)  
}  
  
// match returns true if src matches pattern,  
// false otherwise.  
func match(pattern string, src string) bool {  
    return strings.Contains(src, pattern)  
}
```



```

}

// readInput reads pattern and source string
// from command line arguments and returns them.
func readInput() (pattern, src string, err error) {
    flag.StringVar(&pattern, "p", "", "pattern to match against")
    flag.Parse()
    if pattern == "" {
        return pattern, src, errors.New("missing pattern")
    }
    src = strings.Join(flag.Args(), "")
    if src == "" {
        return pattern, src, errors.New("missing string to match")
    }
    return pattern, src, nil
}

// fail prints the error and exits.
func fail(err error) {
    fmt.Println("match:", err)
    os.Exit(1)
}

```

Если программа компилируется в исполняемый файл, у нее должен быть пакет с названием `main`. Если бы мы делали библиотеку — назвали бы файл `match.go`, а пакет `match`.

Шаг 3 · Пакеты и модули · Stepik

Сборка и установка

`go build` собирает модуль в исполняемый файл:

```

$ go build
$ ./match -p is 'go is awesome'
go is awesome

```

`go install` устанавливает модуль в домашний каталог Go (`$HOME/go` для Linux/macOS, `%USERPROFILE%\go` для Windows):

```

$ go install
$ ~/go/bin/match -p is 'go is awesome'
go is awesome

```

Go компилирует программу в единый исполняемый файл. Чтобы ее запустить, не требуется устанавливать сам Go или зависимости. Копируете бинарник на целевую машину и запускаете,

это все.

Бинарник не кросс-платформенный: чтобы запустить Go-программу на конкретной целевой архитектуре, придется собрать специально под нее. Не будем разбирать это на курсе, но если интересно — вот статья с подробностями: [Building Go Applications for Different Operating Systems and Architectures](#).

Шаг 4 · Пакеты и модули · Stepik

Странный модуль

Есть такой модуль:

```
less
├─ go.sum
└─ less.go

// -- less/less.go --
package main

import "fmt"

func main() {
    fmt.Println("more is less")
}
```

Какие проблемы мешают собрать модуль через `go build`?

Выберите все подходящие ответы из списка

☐ Нет критических проблем

☐ Не хватает файла `go.mod` с описанием модуля

☐ `less.go` должен называться `main.go`

☐ Вместо `package main` должно быть `package less`

Шаг 5 · Пакеты и модули · Stepik

Пакет из нескольких файлов

В первой версии логика функции `match()` довольно примитивная:

```
func match(pattern string, src string) bool {
    return strings.Contains(src, pattern)
}
```

```
}
```

Доработаем утилиту. Чтобы не перегружать файл `main.go`, вынесем логику сравнения в отдельный файл `match.go` в том же пакете:

```
match
```

```
├─ go.mod
├─ main.go
└─ match.go
```

```
// -- match/match.go --
```

```
package main
```

```
import "strings"
```

```
// match returns true if src matches pattern,  
// false otherwise.
```

```
func match(pattern string, src string) bool {  
    return strings.Contains(src, pattern)  
}
```

```
// -- match/main.go --
```

```
// ...
```

```
func main() {  
    pattern, src, err := readInput()  
    if err != nil {  
        fail(err)  
    }  
    isMatch := match(pattern, src)  
    if !isMatch {  
        os.Exit(0)  
    }  
    fmt.Println(src)  
}
```

В `main.go` ничего не изменилось — поскольку функция `match()` объявлена в том же пакете, ее можно вызывать напрямую.

Теперь разрешим использовать в шаблоне специальные символы `?` и `*` по аналогии с командной строкой:

```
// -- match/match.go --
```

```
// ...
```

```
// match returns true if src matches pattern,
// false otherwise.
//
// Supports wildcards:
// ? matches any single character
// * matches everything
func match(pattern string, src string) (bool, error) {
    pat := translate(pattern)
    re, err := regexp.Compile(pat)
    if err != nil {
        return false, err
    }
    isMatch := re.MatchString(src)
    return isMatch, nil
}
```

```
// -- main.go --
// ...

func main() {
    // ...
    isMatch, err := match(pattern, src)
    if err != nil {
        fail(err)
    }
    // ...
}
```

[полный исходный код](#)

Проверим:

```
$ go build
$ ./match -p 'a??some' 'go is awesome'
go is awesome
```

Шаг 6 · Пакеты и модули · Stepik

Пакеты и области видимости

Допустим, логика сопоставления строки с шаблоном усложнилась настолько, что плохо помещается в один файл. Вынесем ее в отдельный пакет `glob`:

```
match
└─ glob
```

```
|   └─ glob.go
|─ go.mod
└─ main.go
```

```
// -- match/glob/glob.go --
/*
glob package matches strings against patterns.
Supports wildcards:
    ? matches any single character
    * matches everything
*/
package glob

// ...

// match returns true if src matches pattern,
// false otherwise.
func match(pattern string, src string) (bool, error) {
    // ...
}

// ...
```

Теперь программа откажется компилироваться:

```
$ go build
# github.com/gothanks/match
./main.go:19:18: undefined: match
```

Функция `match()` переехала из пакета `main` в пакет `glob`, поэтому из `main` больше не доступна. В Go действует правило:

- Если функция, тип, переменная, константа или поле структуры называется со строчной буквы (`match`) — она видна только в пределах родного пакета (*не экспортирована* в терминах Go)*.
- Если называется с прописной буквы (`Match`) — видна и в других пакетах (*экспортирована*).

Экспортируем `match()` и импортируем пакет `glob` в `main.go`:

```
// -- match/glob/glob.go --
// ...

// Match returns true if src matches pattern,
// false otherwise.
func Match(pattern string, src string) (bool, error) {
    // ...
```

```

}

// ...

// -- match/main.go --
package main

import (
    "errors"
    "flag"
    "fmt"
    "os"
    "strings"

    "github.com/gothanks/match/glob"
)

func main() {
    // ...
    isMatch, err := glob.Match(pattern, src)
    // ...
}

```

[полный исходный код](#)

Пакеты импортируются по полному имени с идентификатором модуля (`github.com/gothanks/match/glob`), а в коде доступны по локальному имени (`glob`).

В Go не принято «мельчить» с пакетами. В нашей ситуации создание пакета `glob` не было оправдано, я сделал его только в учебных целях.

Шаг 7 · Пакеты и модули · Stepik

Пакет с пакетами

Есть такой модуль:

```

more
├─ go.mod
├─ more.go
├─ numbers
│   └─ numbers.go
└─ text
    └─ text.go

```

```
// -- more/go.mod --  
module github.com/gothanks/more  
  
go 1.16
```

```
// -- more/more.go --  
package main  
  
import (  
    "fmt"  
  
    "github.com/gothanks/more/numbers"  
    "github.com/gothanks/more/text"  
)  
  
func main() {  
    ok := text.ArePermutation("hello", "lehol")  
    fmt.Println("hello / lehol →", ok)  
  
    ok = numbers.IsEven(42)  
    fmt.Println("42 is even →", ok)  
}
```

```
// -- more/numbers/numbers.go --  
package numbers  
  
// IsEven checks if the number is even.  
func IsEven(n int) bool {  
    return n%2 == 0  
}
```

```
// -- more/text/text.go --  
package text  
  
import (  
    "reflect"  
    "sort"  
)  
  
// ArePermutation checks if two strings are permutations of each other.  
func ArePermutation(first, second string) bool {  
    if len(first) != len(second) {  
        return false  
    }  
}
```

```

    runes1st := []rune(first)
    runes2nd := []rune(second)
    sort.Slice(runes1st, func(i, j int) bool { return runes1st[i] <
runes1st[j] })
    sort.Slice(runes2nd, func(i, j int) bool { return runes2nd[i] <
runes2nd[j] })
    return reflect.DeepEqual(runes1st, runes2nd)
}

```

Какие проблемы мешают собрать модуль через `go build`?

Выберите все подходящие ответы из списка

Пакет `main` не имеет доступа к `text.ArePermutation`

Нет критических проблем

Неправильные импорты в `more.go`

Пакет `main` не имеет доступа к `numbers.IsEven`

Шаг 8 · Пакеты и модули · Stepik

Строки, числа, снова строки

Есть такой модуль:

```

more
├─ go.mod
├─ more.go
├─ numbers
│   └─ numbers.go
└─ text
    └─ text.go

```

```

// -- more/go.mod --
module github.com/gothanks/more

```

```

go 1.16

```

```

// -- more/more.go --
package main

```

```

import (
    "fmt"

```



```
    "github.com/gothanks/more/numbers"
)

func main() {
    digits := numbers.AsDigits(42513)
    fmt.Println("42513 →", digits)
}
```

```
// -- more/numbers/numbers.go --
```

```
package numbers
```

```
import (
    "github.com/gothanks/more/text"
)
```

```
// IsEven checks if the number is even.
```

```
func IsEven(n int) bool {
    return n%2 == 0
}
```

```
// AsDigits returns a slice of digits that make up the number.
```

```
func AsDigits(n int) []int {
    runes := text.AsRunes(n)
    count := len(runes)
    zero := int('0')
    digits := make([]int, count)
    for idx, char := range runes {
        digits[idx] = int(char) - zero
    }
    return digits
}
```

```
// -- more/text/text.go --
```

```
package text
```

```
import (
    "strconv"
    "strings"

    "github.com/gothanks/more/numbers"
)
```

```
// AsDigitString returns the number as a string composed of it's digits,  
// separated by dashes. E.g. 42513 → 4-2-5-1-3
```

```
func AsDigitString(n int) string {
    digits := numbers.AsDigits(n)
    parts := make([]string, len(digits))
    for idx, d := range digits {
        parts[idx] = strconv.Itoa(d)
    }
    return strings.Join(parts, "-")
}

// AsRunes returns the number as a slice of it's digits runes.
func AsRunes(n int) []rune {
    return []rune(strconv.Itoa(n))
}
```

Какие проблемы мешают собрать модуль через `go build`?

Выберите все подходящие ответы из списка

Не используется функция `numbers.IsEven()`

Преобразование вида `int('0')` запрещено

Циклическая зависимость между `numbers` и `text`

Нет критических проблем

Шаг 9 · Пакеты и модули · Stepik

Внешние зависимости

Допустим, мы решили отказаться от проверки по шаблону и сравнивать строки по похожести. Воспользуемся для этого сторонним модулем github.com/sahilm/fuzzy.

Скачаем пакет:

```
$ go get github.com/sahilm/fuzzy
go: downloading github.com/sahilm/fuzzy v0.1.0
go get: added github.com/sahilm/fuzzy v0.1.0
```

`go get` помещает скачанный модуль в домашний каталог Go (`~/go`). В каталог с исходниками нашего модуля он не попадает. Таким образом, все внешние зависимости хранятся в общей куче, а не в конкретных проектах (в противоположность `venv` в питоне и `node_modules` в js).

Подключим и используем в `main.go`:

```
import (
    // ...
    "github.com/sahilm/fuzzy"
)

func main() {
    // ...
    matches := fuzzy.Find(pattern, []string{src})
    isMatch := len(matches) > 0
    // ...
}
```

[полный исходный код](#)

Соберем и проверим:

```
$ go build

$ ./match -p awesome 'go is awesome'
go is awesome

$ ./match -p php 'go is awesome'
(пусто)
```

Go компилирует зависимости вместе с основными исходниками в единый бинарный файл. Никаких внешних бинарных зависимостей, все включено в общий бинарник `match`. Поэтому модули в Go распространяются в виде исходников на гитхабе. Не существует общего репозитория с собранными артефактами (как PyPI в питоне или npm в js).

Зависимость зафиксирована в `go.mod`:

```
module github.com/gothanks/match

go 1.16

require (
    github.com/kylelemons/godebug v1.1.0 // indirect
    github.com/sahilm/fuzzy v0.1.0
)
```

Модуль `github.com/kylelemons/godebug` — транзитивная зависимость (от него зависит `github.com/sahilm/fuzzy`), поэтому записан как *indirect*.

`go build` автоматически проверяет зависимости, перечисленные в `import`. И скачивает их, если чего-то не хватает. Так что если у вас проект с кучей зависимостей — не придется выполнять `go`

`get` по каждой, достаточно одного `go build`.

Шаг 10 · Пакеты и модули · Stepik

Обновление зависимостей

Вот несколько полезных команд.

Показать основной модуль и его зависимости:

```
$ go list -m all
github.com/gothinks/match
github.com/kylelemons/godebug v1.1.0
github.com/sahilm/fuzzy v0.1.0
```

Посмотреть все версии конкретного модуля:

```
$ go list -m -versions github.com/sahilm/fuzzy
github.com/sahilm/fuzzy v0.0.1 v0.0.2 v0.0.3 v0.0.4 v0.0.5 v0.1.0
```

Обновить конкретный модуль на последнюю patch-версию в пределах действующей minor-версии (например, 1.1.0 → 1.1.5):

```
$ go get -u=patch github.com/sahilm/fuzzy
```

Обновить конкретный модуль на последнюю minor-версию в пределах действующей major-версии (1.1.0 → 1.2.1):

```
$ go get -u github.com/sahilm/fuzzy
```

Go предполагает, что авторы модулей следуют правилам семантического версионирования, при котором minor- и patch-версии остаются обратно совместимыми — поэтому `go get -u` обновляет на последнюю выпущенную версию.

Если меняется major-версия (например, 1.2.1 → 2.0.0), у модуля меняется идентификатор (`github.com/sahilm/fuzzy` → `github.com/sahilm/fuzzy/v2`). `go get -u` автоматически на такую версию не обновит. Это сделано специально, потому что новая major-версия может быть несовместима с предыдущей.

Удалить неактуальные зависимости из `go.mod`:

```
$ go mod tidy
```

Шаг 11 · Пакеты и модули · Stepik

Утилита подсчета слов в предложении

Ваша задача — написать утилиту `wordcount`, которая считает количество слов в предложении:

```
$ ./wordcount 'go is awesome'
3
```

Вот что нужно сделать:




1. Форкнуть репозиторий github.com/gothanks/wordcount
2. Перейти по адресу `https://github.com/юзернейм/wordcount/actions` (вместо `юзернейм` подставьте свое имя пользователя на гитхабе). Нажать на длинную зеленую кнопку «I understand...»
3. Клонировать репозиторий на локальную машину.
4. Создать go-модуль, как мы делали для `match`. Идентификатор модуля должен быть `github.com/юзернейм/wordcount`.
5. Реализовать логику утилиты на go. Проверить, что утилита корректно считает слова.
6. Закоммитить и запустить изменения.
7. Убедиться, что проходят тесты:
 - перейти на `https://github.com/юзернейм/wordcount/actions`;
 - дождаться, пока выполнятся тесты (обычно занимает минуту-другую, но в редких случаях гитхаб может тупить до 30 минут);
 - проверить, что workflow успешно выполнен (иконка с «галочкой»).

Должно получиться примерно так:

All workflows

Showing runs from all workflows

🔍 Filter workflow runs

1 workflow run	Event ▾	Status ▾	Branch ▾	Actor ▾
<div> readme</div> <div>build #1: Commit eccc9f5 pushed by nalgeon</div>			<div>main</div>	<div> 2 minutes ago ...</div> <div> 27s</div>

К сожалению, Степик не разрешает проверять такие задания в автоматическом режиме. Поэтому воспользуйтесь [внешним грейдером](#) и укажите в качестве ответа токен, который он вернет.

Напишите текст

```
$ git clone https://github.com/gothanks/wordcount
$ ./wordcount
$ git config --global --add safe.directory <my local path (root-owned due to
remote mount)>/wordcount
```

```
$ git remote set-url origin git@github.com:xelad0m/wordcount.git
$ go mod init github.com/xelad0m/wordcount
go: creating new go.mod: module github.com/xelad0m/wordcount
go: to add module requirements and sums:
    go mod tidy
$ go mod tidy
$ cat go.mod
module github.com/xelad0m/wordcount

go 1.18
$ vim main.go
...
$ cat main.go
package main

import (
    "fmt"
    "os"
    "strings"
)

func main() {
    str := os.Args[1]
    //fmt.Printf("%q\n", strings.Split(str, " "))

    count := 0
    for _, s := range strings.Split(str, " ") {
        if len(s) != 0 {
            count++
        }
    }

    fmt.Println(count)
}

$ go test -v
=== RUN    TestWordcount
=== RUN    TestWordcount/empty
=== RUN    TestWordcount/single
=== RUN    TestWordcount/several
=== RUN    TestWordcount/even_more
--- PASS: TestWordcount (0.00s)
    --- PASS: TestWordcount/empty (0.00s)
    --- PASS: TestWordcount/single (0.00s)
```

```
--- PASS: TestWordcount/several (0.00s)
--- PASS: TestWordcount/even_more (0.00s)
PASS
ok      github.com/xelad0m/wordcount    0.002s
$ git add .
$ git commit -m "init"
[main 70be23c] init
 2 files changed, 24 insertions(+)
 create mode 100644 go.mod
 create mode 100644 main.go
$ git status
On branch main
Your branch is ahead of 'origin/main' by 1 commit.
  (use "git push" to publish your local commits)

nothing to commit, working tree clean
$ git:(main) git push
Enumerating objects: 5, done.
Counting objects: 100% (5/5), done.
Delta compression using up to 4 threads
Compressing objects: 100% (3/3), done.
Writing objects: 100% (4/4), 511 bytes | 511.00 KiB/s, done.
Total 4 (delta 1), reused 0 (delta 0), pack-reused 0
remote: Resolving deltas: 100% (1/1), completed with 1 local object.
To github.com:xelad0m/wordcount.git
 1f87090..70be23c  main -> main
```

Дополнительное чтение

Спецификация: [packages](#) • [package initialization](#)

[Go Modules Reference](#)

[Organizing Go code](#)

Чистый код. 2. Тесты

Шаг 1 · Тесты · Stepik

В Go есть все необходимое для хороших тестов.

Проверим функцию, которая возвращает минимальное из двух чисел:

```
func IntMin(a, b int) int {
    if a < b {
        return a
    }
    return b
}
```

Тесты принято писать в отдельном файле с суффиксом `_test`. Например, если `IntMin()` определена в файле `ints.go`, то тесты будут в том же пакете, но в отдельном файле `ints_test.go`.

Тест — это функция с префиксом `Test`:

```
func TestIntMin(t *testing.T) {
    got := IntMin(2, -2)
    want := -2
    if got != want {
        t.Errorf("got %d; want %d", got, want)
    }
}
```

Тест всегда принимает указатель на `testing.T` — это такой сборник полезных функций. Например, `t.Errorf()` выводит сообщение об ошибке, не прекращая выполнение теста.

Выполним тест с подробными результатами (опция `-v`):

```
$ go test -v
=== RUN   TestIntMin
--- PASS: TestIntMin (0.00s)
PASS
ok      ints    0.004s
```

[песочница](#)

Шаг 2 · Тесты · Stepik

Тесты на сумму

Есть функция, которая суммирует целые числа:

```
func Sum(nums ...int) int {
    total := 0
    for _, num := range nums {
        total += num
    }
}
```



```
    return total
}
```

Помогите мне написать на нее тесты.

Sample Input:

Sample Output:

PASS

```
package main

import (
    "testing"
)

func Sum(nums ...int) int {
    total := 0
    for _, num := range nums {
        total += num
    }
    return total
}

func TestSumZero(t *testing.T) {
    if Sum() != 0 {
        t.Errorf("Expected Sum() == 0")
    }
}

func TestSumOne(t *testing.T) {
    if Sum(1) != 1 {
        t.Errorf("Expected Sum(1) == 1")
    }
}

func TestSumPair(t *testing.T) {
    if Sum(1, 2) != 3 {
        t.Errorf("Expected Sum(1, 2) == 3")
    }
}

func TestSumMany(t *testing.T) {
```

```
    if Sum(1, 2, 3, 4, 5) != 15 {
        t.Errorf("Expected Sum(1, 2, 3, 4, 5) == 15")
    }
}
```

Шаг 3 · Тесты · Stepik

Табличные тесты

Чтобы проверить граничные условия и различные сочетания параметров, нам пришлось бы написать целую кучу тестовых функций. Обычно вместо этого используют *табличные* (table-driven) тесты:

- отдельно описывают входные данные и ожидаемые значения;
- последовательно выполняют каждую проверку с помощью `t.Run()`.

```
func TestIntMin(t *testing.T) {
    var tests = []struct {
        a, b int
        want int
    }{
        {0, 1, 0},
        {1, 0, 0},
        {1, 1, 1},
    }

    for _, test := range tests {
        name := fmt.Sprintf("case(%d,%d)", test.a, test.b)
        t.Run(name, func(t *testing.T) {
            got := IntMin(test.a, test.b)
            if got != test.want {
                t.Errorf("got %d, want %d", got, test.want)
            }
        })
    }
}
```

Срез `tests` — наша таблица с отдельными проверками. У каждой проверки есть входные параметры (`a`, `b`) и ожидаемый результат (`want`). `t.Run()` запускает конкретную проверку и выводит ошибку, если она не прошла.

```
$ go test -v
=== RUN    TestIntMin
=== RUN    TestIntMin/case(0,1)
=== RUN    TestIntMin/case(1,0)
```

```
=== RUN    TestIntMin/case(1,1)
--- PASS: TestIntMin (0.00s)
    --- PASS: TestIntMin/case(0,1) (0.00s)
    --- PASS: TestIntMin/case(1,0) (0.00s)
    --- PASS: TestIntMin/case(1,1) (0.00s)
PASS
ok      ints    0.005s
```

[песочница](#)

Шаг 4 · Тесты · Stepik

Табличные тесты на сумму

Есть функция, которая суммирует целые числа:

```
func Sum(nums ...int) int {
    total := 0
    for _, num := range nums {
        total += num
    }
    return total
}
```

Помогите мне написать на нее табличные тесты вместо пачки обычных.

Sample Input:

Sample Output:

PASS

```
package main

import (
    "testing"
)

func Sum(nums ...int) int {
    total := 0
    for _, num := range nums {
        total += num
    }
    return total
}
```

```

func TestSum(t *testing.T) {
    tests := []struct {
        name string
        nums []int
        want int
    }{
        {"zero", []int{}, 0},
        {"one", []int{1}, 1},
        {"dva", []int{1, 2}, 3},
        {"many", []int{0, 5, 10}, 15},
    }
    for _, test := range tests {
        t.Run(test.name, func(t *testing.T) {
            got := Sum(test.nums...) // так выглядит "распаковка"
            if got != test.want {
                t.Errorf("%s: got %d, want %d", test.name, got, test.want)
            }
        })
    }
}

```

Шаг 5 · Тесты · Stepik

Тесты и внешние вызовы

Предположим, у нас есть модуль, который проверяет доступность произвольного URL. Возвращает `true`, если адрес доступен, и `false` в противном случае:

```

ping/
├─ go.mod
├─ ping.go
└─ ping_test.go

```

Основной код:

```

// -- ping/ping.go --
// Пакет ping проверяет доступность URL.
package ping

import "net/http"

// Pinger проверяет доступность URL.
type Pinger struct {

```

```

    client *http.Client
}

// Ping запрашивает указанный URL.
// Возвращает true, если адрес доступен, и false в противном случае.
func (p Pinger) Ping(url string) bool {
    resp, err := p.client.Head(url)
    if err != nil {
        return false
    }
    if resp.StatusCode != 200 {
        return false
    }
    return true
}

```

Тесты:

```

// -- ping/ping_test.go --
package ping

import (
    "net/http"
    "testing"
)

func TestPing(t *testing.T) {
    client := &http.Client{}
    pinger := Pinger{client}
    got := pinger.Ping("https://example.com")
    if !got {
        t.Errorf("Expected example.com to be available")
    }
    got = pinger.Ping("https://example.com/404")
    if got {
        t.Errorf("Expected example.com/404 to be unavailable")
    }
}

```

```
$ go test
```

```
PASS
```

```
ok      ping      0.728s
```

Тесты проходят, но занимают чуть ли не секунду, потому что делают реальные http-запросы. Это совсем не здорово, попробуем исправить.

Шаг 6 · Тесты · Stepik

Заглушки

В питоне или js мы могли бы сделать *заглушку* (stub) http-клиента с единственным методом `.Head()` и подсунуть ее в `Pinger` вместо настоящего клиента. В Go так сделать не получится: он статически типизирован и ожидает переменную типа `http.Client`:

```
type Pinger struct {  
    client *http.Client  
}
```

Заменим эту жесткую зависимость. Введем интерфейс с единственным нужным нам методом — `.Head()`, и будем использовать его в `Pinger` вместо `http.Client`:

```
// HTTPClient - это усеченный http-клиент,  
// который умеет делать HEAD-запросы.  
type HTTPClient interface {  
    Head(url string) (resp *http.Response, err error)  
}  
  
// Pinger проверяет доступность URL.  
type Pinger struct {  
    client HTTPClient  
}
```

В основном коде можем спокойно использовать настоящий `http.Client`, потому что он реализует интерфейс `HTTPClient`:

```
func main() {  
    client := &http.Client{}  
    pinger := Pinger{client}  
    url := "https://ya.ru"  
    alive := pinger.Ping(url)  
    fmt.Println(url, "is alive =", alive)  
}
```

```
$ go run main.go  
https://ya.ru is alive = true
```

А в тестах создадим заглушку, которая вместо обращения к сайту по http сразу возвращает ответ:

```
// MockClient - это заглушка http-клиента для тестов.
type MockClient struct{}

// Head возвращает http-ответ со статусом, указанным в url.
// Например:
// url = https://ya.ru/200 -> статус = 200
// url = https://ya.ru/404 -> статус = 404
func (client *MockClient) Head(url string) (resp *http.Response, err error)
{
    parts := strings.Split(url, "/")
    last := parts[len(parts)-1]
    statusCode, err := strconv.Atoi(last)
    if err != nil {
        return nil, err
    }
    resp = &http.Response{StatusCode: statusCode}
    return resp, nil
}
```

```
Forecast(url string) (resp *http.Response, err error)
```

Используем заглушку в тестах:

```
func TestPing(t *testing.T) {
    client := &MockClient{}
    pinger := Pinger{client}
    got := pinger.Ping("https://example.com/200")
    if !got {
        t.Errorf("Expected example.com/200 to be available")
    }
    got = pinger.Ping("https://example.com/404")
    if got {
        t.Errorf("Expected example.com/404 to be unavailable")
    }
}
```

[песочница](#)

Запускаем тесты:

```
$ go test
PASS
ok      ping    0.012s
```

12 миллисекунд вместо 728. Другое дело!

Вообще, для http-заглушек в Go есть отдельный пакет [net/http/httptest](https://pkg.go.dev/net/http/httptest). Здесь я его не использовал, чтобы продемонстрировать универсальный принцип создания заглушек для внешних или «тяжелых» зависимостей:

1. Заменяем конкретную зависимость на зависимость от интерфейса.
2. Реализуем интерфейс в тестах и используем его.

Шаг 7 · Тесты · Stepik

Тестируем погоду

Предположим у нас есть сервис `WeatherService`, который предсказывает погоду. Он работает по загадочному и непредсказуемому алгоритму, полному искусственного интеллекта, и возвращает прогноз дневной температуры на завтра, в градусах Цельсия:

```
// WeatherService предсказывает погоду.
type WeatherService struct{}

// Forecast сообщает ожидаемую дневную температуру на завтра.
func (ws *WeatherService) Forecast() int {
    // магия
    return value
}
```

Мы сделали структуру `Weather`, которая запрашивает `WeatherService` и возвращает прогноз, но не числом, а текстом:

```
// Weather выдает текстовый прогноз погоды.
type Weather struct {
    service *WeatherService
}

// Forecast сообщает текстовый прогноз погоды на завтра.
func (w Weather) Forecast() string {
    deg := w.service.Forecast()
    switch {
    case deg < 10:
        return "холодно"
    case deg >= 10 && deg < 15:
        return "прохладно"
    case deg >= 15 && deg < 20:
        return "идеально"
    case deg >= 20:
        return "жарко"
    }
```



```

    }
    return "инопланетно"
}

```

Неплохо бы теперь проверить `Weather`. Проблема в том, что мы никак не контролируем поведение `WeatherService`, поэтому тесты не проходят:

```

type testCase struct {
    deg int
    want string
}

var tests []testCase = []testCase{
    {-10, "холодно"},
    {0, "холодно"},
    {5, "холодно"},
    {10, "прохладно"},
    {15, "идеально"},
    {20, "жарко"},
}

func TestForecast(t *testing.T) {
    service := &WeatherService{}
    weather := Weather{service}
    for _, test := range tests {
        name := fmt.Sprintf("%v", test.deg)
        t.Run(name, func(t *testing.T) {
            got := weather.Forecast()
            if got != test.want {
                t.Errorf("%s: got %s, want %s", name, got, test.want)
            }
        })
    }
}

```

```

$ go test
--- FAIL: TestForecast (0.00s)
    --- FAIL: TestForecast/10 (0.00s)
        weather_test.go:60: 10: got холодно, want прохладно
    --- FAIL: TestForecast/15 (0.00s)
        weather_test.go:60: 15: got холодно, want идеально
    --- FAIL: TestForecast/20 (0.00s)
        weather_test.go:60: 20: got холодно, want жарко
FAIL

```

Исправьте `Weather`, чтобы его можно было нормально протестировать. Сделайте заглушку и используйте ее в `TestForecast`.

Не меняйте `WeatherService`, метод `Weather.Forecast()` и набор тестов `tests`.

Sample Input:

Sample Output:

PASS

```
package main

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

// WeatherService предсказывает погоду.
type WeatherService struct{}

// Forecast сообщает ожидаемую дневную температуру на завтра.
func (ws *WeatherService) Forecast() int {
    rand.Seed(time.Now().Unix())
    value := rand.Intn(31)
    sign := rand.Intn(2)
    if sign == 1 {
        value = -value
    }
    return value
}

// интерфейс
type WeatherClient interface {
    Forecast() int
}

// Переключаем Weather на зависимость от интерфейса Forecaster вместо
конкретного типа WeatherService
type Weather struct {
    //service *WeatherService
    service WeatherClient
}
```

```

}

type MockWeatherService struct {
    deg int
}

func (self *MockWeatherService) Forecast() int {
    return self.deg
}

// Forecast сообщает текстовый прогноз погоды на завтра.
func (w Weather) Forecast() string {
    deg := w.service.Forecast()
    switch {
    case deg < 10:
        return "холодно"
    case deg >= 10 && deg < 15:
        return "прохладно"
    case deg >= 15 && deg < 20:
        return "идеально"
    case deg >= 20:
        return "жарко"
    }
    return "инопланетно"
}

type testCase struct {
    deg int
    want string
}

var tests []testCase = []testCase{
    {-10, "холодно"},
    {0, "холодно"},
    {5, "холодно"},
    {10, "прохладно"},
    {15, "идеально"},
    {20, "жарко"},
}

func TestForecast(t *testing.T) {
    //service := &WeatherService{}

```

```

service := &MockWeatherService{}
weather := Weather{service}
for _, test := range tests {
    name := fmt.Sprintf("%v", test.deg)
    t.Run(name, func(t *testing.T) {
        service.deg = test.deg // передача параметра в заглушку
        got := weather.Forecast()
        if got != test.want {
            t.Errorf("%s: got %s, want %s", name, got, test.want)
        }
    })
}
}

```

Шаг 8 · Тесты · Stepik

Подготовка и завершение

Бывает, хочется выполнить какой-то код до старта тестов (setup), а какой-то — после их завершения (teardown). Например, перед тестами подготовить данные, а после — почистить их.

В других языках для этого часто используют отдельные функции. В Go же обходятся одной —

`TestMain()`:

```

func TestMain(m *testing.M) {
    fmt.Println("Setup tests...")
    start := time.Now()

    m.Run()

    fmt.Println("Teardown tests...")
    end := time.Now()
    fmt.Println("Tests took", end.Sub(start))
}

```

```

$ go test -v
Setup tests...
=== RUN    TestIntMin
--- PASS: TestIntMin (0.00s)
PASS
Teardown tests...
Tests took 137.999µs
ok      ints    0.005s

```

`m.Run()` запускает тесты (функции вида `Test*()`). Получается, что любой код до него — это `setup`, а после — `teardown`.

[песочница](#)

Шаг 9 · Тесты · Stepik

Избирательный запуск

Иногда хочется запустить не все тесты, а только часть. Например, если некоторые тесты медленные, и каждый раз гонять их не хочется.

Наша старая знакомая — функция, которая суммирует целые числа:

```
func Sum(nums ...int) int {
    total := 0
    for _, num := range nums {
        total += num
    }
    return total
}
```

И пара тестов:

```
func TestSumFew(t *testing.T) {
    if Sum(1, 2, 3, 4, 5) != 15 {
        t.Errorf("Expected Sum(1, 2, 3, 4, 5) == 15")
    }
}

func TestSumN(t *testing.T) {
    n := 1_000_000_000
    nums := make([]int, n)
    for i := 0; i < n; i++ {
        nums[i] = i + 1
    }
    got := Sum(nums...)
    want := n * (n + 1) / 2
    if got != want {
        t.Errorf("Expected sum[i=1..n]
(https://stepik.org/lesson/545392/step/i) == n*(n+1)/2")
    }
}
```

Прогоним тесты:

```
$ go test -v
=== RUN    TestSum
--- PASS: TestSum (0.00s)
=== RUN    TestSumN
--- PASS: TestSumN (3.78s)
```

Ни малейшего желания каждый раз ждать 4 секунды, пока выполняется `TestSumN` на миллиарде слагаемых. Воспользуемся так называемым short-режимом, который делит все тесты на «короткие» и «длинные». `TestSumFew` будет коротким тестом, а `TestSumN` — длинным:

```
func TestSumN(t *testing.T) {
    if testing.Short() {
        t.Skip("skipping test in short mode.")
    }
    // сам тест
}
```

Теперь тесты в коротком режиме будут игнорировать `TestSumN` и обрабатывать моментально:

```
$ go test -v -short
=== RUN    TestSumFew
--- PASS: TestSumFew (0.00s)
=== RUN    TestSumN
selective_test.go:21: skipping test in short mode.
--- SKIP: TestSumN (0.00s)
```

[песочница](#)

Альтернативный вариант — указать маску названия теста. Так выполнится только `TestSumFew`:

```
$ go test -v -run Few
```

А так — только `TestSumN`:

```
$ go test -v -run N
```

Шаг 10 · Тесты · Stepik

Странные тесты

Есть функция, которая суммирует целые числа:

```
func Sum(nums ...int) int {
    total := 0
    for _, num := range nums {
        total += num
    }
}
```

```

    }
    return total
}

```

И тесты на нее:

```

func TestMain(m *testing.M) {
    fmt.Println("Testing Sum(n1, n2, ...,nk)...")
    fmt.Println("Finished testing")
}

func TestSumZero(t *testing.T) {
    if Sum() != 0 {
        t.Errorf("Expected Sum() == 0")
    }
}

func TestSumOne(t *testing.T) {
    if Sum(1) != 1 {
        t.Errorf("Expected Sum(1) == 1")
    }
}

func TestSumPair(t *testing.T) {
    if Sum(1, 2) != 3 {
        t.Errorf("Expected Sum(1, 2) == 3")
    }
    t.Skip()
}

func TestSumMany(t *testing.T) {
    if Sum(1, 2, 3, 4, 5) != 15 {
        t.Errorf("Expected Sum(1, 2, 3, 4, 5) == 15")
    }
}

```

Что с ними не так? Помимо того, что тестов недостаточно, а вместо кучки отдельных лучше сделать один табличный тест.

Тесты выполняются неоправданно долго
 Тесты выполняются дважды
 Тесты вообще не выполняются
 С тестами все хорошо
 Один из тестов непонятно почему заскипан

Шаг 11 · Тесты · Stepik

Тестовое покрытие

Покрывание (coverage) показывает, насколько полно тесты проверяют основной код. Чтобы его замерить, достаточно запустить `go test` с опцией `-cover`.

Вот наш модуль:

```
ints/  
├─ go.mod  
├─ ints.go  
└─ ints_test.go
```

Основной код:

```
// -- ints/ints.go --  
package ints  
  
func IntMin(a, b int) int {  
    if a < b {  
        return a  
    }  
    return b  
}
```

Тесты:

```
// -- ints/ints_test.go --  
package ints  
  
import (  
    "testing"  
)  
  
func TestIntMin(t *testing.T) {  
    got := IntMin(2, -2)  
    want := -2  
    if got != want {  
        t.Errorf("got %d; want %d", got, want)  
    }  
}
```

Запускаем тесты:

```
$ go test -cover  
PASS
```



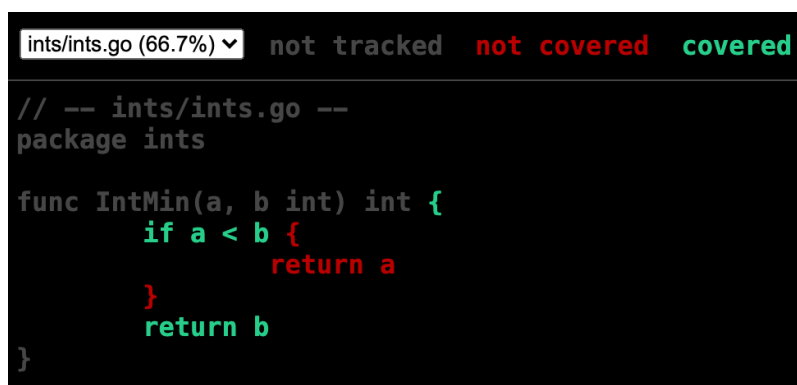
```
coverage: 66.7% of statements
ok      ints      0.004s
```

Только 66%! Чтобы не гадать, какой код не покрыт, попросим Go собрать детальную статистику (profile):

```
$ go test -coverprofile=cover.prof
PASS
coverage: 66.7% of statements
ok      ints      0.004s
```

И посмотрим отчет в HTML:

```
go tool cover -html=cover.prof
```



```
ints/ints.go (66.7%) ▼ not tracked not covered covered
// -- ints/ints.go --
package ints

func IntMin(a, b int) int {
    if a < b {
        return a
    }
    return b
}
```

Так все понятно!

Дополнительное чтение

Пакет [testing](#)

Команда [go test](#)

[Using Subtests and Sub-benchmarks](#)

[Testable Examples in Go](#)

Чистый код. 3. Бенчмарки

Шаг 1 · Бенчмарки · Stepik

Возьмем функцию, которая проверяет строку на соответствие шаблону:

```
// -- match.go --
package match

import (
```

```

    "strings"
)

// MatchContains returns true if the string
// contains the pattern, false otherwise.
func MatchContains(pattern string, src string) bool {
    return strings.Contains(src, pattern)
}

```

Функция рабочая, но довольно примитивная. На вхождение проверяет, а по шаблону — нет:

```

s := "go is awesome"
fmt.Println(MatchContains("is", s))
// true
fmt.Println(MatchContains("go.*awesome", s))
// false

```

Добавим функцию для честной проверки по шаблону:

```

// MatchRegexp returns true if the string
// matches the regexp pattern, false otherwise.
func MatchRegexp(pattern string, src string) bool {
    re, err := regexp.Compile(pattern)
    if err != nil {
        return false
    }
    return re.MatchString(src)
}

```

Теперь работает:

```

fmt.Println(MatchRegexp("go.*awesome", s))
// true
fmt.Println(MatchRegexp("^go", s))
// true
fmt.Println(MatchRegexp("awesome$", s))
// true

```

Итак, у нас есть:

- примитивная (но предположительно быстрая) `MatchContains()`
- мощная (но предположительно медленная) `MatchRegexp()`

Однако, насчет «быстро-медленно» никогда нельзя исходить только из предположений. Давайте выясним наверняка.

Шаг 2 · Бенчмарки · Stepik

Бенчмарки

Сравнить производительность в Go помогут *бенчмарки* (benchmark).

Бенчмарки похожи на обычные тесты, только начинаются со слова `Benchmark` вместо `Test`, и принимают параметр `testing.B` вместо `testing.T`.

Поскольку сравниваем два варианта — `MatchContains()` и `MatchRegexp()` — подготовим по бенчмарку на каждого:

```
// -- match_test.go --
package match

import (
    "testing"
)

// pretty long string
const src = "Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do
eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim
veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea
commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit
esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat
cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id
est laborum."

// matches in the middle of the string
const pattern = "commodo"

func BenchmarkMatchContains(b *testing.B) {
    for n := 0; n < b.N; n++ {
        MatchContains(pattern, src)
    }
}

func BenchmarkMatchRegexp(b *testing.B) {
    for n := 0; n < b.N; n++ {
        MatchRegexp(pattern, src)
    }
}
```

Бенчмарк всегда устроен одинаково — внутри у него цикл от `0` до `b.N`, а в теле цикла вызывается целевая функция. Количество итераций `b.N` Go определяет самостоятельно — так, чтобы получились статистически значимые результаты.

Запускаем и смотрим:

```
$ go test -bench=.
BenchmarkMatchContains-8      15159487      74.76 ns/op
BenchmarkMatchRegex-8        633741      1860 ns/op
```

Первое число — сколько раз был выполнен бенчмарк, второе — сколько наносекунд в среднем занял один прогон ($1\text{ с} = 10^9\text{ нс}$).

Интуиция не подвела. Проверка по регулярке работает аж в 25 раз медленнее 🤖

Шаг 3 · Бенчмарки · Stepik

Сравниваем библиотечную функцию с самописной

Временно помутившись рассудком, я решил, что вместо библиотечной функции

`strings.Contains()` лучше использовать самописную:

```
// Библиотечная
func MatchContains(pattern string, src string) bool {
    return strings.Contains(src, pattern)
}

// Самописная
func MatchContainsCustom(pattern string, src string) bool {
    if pattern == "" {
        return true
    }
    if len(pattern) > len(src) {
        return false
    }
    pat_len := len(pattern)
    for idx := 0; idx < len(src)-pat_len+1; idx++ {
        if src[idx:idx+pat_len] == pattern {
            return true
        }
    }
    return false
}
```

Разрушайте мои иллюзии, написав бенчмарки на оба варианта.

Не меняйте названия функций — они используются при проверке задания.

Sample Input:

Sample Output:

PASS

```
package main

// не удаляйте импорты, они используются при проверке
import (
    "fmt"
    "os"
    "strings"
    "testing"
)

// используйте эту переменную в бенчмарках
const src = "Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do
eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim
veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea
commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit
esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat
cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id
est laborum."

// используйте эту переменную в бенчмарках
const pattern = "commodo"

// реализуйте бенчмарк для MatchContains
func BenchmarkMatchContains(b *testing.B) {
    for n := 0; n < b.N; n++ {
        MatchContains(pattern, src)
    }
}

// реализуйте бенчмарк для MatchContainsCustom
func BenchmarkMatchContainsCustom(b *testing.B) {
    for n := 0; n < b.N; n++ {
        MatchContainsCustom(pattern, src)
    }
}
```

```
// 

не меняйте код ниже этой строки


//

// Библиотечная
func MatchContains(pattern string, src string) bool {
    return strings.Contains(src, pattern)
}

// Самописная
func MatchContainsCustom(pattern string, src string) bool {
    if pattern == "" {
        return true
    }
    if len(pattern) > len(src) {
        return false
    }
    pat_len := len(pattern)
    for idx := 0; idx < len(src)-pat_len+1; idx++ {
        if src[idx:idx+pat_len] == pattern {
            return true
        }
    }
    return false
}
```

Шаг 4 · Бенчмарки · Stepik

Пишем множество

Бенчмарки позволяют сравнить разные алгоритмы. Именно так мы сравнили `MatchContains()` и `MatchContainsCustom()` на предыдущем шаге. А еще они помогают понять, как ведет себя один и тот же алгоритм в зависимости от объема входных данных.

Допустим, мы реализовали структуру данных *множество* (set):

```
// -- set.go --
package set

// IntSet implements a mathematical set
// of integers, elements are unique.
type IntSet struct {
    elems *[]int
}
```

```

// MakeIntSet creates an empty set.
func MakeIntSet() IntSet {
    elems := []int{}
    return IntSet{&elems}
}

// Contains reports whether an element is within the set.
func (s IntSet) Contains(elem int) bool {
    for _, el := range *s.elems {
        if el == elem {
            return true
        }
    }
    return false
}

// Add adds an element to the set.
func (s IntSet) Add(elem int) bool {
    if s.Contains(elem) {
        return false
    }
    *s.elems = append(*s.elems, elem)
    return true
}

```

Наше простенькое множество поддерживает только две операции:

- добавить новый элемент;
- проверить, есть ли элемент в множестве.

```

set := MakeIntSet()

set.Add(5)
fmt.Println(set.Contains(5))
// true

fmt.Println(set.Contains(42))
// false

// элементы множества уникальны,
// добавить дважды один и тот же элемент не получится
added := set.Add(5)

```

```
fmt.Println(added)
// false
```

Работает: можно добавить элемент и проверить на вхождение. Теперь посмотрим на быстроедействие.

Шаг 5 · Бенчмарки · Stepik

Варьируем размер

Метод `IntSet.Contains()` отлично работает на множестве из 3 или 10 элементов. Но как он поведет себя на 1000 элементов? 10000? 100000? Ответить помогут бенчмарки.

Возможно, вы помните, как на прошлом уроке мы выполняли табличные тесты — один большой тест, внутри которого запускается много маленьких. Когда тестируют производительность одной и той же функции на разных входных данных — поступают аналогично:

```
// -- set_test.go --
package set

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

func BenchmarkIntSet(b *testing.B) {
    rand.Seed(0)
    for _, size := range []int{1, 10, 100, 1000, 10000, 100000} {
        set := randomSet(size)
        name := fmt.Sprintf("Contains-%d", size)
        b.Run(name, func(b *testing.B) {
            for n := 0; n < b.N; n++ {
                elem := rand.Intn(100000)
                set.Contains(elem)
            }
        })
    }
}

func randomSet(size int) IntSet {
    set := MakeIntSet()
    for i := 0; i < size; i++ {
```



```
        n := rand.Intn(100000)
        set.Add(n)
    }
    return set
}
```

`BenchmarkIntSet()` работает так:

1. Берет конкретный размер множества (1, 10, 100...).
2. Создает множество указанного размера и заполняет его случайными числами.
3. Запускает бенчмарк, который меряет производительность `IntSet.Contains()` на этом множестве.
4. Переходит к следующему размеру и повторяет шаги 2–3.

Таким образом, выполняется по одному независимому бенчмарку для каждого размера множества.

Посмотрим на результаты:

```
$ go test -bench=.
BenchmarkIntSet/Contains-1-8          56203182          19.70 ns/op
BenchmarkIntSet/Contains-10-8         47717664          22.26 ns/op
BenchmarkIntSet/Contains-100-8        17775404          62.53 ns/op
BenchmarkIntSet/Contains-1000-8       3852343           309.0 ns/op
BenchmarkIntSet/Contains-10000-8      418706            2645 ns/op
BenchmarkIntSet/Contains-100000-8     85210             13574 ns/op
```

Не слишком радужно. С увеличением размера множества скорость `Contains()` падает в сотни раз. Хорошие алгоритмы так себя не ведут.

[песочница](#)

Шаг 6 · Бенчмарки · Stepik

Ускоряем множество

Переделайте множество так, чтобы `IntSet.Contains()` продолжал работать быстро даже на больших объемах. На множествах размером от 1 до 10000 элементов метод должен выполняться за 100 нс или меньше.

Бенчмарки в этом задании добавляются автоматически при проверке. Чтобы увидеть их результаты, установите константу `verbose` = `true` и нажмите кнопку «Запустить код». Обязательно установите в `false` перед отправкой на проверку (кнопка «Отправить»), иначе результат не будет засчитан.

Подсказка

Медленное множество внутри использует срез. У среза при увеличении размера время поиска элемента растет линейно — $O(n)$. Чтобы множество работало быстро, нужна структура данных, у которой время поиска элемента не зависит от размера — $O(1)$. Такая структура в Go есть — это карта.

Если не знакомы с определением скорости алгоритма в терминах $O(x)$, посмотрите [разбор на котиках](#).

Sample Input:

Sample Output:

PASS

Шаг 7 · Бенчмарки · Stepik

Использование памяти

Бенчмарки измеряют не только время выполнения, но и использование памяти.

Допустим, мы хотим посчитать, сколько раз каждое слово встречается в фразе. Для простоты будем считать, что разделитель слов — только пробел. Напишем функцию:

```
// -- wc.go --
package wc

import (
    "regexp"
)

// Counter maps words to their counts
type Counter map[string]int

var splitter *regexp.Regexp = regexp.MustCompile(" ")

// WordCountRegexp counts absolute frequencies of words in a string.
// Uses Regexp.Split() to split the string into words.
func WordCountRegexp(s string) Counter {
    counter := make(Counter)
    for _, word := range splitter.Split(s, -1) {
        word = strings.ToLower(word)
        counter[word]++
    }
}
```

```
    return counter
}
```

Работает так:

1. Разбиваем строку на слова с помощью регулярного выражения (`Go is awesome` → `go`, `is`, `awesome`).
2. Проходим по словам, приводя каждое слово к нижнему регистру (`Go` → `go`).
3. Учитываем в карте слов, сколько раз встретилось каждое.

Напишем бенчмарк:

```
// -- wc_test.go --
package wc

import (
    "fmt"
    "math/rand"
    "strings"
    "testing"
)

func BenchmarkRegexp(b *testing.B) {
    for _, length := range []int{10, 100, 1000, 10000} {
        rand.Seed(0)
        phrase := randomPhrase(length)
        name := fmt.Sprintf("Regexp-%d", length)
        b.Run(name, func(b *testing.B) {
            for n := 0; n < b.N; n++ {
                WordCountRegexp(phrase)
            }
        })
    }
}

// randomPhrase returns a phrase of n random words
func randomPhrase(n int) string {
    // ...
}
```

Бенчмарк считает слова на случайных фразах из 10, 100, 1000 и 10000 слов. Запустим и посмотрим на результат:

```
$ go test -bench=. -benchmem
```

Regex-10-8	541161	2268 ns/op	1503 B/op
15 allocs/op			
Regex-100-8	50229	23702 ns/op	22686 B/op
115 allocs/op			
Regex-1000-8	4534	261560 ns/op	240035 B/op
1050 allocs/op			
Regex-10000-8	270	4572228 ns/op	2497343 B/op
10302 allocs/op			

Ключ `benchmem` включает отслеживание памяти. Теперь в ответе четыре показателя вместо двух:

- количество прогонов бенчмарка;
- среднее время выполнения одного прогона;
- среднее количество байт, которые пришлось выделить за один прогон;
- сколько в среднем раз приходилось выделять память за один прогон.

Что можно заметить по результатам нашего бенчмарка:

- все показатели линейно растут с увеличением длины строки (вполне логично);
- кажется, что слишком часто выделяется память (примерно столько же, сколько слов в строке).

Давайте проверим, можно ли как-то уменьшить потребление памяти и выделять ее пореже.

[песочница](#)

Шаг 8 · Бенчмарки · Stepik

Оптимизируем использование памяти

Мы договорились, что разделитель слов — только пробел. Раз так, можно обойтись без регулярных выражений. Будем использовать функцию `strings.Fields()`:

```
// WordCountFields counts absolute frequencies of words in a string.
// Uses strings.Fields() to split the string into words.
func WordCountFields(s string) Counter {
    counter := make(Counter)
    for _, word := range strings.Fields(s) {
        word = strings.ToLower(word)
        counter[word]++
    }
    return counter
}
```

Напишем бенчмарк `BenchmarkFields` по аналогии с `BenchmarkRegex` и прогоним оба:

Regexp-10-8	541161	2268 ns/op	1503 B/op
15 allocs/op			
Regexp-100-8	50229	23702 ns/op	22686 B/op
115 allocs/op			
Regexp-1000-8	4534	261560 ns/op	240035 B/op
1050 allocs/op			
Regexp-10000-8	270	4572228 ns/op	2497343 B/op
10302 allocs/op			
Fields-10-8	1358196	873.7 ns/op	836 B/op
4 allocs/op			
Fields-100-8	125509	9577 ns/op	9864 B/op
10 allocs/op			
Fields-1000-8	9250	114056 ns/op	138743 B/op
42 allocs/op			
Fields-10000-8	1039	1175472 ns/op	1138390 B/op
282 allocs/op			

Видим:

- радикально сократилось количество раз, когда выделяется память;
- в два раза уменьшилось количество используемой памяти;
- в разы сократилось время выполнения.

Видимо, основной виновник частого выделения памяти — `Regexp.Split()`. Хорошо, что можно без него обойтись.

Вдохновившись успехами, я вспомнил, что есть функция `strings.Split()`, которая тоже умеет разбивать строку на слова. Как понять, превосходит ли она `strings.Fields()`? Ответят, конечно же, бенчмарки:

```
// WordCountSplit counts absolute frequencies of words in a string.
// Uses strings.Split() to split the string into words.
func WordCountSplit(s string) Counter {
    counter := make(Counter)
    for _, word := range strings.Split(s, " ") {
        word = strings.ToLower(word)
        counter[word]++
    }
    return counter
}
```

Fields-10-8	1358196	873.7 ns/op	836 B/op
4 allocs/op			

Fields-100-8 10 allocs/op	125509	9577 ns/op	9864 B/op
Fields-1000-8 42 allocs/op	9250	114056 ns/op	138743 B/op
Fields-10000-8 282 allocs/op	1039	1175472 ns/op	1138390 B/op
Split-10-8 4 allocs/op	1340078	894.3 ns/op	836 B/op
Split-100-8 10 allocs/op	121958	9460 ns/op	9864 B/op
Split-1000-8 42 allocs/op	10000	122222 ns/op	138754 B/op
Split-10000-8 283 allocs/op	1018	1128942 ns/op	1138498 B/op

Не похоже. Использование памяти идентичное, а 1% выигрыш во времени выполнения может быть случайностью.

Что если приводить к нижнему регистру не отдельные слова, а всю строку заранее? Проверим:

```
// WordCountLowerPhrase counts absolute frequencies of words in a string.
// Converts the whole string to lower case before splitting.
func WordCountLowerPhrase(s string) Counter {
    counter := make(Counter)
    phrase := strings.ToLower(s)
    for _, word := range strings.Split(phrase, " ") {
        counter[word]++
    }
    return counter
}
```

Split-10-8 4 allocs/op	1340078	894.3 ns/op	836 B/op
Split-100-8 10 allocs/op	121958	9460 ns/op	9864 B/op
Split-1000-8 42 allocs/op	10000	122222 ns/op	138754 B/op
Split-10000-8 283 allocs/op	1018	1128942 ns/op	1138498 B/op
LowerPhr-10-8 4 allocs/op	1323583	884.2 ns/op	836 B/op
LowerPhr-100-8	123859	9763 ns/op	9864 B/op

10 allocs/op			
LowerPhr-1000-8	9578	115119 ns/op	138732 B/op
42 allocs/op			
LowerPhr-10000-8	1062	1143585 ns/op	1138566 B/op
283 allocs/op			

Никакой разницы.

Я предполагаю, что больше всего выделений памяти приходится на заполнение карты `counter`. Мы начинаем с пустой карты, так что при добавлении новых слов памяти будет постоянно не хватать, и Go будет выделять все новую и новую. Проверим эту гипотезу и выделим память заранее:

```
// WordCountAllocate counts absolute frequencies of words in a string.
// Pre-allocates memory for the counter.
func WordCountAllocate(s string) Counter {
    words := strings.Split(s, " ")
    size := len(words) / 2
    if size > 10000 {
        size = 10000
    }
    counter := make(Counter, size)
    for _, word := range words {
        word = strings.ToLower(word)
        counter[word]++
    }
    return counter
}
```

Здесь сходу создается карта на половину всех слов, но не более 10000 (предполагаю, что чем длиннее строка, тем чаще повторяются слова, так что делать уж очень большую карту смысла нет). Прогоним бенчмарки:

Split-10-8	1340078	894.3 ns/op	836 B/op
4 allocs/op			
Split-100-8	121958	9460 ns/op	9864 B/op
10 allocs/op			
Split-1000-8	10000	122222 ns/op	138754 B/op
42 allocs/op			
Split-10000-8	1018	1128942 ns/op	1138498 B/op
283 allocs/op			
Allocate-10-8	1329836	908.2 ns/op	836 B/op
4 allocs/op			

Allocate-100-8	147991	8044 ns/op	8139 B/op
6 allocs/op			
Allocate-1000-8	13580	94719 ns/op	106147 B/op
22 allocs/op			
Allocate-10000-8	1200	960448 ns/op	880209 B/op
140 allocs/op			

Память выделяется в два раза реже, а скорость подросла на 15%. Неплохо! Остановимся на этом варианте.

Таким образом, бенчмарки помогли оптимизировать программу как по времени выполнения, так и по использованию памяти.

[песочница](#)

Шаг 9 · Бенчмарки · Stepik

Оптимизируем поиск слов

Я сделал тип `Words`, который находит индекс конкретного слова в строке:

```
// Words represents words in a string.
type Words struct {
    str string
}

// MakeWords creates words from a string.
func MakeWords(s string) Words {
    return Words{s}
}

// Index returns the index of the first instance of word in words,
// or -1 if word is not present in words.
func (w Words) Index(word string) int {
    words := strings.Fields(w.str)
    for idx, item := range words {
        if item == word {
            return idx
        }
    }
    return -1
}
```

Используется так:


```
s := "go is awesome, php is not"
w := MakeWords(s)

fmt.Println(w.Index("go"))
// 0
fmt.Println(w.Index("is"))
// 1
fmt.Println(w.Index("is not"))
// -1
fmt.Println(w.Index("python"))
// -1
```

К сожалению, метод `Words.Index()` потребляет много памяти:

Index-10:	314 ns	166 B	3 allocs
Index-100:	1917 ns	1798 B	3 allocs
Index-1000:	17398 ns	16390 B	3 allocs
Index-10000:	163084 ns	163847 B	3 allocs

Исправьте эту проблему. На строках размером от 1 до 10000 слов метод должен использовать 1000 байт памяти или меньше. Постарайтесь заодно исправить и время поиска, чтобы оно не увеличивалось с ростом длины строки.

Можете менять состав типа `Words`, а также содержимое функций `MakeWords()` и `Index()`. Главное — сохраняйте сигнатуры.

Исходите из того, что `strings.Fields()` подходит для разбиения строки на слова.

Бенчмарки в этом задании добавляются автоматически при проверке. Чтобы увидеть их результаты, установите константу `verbose = true` и нажмите кнопку «Запустить код». Обязательно установите в `false` перед отправкой на проверку (кнопка «Отправить»), иначе результат не будет засчитан.

Подсказка

Обратите внимание, что мы оптимизируем конкретно метод `Words.Index()`. Если для этого придется усложнить конструктор `MakeWords()` — это допустимо.

Sample Input:

Sample Output:

PASS

```
package main
```

```
// не удаляйте импорты, они используются при проверке
import (
    "fmt"
    "math/rand"
    "os"
    "strings"
    "testing"
)

// поменяйте на true, чтобы увидеть результаты бенчмарков
// отключите перед отправкой на проверку,
// иначе результат не будет засчитан
const verbose = false

type Words struct {
    indexer map[string]int
}

func MakeWords(s string) Words {
    W := Words{make(map[string]int)}
    words := strings.Fields(s)

    for idx, item := range words {
        if _, ok := W.indexer[item]; !ok {
            W.indexer[item] = idx
        }
    }
    return W
}

func (w Words) Index(word string) int {
    idx, ok := w.indexer[word]
    if ok {
        return idx
    }
    return -1
}
```

Шаг 10 · Бенчмарки · Stepik

Параметры запуска бенчмарков

Команда, которой мы пользовались весь урок:

```
$ go test -bench=.
```

— запускает все бенчмарки модуля. Чтобы запустить конкретный, укажите его название:

```
$ go test -bench=BenchmarkFields
```

```
BenchmarkFields/Fields-10-8      1203883      970.4 ns/op
...
```

Можно указать шаблон — тогда выполнятся все бенчмарки, которые под него подходят:

```
$ go test -bench="Fields|Split"
```

```
BenchmarkFields/Fields-10-8      1267838      981.4 ns/op
...
BenchmarkSplit/Split-10-8        1243818      942.8 ns/op
...
```

Помимо бенчмарков, команда `test` заодно выполняет все обычные тесты. Чтобы их исключить, задайте заведомо несуществующее значение параметра `run`:

```
$ go test -run=! -bench=.
```

Так выполнятся только бенчмарки, без тестов.

Чистый код. 4. Профайлер

Шаг 1 · Профайлер · Stepik

Когда мы ускоряли код на прошлом уроке, всегда был конкретный «подозреваемый» — операция, которая предположительно работает неоптимально. Но бывает, что подозреваемых несколько, и вклад каждого заранее неизвестен. Тут-то и помогает профилирование.

Предположим, нам достался легаси-код с функцией, которая бьет строку на слова, убирает дубли, и возвращает уникальные слова, отсортированные по алфавиту:

```
// UniqWords splits a string into words, removes duplicates
// and returns unique words in sorted order.
func UniqWords(str string) []string {
    words := splitString(str)
    words = sortWords(words)
    words = uniqWords(words)
    return words
}

func splitString(str string) []string {
```

```

    // ...
}

func sortWords(words []string) []string {
    // ...
}

func uniqWords(words []string) []string {
    // ...
}

```

Удобно, что неизвестный автор явно выделил основные операции в отдельные функции. И неудобно, что каждая из них занимает по странице слабочитаемого кода. Прежде чем что-то оптимизировать, хочется понять: от кого из трех кандидатов — `splitString()`, `sortWords()` или `uniqWords()` — больше всего зависит время выполнения `UniqWords()`.

Для начала напишем бенчмарк, как делали уже неоднократно:

```

func BenchmarkUniqWords(b *testing.B) {
    for _, size := range []int{10, 100, 1000, 10000, 100000} {
        name := fmt.Sprintf("UniqWords-%d", size)
        phrase := randomPhrase(size)
        b.Run(name, func(b *testing.B) {
            for n := 0; n < b.N; n++ {
                UniqWords(phrase)
            }
        })
    }
}

```

И запустим с парой специальных настроек:

```
$ GOGC=off go test -bench=. -cpuprofile=cpu.prof
```

`GOGC=off` отключает сборку мусора во время выполнения. Благодаря этому, сборщик мусора не будет искажать картину использования CPU, которую мы собираемся изучить. Ключ же `cpuprofile` включает запись CPU-профиля (той самой «картины») в файл `cpu.prof`.

Переходим к анализу профиля.

Шаг 2 · Профайлер · Stepik

Анализируем CPU-профиль

Откроем собранный профиль в интерактивной утилите `pprof`:

```
$ go tool pprof cpu.prof
```

```
Type: cpu
```

```
Time: May 24, 2022 at 1:04am (MSK)
```

```
Duration: 7.26s, Total samples = 6.55s (90.16%)
```

```
Entering interactive mode (type "help" for commands, "o" for options)
(pprof)
```

Теперь можно выполнять команды. Начнем с `topN`:

```
(pprof) top10
```

```
Showing nodes accounting for 5390ms, 82.29% of 6550ms total
```

```
Dropped 65 nodes (cum <= 32.75ms)
```

```
Showing top 10 nodes out of 60
```

	flat	flat%	sum%		cum	cum%	
	1950ms	29.77%	29.77%		1960ms	29.92%	runtime.madvise
	790ms	12.06%	41.83%		790ms	12.06%	runtime.memmove
	740ms	11.30%	53.13%		1480ms	22.60%	sort.doPivot
	530ms	8.09%	61.22%		530ms	8.09%	cmpbody
	480ms	7.33%	68.55%		1060ms	16.18%	runtime.mallocgc
	230ms	3.51%	72.06%		230ms	3.51%	runtime.memclrNoHeapPointers
	190ms	2.90%	74.96%		1940ms	29.62%	words.splitString
	180ms	2.75%	77.71%		360ms	5.50%	runtime.slicerunetosting
	170ms	2.60%	80.31%		730ms	11.15%	sort.StringSlice.Less
	130ms	1.98%	82.29%		130ms	1.98%	runtime.heapBitsSetType

Операции отсортированы по *чистому времени* использования процессора (столбец `flat`).

Чистое время не учитывает вызов других операций. Например, если функция `a()` вызывает функции `b()` и `c()`, то показатели `flat` и `flat%` для `a()` не будут учитывать время выполнения `b()` и `c()`.

Команда `topN` хорошо выявляет совсем уж явных злодеев. Когда какая-то пользовательская функция оказывается в топе по `flat`, с ней явно что-то не то. У нас такого нет — в топе находятся операции рантайма (среды исполнения) Go. Само по себе это не хорошо и не плохо.

Попробуем другую команду — `topN -cum`:

```
(pprof) top10 -cum
```

```
Showing nodes accounting for 2.15s, 32.82% of 6.55s total
```

```
Dropped 65 nodes (cum <= 0.03s)
```

```
Showing top 10 nodes out of 60
```

	flat	flat%	sum%		cum	cum%	
	0	0%	0%		4.41s	67.33%	testing.(*B).runN
	0	0%	0%		4.33s	66.11%	words.BenchmarkUniqWords.func1
	0	0%	0%		4.33s	66.11%	words.UniqWords

0	0%	0%	4.30s	65.65%	testing.(*B).launch
0	0%	0%	2.13s	32.52%	runtime.systemstack
1.95s	29.77%	29.77%	1.96s	29.92%	runtime.madvise
0.19s	2.90%	32.67%	1.94s	29.62%	words.splitString (inline)
0	0%	32.67%	1.91s	29.16%	words.sortWords (inline)
0	0%	32.67%	1.88s	28.70%	runtime.(*mheap).alloc.func1
0.01s	0.15%	32.82%	1.88s	28.70%	runtime.(*mheap).allocSpan

Теперь операции отсортированы по *кумулятивному времени* использования процессора (столбец `cum`). Кумулятивное время учитывает вызов всех дочерних операций. Например, если функция `a()` вызывает функции `b()` и `c()`, а `c()` в свою очередь вызывает `d()` — то показатели `cum` и `cum%` для `a()` будут учитывать время выполнения `b()`, `c()` и `d()`.

Команда `topN -cum` хорошо показывает основные операции в коде. У нас видно, что это `words.UniqWords()`. Вот только мы и так это знали — ведь именно ее тестирует бенчмарк ♪ В остальном выдача по-прежнему сильно забита операциями рантайма, что совсем нам не помогает.

Шаг 3 · Профайлер · Stepik

Кто виноват?

Нам нужен ответ на конкретный вопрос:

- если принять время выполнения `UniqWords()` за 100%,
- то какую долю занимает каждая из функций `splitString()`, `sortWords()` и `uniqWords()`?

Ровно на этот вопрос дает ответ команда `peek`, которая показывает непосредственных «детей» функции:

```
(pprof) peek words.UniqWords
Showing nodes accounting for 6.55s, 100% of 6.55s total
-----+-----
      flat  flat%   sum%        cum  cum%   calls calls% + context
-----+-----
                                4.33s   100% |
words.BenchmarkUniqWords.func1
      0      0%      0%    4.33s  66.11% | words.UniqWords
                                1.94s  44.80% |
words.splitString (inline)
                                1.91s  44.11% |
words.sortWords (inline)
                                0.48s  11.09% |
words.uniqWords (inline)
-----+-----
```

Теперь понятно:

- `splitString()` = 45%
- `sortWords()` = 44%
- `uniqWords()` = 11%

`uniqWords()` с 11% точно не стоит трогать в первую очередь. 44% для сортировки выглядит адекватным. Но чуть ли не половина времени на разбивку строки по словам? С функцией `splitString()` явно что-то не то.

Таким образом, профилирование помогло нам определить самую «проблемную» часть кода.

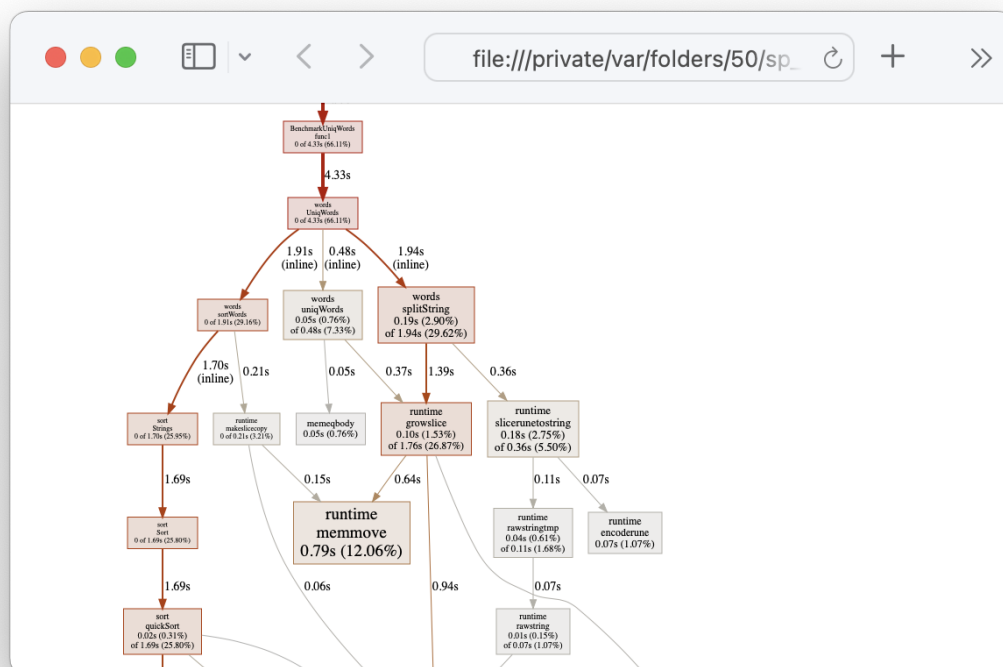
Шаг 4 · Профайлер · StepiK

Визуальный профиль

С профилем не обязательно работать в консоли. Команда `web` отрисует полный граф вызовов в `svg` и откроет его в браузере:

(pprof) web

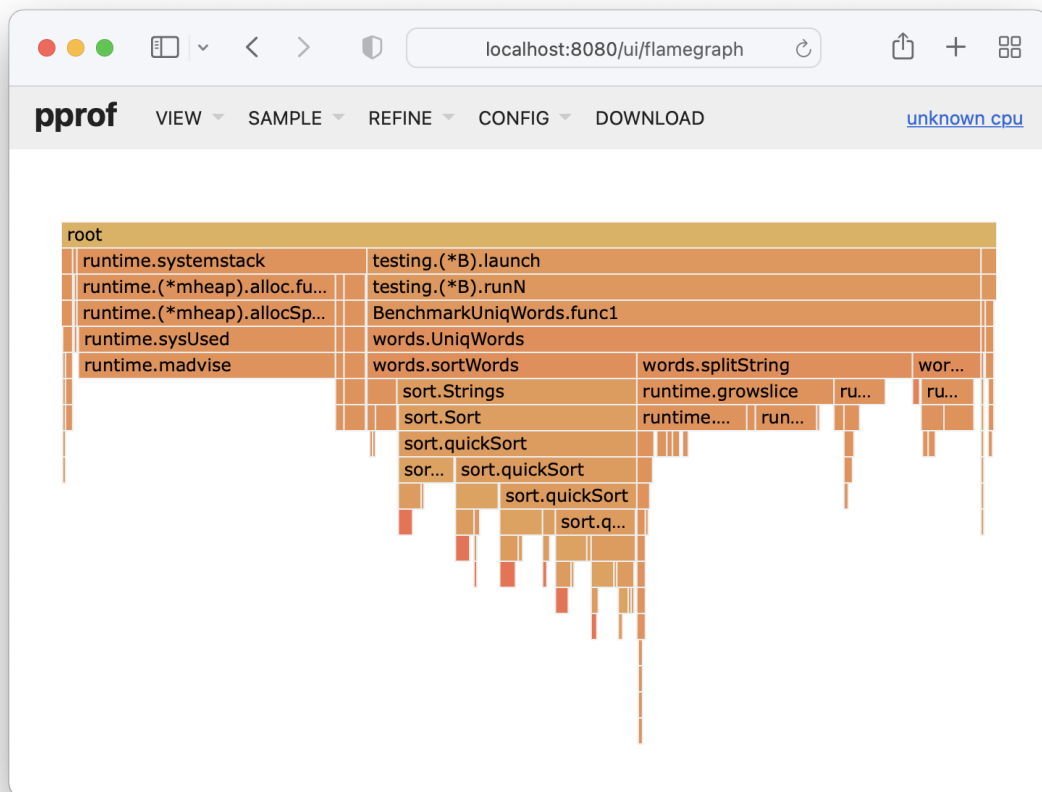
Команда требует установленного [Graphviz](#). Смотрите инструкцию для своей ОС на сайте.



А можно и вовсе запустить rprof с веб-интерфейсом:

```
$ go tool pprof -http=localhost:8080 cpu.prof
```

Тогда в браузере можно посмотреть и топ вызовов, и граф, и flame-граф:



Хотя как по мне, смотреть в консоли в режиме `peek` даже удобнее.

Шаг 5 · Профайлер · Stepik

Кто съел время?

Я написал функцию `JoinWords()`, которая объединяет слова из двух фраз в единый список:

```
// JoinWords combines words from two strings, removes duplicates
// and returns resulting words in sorted order.
func JoinWords(first, second string) []string {
    words1 := split(first)
    words2 := split(second)
    words := join(words1, words2)
    words = lower(words)
    words = sorted(words)
    words = uniq(words)
    return words
}
```

И бенчмарк:


```

func BenchmarkJoinWords(b *testing.B) {
    rand.Seed(0)
    for _, size := range []int{10, 100, 1000, 10000} {
        name := fmt.Sprintf("JoinWords-%d", size)
        s1 := randomPhrase(size)
        s2 := randomPhrase(size)
        b.Run(name, func(b *testing.B) {
            for n := 0; n < b.N; n++ {
                JoinWords(s1, s2)
            }
        })
    }
}

```

Запустите бенчмарк с профайлером, изучите результаты с помощью `pprof` и расположите функции (`split`, `join` и прочие) от самой медленной к самой быстрой.

Полный код возьмите [в песочнице](#). Запускать придется локально — профайлер в песочнице не работает. Перед запуском должно получиться три файла:

```

join
├─ go.mod
├─ join.go
└─ join_test.go

```

Хорошие новости, верно!

`sorted()`

`uniq()`

`split()`

`join()`

`lower()`

```

→ join git:(master) go build
→ join git:(master) GOGC=off go test -bench=. -cpuprofile=cpu.prof
goos: linux
goarch: amd64
pkg: wtf/join
cpu: Intel(R) Core(TM) i5-4440 CPU @ 3.10GHz
BenchmarkJoinWords/JoinWords-10-4          422427          2774
ns/op

```

```
BenchmarkJoinWords/JoinWords-100-4          41622          30334
ns/op
BenchmarkJoinWords/JoinWords-1000-4          2740           442679
ns/op
BenchmarkJoinWords/JoinWords-10000-4         222           5379383
ns/op
PASS
```

```
ok      wtf/join      6.170s
```

```
→ join git:(master) l
```

```
итого 3,0M
```

```
drwxrwxrwx 1 root root 4,0K мая 30 20:37 .
```

```
drwxrwxrwx 1 root root 4,0K мая 30 20:34 ..
```

```
-rwxrwxrwx 1 root root 11K мая 30 20:37 cpu.prof
```

```
-rwxrwxrwx 1 root root 25 мая 30 20:35 go.mod
```

```
-rwxrwxrwx 1 root root 1,4K мая 30 20:36 join.go
```

```
-rwxrwxrwx 1 root root 3,0M мая 30 20:37 join.test
```

```
-rwxrwxrwx 1 root root 1,4K мая 30 20:36 join_test.go
```

```
→ join git:(master) go tool pprof cpu.prof
```

```
File: join.test
```

```
Type: cpu
```

```
Time: May 30, 2022 at 8:37pm (MSK)
```

```
Duration: 6.15s, Total samples = 6.24s (101.47%)
```

```
Entering interactive mode (type "help" for commands, "o" for options)
```

```
(pprof) top10
```

```
Showing nodes accounting for 4670ms, 74.84% of 6240ms total
```

```
Dropped 58 nodes (cum <= 31.20ms)
```

```
Showing top 10 nodes out of 61
```

flat	flat%	sum%	cum	cum%	
1270ms	20.35%	20.35%	2780ms	44.55%	sort.doPivot
1160ms	18.59%	38.94%	1160ms	18.59%	cmpbody
370ms	5.93%	44.87%	560ms	8.97%	strings.Fields
350ms	5.61%	50.48%	410ms	6.57%	runtime.heapBitsSetType
350ms	5.61%	56.09%	350ms	5.61%	runtime.memclrNoHeapPointers
340ms	5.45%	61.54%	1640ms	26.28%	sort.StringSlice.Less
290ms	4.65%	66.19%	680ms	10.90%	sort.insertionSort
230ms	3.69%	69.87%	230ms	3.69%	sort.StringSlice.Swap
160ms	2.56%	72.44%	160ms	2.56%	runtime.memmove
150ms	2.40%	74.84%	1000ms	16.03%	wtf/join.uniq

```
(pprof) top10 -cum
```

```
Showing nodes accounting for 1.71s, 27.40% of 6.24s total
```

```
Dropped 58 nodes (cum <= 0.03s)
```

```
Showing top 10 nodes out of 61
```

flat	flat%	sum%	cum	cum%
------	-------	------	-----	------

0	0%	0%	5.91s	94.71%	testing.(*B).runN
0	0%	0%	5.76s	92.31%	testing.(*B).launch
0	0%	0%	5.75s	92.15%	
wtf/join.BenchmarkJoinWords.func1					
0	0%	0%	5.75s	92.15%	wtf/join.JoinWords
0	0%	0%	3.64s	58.33%	sort.Strings (inline)
0	0%	0%	3.64s	58.33%	wtf/join.sorted (inline)
0.02s	0.32%	0.32%	3.61s	57.85%	sort.Sort
0.08s	1.28%	1.60%	3.59s	57.53%	sort.quickSort
1.27s	20.35%	21.96%	2.78s	44.55%	sort.doPivot
0.34s	5.45%	27.40%	1.64s	26.28%	sort.StringSlice.Less
(pprof) peek JoinWords					
Showing nodes accounting for 6.24s, 100% of 6.24s total					
-----+-----					
flat	flat%	sum%	cum	cum%	calls calls% + context
-----+-----					
					0.01s 100% testing.
(*B).runN					
0	0%	0%	0.01s	0.16%	
wtf/join.BenchmarkJoinWords					
					0.01s 100%
wtf/join.randomPhrase					
-----+-----					
					5.75s 100% testing.
(*B).runN					
0	0%	0%	5.75s	92.15%	
wtf/join.BenchmarkJoinWords.func1					
					5.75s 100%
wtf/join.JoinWords					
-----+-----					
					5.75s 100%
wtf/join.BenchmarkJoinWords.func1					
0	0%	0%	5.75s	92.15%	
wtf/join.JoinWords					
					3.64s 63.30%
wtf/join.sorted (inline)					
					1s 17.39% wtf/join.uniq
(inline)					
					0.56s 9.74% wtf/join.split
(inline)					
					0.28s 4.87% wtf/join.join
(inline)					
					0.27s 4.70% wtf/join.lower

```
(inline)
```

Шаг 6 · Профайлер · Stepik

Анализируем memory-профиль

Помимо времени выполнения, профилирование помогает замерить использование памяти.

Запускаем бенчмарки с настройкой `mempfile`:

```
$ GOGC=off go test -bench=. -mempfile=mem.prof
```

Открываем собранный профиль в утилите `pprof`:

```
$ go tool pprof mem.prof
```

```
Type: alloc_space
```

```
Time: May 27, 2022 at 12:30am (+06)
```

```
Entering interactive mode (type "help" for commands, "o" for options)
(pprof)
```

Работают уже знакомые нам команды, только теперь они показывают использованную память, а не время занятости процессора.

`topN` — самые прожорливые функции по чистому использованию памяти (без учета дочерних функций):

```
(pprof) top10
```

```
Showing nodes accounting for 4005.14MB, 99.44% of 4027.75MB total
```

```
Dropped 27 nodes (cum <= 20.14MB)
```

flat	flat%	sum%	cum	cum%	
2385.19MB	59.22%	59.22%	2385.19MB	59.22%	words.splitString (inline)
1180.04MB	29.30%	88.52%	1180.04MB	29.30%	words.UniqWords (inline)
439.91MB	10.92%	99.44%	457.91MB	11.37%	words.sortWords (inline)
0	0%	99.44%	4008.76MB	99.53%	testing.(*B).launch
0	0%	99.44%	4025.25MB	99.94%	testing.(*B).runN
0	0%	99.44%	4023.14MB	99.89%	words.BenchmarkUniqWords.func1
0	0%	99.44%	4023.14MB	99.89%	words.UniqWords

`topN -cum` — самые прожорливые по кумулятивному использованию памяти (с учетом дочерних функций):

```
(pprof) top10 -cum
```

```
Showing nodes accounting for 4005.14MB, 99.44% of 4027.75MB total
```

```
Dropped 27 nodes (cum <= 20.14MB)
```

flat	flat%	sum%	cum	cum%	
0	0%	0%	4025.25MB	99.94%	testing.(*B).runN
0	0%	0%	4023.14MB	99.89%	words.BenchmarkUniqWords.func1
0	0%	0%	4023.14MB	99.89%	words.UniqWords
0	0%	0%	4008.76MB	99.53%	testing.(*B).launch
2385.19MB	59.22%	59.22%	2385.19MB	59.22%	words.splitString (inline)
1180.04MB	29.30%	88.52%	1180.04MB	29.30%	words.uniqWords (inline)
439.91MB	10.92%	99.44%	457.91MB	11.37%	words.sortWords (inline)

`peek` — разбивка использования памяти по дочерним функциям:

```
(pprof) peek words.UniqWords
```

Showing nodes accounting for 4027.75MB, 100% of 4027.75MB total

flat	flat%	sum%	cum	cum%	calls	calls%	+ context
			4023.14MB	100%			
					words.BenchmarkUniqWords.func1		
0	0%	0%	4023.14MB	99.89%			words.UniqWords
					2385.19MB	59.29%	
					words.splitString (inline)		
					1180.04MB	29.33%	
					words.uniqWords (inline)		
					457.91MB	11.38%	
					words.sortWords (inline)		

Приятно видеть, что `splitString()` не только отъела половину процессора, но и по использованию памяти далеко всех опередила. Явный кандидат на рефакторинг!

Шаг 7 · Профайлер · Stepik

Кто съел память?

Помните функцию `JoinWords()`, которая объединяет слова из двух фраз в единый список?

```
// JoinWords combines words from two strings, removes duplicates
// and returns resulting words in sorted order.
func JoinWords(first, second string) []string {
    words1 := split(first)
    words2 := split(second)
    words := join(words1, words2)
    words = lower(words)
    words = sorted(words)
```

```

    words = uniq(words)
    return words
}

```

Запустите бенчмарк с профайлером, изучите результаты с помощью rprof и расположите функции (`split`, `join` и прочие) по использованию памяти от большего к меньшему.

Полный код возьмите [в песочнице](#). Запускать придется локально — профайлер в песочнице не работает. Перед запуском должно получиться три файла:

```

join
├─ go.mod
├─ join.go
└─ join_test.go

```

Нюанс: функции `split()` и `join()` занимают примерно одинаковое количество памяти, поэтому могут меняться местами при разных прогонах бенчмарка. Чтобы ответ был принят, указывайте `split()` перед `join()`.

Так точно!

`uniq()`

`split()`

`join()`

`sorted()`

`lower()`

```

→ join git:(master) GOGC=off go test -bench=. -memprofile=mem.prof
goos: linux
goarch: amd64
pkg: wtf/join
cpu: Intel(R) Core(TM) i5-4440 CPU @ 3.10GHz
BenchmarkJoinWords/JoinWords-10-4          444027          2754
ns/op
BenchmarkJoinWords/JoinWords-100-4         41847          30406
ns/op
BenchmarkJoinWords/JoinWords-1000-4        2677          440133
ns/op
BenchmarkJoinWords/JoinWords-10000-4       218          5316363
ns/op
PASS

```

```

ok      wtf/join      5.981s
→ join git:(master) go tool pprof mem.prof
File: join.test
Type: alloc_space
Time: May 30, 2022 at 8:46pm (MSK)
Entering interactive mode (type "help" for commands, "o" for options)
(pprof) peek JoinWords
Showing nodes accounting for 2403.25MB, 100% of 2403.25MB total
-----+-----
      flat  flat%   sum%        cum   cum%   calls  calls% + context
-----+-----
                                0.58MB   100% |   testing.
(*B).runN
      0      0%      0%      0.58MB 0.024% |
wtf/join.BenchmarkJoinWords
                                0.58MB   100% |
wtf/join.randomPhrase
-----+-----
                                2400.15MB 100% |   testing.
(*B).runN
      0      0%      0%  2400.15MB 99.87% |
wtf/join.BenchmarkJoinWords.func1
                                2400.15MB 100% |
wtf/join.JoinWords
-----+-----
                                2400.15MB 100% |
wtf/join.BenchmarkJoinWords.func1
      0      0%      0%  2400.15MB 99.87% |
wtf/join.JoinWords
                                1412.46MB 58.85% |   wtf/join.uniq
(inline)
                                504.35MB 21.01% |   wtf/join.split
(inline)
                                470.34MB 19.60% |   wtf/join.join
(inline)
                                13MB    0.54% |
wtf/join.sorted (inline)
-----+-----

```

Дополнительное чтение

[Profiling Go Programs](#)

Многозадачность. Резюме

1

Горутины

Основа многозадачности в Go — горутины. Это функции, запущенные через `go`:

```
func main() {
    var wg sync.WaitGroup
    wg.Add(2)
    go func() {
        defer wg.Done()
        fmt.Println("worker 1")
    }()
    go func() {
        defer wg.Done()
        fmt.Println("worker 2")
    }()
    wg.Wait()
}
```

Среда исполнения Go жонглирует горутинами, распределяя их по потокам операционной системы, которые, в свою очередь, выполняются на разных ядрах CPU. По сравнению с потоками ОС горутины очень легкие, так что их можно создавать сотнями и тысячами.

Горутины полностью независимы. `main()` — тоже горутина. Только среда исполнения Go запускает ее неявно, при старте программы.

2

Каналы

Горутины могут передавать друг другу значения через каналы. Канал — это окошко, в которое одна горутина может что-нибудь бросить, а вторая — поймать.

```
func main() {
    messages := make(chan string)

    go func() { messages <- "ping" }()

    msg := <-messages
    fmt.Println(msg)
}
```


Передача значения через канал — синхронная операция. Когда горутина пишет значение в канал `ch <- val`, она блокируется и ждет, пока с другой стороны кто-нибудь прочитает значение из канала `val := <-ch`. Только после этого она возобновит выполнение.

Закрывать канал

Чтобы сигнализировать читателям канала, что значений больше не будет, горутина-писатель закрывает канал функцией `close()`:

```
out := make(chan string)
go func() {
    defer close(out)
    for _, word := range words {
        out <- word
    }
}()
```

Читатель проверяет статус канала вторым значением при считывании:

```
for {
    word, ok := <-in
    if !ok {
        break
    }
}
```

Пока канал открыт, читатель получает очередное значение и статус `true`. Если канал закрыт, читатель получает нулевое значение и статус `false`.

Закрывать канал можно только один раз. Повторное закрытие или попытка записи в закрытый канал приведет к панике.

Закрывать канал стоит с единственной целью — сообщить его читателям, что все данные отправлены. Если читателей у канала нет, то и закрывать его не нужно. Когда канал перестанет использоваться, сборщик мусора Go освободит занятые им ресурсы — вне зависимости от того, закрыт канал или нет.

range по каналу

`range` автоматически считывает очередное значение из канала и проверяет, не закрыт ли тот. Если закрыт — выходит из цикла:

```
for word := range in {
    fmt.Println(word)
}
```

`range` по каналу возвращает одно значение, а не пару, в отличие от `range` по срезу.

Направленные каналы

Каналу можно задать направление:

- `chan`: для чтения и записи (по умолчанию);
- `chan<-`: только для записи (send-only);
- `<-chan`: только для чтения (receive-only).

Из send-only канала нельзя читать, а в receive-only канал нельзя писать (и закрыть тоже нельзя).

Обычно каналы инициализируют для чтения и записи, а в параметрах конкретных функций заявляют как однонаправленные. Go конвертирует обычный канал в направленный автоматически:

```
stream := make(chan int)

go func(in chan<- int) {
    in <- 42
}(stream)

func(out <-chan int) {
    fmt.Println(<-out)
}(stream)
```

Буферизованные каналы

Буферизованные каналы работают как маленькая очередь: у них есть собственный буфер фиксированного размера, в котором можно хранить значения. Пока в буфере есть свободные места, запись в канал не блокирует горутину. Аналогично, пока в буфере есть значения, чтение из канала не блокирует горутину.

```
stream := make(chan int, 3)
stream <- 1
stream <- 2
stream <- 3

fmt.Println(<-stream)
fmt.Println(<-stream)
```

По умолчанию, если не указать размер буфера, будет создан канал с буфером размера 0.

На буферизованных каналах работают встроенные функции `len()` и `cap()`:

```
stream := make(chan int, 3)
stream <- 7
```

```
fmt.Println(cap(stream), len(stream))  
// 3 1
```

Если закрыть буферизованный канал, то при чтении он будет отдавать значения из буфера и признак `true`. Когда все значения выбраны — станет отдавать нулевое значение и признак `false`, как обычный канал.

```
stream := make(chan int, 1)  
stream <- 7  
close(stream)
```

```
val, ok := <-stream  
fmt.Println(val, ok)  
// 7 true
```

```
val, ok = <-stream  
fmt.Println(val, ok)  
// 0 false
```

nil-канал

Как у любого типа в Go, у каналов тоже есть нулевое значение. Это `nil`:

- Запись в nil-канал навсегда блокирует горутину.
- Чтение из nil-канала навсегда блокирует горутину.
- Закрывание nil-канала приводит к панике.

3

Конвейер

Конвейер — это последовательность операций, каждая из которых принимает на входе данные, обрабатывает их определенным образом и отдает на выход. Входом и выходом для каждой операции выступает канал.

Типичный конвейер выглядит так:

- читатель (начитывает исходные данные из файла, базы или по сети);
- N обработчиков (преобразуют данные, фильтруют, агрегируют, дополняют информацией из внешних источников);
- писатель (записывает результат обработки в файл, базу или по сети).

Канал завершения

Горутина может сигнализировать другим горутинам, что закончила работу. Для этого используют канал результатов:

```

func send(n int) <-chan int {
    out := make(chan int)
    go func() {
        defer close(out)
        for i := 1; i <= n; i++ {
            out <- i
        }
    }()
    return out
}

func main() {
    ch := send(5)
    for n := range ch {
        fmt.Print(n)
    }
}

```

Либо канал завершения, если от горутины не ожидают результатов, а просто хотят дождаться завершения:

```

func work() <-chan struct{} {
    done := make(chan struct{})
    go func() {
        defer close(done)
        fmt.Println("work done")
    }()
    return done
}

func main() {
    done := work()
    <-done
}

```

Канал отмены

Чтобы досрочно завершить горутину, используют канал отмены и инструкцию `select`:

```

func send(cancel chan struct{}, n int) <-chan int {
    out := make(chan int)
    go func() {
        defer close(out)
        for i := 1; i <= n; i++ {
            select {

```

```

        case out <- i:
        case <-cancel:
            return
    }
}
}()
return out
}

func main() {
    cancel := make(chan struct{})
    defer close(cancel)
    ch := send(cancel, 100)
    fmt.Println(<-ch)
}

```

4

select

Селект отчасти похож на `switch`, только создан специально для работы с каналами. Вот что он делает:

1. Проверяет, какие ветки не заблокированы.
2. Если таких веток несколько, выбирает одну из них случайным образом и выполняет ее.
3. Если все ветки заблокированы и нет блока `default`, блокирует выполнение, пока хотя бы одна ветка не разблокируется.
4. Если все ветки заблокированы и есть блок `default` — выполняет его.

Селект используют для управления потоком данных в конвейерах:

```

func work(in1, in2 <-chan int) <-chan int {
    out := make(chan int)
    go func() {
        defer close(out)
        for {
            select {
            case out <- <-in1:
            case out <- <-in2:
            case <-cancel:
                return
            }
        }
    }
}
}()

```

```
    return out
}
```

Для отмены горутин:

```
func work(cancel chan struct{}, in, out chan int) {
    for val := range in {
        select {
            case out <- val:
            case <-cancel:
                return
        }
    }
}
```

Для неблокирующих операций:

```
func work(ch chan int) {
    select {
    case ch <- val:
    default:
        return errors.New("empty")
    }
}
```

И многого другого.

5

Время

After

`time.After()` возвращает канал, который изначально пуст, а через `timeout` времени отправляет в него значение. Подходит, чтобы таймаутить операции:

```
func withTimeout(fn func() int, timeout time.Duration) (int, error) {
    var result int

    done := make(chan struct{})
    go func() {
        defer close(done)
        result = fn()
    }()

    select {
```

```

    case <-done:
        return result, nil
    case <-time.After(timeout):
        return 0, errors.New("timeout")
}

```

Таймер

Таймер `time.Timer` — это структура с каналом `C`, в который он запишет текущее время, когда сработает. Подходит, чтобы планировать операции в будущем:

```

timer := time.NewTimer(100 * time.Millisecond)
go func() {
    eventTime = <-timer.C
    work()
}()

```

`Stop()` останавливает таймер и возвращает `true`, если он еще не успел сработать, и `false` в противном случае:

```

timer := time.NewTimer(100 * time.Millisecond)
go func() {
    eventTime = <-timer.C
    work()
}()

if !timer.Stop() {
    fmt.Println("too late to cancel")
}

```

Часто удобнее воспользоваться оберткой `time.AfterFunc()`:

```

work := func() {
    fmt.Println("work done")
}

time.AfterFunc(100*time.Millisecond, work)

```

Она ждет в течение времени `d`, после чего выполняет функцию `f`. Возвращает таймер, через который можно отменить выполнение, пока оно не началось.

Тикер

Тикер похож на таймер, только срабатывает не один раз, а регулярно, пока не остановить. Подходит для периодических операций:

```

ticker := time.NewTicker(50 * time.Millisecond)
defer ticker.Stop()

go func() {
    for {
        at := <-ticker.C
        work(at)
    }
}()

```

`NewTicker(d)` создает тикер, который с периодичностью `d` отправляет текущее время в канал `C`. Тикер обязательно надо рано или поздно остановить через `Stop()`, чтобы он освободил занятые ресурсы.

Если получатель опаздывает, тикер пропускает тики, чтобы подстроиться под него.

6

Контекст

Основное назначение контекста — отмена операций.

Ручная отмена:

```

func work(ctx context.Context) error {
    done := make(chan struct{})

    go func() {
        defer close(done)
        fmt.Println("do stuff")
    }()

    select {
    case <-done:
        return nil
    case <-ctx.Done():
        return ctx.Err()
    }
}

func main() {
    ctx := context.Background()
    ctx, cancel := context.WithCancel(ctx)
    defer cancel()
}

```



```
    work(ctx)
}
```

Отмена по таймауту:

```
func main() {
    ctx := context.Background()
    ctx, cancel := context.WithCancel(ctx)
    ctx, cancel = context.WithTimeout(ctx, 3*time.Second)
    defer cancel()
    work(ctx)
}
```

Отмена по дедлайну:

```
func main() {
    ctx := context.Background()
    ctx, cancel := context.WithCancel(ctx)
    deadline := time.Now().Add(3 * time.Second)
    ctx, cancel = context.WithDeadline(ctx, deadline)
    defer cancel()
    work(ctx)
}
```

Контекст — это матрешка. Объект контекста неизменяемый. Чтобы добавить контексту новые свойства, создают новый контекст («дочерний») на основе старого («родительского»). Из таймаутов, наложенных родительским и дочерним контекстами, всегда срабатывает более жесткий. Дочерние контексты могут только ужесточить таймаут родительского, но не ослабить его.

Многократная отмена безопасна. Вызывать `cancel()` контекста можно сколько угодно. Первая отмена работает, а остальные будут проигнорированы.

Еще одна задача контекста — передача дополнительной информации о вызове. За это отвечает `context.WithValue()`, которая создает контекст со значением по ключу. Использовать эту возможность, кроме как для обработки HTTP-запросов, не стоит.

7

Синхронизация

`sync.WaitGroup`

Группа ожидания позволяет дождаться, пока отработают запущенные горутини. Внутри у нее счетчик, который мы увеличиваем методом `Add()` и уменьшаем методом `Done()`. Метод `Wait()` блокирует выполнение горутини, пока счетчик не достигнет 0.

```
func main() {
    var wg sync.WaitGroup

    wg.Add(1)
    go say(&wg, 1, "go is awesome")

    wg.Add(1)
    go say(&wg, 2, "cats are cute")

    wg.Wait()
}
```

Гонки

Ситуация, когда несколько горутин одновременно обращаются к одной и той же переменной, причем как минимум одна из горутин ее модифицирует — называется гонками. Не всегда гонки приводят к runtime-ошибке, поэтому Go предоставляет специальный инструмент — детектор гонок. Он включается флагом `race`, который доступен для команд `test`, `run`, `build` и `install`.

```
func main() {
    var total int

    var wg sync.WaitGroup
    wg.Add(2)

    go func() {
        defer wg.Done()
        total++
    }()

    go func() {
        defer wg.Done()
        total++
    }()

    wg.Wait()
    fmt.Println(total)
}
```

```
$ go run -race race.go
```

```
=====
WARNING: DATA RACE
...
=====
```

2

Found 1 data race(s)

Каналы безопасны для одновременного доступа и не создают гонок.

Способы борьбы с гонками:

- исключить одновременную модификацию данных (как правило, через каналы);
- синхронизировать доступ через мьютексы;
- использовать только атомарные операции.

sync.Mutex

Мьютекс защищает общие данные и участки кода от одновременного доступа:

```
var lock sync.Mutex

go func(lock *sync.Mutex) {
    lock.Lock()
    defer lock.Unlock()
    total++
}(&lock)
```

Мьютекс гарантирует, что участок кода между `Lock()` и `Unlock()` выполняется только одной горутиной в каждый момент времени.

Мьютекс используют в следующих ситуациях:

- Если несколько горутин модифицируют одни и те же данные.
- Если одна горутина модифицирует данные, а несколько читают.

Если все горутины только читают данные, мьютекс не нужен.

sync.RWMutex

Мьютекс `sync.RWMutex` разделяет писателей и читателей. У него две пары методов:

- `Lock()` / `Unlock()` запирает мьютекс для чтения и записи.
- `RLock()` / `RUnlock()` запирает мьютекс для чтения.

Работает так:

- Если горутина заперла мьютекс через `Lock()`, прочие заблокируются при попытке `Lock()` или `RLock()`.
- Если горутина заперла мьютекс через `RLock()`, прочие тоже могут запирать его через `RLock()`, и не блокируются при этом.

- Если хотя бы одна горутина заперла мьютекс через `RLock()`, прочие заблокируются при попытке `Lock()`.

Получается схема «один одновременный писатель, много читателей».

sync.Map

`sync.Map` — это карта, которую безопасно использовать из нескольких горутин. Она нетипизированная и оптимизирована для двух специфических сценариев:

- если значение по ключу записывается только однажды, а читается много раз;
- если горутин работают с непересекающимися множествами ключей (каждая горутина работает только со своими ключами и не трогает чужие).

sync/atomic

Атомарной операция может быть только в одном случае — если она превращается в одну инструкцию процессора. Атомарные операции не требуют блокировок и не создают проблем при одновременном вызове. Даже операции записи.

Атомарных операций немного, и все они сосредоточены в пакете `sync/atomic`:

```
func AddInt32(addr *int32, delta int32) (new int32)
func LoadInt32(addr *int32) (val int32)
func StoreInt32(addr *int32, val int32)
```

`Store()` записывает значение переменной, `Load()` считывает значение переменной а `Add()` увеличивает значение переменной на указанную дельту. Функции принимают адрес переменной, чтобы работать с оригиналом, а не копией. Все они выполняются за одну инструкцию процессора, поэтому безопасны для одновременных вызовов.

8

Дополнительное чтение

Спецификация: [channel](#) • [send](#) • [receive](#) • [go](#) • [select](#)

[Effective Go: Concurrency](#)

[Share Memory By Communicating](#)

[Go Concurrency Patterns](#)

[Advanced Go Concurrency Patterns](#)

[Pipelines and cancellation](#)

[Context](#)

Многозадачность 1. Горутины

Многозадачность 1. Горутины

3.1 Горутины

Давайте обойдемся без долгих рассуждений о многозадачности и параллелизме. Вместо этого сразу напишем многозадачную программу на Go!

Есть у нас функция, которая произносит фразу по словам (с некоторыми задержками):

```
func say(phrase string) {  
    for _, word := range strings.Fields(phrase) {  
        fmt.Printf("Simon says: %s...\n", word)  
        dur := time.Duration(rand.Intn(100)) * time.Millisecond  
        time.Sleep(dur)  
    }  
}
```

Вызываем ее из `main()`:

```
func main() {  
    say("go is awesome")  
}
```

Simon says: go...

Simon says: is...

Simon says: awesome...

Пусть теперь будет две болтушки, каждая из которых озвучивает свою фразу:

```
func main() {  
    say(1, "go is awesome")  
    say(2, "cats are cute")  
}  
  
func say(id int, phrase string) {  
    for _, word := range strings.Fields(phrase) {  
        fmt.Printf("Worker #%d says: %s...\n", id, word)  
        dur := time.Duration(rand.Intn(100)) * time.Millisecond  
        time.Sleep(dur)  
    }  
}
```

```
}  
}
```

Запускаем:

```
Worker #1 says: go...  
Worker #1 says: is...  
Worker #1 says: awesome...  
Worker #2 says: cats...  
Worker #2 says: are...  
Worker #2 says: cute...
```

Неплохо, только срабатывают функции последовательно. Чтобы запустить одновременно, добавим `go` перед вызовом `say()`:

```
func main() {  
    go say(1, "go is awesome")  
    go say(2, "cats are cute")  
    time.Sleep(500 * time.Millisecond)  
}
```

И функции говорят наперебой:

```
Worker #2 says: cats...  
Worker #1 says: go...  
Worker #2 says: are...  
Worker #1 says: is...  
Worker #2 says: cute...  
Worker #1 says: awesome...
```

Вот и все. Написали `go f()` → функция `f()` выполнялась независимо от прочих.

В предыдущих уроках я иногда проводил параллели с Python и JavaScript. В этом модуле не буду, потому что многозадачность в Go сделана много проще и удобнее, чем в этих языках. Если вы привыкли к `async/await` — постарайтесь забыть все, что знаете. Воспринимайте подход Go «с чистого листа».

Функции, запущенные через `go`, называются *горутинами* (goroutine). Среда исполнения Go жонглирует горутинами, распределяя их по *потокам* операционной системы (threads), которые, в свою очередь, выполняются на разных *ядрах CPU* (CPU cores). По сравнению с потоками ОС горутини очень легкие, так что их можно создавать сотнями и тысячами.

Возможно, у вас возник вопрос: зачем `time.Sleep()` в `main()`? Сейчас разберемся.

[песочница](#)

Горутины2

Зависимые и независимые горутины

Дело в том, что горутины полностью независимы. Написав `go say(...)`, мы отправили функцию в свободное плавание. `main()` больше нет до нее дела. Поэтому если написать так:

```
func main() {
    go say(1, "go is awesome")
    go say(2, "cats are cute")
}
```

то программа вовсе ничего не напечатает. `main()` завершится прежде, чем успеют выполняться наши болтушки. А поскольку `main()` самая главная, то вместе с ней завершится и вся программа.

`main()` — тоже горутина. Только среда исполнения Go запускает ее неявно, при старте программы. Так что в нашей программе три горутины: `main()`, `say(1)` и `say(2)`. Все независимы друг от друга, единственное ограничение — когда заканчивается `main()`, заканчиваются и все прочие.

Конечно, ожидать окончания горутин через `time.Sleep()` неправильно — мы не знаем, сколько времени они займут. Правильно использовать *группу ожидания* (wait group):

```
func main() {
    var wg sync.WaitGroup

    wg.Add(1)
    go say(&wg, 1, "go is awesome")

    wg.Add(1)
    go say(&wg, 2, "cats are cute")

    wg.Wait()
}

func say(wg *sync.WaitGroup, id int, phrase string) {
    for _, word := range strings.Fields(phrase) {
        fmt.Printf("Worker #%d says: %s...\n", id, word)
        dur := time.Duration(rand.Intn(100)) * time.Millisecond
        time.Sleep(dur)
    }
    wg.Done()
}
```

У `wg` внутри живет счетчик. Вызывая `wg.Add(1)`, мы увеличиваем его на единицу. `wg.Done()`, напротив, уменьшает счетчик на единицу. `wg.Wait()` блокирует горутину (в данном случае `main`) до тех пор, пока счетчик не обнулится. Таким образом, `main()` дожидается, пока отработают `say(1)` и `say(2)`, после чего завершается:

```
Worker #2 says: cats...
Worker #1 says: go...
Worker #2 says: are...
Worker #1 says: is...
Worker #2 says: cute...
Worker #1 says: awesome...
```

У такого кода есть недостаток: мы смешали бизнес-логику (`say`) с логикой многозадачности (`wg`). В результате теперь `say()` не получится запустить в обычном, однозадачном коде.

В Go так не принято. Обычно «многозадачную» логику стараются отделить от «бизнес». Для этого используют отдельные функции. В простых случаях, вроде нашего, подойдут даже анонимные:

```
func main() {
    var wg sync.WaitGroup
    wg.Add(2)

    go func() {
        defer wg.Done()
        say(1, "go is awesome")
    }()

    go func() {
        defer wg.Done()
        say(2, "cats are cute")
    }()

    wg.Wait()
}

func say(id int, phrase string) {
    for _, word := range strings.Fields(phrase) {
        fmt.Printf("Worker #%d says: %s...\n", id, word)
        dur := time.Duration(rand.Intn(100)) * time.Millisecond
        time.Sleep(dur)
    }
}
```

Обратите внимание:

- мы заранее знаем, что будет две горутины, поэтому сразу вызываем `wg.Add(2)`;
- анонимные функции запускаются через `go` точно так же, как обычные;
- `defer wg.Done()` гарантирует, что горутина уменьшит счетчик перед выходом, даже если `say()` сломается с паникой;
- сама `say()` знать не знает о многопоточности и живет счастливой жизнью.

[песочница](#)

Горутины3

Сколько цифр в каждом слове?

Есть функция, которая считает количество цифр в слове:

```
func countDigits(str string) int {
    count := 0
    for _, char := range str {
        if unicode.IsDigit(char) {
            count++
        }
    }
    return count
}
```

Напишите функцию, которая возьмет исходную строку, разобьет ее на слова и посчитает количество цифр в каждом слове. Причем подсчет для каждого слова запустите в отдельной горутине.

Мы пока не обсуждали, как модифицировать общие данные из разных горутин. Поэтому в задании есть готовая переменная `syncStats`, к которой можно спокойно обращаться из горутин.

[песочница](#)

Горутины в цикле

Если вы запускаете горутины в цикле, вот так:

```
values := []int{1, 2, 3, 4, 5}
for _, val := range values {
    go func() {
        fmt.Printf("%d ", val)
    }()
}
```

— то наверняка ожидаете, что напечатается `1 2 3 4 5` (в любом порядке). Но вместо этого увидите `5 5 5 5 5`.

Дело в том, что `val` — общая переменная для всех горутин. И к моменту, когда они стартовали, цикл `for` уже закончил работу, поэтому `val` = 5. Вот все горутины и печатают 5.

Есть два способа решения проблемы. Первый — передавать `val` параметром, чтобы у каждой горутины была своя копия:

```
for _, val := range values {
    go func(val int) {
        fmt.Printf("%d ", val)
    }(val)
}
```

Второй — создавать копию переменной на каждой итерации цикла (так каждая горутина тоже будет работать со своей копией):

```
for _, val := range values {
    val := val
    go func() {
        fmt.Printf("%d ", val)
    }()
}
```

О заданиях в этом модуле

Упражнения в этом модуле используют одну и ту же задачу — подсчет цифр в словах. Так сделано специально. Понятно, что в реальной жизни определить количество цифр проще всего обычным циклом. Но многозадачность — сама по себе сложная тема. Поэтому лучше осваивать ее на одном и том же игрушечном примере. Затем вы сможете применить полученные навыки на более сложных задачах.

Еще нюанс. Извне многозадачная программа выглядит так же, как обычная. Мне нужно видеть «внутренности» вашей программы, чтобы нормально ее проверить. Поэтому сделаем так:

- Я даю полный код в песочнице, чтобы вы могли его отлаживать.
- Вы отправляете в качестве решения не весь код, а только фрагмент, отмеченный комментариями «начало решения» и «конец решения».

И заглядывайте в раздел «решения».

Sample Input:

Sample Output:

PASS

```

package main

import (
    "fmt"
    "strings"
    "sync"
    "unicode"
)

// counter stores the number of digits in each word.
// each key is a word and value is the number of digits in the word.
type counter map[string]int

// countDigitsInWords counts digits in phrase words
func countDigitsInWords(phrase string) counter {
    words := strings.Fields(phrase)
    syncStats := sync.Map{}

    var wg sync.WaitGroup

    // начало решения
    // wg.Add(len(words))

    // Посчитайте количество цифр в словах,
    // используя отдельную горутину для каждого слова.
    for _, word := range words {
        wg.Add(1) // лучше наращивать счетчик горутин в самом цикле,
        // мало-ли что
        go func(w string) {
            defer wg.Done()
            count := countDigits(w)
            syncStats.Store(w, count) // потоко- (горутино-) безопасный
            // словарь уот так уот
        }(word)
    }
    wg.Wait()
    // конец решения

    return asStats(syncStats)
}

// countDigits returns the number of digits in a string
func countDigits(str string) int {

```

```

count := 0
for _, char := range str {
    if unicode.IsDigit(char) {
        count++
    }
}
return count
}

// asStats converts stats from sync.Map to ordinary map
func asStats(m sync.Map) counter {
    stats := counter{}
    m.Range(func(word, count any) bool {
        stats[word.(string)] = count.(int)
        return true
    })
    return stats
}

// printStats prints words and their digit counts
func printStats(stats counter) {
    for word, count := range stats {
        fmt.Printf("%s: %d\n", word, count)
    }
}

func main() {
    phrase := "One two thr33 4068"
    counts := countDigitsInWords(phrase)
    printStats(counts)
}

```

Горутины4

Каналы

Запускать горутины в неприличных количествах — это, конечно, здорово. Но как им обмениваться данными? В Go горутины могут передавать друг другу значения через *каналы* (channels). Канал — это окошко, в которое одна горутина может что-нибудь бросить, а вторая — поймать.





Горутину B передает значение X в горутину A. Холст, масло.

Вот как это работает:

```
func main() {
    // Канал создается через `make(chan тип)`
    // и может передавать только значения указанного типа:
    messages := make(chan string)

    // Чтобы отправить значение в канал,
    // используют синтаксис `канал <-`
    // Отправим «пинг»:
    go func() { messages <- "ping" }()

    // Чтобы получить значение из канала,
    // используют синтаксис `<-канал`
    // Получим «пинг» и напечатаем его:
    msg := <-messages
    fmt.Println(msg)
}
```

Когда запустим программу, одна горутина (анонимная) передаст сообщение второй (main) через канал messages:

```
$ go run channels.go
ping
```

Передача значения через канал — синхронная операция. Когда горутина-отправитель говорит `messages <- "ping"`, она блокируется и ждет, пока с другой стороны кто-нибудь примет значение через `<-messages`. Только после этого она возобновит выполнение:

```
func main() {
    messages := make(chan string)

    go func() {
        fmt.Println("B: Sending message...")
        messages <- "ping" // (1)
        fmt.Println("B: Message sent!") // (2)
    }()

    fmt.Println("A: Doing some work...")
}
```

```

    time.Sleep(500 * time.Millisecond)
    fmt.Println("A: Ready to receive a message...")

    <-messages                                // (3)

    fmt.Println("A: Messege received!")
    time.Sleep(100 * time.Millisecond)
}

```

```

$ go run channels.go
A: Doing some work...
B: Sending message...
A: Ready to receive a message...
A: Messege received!
B: Message sent!

```

Отправив сообщение в канал ❶, горутинa B заблокировалась. Только после того, как горутинa A приняла сообщение ❸, горутинa B возобновила выполнение и смогла напечатать «message sent» ❷.

Таким образом, через каналы не только передают данные, но еще и используют их, чтобы синхронизировать независимые горутинy. Это нам еще пригодится.

[песочница](#)

Горутинy5

Канал с результатами

В программировании на каждом шагу встречается шаблон «поставщик — потребитель»:

- поставщик предоставляет данные;
- потребитель получает их и обрабатывает.

В этом и следующих заданиях посмотрим, как поставщик и потребитель могут взаимодействовать через каналы.

Работаем с функцией, которая считает цифры по словам:

```

type counter map[string]int

func countDigitsInWords(phrase string) counter {
    words := strings.Fields(phrase)
    // ...
}

```

```
    return stats
}
```

Сделайте вот что:

- Запустите одну горутины.
- В горутике пройдитесь по словам, посчитайте количество цифр в каждом, и запишите в канал `counted`.
- В основной функции считайте значения из канала и заполните `stats`.

Полный код — в песочнице. Отправляйте в качестве решения только фрагмент, отмеченный комментариями «начало решения» и «конец решения».

P.S. Если вы уже работали с многозадачностью в Go, код этого урока может выглядеть несколько... неканонично, если не сказать «топорно». Так сделано специально, чтобы не перегружать тех, кто с многозадачностью не знаком. Через пару уроков мы освоим все основные концепции, и станет красиво ٩

И пожалуйста, не публикуйте решения с конструкциями и приемами, которые мы пока не прошли в лекциях.

[песочница](#)

Sample Input:

Sample Output:

PASS

```
package main

import (
    "fmt"
    "strings"
    "unicode"
)

// counter хранит количество цифр в каждом слове.
// ключ карты - слово, а значение - количество цифр в слове.
type counter map[string]int

// countDigitsInWords считает количество цифр в словах фразы
func countDigitsInWords(phrase string) counter {
    words := strings.Fields(phrase)
    counted := make(chan int)
```

```

// начало решения

go func() {
    // Пройдите по словам,
    for _, word := range words {
        // посчитайте количество цифр в каждом,
        count := countDigits(word)
        // и запишите его в канал counted
        counted <- count
    }
}()

// Считайте значения из канала counted
// и заполните stats.
stats := counter{}
for _, word := range words {
    // посчитайте количество цифр в каждом,
    cnt := <-counted
    stats[word] = cnt
}
// В результате stats должна содержать слова
// и количество цифр в каждом.

// конец решения

return stats
}

// countDigits возвращает количество цифр в строке
func countDigits(str string) int {
    count := 0
    for _, char := range str {
        if unicode.IsDigit(char) {
            count++
        }
    }
    return count
}

// printStats печатает слова и количество цифр в каждом
func printStats(stats counter) {
    for word, count := range stats {
        fmt.Printf("%s: %d\n", word, count)
    }
}

```



```

    }
}

func main() {
    phrase := "One two thr33 4068"
    stats := countDigitsInWords(phrase)
    printStats(stats)
}

```

Горутины6

Генератор

До сих пор мы предполагали, что функция `countDigitsInWords()` заранее знает все слова:

```

func countDigitsInWords(phrase string) counter {
    words := strings.Fields(phrase)
    // ...
    return stats
}

```

На такую роскошь рассчитывать в реальной жизни не приходится. Данные могут приходить из базы или по сети, и функция понятия не имеет, сколько всего их будет.

Давайте смоделируем эту ситуацию, и вместо фразы `phrase` будем передавать функцию-генератор `next`. Каждый вызов `next()` выдает следующее слово из какого-то источника. Когда в источнике заканчиваются слова, он передает пустую строку.

Тогда обычная, однозадачная программа будет выглядеть так:

```

// nextFunc returns the next word from the generator
type nextFunc func() string

```

```

// counter stores the number of digits in each word.
// each key is a word and value is the number of digits in the word.
type counter map[string]int

```

```

// countDigitsInWords counts digits in words,
// fetching each word with the next() function
func countDigitsInWords(next nextFunc) counter {
    stats := counter{}

    for {
        word := next()
        if word == "" {

```

```

        break
    }
    count := countDigits(word)
    stats[word] = count
}

return stats
}

// wordGenerator returns a generator,
// which emits words from a phrase.
func wordGenerator(phrase string) nextFunc {
    words := strings.Fields(phrase)
    idx := 0
    return func() string {
        if idx == len(words) {
            return ""
        }
        word := words[idx]
        idx++
        return word
    }
}

func main() {
    phrase := "One two thr33 4068"
    next := wordGenerator(phrase)
    stats := countDigitsInWords(next)
    printStats(stats)
}

```

А теперь добавим многозадачности.

[песочница](#)

Горутиньы7

Выборка из генератора

Работаем с функцией, которая считает цифры по словам:

```

func countDigitsInWords(next nextFunc) counter {
    // ...
}

```

```
    return stats
}
```

Сделайте вот что:

- Запустите одну горутину.
- В горутине пройдите по словам, посчитайте количество цифр в каждом, и запишете в канал `counted`.
- В основной функции считайте значения из канала и заполните `stats`.

Если вы попытаетесь сделать задание по аналогии с предыдущим, возникнет пара проблем:

```
counted := make(chan int)

go func() {
    for {
        word := next()
        count := countDigits(word)
        counted <- count
    }
}()

stats := counter{}
for {
    count := <-out
    // как понять, что слова закончились?
    if ... {
        break
    }
    // откуда взять слово?
    stats[...] = count
}
```

Подумайте, что надо передавать в канал `counted`, чтобы решить обе проблемы. Обратите внимание на тип `pair`.

Полный код — в песочнице. Отправляйте в качестве решения только фрагмент, отмеченный комментариями «начало решения» и «конец решения».

[песочница](#)

Sample Input:

Sample Output:

PASS

```
package main

import (
    "fmt"
    "strings"
    "unicode"
)

// nextFunc возвращает следующее слово из генератора
type nextFunc func() string

// counter хранит количество цифр в каждом слове.
// ключ карты - слово, а значение - количество цифр в слове.
type counter map[string]int

// pair хранит слово и количество цифр в нем
type pair struct {
    word  string
    count int
}

// countDigitsInWords считает количество цифр в словах,
// выбирая очередные слова с помощью next()
func countDigitsInWords(next nextFunc) counter {
    counted := make(chan pair)

    // начало решения

    go func() {
        // Пройдите по словам,
        for {
            word := next()
            // посчитайте количество цифр в каждом,
            p := pair{word, countDigits(word)}
            // и запишите его в канал counted
            counted <- p

            if word == "" { // по пустой строке потом сможем определить
                // конец записи в канал
                break // (закрывать канал мы пока "не умеем")
            }
        }
    }
}
```

```

    }
}()

// Считайте значения из канала counted
// и заполните stats.
stats := counter{}
for {
    p := <-counted
    if p.word == "" {
        break
    }
    stats[p.word] = p.count
}

// В результате stats должна содержать слова
// и количество цифр в каждом.

// конец решения

return stats
}

// countDigits возвращает количество цифр в строке
func countDigits(str string) int {
    count := 0
    for _, char := range str {
        if unicode.IsDigit(char) {
            count++
        }
    }
    return count
}

// printStats печатает слова и количество цифр в каждом
func printStats(stats counter) {
    for word, count := range stats {
        fmt.Printf("%s: %d\n", word, count)
    }
}

// wordGenerator возвращает генератор, который выдает слова из фразы
func wordGenerator(phrase string) nextFunc {
    words := strings.Fields(phrase)

```

```

    idx := 0
    return func() string {
        if idx == len(words) {
            return ""
        }
        word := words[idx]
        idx++
        return word
    }
}

func main() {
    phrase := "One two thr33 4068"
    next := wordGenerator(phrase)
    stats := countDigitsInWords(next)
    printStats(stats)
}

```

Горутинны8

Читатель и счетовод

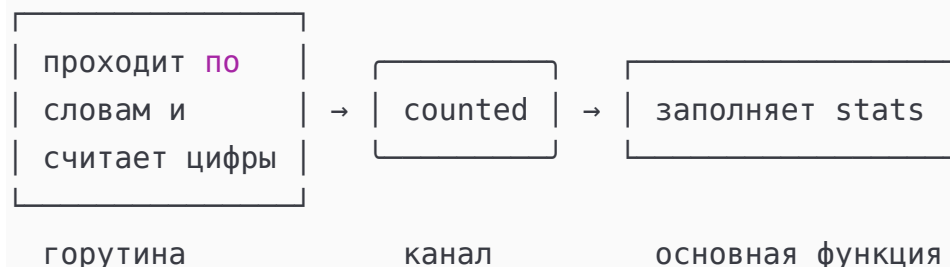
Работаем с функцией, которая считает цифры по словам:

```

func countDigitsInWords(next nextFunc) counter {
    // ...
    return stats
}

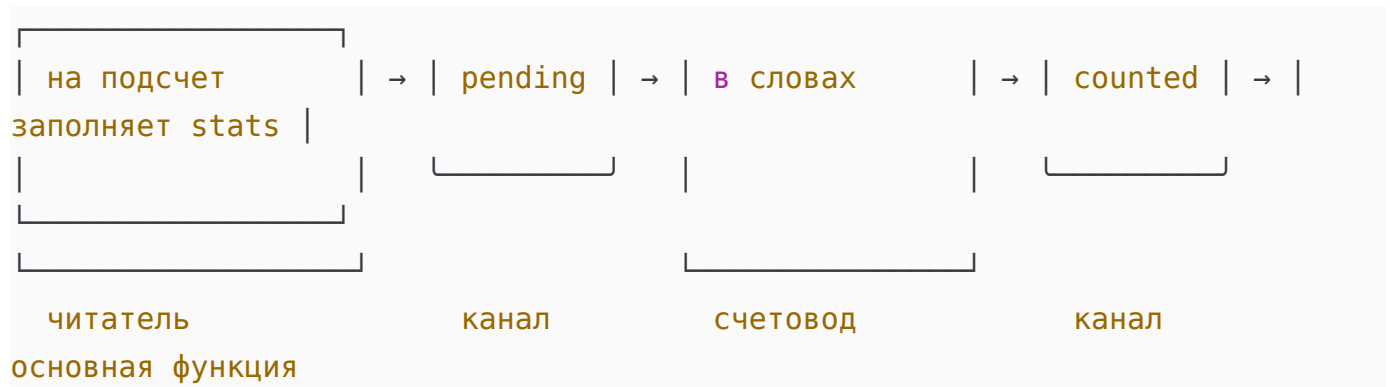
```

В предыдущем задании горутина проходила по словам и считала цифры, а основная функция получала результаты подсчетов и заполняла счетчик:



В более сложных задачах бывает удобно использовать отдельную горутину для считывания данных («читатель»), и отдельную для расчетов («счетовод»). Давайте применим этот подход в нашей функции:





Сделайте вот что:

- Запустите одну горутину, которая проходит по словам и отправляет их в канал `pending` («читатель»).
- Запустите вторую горутину, которая начитывает из `pending`, считает цифры и пишет в канал `counted` («счетовод»).
- В основной функции начитывайте из `counted` и заполняйте итоговый счетчик `stats`.

Полный код — в песочнице. Отправляйте в качестве решения только фрагмент, отмеченный комментариями «начало решения» и «конец решения».

[песочница](#)

Sample Input:

Sample Output:

PASS

```
package main

import (
    "fmt"
    "strings"
    "unicode"
)

// nextFunc возвращает следующее слово из генератора
type nextFunc func() string

// counter хранит количество цифр в каждом слове.
// ключ карты - слово, а значение - количество цифр в слове.
type counter map[string]int

// pair хранит слово и количество цифр в нем
```

```

type pair struct {
    word  string
    count int
}

// countDigitsInWords считает количество цифр в словах,
// выбирая очередные слова с помощью next()
func countDigitsInWords(next nextFunc) counter {
    pending := make(chan string)
    counted := make(chan pair)

    // начало решения

    // отправляет слова на подсчет
    go func() {
        // Пройдите по словам и отправьте их
        for {
            word := next()
            // в канал pending
            pending <- word
            if word == "" {
                break
            }
        }
    }()

    // считает цифры в словах
    go func() {
        // Читайте слова из канала pending,
        for {
            word := <-pending
            // посчитайте количество цифр в каждом,
            p := pair{word, countDigits(word)}
            // и запишите его в канал counted
            counted <- p
            if word == "" {
                break
            }
        }
    }()

    // Читайте значения из канала counted
    // и заполните stats.

```



```

stats := counter{}
for {
    // var p pair
    p := <-counted
    if p.word == "" {
        break
    }
    stats[p.word] = p.count
}

// В результате stats должна содержать слова
// и количество цифр в каждом.

// конец решения

return stats
}

// countDigits возвращает количество цифр в строке
func countDigits(str string) int {
    count := 0
    for _, char := range str {
        if unicode.IsDigit(char) {
            count++
        }
    }
    return count
}

// printStats печатает слова и количество цифр в каждом
func printStats(stats counter) {
    for word, count := range stats {
        fmt.Printf("%s: %d\n", word, count)
    }
}

// wordGenerator возвращает генератор, который выдает слова из фразы
func wordGenerator(phrase string) nextFunc {
    words := strings.Fields(phrase)
    idx := 0
    return func() string {
        if idx == len(words) {
            return ""
        }

```

```

    }
    word := words[idx]
    idx++
    return word
}
}

func main() {
    phrase := "One two thr33 4068"
    next := wordGenerator(phrase)
    stats := countDigitsInWords(next)
    printStats(stats)
}

```

Горутинны9

Горутинны в отдельных функциях

Работаем с функцией, которая считает цифры по словам:

```

func countDigitsInWords(next nextFunc) counter {
    // ...
    return stats
}

```

После разделения логики на читателя и счетовода функция получилась достаточно увесистой:

```

func countDigitsInWords(next nextFunc) counter {
    // ...

    // отправляет слова на подсчет
    go func() {
        // ...
    }()

    // считает цифры в словах
    go func() {
        // ...
    }()

    // заполняет stats
    // ...
}

```

```
    return stats
}
```

Явно просматриваются три логических блока:

1. Отправить слова на подсчет.
2. Посчитать цифры в словах.
3. Заполнить итоговые результаты.

Удобно было бы вынести блоки в отдельные функции, которые обмениваются данными через каналы. Вот так:

```
func countDigitsInWords(next nextFunc) counter {
    pending := make(chan string)
    go submitWords(next, pending)

    counted := make(chan pair)
    go countWords(pending, counted)

    return fillStats(counted)
}
```

Сделайте это.

Полный код — в песочнице. Отправляйте в качестве решения только фрагмент, отмеченный комментариями «начало решения» и «конец решения».

[песочница](#)

Sample Input:

Sample Output:

PASS

```
package main

import (
    "fmt"
    "strings"
    "unicode"
)

// nextFunc возвращает следующее слово из генератора
type nextFunc func() string
```

```
// counter хранит количество цифр в каждом слове.
// ключ карты - слово, а значение - количество цифр в слове.
type counter map[string]int

// pair хранит слово и количество цифр в нем
type pair struct {
    word  string
    count int
}

// countDigitsInWords считает количество цифр в словах,
// выбирая очередные слова с помощью next()
func countDigitsInWords(next nextFunc) counter {
    pending := make(chan string)
    go submitWords(next, pending)

    counted := make(chan pair)
    go countWords(pending, counted)

    return fillStats(counted)
}

// начало решения

// submitWords отправляет слова на подсчет
func submitWords(next nextFunc, ch chan string) {
    for {
        word := next()
        ch <- word
        if word == "" {
            break
        }
    }
}

// countWords считает цифры в словах
func countWords(ch_in chan string, ch_out chan pair) {
    for {
        word := <-ch_in
        ch_out <- pair{word, countDigits(word)}
        if word == "" {
            break
        }
    }
}
```

```

    }
}

// fillStats готовит итоговую статистику
func fillStats(ch chan pair) counter {
    stats := counter{}
    for {
        p := <-ch
        if p.word == "" {
            break
        }
        stats[p.word] = p.count
    }
    return stats
}

// конец решения

// countDigits возвращает количество цифр в строке
func countDigits(str string) int {
    count := 0
    for _, char := range str {
        if unicode.IsDigit(char) {
            count++
        }
    }
    return count
}

// printStats печатает слова и количество цифр в каждом
func printStats(stats counter) {
    for word, count := range stats {
        fmt.Printf("%s: %d\n", word, count)
    }
}

// wordGenerator возвращает генератор, который выдает слова из фразы
func wordGenerator(phrase string) nextFunc {
    words := strings.Fields(phrase)
    idx := 0
    return func() string {
        if idx == len(words) {

```

```

        return ""
    }
    word := words[idx]
    idx++
    return word
}
}

func main() {
    phrase := "One two thr33 4068"
    next := wordGenerator(phrase)
    stats := countDigitsInWords(next)
    printStats(stats)
}

```

Многозадачность 2. Каналы 1

Многозадачность 2. Каналы 1

Мы научились запускать горутин и передавать данные через каналы. Но у каналов еще много интересных возможностей. Давайте разбираться!

Есть программа, которая бьет строку по запятым и фильтрует пустые кусочки:

```

str := "one,two,,four"

in := make(chan string)
go func() {                                     // (1)
    words := strings.Split(str, ",")
    for _, word := range words {
        in <- word
    }
}()

for {                                           // (2)
    word := <-in
    if word != "" {
        fmt.Printf("%s ", word)
    }
}
// one two four

```

Горутина ❶ бьет строку по словам и отправляет их в канал in. Цикл ❷ читает слова из канала и печатает непустые.

К сожалению, программа не работает:

```
$ go run filter.go
fatal error: all goroutines are asleep - deadlock!
```

Причина в бесконечном цикле ❷:

```
for {
    word := <-in
    if word != "" {
        fmt.Printf("%s ", word)
    }
}
```

Как понять, что слов в `in` больше не будет, и пора выходить из цикла? На прошлом уроке решали эту проблему проверкой на пустую строку:

```
for {
    word := <-in
    if word == "" {
        break
    }
}
```

Но теперь пустая строка у нас — корректное значение. Его следует пропустить и перейти к следующему, а не выходить из цикла.

Можно, конечно, договориться так:

- Горутина отправляет в `in` какое-нибудь специальное значение после того, как слова закончились ❶.
- Цикл отслеживает это специальное значение и прерывает работу ❷.

```
const eof = "__EOF__"

str := "one,two,,four"

in := make(chan string)
go func() {
    words := strings.Split(str, ",")
    for _, word := range words {
        in <- word
    }
    in <- eof           // (1)
}()
```

```

for {
    word := <-in
    if word == eof { // (2)
        break
    }
    if word != "" {
        fmt.Printf("%s ", word)
    }
}

```

Но, думаю, вы и сами понимаете, насколько это слабое решение. К счастью, в Go есть нормальное.

[песочница](#)

Каналы2

Заккрыть канал

Мы столкнулись с типичной проблемой взаимодействия двух участников в многозадачной среде:

- писатель пишет значения в канал;
- читатель читает значения из канала;
- как писателю информировать читателя, что значения закончились?

В Go есть механизм, который решает ровно эту задачу:

- писатель может *заккрыть* (close) канал;
- читатель может понять, что канал закрыт.

Писатель закрывает канал функцией `close()`:

```

in := make(chan string)
go func() {
    words := strings.Split(str, ",")
    for _, word := range words {
        in <- word
    }
    close(in)
}()

```

Читатель проверяет статус канала вторым значением при считывании:


```

for {
    word, ok := <-in
    if !ok {
        break
    }
    if word != "" {
        fmt.Printf("%s ", word)
    }
}

```

Допустим, в канал передают строки `one`, `two`, после чего закрывают его. Вот что получит при этом читатель:

```

// in <- "one"
word, ok := <-in
// word = "one", ok = true

// in <- "two"
word, ok := <-in
// word = "two", ok = true

// close(in)
word, ok := <-in
// word = "", ok = false

word, ok := <-in
// word = "", ok = false

word, ok := <-in
// word = "", ok = false

```

Пока канал открыт, читатель получает очередное значение и статус `true`. Если канал закрыт, читатель получает нулевое значение (для строки это `""`) и статус `false`.

Как видно из примера, читать из закрытого канала можно сколько угодно — каждый раз вернется нулевое значение и статус `false`. Это неспроста — через несколько шагов разберемся, зачем так сделано.

Закрыть канал можно только один раз. Повторное закрытие приведет к панике:

```

in := make(chan string)
close(in)
close(in)
// panic: close of closed channel

```

Записать в закрытый канал тоже не получится:

```
in := make(chan string)
go func() {
    in <- "hi"
    close(in)
}()
fmt.Println(<-in)
// hi

in <- "bye"
// panic: send on closed channel
```

Отсюда два важных правила:

1. *Заккрыть канал имеет право только писатель, но не читатель.* Если читатель закроет канал, то писатель словит панику при следующей записи.
2. *Писатель имеет право закрыть канал, только если владеет им единолично.* Если писателей несколько, и один из них закроет канал, то остальные словят панику при следующей записи или попытке закрыть канал со своей стороны.

Всегда ли закрывать канал

Наверняка вы раньше работали с внешними ресурсами (файлами, соединениями БД) — и знаете, что их всегда следует закрывать, чтобы не было утечки. Но канал — не внешний ресурс. Когда канал перестанет использоваться, сборщик мусора Go освободит занятые им ресурсы — вне зависимости от того, закрыт канал или нет.

Закрывать канал стоит с единственной целью — сообщить его читателям, что все данные отправлены. Если читателей у канала нет, то и закрывать его не нужно.

[песочница](#)

Каналы3

Итерирование по каналу

На предыдущем шаге мы заставили читателя постоянно проверять, открыт ли канал:

```
for {
    word, ok := <-in
    if !ok {
        break
    }
    if word != "" {
```

```
        fmt.Printf("%s ", word)
    }
}
```

Довольно громоздко. Чтобы не делать этого вручную, Go поддерживает конструкцию `range` для чтения из канала:

```
for word := range in {
    if word != "" {
        fmt.Printf("%s ", word)
    }
}
```

`range` автоматически считывает очередное значение из канала и проверяет, не закрыт ли тот. Если закрыт — выходит из цикла. Удобно, не правда ли?

Обратите внимание, что `range` по каналу возвращает одно значение, а не пару, в отличие от `range` по срезу. Сравните:

```
// срез
words := []string{"1", "2", "3"}
for idx, val := range words {
    fmt.Println(idx, val)
}

// канал
in := make(chan string)
go func() {
    in <- "1"
    in <- "2"
    in <- "3"
    close(in)
}()
for val := range in {
    fmt.Println(val)
}
```

[песочница](#)

Каналы4

Итерирование с закрывашкой

Работаем с функцией, которая считает цифры по словам:

```
func countDigitsInWords(next nextFunc) counter {
    pending := make(chan string)
    go submitWords(next, pending)

    counted := make(chan pair)
    go countWords(pending, counted)

    return fillStats(counted)
}
```

Напишите код функций:

- `submitWords()` проходит по словам и пишет их в канал `pending`.
- `countWords()` читает из `pending`, считает цифры и пишет в канал `counted`.
- `fillStats()` читает из `counted` и заполняет итоговый счетчик `stats`.

В горутинах-писателях используйте `close()`, чтобы сигнализировать читателям, что значений в канале больше не предвидится. В горутинах-читателях используйте `range` для чтения из каналов.

Полный код — в песочнице. Отправляйте в качестве решения только фрагмент, отмеченный комментариями «начало решения» и «конец решения».

[песочница](#)

Sample Input:

Sample Output:

PASS

```
package main

import (
    "fmt"
    "strings"
    "unicode"
)

// nextFunc возвращает следующее слово из генератора
type nextFunc func() string

// counter хранит количество цифр в каждом слове.
// ключ карты - слово, а значение - количество цифр в слове.
type counter map[string]int
```

```
// pair хранит слово и количество цифр в нем
type pair struct {
    word  string
    count int
}

// countDigitsInWords считает количество цифр в словах,
// выбирая очередные слова с помощью next()
func countDigitsInWords(next nextFunc) counter {
    pending := make(chan string)
    go submitWords(next, pending)

    counted := make(chan pair)
    go countWords(pending, counted)

    return fillStats(counted)
}

// начало решения

// submitWords отправляет слова на подсчет
func submitWords(next nextFunc, out chan string) {
    for {
        word := next()
        if word == "" {
            break
        }
        out <- word
    }
    close(out)
}

// countWords считает цифры в словах
func countWords(in chan string, out chan pair) {
    for word := range in {
        p := pair{word, countDigits(word)}
        out <- p
    }
    close(out)
}

// fillStats готовит итоговую статистику
```

```

func fillStats(in chan pair) counter {
    stats := counter{}
    for p := range in {
        stats[p.word] = p.count
    }
    return stats
}

// конец решения

// countDigits возвращает количество цифр в строке
func countDigits(str string) int {
    count := 0
    for _, char := range str {
        if unicode.IsDigit(char) {
            count++
        }
    }
    return count
}

// printStats печатает слова и количество цифр в каждом
func printStats(stats counter) {
    for word, count := range stats {
        fmt.Printf("%s: %d\n", word, count)
    }
}

// wordGenerator возвращает генератор, который выдает слова из фразы
func wordGenerator(phrase string) nextFunc {
    words := strings.Fields(phrase)
    idx := 0
    return func() string {
        if idx == len(words) {
            return ""
        }
        word := words[idx]
        idx++
        return word
    }
}

func main() {

```

```
phrase := "0ne 1wo thr33 4068"  
next := wordGenerator(phrase)  
stats := countDigitsInWords(next)  
printStats(stats)  
}
```

Каналы5

Направленные каналы

Есть программа, которая фильтрует пустые строки:

```
func main() {  
    str := "one,two,,four"  
    stream := make(chan string)  
    go submit(str, stream)  
    print(stream)  
}  
  
func submit(str string, stream chan string) {  
    words := strings.Split(str, ",")  
    for _, word := range words {  
        stream <- word  
    }  
    close(stream)  
}  
  
func print(stream chan string) {  
    for word := range stream {  
        if word != "" {  
            fmt.Printf("%s ", word)  
        }  
    }  
    fmt.Println()  
}
```

Здесь все работает. Но если вернуться к коду спустя месяц и буду не слишком внимателен, то легко могу все сломать.

Например, закрою канал из функции-читателя:

```
func print(stream chan string) {  
    for word := range stream {  
        if word != "" {  
            fmt.Printf("%s ", word)
```

```

    }
}
close(stream)    // (!)
fmt.Println()
}

```

```

$ go run filter.go
panic: send on closed channel

```

Или случайно прочитаю из канала в функции-писателе:

```

func submit(str string, stream chan string) {
    words := strings.Split(str, ",")
    for _, word := range words {
        stream <- word
    }
    <-stream    // (!)
    close(stream)
}

```

```

$ go run filter.go
fatal error: all goroutines are asleep - deadlock!

```

Ошибки происходят во время выполнения (runtime errors), то есть замечу я их только после того, как запущу программу. А хорошо бы отлавливать еще на этапе компиляции.

Уменьшить путаницу с каналами можно, если задать им *направление* (direction). Каналы бывают:

- `chan`: для чтения и записи (по умолчанию);
- `chan<-`: только для записи (send-only);
- `<-chan`: только для чтения (receive-only).

Функции `submit()` подойдет канал «только для записи»:

```

func submit(str string, stream chan<- string) { // (1)
    words := strings.Split(str, ",")
    for _, word := range words {
        stream <- word
    }
    // <-stream    // (2)
    close(stream)
}

```

В сигнатуре функции ❶ мы указали, что канал только для записи, так что теперь прочитать из него не получится. Если раскомментировать строчку ❷, получим ошибку при компиляции:


```
invalid operation: cannot receive from send-only channel stream
```

Функции `print()` подойдет канал «только для чтения»:

```
func print(stream <-chan string) { // (1)
    for word := range stream {
        if word != "" {
            fmt.Printf("%s ", word)
        }
    }
    // stream <- "oops" // (2)
    // close(stream) // (3)
    fmt.Println()
}
```

В сигнатуре функции ❶ мы указали, что канал только для чтения. Записать в него не получится. Если раскомментировать строчку ❷, получим ошибку при компиляции:

```
invalid operation: cannot send to receive-only channel stream
```

Закрывать receive-only канал тоже нельзя. Если раскомментировать строчку ❸, получим ошибку при компиляции:

```
invalid operation: cannot close receive-only channel stream
```

Задать направление канала можно и при инициализации. Но толку от этого мало:

```
func main() {
    str := "one,two,,four"
    stream := make(chan<- string) // (!)
    go submit(str, stream)
    print(stream)
}
```

Здесь `stream` объявлен только для записи, так что для функции `print()` он больше не подходит. А если объявить только для чтения — не подойдет для `submit()`. Поэтому обычно каналы инициализируют для чтения и записи, а в параметрах конкретных функций заявляют как однонаправленные. Go конвертирует обычный канал в направленный автоматически:

```
stream := make(chan int)

go func(in chan<- int) {
    in <- 42
}(stream)

func(out <-chan int) {
```

```
    fmt.Println(<-out)
}(stream)
// 42
```

Старайтесь всегда указывать направление канала в параметрах функции, чтобы застраховаться от ошибок во время выполнения программы.

[песочница](#)

Каналы6

Чиним направления

Я написал программу, которая кодирует фразу шифром Цезаря:

```
src := "go is awesome"
res := encode(src)
fmt.Println(res)
// tb vf njrfbzr
```

К сожалению, она так и не заработала. Внутри функции `encode()` творится черти что. Найдите проблемы и исправьте их.

Помимо прочего, исправьте сигнатуры функций `submitter()`, `encoder()` и `receiver()` так, чтобы они корректно работали с направленными каналами. Не меняйте названия этих функций, они используются при проверке.

Логика шифровальщика `encodeWord()` не имеет значения. Считайте, что она работает корректно.

Полный код — в песочнице. Отправляйте в качестве решения только фрагмент, отмеченный комментариями «начало решения» и «конец решения».

[песочница](#)

Sample Input:

Sample Output:

PASS

```
package main

import (
    "fmt"
    "strings"
)
```

```

// encode кодирует строку шифром Цезаря
func encode(str string) string {
    // начало решения

    submitter := func(str string) <-chan string {
        ch := make(chan string)
        go func() {
            words := strings.Fields(str)
            for _, word := range words {
                ch <- word
            }
            close(ch)
        }()
        return ch
    }

    encoder := func(ch1 <-chan string) <-chan string {
        ch2 := make(chan string)
        go func() {
            for word := range ch1 {
                ch2 <- encodeWord(word)
            }
            close(ch2)
        }()
        return ch2
    }

    receiver := func(ch <-chan string) []string {
        words := []string{}
        for word := range ch {
            words = append(words, word)
        }
        return words
    }

    // конец решения

    pending := submitter(str)
    encoded := encoder(pending)
    words := receiver(encoded)
    return strings.Join(words, " ")
}

```

```
// encodeWord кодирует слово шифром Цезаря
func encodeWord(word string) string {
    const shift = 13
    const char_a byte = 'a'
    encoded := make([]byte, len(word))
    for idx, char := range []byte(word) {
        delta := (char - char_a + shift) % 26
        encoded[idx] = char_a + delta
    }
    return string(encoded)
}

func main() {
    src := "go is awesome"
    res := encode(src)
    fmt.Println(res)
}
```

Многозадачность 3. Каналы 2

Многозадачность 3. Каналы 2

Продолжим разбираться с каналами.

Канал завершения

Есть функция, которая произносит фразу по словам (с некоторыми задержками):

```
func say(id int, phrase string) {
    for _, word := range strings.Fields(phrase) {
        fmt.Printf("Worker #%d says: %s...\n", id, word)
        dur := time.Duration(rand.Intn(100)) * time.Millisecond
        time.Sleep(dur)
    }
}
```

Запускаем несколько одновременных болтушек, по одной на каждую фразу:

```
func main() {
    phrases := []string{
        "go is awesome",
        "cats are cute",
        "rain is wet",
    }
```

```

    "channels are hard",
    "floor is lava",
}
for idx, phrase := range phrases {
    go say(idx+1, phrase)
}
}

```

Программа, конечно, ничего не печатает — функция `main()` завершается до того, как отработает хотя бы одна болтушка:

```

$ go run say.go
<пусто>

```

Раньше мы использовали `[sync.WaitGroup](https://pkg.go.dev/sync#WaitGroup)`, чтобы дожидаться завершения горутин. А можно использовать прием «канал завершения» (done channel):

```

func say(done chan<- struct{}, id int, phrase string) {
    for _, word := range strings.Fields(phrase) {
        fmt.Printf("Worker #%d says: %s...\n", id, word)
        dur := time.Duration(rand.Intn(100)) * time.Millisecond
        time.Sleep(dur)
    }
    done <- struct{}{} // (1)
}

func main() {
    phrases := []string{
        "go is awesome",
        "cats are cute",
        "rain is wet",
        "channels are hard",
        "floor is lava",
    }

    done := make(chan struct{}) // (2)

    for idx, phrase := range phrases {
        go say(done, idx+1, phrase) // (3)
    }

    // wait for goroutines to finish
    for i := 0; i < len(phrases); i++ { // (4)
        <-done
    }
}

```

```
}  
}
```

Вот что здесь происходит:

- создаем отдельный канал ❷ и передаем его в каждую горутину ❸;
- в горутине записываем значение в канал по окончании работы ❶;
- в основной функции ждем, пока каждая горутина запишет в канал ❹.

Чтобы это работало, основная функция должна точно знать, сколько горутин запущено (в нашем случае — по одной на каждую исходную строку). Иначе непонятно, сколько значений читать из `done`.

Теперь все в порядке:

```
$ go run say.go  
Worker #5 says: floor...  
Worker #1 says: go...  
Worker #4 says: channels...  
Worker #3 says: rain...  
Worker #2 says: cats...  
Worker #4 says: are...  
Worker #3 says: is...  
Worker #4 says: hard...  
Worker #2 says: are...  
Worker #5 says: is...  
Worker #5 says: lava...  
Worker #3 says: wet...  
Worker #1 says: is...  
Worker #2 says: cute...  
Worker #1 says: awesome...
```

Если прием с каналом завершения вам не по душе, вместо него всегда можно использовать `sync.WaitGroup`.

[песочница](#)

Каналы2

Четыре счетовода

Работаем с функцией, которая считает цифры по словам:

```
func countDigitsInWords(next nextFunc) counter {  
    pending := make(chan string)
```

```

    go submitWords(next, pending)

    counted := make(chan pair)
    go countWords(pending, counted)

    return fillStats(counted)
}

```

Модифицируйте функцию, чтобы она запускала четыре горютины `countWords()` вместо одной. Используйте канал завершения, чтобы дождаться окончания обработки и закрыть канал `counted`. Реализуйте логику подсчета цифр в `countWords()` с использованием каналов `done`, `in` и `out`.

Полный код — в песочнице. Отправляйте в качестве решения только фрагмент, отмеченный комментариями «начало решения» и «конец решения».

[песочница](#)

Дедлоки и как с ними бороться

Самая распространенная проблема в многозадачных программах — *дедлок* (deadlock). Дедлок возникает, когда одна горютина ждет вторую, а та первую. Go распознает такие ситуации и завершает программу с ошибкой.

```
fatal error: all goroutines are asleep - deadlock!
```

Чтобы победить дедлок, надо ~~стать дедлоком~~ понять, из-за чего он возник. Рассмотрим пример:

```

func work(done chan struct{}, out chan int) {
    out <- 42
    done <- struct{}{}}
}

func main() {
    out := make(chan int)
    done := make(chan struct{})

    go work(done, out)    // (1)

    <-done                // (2)

    fmt.Println(<-out)    // (3)
}

```

В основной функции запускаем горютину `work` ❶. Она пишет результат в канал `out` и отчитывается о завершении в канал `done`. Основная функция тем временем ожидает канала

завершения ❷, после чего начитывает из канала результата ❸.

Видите проблему? `work()` ❶ хочет записать в `out`, то есть ожидает читателя ❸. А ❷ хочет прочитать из `done`, то есть ожидает ❶. Получается, что `work()` ждет `main()`, а `main()` ждет `work()`. Дедлок.

В данном случае можно вовсе убрать канал `done`, и программа заработает. Но если он вам зачем-то нужен, то решение — выполнять ❷ и ❸ независимо:

```
out := make(chan int)
done := make(chan struct{})

go work(done, out)           // (1)

go func() {                  // (2)
    <-done
    fmt.Println("work done")
    done <- struct{}{}
}()

fmt.Println(<-out)           // (3)
<-done
fmt.Println("all goroutines done")
```

Когда сталкиваетесь с дедлоком — поймите, из-за чего он возник. Тогда решение найдется.

Sample Input:

Sample Output:

PASS

```
package main

import (
    "fmt"
    "strings"
    "unicode"
)

// nextFunc возвращает следующее слово из генератора
type nextFunc func() string

// counter хранит количество цифр в каждом слове.
// ключ карты - слово, а значение - количество цифр в слове.
```



```

type counter map[string]int

// pair хранит слово и количество цифр в нем
type pair struct {
    word  string
    count int
}

// countDigitsInWords считает количество цифр в словах,
// выбирая очередные слова с помощью next()
func countDigitsInWords(next nextFunc) counter {
    pending := make(chan string)
    go submitWords(next, pending)

    done := make(chan struct{}) // пустая структура в качестве "флага"
    (struct{} тип, struct{}{} объект)
    counted := make(chan pair)

    // начало решения
    // запустите четыре горютины countWords()
    // вместо одной
    for i := 1; i <= 4; i++ {
        go countWords(done, pending, counted)
    }

    // используйте канал завершения, чтобы дождаться
    // окончания обработки и закрыть канал counted
    go func() {
        for i := 1; i <= 4; i++ {
            <-done
        }
        close(counted)
    }()
    // конец решения

    return fillStats(counted)
}

// submitWords отправляет слова на подсчет
func submitWords(next nextFunc, out chan<- string) {
    for {
        word := next()
        if word == "" {

```

```

        break
    }
    out <- word
}
close(out)
}

// countWords считает цифры в словах
func countWords(done chan<- struct{}, in <-chan string, out chan<- pair) {
    // например так
    for word := range in {
        p := pair{word, countDigits(word)}
        out <- p
    }
    done <- struct{}{}
}

// fillStats готовит итоговую статистику
func fillStats(in <-chan pair) counter {
    stats := counter{}
    for p := range in {
        stats[p.word] = p.count
    }
    return stats
}

// countDigits возвращает количество цифр в строке
func countDigits(str string) int {
    count := 0
    for _, char := range str {
        if unicode.IsDigit(char) {
            count++
        }
    }
    return count
}

// printStats печатает слова и количество цифр в каждом
func printStats(stats counter) {
    for word, count := range stats {
        fmt.Printf("%s: %d\n", word, count)
    }
}

```

```
// wordGenerator возвращает генератор, который выдает слова из фразы
func wordGenerator(phrase string) nextFunc {
    words := strings.Fields(phrase)
    idx := 0
    return func() string {
        if idx == len(words) {
            return ""
        }
        word := words[idx]
        idx++
        return word
    }
}

func main() {
    phrase := "1 22 333 4444 55555 666666 7777777 88888888"
    next := wordGenerator(phrase)
    stats := countDigitsInWords(next)
    printStats(stats)
}
```

Каналы3

Буферизованные каналы

Есть горутина `send()`, которая передает значение горутине `receive()` через канал `stream`:

```
var wg sync.WaitGroup
wg.Add(2)

stream := make(chan bool)

send := func() {
    defer wg.Done()
    fmt.Println("Sender ready to send...")
    stream <- true // (1)
    fmt.Println("Sent!")
}

receive := func() {
    defer wg.Done()
    fmt.Println("Receiver not ready yet...")
    time.Sleep(500 * time.Millisecond)
```

```

    fmt.Println("Receiver ready to receive...")
    <-stream // (2)
    fmt.Println("Received!")
}

go send()
go receive()
wg.Wait()

```

`send()` сразу после запуска хочет передать значение в канал, но `receive()` пока не готова. Поэтому `send()` вынуждена заблокироваться в точке ❶ и ждать 500 миллисекунд, пока `receive()` не придет в точку ❷ и не согласится принять значение из канала. Получается, что горютины *синхронизируются* в точке приема/передачи:

```

Receiver not ready yet...
Sender ready to send...
Receiver ready to receive...
Received!
Sent!

```

Чаще всего такое поведение нас устраивает. Но что делать, если мы хотим, чтобы отправитель не ждал получателя? Хотим, чтобы он отправил значение в канал и занялся своими делами. А получатель пусть заберет, когда будет готов. Ах, если бы только в канал можно было сложить значение, как в очередь! И как хорошо, что Go предоставляет ровно такую возможность:

```

// второй аргумент - размер буфера канала
// то есть количество значений, которые он может хранить
stream := make(chan int, 3)
// □ □ □

stream <- 1
// 1 □ □

stream <- 2
// 1 2 □

stream <- 3
// 1 2 3

fmt.Println(<-stream)
// 1
// 2 3 □

fmt.Println(<-stream)

```

```
// 2
// 3  □ □

stream <- 4
stream <- 5
// 3  4  5

stream <- 6
// в канале больше нет места,
// горутина блокируется
```

Такие каналы называются *буферизованными* (buffered), потому что у них есть собственный буфер фиксированного размера, в котором можно хранить значения. По умолчанию, если не указать размер буфера, будет создан канал с буфером размера 0 — именно с такими каналами мы работали до сих пор:

```
// канал без буфера
unbuffered := make(chan int)

// канал с буфером
buffered := make(chan int, 3)
```

На буферизованных каналах работают встроенные функции `len()` и `cap()`:

- `cap()` возвращает общую емкость канала;
- `len()` — количество значений в канале.

```
stream := make(chan int, 2)
fmt.Println(cap(stream), len(stream))
// 2 0

stream <- 7
fmt.Println(cap(stream), len(stream))
// 2 1

stream <- 7
fmt.Println(cap(stream), len(stream))
// 2 2

<-stream
fmt.Println(cap(stream), len(stream))
// 2 1
```

Чтобы отвязать `send()` от `receive()` с помощью буферизованного канала, достаточно изменить единственную строчку, оставив остальной код без изменений:

```
// создаем канал с буфером 1
// вместо обычного
stream := make(chan bool, 1)

send := func() {
    // ...
}

receive := func() {
    // ...
}

go send()
go receive()
```

Теперь отправитель не ждет получателя:

```
Receiver not ready yet...
Sender ready to send...
Sent!
Receiver ready to receive...
Received!
```

Буферизованные каналы нужны не всегда. Не злоупотребляйте ими и применяйте только тогда, когда обычные каналы по каким-то причинам не подходят. Разберем пример на одном из следующих шагов.

[песочница](#)

Каналы4

async / await

async/await — это распространенная в разных языках программирования концепция, при которой функции бывают двух видов: синхронные (выполняются последовательно) и асинхронные (могут выполняться одновременно). Асинхронные функции обязательно маркируют словом `async`. Чтобы дождаться результатов выполнения асинхронной функции, используют слово `await`. Если бы Go поддерживал эту концепцию, выглядело бы это как-то так:

```
async func answer() int {
    time.Sleep(100 * time.Millisecond)
    return 42
}

n := await answer()
```

К счастью, никаких `async/await` в Go нет. Надеюсь, вы по ним не соскучились. Но если что, реализовать можно в пять строчек кода:

```
// await executes fn concurrently
// and waits for results
func await(fn func() any) any {
    done := make(chan any, 1) // (1)
    go func() {
        done <- fn()           // (2)
    }()
    return <-done
}

func main() {
    slowpoke := func() any {
        fmt.Print("I'm so... ")
        time.Sleep(500 * time.Millisecond)
        fmt.Println("slow")
        return "okay"
    }

    result := await(slowpoke)
    fmt.Println(result.(string))
}
```

Если забыли: тип `any` — это синоним пустого интерфейса `interface{}`. То есть `await()` принимает функцию, которая возвращает одно значение, а какого оно типа — неважно. Ответственность за приведение типа лежит на клиенте.

Как видите, ничего особенного `await()` не делает:

- создает канал завершения;
- запускает горутину, в которой выполняет переданную функцию;
- дожидается завершения;
- возвращает результат клиенту.

Благодаря буферизованному каналу ❶, горутина в точке ❷ не блокируется, а сразу завершает работу. Таким образом, она перестает зависеть от вызывающей стороны. Конкретно в этой задаче можно обойтись и обычным каналом, ведь `await()` сразу начитывает результат. Но если такой гарантии нет — канал с буфером будет полезен.

Каналы5

Promise.all()

В JavaScript есть функция `Promise.all()`, которая асинхронно запускает переданные ей функции и возвращает результат, когда все отработают:

```
async function squared(n) {
    return n * n;
}

funcs = [squared(2), squared(3), squared(4)];
nums = await Promise.all(funcs);
console.log(nums);

// [4, 9, 16]
```

Есть аналогичная функция и в Python — `asyncio.gather()`:

```
async def squared(n):
    return n*n

funcs = [squared(2), squared(3), squared(4)]
nums = await asyncio.gather(*funcs)
print(nums)

# [4, 9, 16]
```

Реализуйте аналогичную функцию `gather()` на Go:

```
func gather(funcs []func() any) []any {
    // выполните все переданные функции,
    // соберите результаты в срез
    // и верните его
}

func squared(n int) func() any {
    return func() any {
        time.Sleep(time.Duration(n) * 100 * time.Millisecond)
        return n * n
    }
}

func main() {
    funcs := []func() any{squared(2), squared(3), squared(4)}

    start := time.Now()
    nums := gather(funcs)
```



```

    elapsed := float64(time.Since(start)) / 1_000_000

    fmt.Println(nums)
    fmt.Printf("Took %.0f ms\n", elapsed)
}

```

```

[4 9 16]
Took 401 ms

```

Функции, переданные в `gather()`, должны выполняться не последовательно, а одновременно.

Полный код — в песочнице. Отправляйте в качестве решения только фрагмент, отмеченный комментариями «начало решения» и «конец решения».

Подсказка

Функция должна возвращать результаты в том же порядке, в каком переданы исходные функции. Если на входе было `[squared(4), squared(3), squared(2)]` — на выходе должно быть `[16, 9, 4]`, а не `[9, 16, 4]` или `[4, 9, 16]`.

[песочница](#)

Sample Input:

Sample Output:

PASS

```

package main

import (
    "fmt"
    "time"
)

// gather выполняет переданные функции одновременно
// и возвращает срез с результатами, когда они готовы
func gather(funcs []func() any) []any {
    // начало решения
    done := make(chan any, len(funcs))
    idxs := make(chan int, len(funcs)) // заведем канал для сохранения
    порядка

    // выполните все переданные функции,
    for i := 0; i < len(funcs); i++ {

```

```

        go func(i int) {
            done <- funcs[i]()
            idxs <- i
        }(i)
    }
    // соберите результаты в срез
    res := make([]any, len(funcs))
    for range funcs {
        res[<-idxs] = <-done
    }
    // и верните его
    return res
    // конец решения
}

// squared возвращает функцию,
// которая считает квадрат n
func squared(n int) func() any {
    return func() any {
        time.Sleep(time.Duration(n) * 100 * time.Millisecond)
        return n * n
    }
}

func main() {
    funcs := []func() any{
        squared(2), squared(3), squared(4)
    }
    start := time.Now()
    nums := gather(funcs)
    elapsed := float64(time.Since(start)) / 1_000_000

    fmt.Println(nums)
    fmt.Printf("Took %.0f ms\n", elapsed)
}

```

Каналы6

N обработчиков

Помните, как мы запускали болтушки в горутинах — по одной на каждую фразу?

```

func main() {
    phrases := []string{
        // ...
    }
}

```

```

    for idx, phrase := range phrases {
        go say(idx+1, phrase)
    }
}

```

Горутины — легковесные объекты. Вполне можно запустить одновременно 10, 100 или 1000 штук. Но что, если исходных фраз будет 100 тысяч или миллион? Понятно, что реальная многозадачность все равно ограничена количеством ядер. Так что нет смысла впустую расходовать память на сотни тысяч горутин, если параллельно выполняться все равно будут только восемь (или сколько у вас там CPU).

Скажем, мы хотим, чтобы одновременно существовали только N say-горутин. Добиться этого поможет буферизованный канал. Идея следующая:

- создаем канал с буфером размера N и заполняем его «токенами» (любыми значениями);
- перед запуском горутин забирает токен из канала;
- по завершении работы горутин возвращает токен в канал.

Таким образом, если в канале не осталось токенов, то очередная горутин не запустится и будет ждать, пока кто-нибудь вернет токен в канал. В результате одновременно запущены будут не более N горутин.

Вот как это может выглядеть при N = 2:

```

func main() {
    phrases := []string{
        // ...
    }

    // пул идентификаторов для 2 горутин
    pool := make(chan int, 2)
    pool <- 1
    pool <- 2

    for _, phrase := range phrases {
        // получаем идентификатор из пула,
        // если есть свободные
        id := <-pool
        go say(pool, id, phrase)
    }

    // ждем, пока все горутин закончат работу
    // (то есть все идентификаторы вернутся в пул)
    <-pool
}

```

```

    <-pool
}

func say(pool chan<- int, id int, phrase string) {
    for _, word := range strings.Fields(phrase) {
        fmt.Printf("Worker #%d says: %s...\n", id, word)
        dur := time.Duration(rand.Intn(100)) * time.Millisecond
        time.Sleep(dur)
    }
    // возвращаем идентификатор в пул
    pool <- id
}

```

`main()` проходит по исходным фразам, для каждой фразы получает токен из пула и запускает `say`-горутину. `Say`-горутинка печатает фразу и возвращает токен в пул. Таким образом, фразы обрабатываются одновременно, причем каждая фраза печатается только один раз:

```

Worker #2 says: go...
Worker #1 says: cats...
Worker #2 says: is...
Worker #1 says: are...
Worker #2 says: awesome...
Worker #1 says: cute...
Worker #1 says: rain...
Worker #1 says: is...
...

```

В качестве токенов используются идентификаторы (порядковые номера 1, 2), но это исключительно для наглядности вывода в `say()`. Токенами могут быть пустые структуры или любые другие значения.

[песочница](#)

Альтернативный вариант

Решить задачу можно было и без пула, как мы это делали на шаге [Четыре счетовода](#). Бросаем исходные данные в канал и запускаем `N` горутин, которые его разгребают:

```

func main() {
    phrases := []string{
        // ...
    }

    pending := make(chan string)
}

```

```

go func() {
    for _, phrase := range phrases {
        pending <- phrase
    }
    close(pending)
}()

done := make(chan struct{})

go say(done, pending, 1)
go say(done, pending, 2)

<-done
<-done
}

func say(done chan<- struct{}, pending <-chan string, id int) {
    for phrase := range pending {
        for _, word := range strings.Fields(phrase) {
            fmt.Printf("Worker #%d says: %s...\n", id, word)
            dur := time.Duration(rand.Intn(100)) * time.Millisecond
            time.Sleep(dur)
        }
    }
    done <- struct{}{}
}

```

В таком варианте мы зависим от поставщика данных в канал `pending`, который в нужный момент закрывает канал. Если исходные данные заранее неизвестны и приходят с непонятной частотой — вариант с пулом токенов может быть удобнее.

Разница еще в том, что в первом подходе (с пулом токенов) мы запускаем множество короткоживущих горутин, а во втором (с разгребанием канала) — две долгоживущие.

[песочница](#)

Каналы7

Пул обработчиков

Продолжаем работать с функциями-болтушками. На предыдущем шаге мы заставили их обрабатывать исходные данные в две горуты, с помощью пула:

```

func main() {
    phrases := []string{

```

```

    // ...
}

// пул идентификаторов для 2 горутин
pool := make(chan int, 2)
pool <- 1
pool <- 2

for _, phrase := range phrases {
    // получаем идентификатор из пула,
    // если есть свободные
    id := <-pool
    go say(pool, id, phrase)
}

// ожидаем, пока все горутин закончат работу
// (то есть все идентификаторы вернутся в пул)
<-pool
<-pool
}

func say(pool chan<- int, id int, phrase string) {
    for _, word := range strings.Fields(phrase) {
        // ...
    }
    // возвращаем идентификатор в пул
    pool <- id
}

```

Это работает, но как-то... кривовато. Во-первых, получился пул строго на две горутин (а вдруг понадобится десять?). Во-вторых, логика пула размазана по всей программе — это чревато ошибками.

Хочется, чтобы функция `say()` ничего не знала о пуле:

```

func say(id int, phrase string) {
    for _, word := range strings.Fields(phrase) {
        // ...
    }
}

```

А функция `main()` чтобы выглядела так:

```

func main() {
    phrases := []string{

```

```

    // ...
}

handle, wait := makePool(2, say)
for _, phrase := range phrases {
    handle(phrase)
}
wait()
}

```

Конструктор `makePool()` создает пул на указанное количество обработчиков. Он также принимает функцию-обработчик, которую будет вызывать позже. Конструктор возвращает две функции:

- `handle()` выбирает токен из пула и обрабатывает переданную фразу через функцию-обработчик;
- `wait()` дожидается, пока все токены вернутся в пул.

Вот шаблон `makePool()`:

```

func makePool(n int, handler func(int, string)) (func(string), func()) {
    // создайте пул на n обработчиков
    // используйте для канала имя pool и тип chan int
    // определите функции handle() и wait()

    // handle() выбирает токен из пула
    // и обрабатывает переданную фразу через handler()

    // wait() дожидается, пока все токены вернутся в пул

    return handle, wait
}

```

Реализуйте внутренности.

Чтобы не усложнять задачу, гарантируется следующее:

- `handle()` и `wait()` всегда вызываются в одной и той же горутине;
- `wait()` вызывается только после того, как вызваны все `handle()`;
- `handle()` никогда не вызывается после `wait()`;
- `wait()` вызывается только один раз.

Полный код — в песочнице. Отправляйте в качестве решения только фрагмент, отмеченный комментариями «начало решения» и «конец решения».

[песочница](#)

Sample Input:

Sample Output:

PASS

```
package main

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

// say печатает фразу от имени обработчика
func say(id int, phrase string) {
    for _, word := range strings.Fields(phrase) {
        fmt.Printf("Worker #%d says: %s...\n", id, word)
        dur := time.Duration(rand.Intn(100)) * time.Millisecond
        time.Sleep(dur)
    }
}

// makePool создает пул на n обработчиков
// возвращает функции handle и wait
func makePool(n int, handler func(int, string)) (func(string), func()) {
    // начало решения

    // создайте пул на n обработчиков
    // используйте для канала имя pool и тип chan int
    pool := make(chan int, n)
    for i := 1; i <= n; i++ {
        pool <- i
    }

    // определите функции handle() и wait()
    handle := func(phrase string) {
        // handle() выбирает токен из пула
        id := <-pool
        // и обрабатывает переданную фразу через handler()
        go func() {
```



```

        handler(id, phrase) // отработал обработчик
        pool <- id          // вернули id в пул
    }()
}

// wait() дожидается, пока все токены вернутся в пул
wait := func() {
    for i := 1; i <= n; i++ {
        <-pool
    }
}

// конец решения

return handle, wait
}

func main() {
    phrases := []string{
        "go is awesome",
        "cats are cute",
        "rain is wet",
        "channels are hard",
        "floor is lava",
    }

    handle, wait := makePool(2, say)
    for _, phrase := range phrases {
        handle(phrase)
    }
    wait()
}

```

Каналы8

Еще пара нюансов, и заканчиваем ツ

Закрывать буферизованный канал

Как мы знаем, обычный канал после закрытия отдает нулевое значение и признак `false`:

```

stream := make(chan int)
close(stream)

val, ok := <-stream

```

```

fmt.Println(val, ok)
// 0 false

val, ok = <-stream
fmt.Println(val, ok)
// 0 false

val, ok = <-stream
fmt.Println(val, ok)
// 0 false

```

Канал с буфером ведет себя точно так же, если буфер пуст. А вот если в буфере есть значения — иначе:

```

stream := make(chan int, 2)
stream <- 1
stream <- 2
close(stream)

val, ok := <-stream
fmt.Println(val, ok)
// 1 true

val, ok = <-stream
fmt.Println(val, ok)
// 2 true

val, ok = <-stream
fmt.Println(val, ok)
// 0 false

```

Пока в буфере есть значения, канал отдает их и признак `true`. Когда все значения выбраны — отдает нулевое значение и признак `false`, как обычный канал.

Благодаря этому отправитель может в любой момент закрыть канал, не задумываясь о том, остались в нем значения или нет. Получатель в любом случае их считает:

```

stream := make(chan int, 3)

go func() {
    fmt.Println("Sending...")
    stream <- 1
    stream <- 2
    stream <- 3
}

```

```

    close(stream)
    fmt.Println("Sent and closed!")
}()

time.Sleep(500 * time.Millisecond)
fmt.Println("Receiving...")
for val := range stream {
    fmt.Printf("%v ", val)
}
fmt.Println()
fmt.Println("Received!")

```

```

Sending...
Sent and closed!
Receiving...
1 2 3
Received!

```

[песочница](#)

nil-канал

Как у любого типа в Go, у каналов тоже есть нулевое значение. Это `nil`:

```

var stream chan int
fmt.Println(stream)
// <nil>

```

nil-канал — малоприятная штука:

- Запись в nil-канал навсегда блокирует горутину.
- Чтение из nil-канала навсегда блокирует горутину.
- Заккрытие nil-канала приводит к панике.

```

var stream chan int

go func() {
    stream <- 1
}()

<-stream

// fatal error: all goroutines are asleep - deadlock!

```

```
var stream chan int
close(stream)

// panic: close of nil channel
```

У nil-каналов есть некоторые очень специфические сценарии использования. Один из них мы рассмотрим на следующем уроке. В целом — старайтесь избегать nil-каналов до тех пор, пока не почувствуете, что никак не можете без них обойтись.

[песочница](#)

Многозадачность 4. Композиция

Многозадачность 4. Композиция

Зависшая горутина

Есть функция, которая отправляет в канал числа в указанном диапазоне:

```
func rangeGen(start, stop int) <-chan int {
    out := make(chan int)
    go func() {
        for i := start; i < stop; i++ {
            out <- i
        }
        close(out)
    }()
    return out
}
```

Работает вроде бы корректно:

```
func main() {
    generated := rangeGen(41, 46)
    for val := range generated {
        fmt.Println(val)
    }
}
```

```
$ go run rangeGen.go
41
42
43
44
45
```

Проверим, что будет, если завершить цикл досрочно:

```
func main() {
    generated := rangeGen(41, 46)
    for val := range generated {
        fmt.Println(val)
        if val == 42 {
            break
        }
    }
}
```

41

42

На первый взгляд, по-прежнему корректно. Но не совсем — горутина `rangeGen()` зависла:

```
func rangeGen(start, stop int) <-chan int {
    out := make(chan int)
    go func() {
        for i := start; i < stop; i++ {    // (1)
            out <- i                      // (2)
        }
        close(out)
    }()
    return out
}
```

Поскольку `main()` вышла из цикла на числе 42, то цикл ❶ внутри `rangeGen()` тоже не завершился. Он навсегда заблокировался на строчке ❷ при попытке отправить число 43 в канал `out`. Горутина зависла. Канал `out` тоже не закрылся, так что если бы от него зависели другие горутыны — зависли бы и они.

В данном случае большой беды в этом нет: когда `main()` завершится, среда выполнения завершит и все прочие горутыны. Но если бы `main()` продолжала работать, и снова и снова вызывала `rangeGen()` — зависшие горутыны бы накапливались. Это плохо: горутыны хоть и легковесные, но не совсем «бесплатные». Рано или поздно память может закончиться (сборщик мусора горутыны не собирает).

Получается, нужна возможность досрочно завершить горутину.

[песочница](#)

Перевернуть уникальные слова

Реализуйте конвейер, который делает следующее:

- генерит слова из 5 букв;
- отсеивает слова, в которых буквы повторяются;
- переворачивает оставшиеся слова;
- печатает пары «исходное-перевернутое» для первых n из них.

Пример. Если исходно сгенерились слова `water`, `happy` и `lemon`, то в итоге должно напечататься:

```
water -> retaw
happy -> yppah
lemon -> nomeɹ
```

Все этапы должны поддерживать отмену.

Пример вызова конвейера:

```
func main() {
    cancel := make(chan struct{})
    defer close(cancel)

    c1 := generate(cancel)
    c2 := takeUnique(cancel, c1)
    c3_1 := reverse(cancel, c2)
    c3_2 := reverse(cancel, c2)
    c4 := merge(cancel, c3_1, c3_2)
    print(cancel, c4, 10)
}
```

Шаблон решения — в песочнице. Отправляйте в качестве решения только фрагмент, отмеченный комментариями «начало решения» и «конец решения».

[песочница](#)

Зависшие горютины и как с ними бороться

Зависшие горютины — вторая по популярности проблема многозадачных программ после дедлоков. Go на них не ругается, так что они часто остаются незамеченными.

На курсе зависшие горютины приводят к ошибке при проверке задания:

```
ERROR: there are leaked goroutines
```

Основные причины зависания:

1. Вы забыли сделать канал отмены и подключить его через select.
2. Канал отмены есть, но горютина зависает внутри селекта (что? да!)

С первой причиной мы разобрались еще в начале урока, так что посмотрим на вторую. Для многих она оказывается неожиданной.

Допустим, есть функция, которая отправляет числа в канал:

```
func generate(cancel <-chan struct{}) chan int {
    out := make(chan int)
    go func() {
        defer close(out)
        for i := 0; ; i++ {
            select {
            case out <- i:
            case <-cancel:
                return
            }
        }
    }()
    return out
}
```

И функция, которая модифицирует числа:

```
func modify(cancel <-chan struct{}, in <-chan int) <-chan int {
    out := make(chan int)
    go func() {
        defer fmt.Println("modify done") // (1)
        defer close(out)
        for {
            select {
            case num := <-in:
                out <- num * 2
            case <-cancel:
                return
            }
        }
    }()
    return out
}
```

Благодаря ❶ горютина напишет, когда завершится.

В связке эти две функции будут работать бесконечно, так что добавим третью, которая начитывает первые 10 результатов и заканчивает:

```
func print(in <-chan int) {
    for i := 0; i < 10; i++ {
        <-in
        fmt.Printf(".")
    }
    fmt.Println()
}

func main() {
    cancel := make(chan struct{})
    c1 := generate(cancel)
    c2 := modify(cancel, c1)
    print(c2)

    close(cancel)
    // времени с запасом, чтобы горутины
    // закончили работать после закрытия
    // канала отмены
    time.Sleep(50 * time.Millisecond)
}
```

Проверим:

```
.....
modify done
```

Работает! Ну еще бы, мы же везде использовали select + cancel, значит ошибки быть не могло.

А вот и нет. Чтобы в этом убедиться, достаточно добавить 10 мс задержку:

```
func modify(cancel <-chan struct{}, in <-chan int) <-chan int {
    out := make(chan int)
    go func() {
        defer fmt.Println("modify done")
        defer close(out)
        for {
            select {
            case num := <-in:
                time.Sleep(10 * time.Millisecond) // (1)
                out <- num * 2                      // (2)
            case <-cancel:
            }
        }
    }
    return out
}
```



```

        return
    }
}
}()
return out
}

```

Запускаем:

```
.....
```

Горутина `modify()` зависла в точке ❷. Когда закрылся канал `cancel`, она ожидала записи в `out` внутри конкретной ветки селекта, так что селект уже ничего не мог помочь.

Есть два способа с этим бороться. Первый — держать ветки селекта пустыми:

```

func modify(cancel <-chan struct{}, in <-chan int) <-chan int {
    out := make(chan int)

    multiply := func(num int) int {
        time.Sleep(10 * time.Millisecond)
        return num * 2
    }

    go func() {
        defer fmt.Println("modify done")
        defer close(out)
        for num := range in {
            select {
            case out <- multiply(num):
            case <-cancel:
                return
            }
        }
    }()
    return out
}

```

```

.....
modify done

```

Как бы ни тормозила функция `multiply()`, мы не проваливаемся внутрь ветки селекта, так что закрытие `cancel` гарантированно сработает.

Второй способ — использовать вложенные селекты везде, где идет запись или чтение из канала:

```

func modify(cancel <-chan struct{}, in <-chan int) <-chan int {
    out := make(chan int)
    go func() {
        defer close(out)
        for {
            select {
            case num, ok := <-in:
                if !ok {
                    return
                }
                time.Sleep(10 * time.Millisecond)
                select {
                case out <- num * 2:
                case <-cancel:
                    return
                }
            case <-cancel:
                return
            }
        }
    }()
    return out
}

```

```

.....
modify done

```

Тут вложенный селект страхует нас при записи в `out`, так что зависнуть тоже нечему.

Второй пример демонстрирует еще один важный принцип: если используете обычный цикл `for` вместо `for range`, обязательно проверяйте, не закрыт ли входной канал. А лучше — всегда по возможности используйте `for range`, чтобы он проверял за вас.

Многозадачные программы — это сложно. Go предоставляет удобные инструменты вроде каналов и селекта, но это не панацея. Всегда тестируйте свой код: как отдельные горутины, так и их композицию.

Sample Input:

Sample Output:

PASS

```

package main

```

```

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

// начало решения

// генерит случайные слова из 5 букв
// с помощью randomWord(5)
func generate(cancel <-chan struct{}) <-chan string {
    out := make(chan string)
    go func() {
        // defer fmt.Println("generate done...")
        defer close(out)
        for {
            word := randomWord(5)
            select {
            case out <- word:
                // fmt.Printf("'s' generated and sent to chan...\n", word)
            case <-cancel:
                return
            }
        }
    }()
    return out
}

// выбирает слова, в которых не повторяются буквы,
// abcde - подходит
// abcda - не подходит
func takeUnique(cancel <-chan struct{}, in <-chan string) <-chan string {
    out := make(chan string)
    go func() {
        // defer fmt.Println("unique done...")
        defer close(out)
        for {
            select {
            case word, ok := <-in:
                word_arr := strings.Split(word, "")
                word_dict := make(map[string]bool)
                for _, ch := range word_arr {
                    word_dict[ch] = true
                }
            }
        }
    }()
    return out
}

```

```

    }
    if !ok {
        return
    }

    if len(word_arr) != len(word_dict) {
        // fmt.Printf("'s' is NOT unique...\n", word)
        continue // не return!
    }

    select {
    case out <- word:
    case <-cancel:
        return
    }
}

}

}()
return out
}

// переворачивает слова
// abcde -> edcba
func reverse(cancel <-chan struct{}, in <-chan string) <-chan string {
    out := make(chan string)

    rev := func(s string) string { //
https://github.com/golang/example/blob/master/stringutil/reverse.go
        r := []rune(s)
        for i, j := 0, len(r)-1; i < len(r)/2; i, j = i+1, j-1 {
            r[i], r[j] = r[j], r[i]
        }
        return string(r)
    }

    go func() {
        // defer fmt.Println("reverse done...")
        defer close(out)
        for {
            select {
            case word, ok := <-in:
                if !ok {
                    return
                }
            }
        }
    }()
}

```

```

    }
    rev_word := rev(word)
    // fmt.Printf("' %s' is reversed '%s'...\n", rev_word, word)
    select {
    case out <- fmt.Sprintf("%s -> %s", word, rev_word):
    case <-cancel:
        return
    }
    }
}

}()
return out
}

// объединяет c1 и c2 в общий канал
func merge(cancel <-chan struct{}, c1, c2 <-chan string) <-chan string {
    out := make(chan string)
    go func() {
        // defer fmt.Println("merge done...")
        defer close(out)
        for c1 != nil || c2 != nil {
            select {
            case val1, ok := <-c1:
                if ok {
                    out <- val1
                } else {
                    c1 = nil
                }

            case val2, ok := <-c2:
                if ok {
                    out <- val2
                } else {
                    c2 = nil
                }

            case <-cancel:
                return
            }
        }
    }()
    return out
}

```

```

// печатает первые n результатов
func print(cancel <-chan struct{}, in <-chan string, n int) {
    for i := 1; i <= n; i++ {
        // fmt.Printf("Got: %s\n", <-in)
        fmt.Println(<-in)
    }
}

// конец решения

// генерит случайное слово из n букв
func randomWord(n int) string {
    const letters = "aeiourtnsl"
    chars := make([]byte, n)
    for i := range chars {
        chars[i] = letters[rand.Intn(len(letters))]
    }
    return string(chars)
}

func main() {
    cancel := make(chan struct{})
    defer close(cancel)

    c1 := generate(cancel)
    c2 := takeUnique(cancel, c1)
    c3_1 := reverse(cancel, c2)
    c3_2 := reverse(cancel, c2)
    c4 := merge(cancel, c3_1, c3_2)
    print(cancel, c4, 10)
}

```

9

Конвейер

Конвейер (pipeline) — это последовательность операций, каждая из которых принимает на входе данные, обрабатывает их определенным образом и отдает на выход. Входом и выходом для каждой операции выступает канал.

Собственно, последние три урока мы только тем и занимаемся, что строим конвейеры. Но давайте еще разок закрепим для верности.

Типичный конвейер выглядит так:

- читатель (начитывает исходные данные из файла, базы или по сети);
- N обработчиков (преобразуют данные, фильтруют, агрегируют, дополняют информацией из внешних источников);
- писатель (записывает результат обработки в файл, базу или по сети).

Этапов конвейера с обработчиками может быть сколько угодно: сначала фильтруем, затем преобразуем, наконец агрегируем. На каждом этапе может запускаться несколько одновременных обработчиков, а вот читатель и писатель обычно существуют в единственном экземпляре.

Рассмотрим конвейер из 5 этапов:

- `rangeGen()` генерит числа в указанном диапазоне (читатель);
- `takeLucky()` выбирает «счастливые» числа (обработчик);
- `merge()` объединяет независимые каналы (обработчик);
- `sum()` суммирует числа (обработчик);
- `printTotal()` печатает результат (писатель).

```
// количество и сумма счастливых чисел
type Total struct {
    count int
    amount int
}

// генерит числа в указанном диапазоне
func rangeGen(start, stop int) <-chan int {
    out := make(chan int)
    go func() {
        defer close(out)
        for i := start; i < stop; i++ {
            out <- i
        }
    }()
    return out
}

// выбирает счастливые числа
func takeLucky(in <-chan int) <-chan int {
    out := make(chan int)
    go func() {
        defer close(out)
        for num := range in {
```

```

        if num%7 == 0 && num%13 != 0 {
            out <- num
        }
    }
}()
return out
}

// объединяет независимые каналы
func merge(channels []<-chan int) <-chan int {
    out := make(chan int)
    // ...
    return out
}

// суммирует числа
func sum(in <-chan int) <-chan Total {
    out := make(chan Total)
    go func() {
        defer close(out)
        total := Total{}
        for num := range in {
            total.amount += num
            total.count++
        }
        out <- total
    }()
    return out
}

// печатает результат
func printTotal(in <-chan Total) {
    total := <-in
    fmt.Printf("Total of %d lucky numbers = %d\n", total.count,
total.amount)
}

func main() {
    readerChan := rangeGen(1, 1000)
    luckyChans := make([]<-chan int, 4)
    for i := 0; i < 4; i++ {
        luckyChans[i] = takeLucky(readerChan)
    }
}

```



```
mergedChan := merge(luckyChans)
totalChan := sum(mergedChan)
printTotal(totalChan)
// Total of 132 lucky numbers = 66066
}
```

Даже на такой игрушечной задаче конвейер удобнее, чем одна большая функция:

- каждый этап решает ровно одну задачу — легче понять код;
- можно удалять и добавлять отдельные этапы, не трогая остальную логику;
- можно независимо менять степень параллелизма на каждом этапе;
- можно переиспользовать этапы в других конвейерах.

Задачи вида «выгрузить, обработать и загрузить» постоянно встречаются на практике. Конвейеры отлично для них подходят.

8

Объединяем N каналов

Реализуйте функцию `merge()`, которая объединяет произвольное количество каналов в один:

```
func merge(channels ...chan int) <-chan int {
    // ...
}

func main() {
    in1 := rangeGen(11, 15)
    in2 := rangeGen(21, 25)
    in3 := rangeGen(31, 35)

    start := time.Now()
    merged := merge(in1, in2, in3)
    for val := range merged {
        fmt.Print(val, " ")
    }
    fmt.Println()
    fmt.Println("Took", time.Since(start))
}
```

Используйте любую из техник параллельного объединения. Последовательное объединение не подойдет.

Полный код — в песочнице. Отправляйте в качестве решения только фрагмент, отмеченный комментариями «начало решения» и «конец решения».

[песочница](#)

Sample Input:

Sample Output:

PASS

```
package main

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

// rangeGen отправляет в канал числа от start до stop-1
func rangeGen(start, stop int) <-chan int {
    out := make(chan int)
    go func() {
        defer close(out)
        for i := start; i < stop; i++ {
            time.Sleep(50 * time.Millisecond)
            out <- i
        }
    }()
    return out
}

// начало решения

// merge выбирает числа из входных каналов и отправляет в выходной
// func merge(channels ...<-chan int) <-chan int {
//     // объедините все исходные каналы в один выходной
//     // последовательное объединение НЕ подходит
//     out := make(chan int)
//     var wg sync.WaitGroup
//     wg.Add(len(channels))
//     for _, c := range channels {
//         go func(c <-chan int) {
//             for v := range c {
//                 out <- v
//             }
//         }(c)
//     }
//     wg.Wait()
//     return out
// }
```

```

//                                wg.Done()
//                                }(c)
//                                }
//                                go func() {
//                                    wg.Wait()
//                                    close(out)
//                                }()
//                                return out
// }

// конец решения

// РЕКУРСИВНЫЙ ВАРИАНТ
// https://medium.com/justforfunc/analyzing-the-performance-of-go-functions-with-benchmarks-60b8162e61c6
// начало решения
// func mergeTwo(ch1, ch2 <-chan int) <-chan int {
//     out := make(chan int)
//     go func() {
//         defer close(out)
//         for ch1 != nil || ch2 != nil {
//             select {
//                 case val, ok := <-ch1:
//                     if !ok {
//                         ch1 = nil
//                     } else {
//                         out <- val
//                     }
//                 case val, ok := <-ch2:
//                     if !ok {
//                         ch2 = nil
//                     } else {
//                         out <- val
//                     }
//             }
//         }
//     }
//     }()
//     return out
// }

// func merge(channels ...<-chan int) <-chan int {
//     switch len(channels) {
//     case 0:

```

```
//      ch := make(chan int)
//      close(ch)
//      return ch
//  case 1:
//      return channels[0]
//  default:
//      half := len(channels) / 2
//      return mergeTwo(merge(channels[:half]...),
merge(channels[half:]...))
//  }
// }
```

// конец решения

// КАНАЛ ЗАВЕРШЕНИЯ

```
// func merge(channels ...<-chan int) <-chan int {
//     done := make(chan struct{})
//     out := make(chan int)

//     for _, channel := range channels {
//         go func(channel <-chan int) {
//             for value := range channel {
//                 out <- value
//             }
//             done <- struct{}{}
//         }(channel)
//     }

//     go func() {
//         for i := 0; i < len(channels); i++ {
//             <-done
//         }
//         close(out)
//     }()

//     return out
// }
```

// начало решения

// SELECT

```
func merge(channels ...<-chan int) <-chan int {
    out := make(chan int)
```

```

go func() {
    defer close(out)
    for channels != nil {
        for _, channelx := range channels {
            select {
                case val, ok := <-channelx:
                    if ok {
                        out <- val
                    } else {
                        channels = nil
                    }
            }
        }
    }
}()
return out
}

func main() {
    in1 := rangeGen(11, 15)
    in2 := rangeGen(21, 25)
    in3 := rangeGen(31, 35)

    start := time.Now()
    merged := merge(in1, in2, in3)
    for val := range merged {
        fmt.Print(val, " ")
    }
    fmt.Println()
    fmt.Println("Took", time.Since(start))
}

```

7

Объединение каналов (select)

Можно обойтись и одной горутиной, сохранив быстродействие. Поможет инструкция `select`. Вспомним ее полезное свойство:

Если незаблокированных веток несколько, выбирает одну из них случайным образом и выполняет ее.

Таким образом, если селектить одной веткой из `in1`, а второй из `in2` — объединение должно работать почти с такой же скоростью, как две независимые горутины:

```
func merge(in1, in2 <-chan int) <-chan int {
    out := make(chan int)
    go func() {
        defer close(out)
        for {
            select {
            case out <- <-in1:
            case out <- <-in2:
            }
        }
    }()
    return out
}
```

Однако, на практике это выглядит примерно так:

```
21 11 12 22 13 23 24 14 0 0 0 0 0 0 0 0 0 0 0... далее до бесконечности
```

Такая реализация продолжает выбирать значения из входных каналов даже после того, как они закрыты — до бесконечности.

Для корректной работы не хватает трех условий:

- выбирать значения из `in1`, только если он открыт;
- выбирать значения из `in2`, только если он открыт;
- если `in1` и `in2` оба закрыты — выйти из цикла.

`select` может справиться и с этим. Поможет свойство `nil`-каналов, о котором мы говорили на прошлом уроке:

Чтение из `nil`-канала навсегда блокирует горутины.

Если превратить `in1` в `nil` после того, как он закрылся — `select` перестанет из него читать (он ведь игнорирует заблокированные ветки). Аналогично для `in2`:

```
func merge(in1, in2 <-chan int) <-chan int {
    out := make(chan int)
    go func() {
        defer close(out)
        for in1 != nil || in2 != nil {
            select {
            case val1, ok := <-in1:
```

```

        if ok {
            out <- val1
        } else {
            in1 = nil
        }

        case val2, ok := <-in2:
            if ok {
                out <- val2
            } else {
                in2 = nil
            }
        }
    }
}()
return out
}

```

Теперь каждая ветка `select` отключается после того, как соответствующий канал закрыт. А когда закрыты оба канала — прекращается цикл `for`. Проверим:

```

21 11 22 12 23 13 24 14
Took 200ms

```

Работает!

P.S. Помните, я говорил, что `nil`-каналы бывают полезны в некоторых специфических случаях? Вот это один из них.

[песочница](#)

6

Объединение каналов (параллельное)

Чтобы независимо начитывать из входных каналов, запустим две горуты:

```

func merge(in1, in2 <-chan int) <-chan int {
    var wg sync.WaitGroup
    wg.Add(2)

    out := make(chan int)

    // первая горутина начитывает из in1 в out
    go func() {

```

```

    defer wg.Done()
    for val := range in1 {
        out <- val
    }
}()

// вторая горутина начитывает из in2 в out
go func() {
    defer wg.Done()
    for val := range in2 {
        out <- val
    }
}()

// ждем, пока исчерпаются оба входных канала,
// после чего закрываем выходной
go func() {
    wg.Wait()
    close(out)
}()

return out
}

```

Проверяем:

```

11 21 12 22 23 13 14 24
Took 200ms

```

Другое дело.

[песочница](#)

5

Объединение каналов (последовательное)

Бывает, что несколько независимых функций отправляют результаты каждая в свой канал. А работать при этом удобнее с общим каналом результатов. Тогда приходится объединять выходные каналы функций в итоговый канал.

Функция `rangeGen()` отправляет в канал числа в указанном диапазоне:

```

func rangeGen(start, stop int) <-chan int {
    out := make(chan int)

```



```

go func() {
    defer close(out)
    for i := start; i < stop; i++ {
        time.Sleep(50 * time.Millisecond)
        out <- i
    }
}()
return out
}

```

Для простоты изложения на этом и дальнейших шагах будем работать с неотменяемыми горутинами. Как превратить любую неотменяемую горутина в отменяемую — вы уже знаете.

Запускаем ее дважды, объединяем выходные каналы и печатаем результаты:

```

func main() {
    in1 := rangeGen(11, 15)
    in2 := rangeGen(21, 25)

    start := time.Now()
    merged := merge(in1, in2)
    for val := range merged {
        fmt.Print(val, " ")
    }
    fmt.Println()
    fmt.Println("Took", time.Since(start))
}

```

Осталось только реализовать функцию `merge()`.

Вот первое, что приходит в голову:

```

func merge(in1, in2 <-chan int) <-chan int {
    out := make(chan int)
    go func() {
        defer close(out)
        for val := range in1 {
            out <- val
        }
        for val := range in2 {
            out <- val
        }
    }()
}

```

```
    return out
}
```

Последовательно проходим по первому каналу, затем второму, и отправляем результаты в объединенный канал:

```
11 12 13 14 21 22 23 24
Took 350ms
```

Но такая реализация уничтожила многозадачность. Пока `merge()` начитывает результаты от первой горютины `rangeGen()`, вторая горютина `rangeGen()` заблокирована — никто не готов читать ее выходной канал. Поэтому выполнение и заняло 350 мс вместо ожидаемых 200 мс ($8 \text{ значений} * 50 \text{ мс} / 2 \text{ горютины} = 200 \text{ мс}$).

Нужно что-то другое.

[песочница](#)

Композиция 4

Отменяем горютины

Есть функция `count()`, которая отправляет числа в канал от начального и до бесконечности:

```
func count(start int) <-chan int {
    out := make(chan int)
    go func() {
        for i := start; ; i++ {
            out <- i
        }
    }()
    return out
}
```

И функция `take()`, которая выбирает первые `n` чисел из входного канала и отправляет их в выходной:

```
func take(in <-chan int, n int) <-chan int {
    out := make(chan int)
    go func() {
        for i := 0; i < n; i++ {
            out <- <-in
        }
        close(out)
    }()
}
```

```
    return out
}
```

Используются они так:

```
func main() {
    stream := take(count(10), 5)
    first := <-stream
    second := <-stream
    third := <-stream

    fmt.Println(first, second, third)
    // 10 11 12
}
```

Проблема в том, что обе горутины не отработывают до конца и зависают. Исправьте, чтобы они поддерживали отмену:

```
func main() {
    cancel := make(chan struct{})
    defer close(cancel)

    stream := take(cancel, count(cancel, 10), 5)
    first := <-stream
    second := <-stream
    third := <-stream

    fmt.Println(first, second, third)
    // 10 11 12
}
```

Полный код — в песочнице. Отправляйте в качестве решения только фрагмент, отмеченный комментариями «начало решения» и «конец решения».

[песочница](#)

P.S. `take()` сейчас не проверяет, закрыт ли входной канал. Если `in` закрыт — вернет нулевые значения. Для простоты давайте считать такое поведение нормальным.

Sample Input:

Sample Output:

PASS

Канал отмены и канал завершения

Канал отмены (cancel) похож на [канал завершения](#) (done), который мы разбирали на предыдущем уроке.

Канал завершения:

```
// горутина b принимает канал,  
// на котором позже сигнализирует,  
// что закончила работу  
func b(done chan<- struct{}) {  
    // do work...  
    done <- struct{}{}  
}  
  
func a() {  
    done := make(chan struct{})  
    go b(done)  
    // горутина a ждет, пока b закончит работу  
    <- done  
}
```

Канал отмены:

```
// горутина b принимает канал,  
// на котором позже получит сигнал отмены  
func b(cancel <-chan struct{}) {  
    // do work...  
    select {  
    case <-cancel:  
        return  
    }  
}  
  
func a() {  
    cancel := make(chan struct{})  
    go b(cancel)  
    // горутина a сигнализирует b,  
    // что той пора заканчивать  
    close(cancel)  
}
```

На практике для обоих каналов — отмены и завершения — часто используют название `done`, так что не удивляйтесь. На курсе я буду использовать `cancel` для отмены и `done` для завершения, чтобы не возникало путаницы.

2

Канал отмены

Для начала создадим отдельный *канал отмены* (cancel channel), через который `main()` будет сигнализировать `rangeGen()`, что пора завершаться:

```
func main() {
    cancel := make(chan struct{}) // (1)
    defer close(cancel)           // (2)

    generated := rangeGen(cancel, 41, 46) // (3)
    for val := range generated {
        fmt.Println(val)
        if val == 42 {
            break
        }
    }
}
```

Мы создаем канал `cancel` ❶ и сразу настраиваем отложенный вызов `close(cancel)` ❷. Так часто делают, чтобы не отслеживать по коду все места, в которых нужно закрыть канал. `defer` гарантирует, что канал в любом случае закроется при выходе из функции, так что о нем можно не беспокоиться.

Затем передаем канал `cancel` в горутину ❸. Теперь, когда канал закроется, горутина должна как-то понять это и завершить работу. Хотелось бы добавить примерно такую проверку:

```
func rangeGen(cancel <-chan struct{}, start, stop int) <-chan int {
    out := make(chan int)
    go func() {
        defer close(out)
        for i := start; i < stop; i++ {
            out <- i
            if <-cancel { // (1)
                return
            }
        }
    }()
}
```

```
    return out
}
```

Если `cancel` закрыт, то проверка ❶ пройдет (закрытый канал всегда возвращает нулевое значение, помните?), и горутина завершит работу. Но вот беда: если `cancel` не закрыт, то горутина заблокируется и на следующую итерацию цикла не пойдет.

Нам нужна другая, неблокирующая логика:

- если `cancel` закрыт, выйти из горутины;
- иначе отправить очередное значение в `out`.

В Go для этого существует инструкция `select`:

```
func rangeGen(cancel <-chan struct{}, start, stop int) <-chan int {
    out := make(chan int)
    go func() {
        defer close(out)
        for i := start; i < stop; i++ {
            select {
                case out <- i:      // (1)
                case <-cancel:      // (2)
                    return
            }
        }
    }()
    return out
}
```

`select` отчасти похож на `switch`, только создан специально для работы с каналами. Вот что он делает:

1. Проверяет, какие ветки (case) не заблокированы.
2. Если таких веток несколько, выбирает одну из них случайным образом и выполняет ее.
3. Если все ветки заблокированы, блокирует выполнение, пока хотя бы одна ветка не разблокируется.

В нашем случае, пока `cancel` открыт, его ветка ❷ заблокирована (невозможно прочитать значение из канала, если никто не готов в него записать). А вот ветка ❶ `out <- i` разблокирована, потому что `main()` начитывает значения из `out`. Соответственно, `select` на каждой итерации цикла будет выполнять `out <- i`.

Затем `main()` дойдет до числа 42 и прекратит начитывать из `out`. После этого обе ветки `select` заблокируются и горутина (временно) зависнет.

Наконец, в `main()` выполнится отложенный `close(cancel)`, после чего в `select` разблокируется ветка ❷, и горутина завершит работу. Попутно закроется канал `out` (благодаря `defer`).

Если `main()` передумает останавливаться на 42 и снова будет начитывать все значения, подход с cancel-каналом продолжит корректно работать:

```
func main() {
    cancel := make(chan struct{})
    defer close(cancel)

    generated := rangeGen(cancel, 41, 46)
    for val := range generated {
        fmt.Println(val)
    }
}
```

Здесь `rangeGen()` завершится еще до того, как `main()` вызовет `close(cancel)`. Ну и ладно, проблем от этого никаких.

Таким образом, благодаря cancel-каналу и проверке через `select`, горутина `rangeGen()` корректно завершится вне зависимости от того, что происходит в `main()`. И никаких зависших горутин!

[песочница](#)

Многозадачность 5. Время

Многозадачность 5. Время

1

На этом уроке рассмотрим некоторые приемы работы со временем в многозадачных программах.

Обработка с ожиданием

Есть у нас некоторая полезная работа, которую приходится выполнять в больших количествах:

```
work := func() {
    // что-то очень нужное, но не очень быстрое
    time.Sleep(100 * time.Millisecond)
}
```

Проще всего — обрабатывать последовательно:

```
func main() {
    work := func() {
        time.Sleep(100 * time.Millisecond)
    }

    start := time.Now()

    work()
    work()
    work()
    work()

    fmt.Println("4 calls took", time.Since(start))
}
```

4 calls took 400ms

Четыре вызова по 100 мс каждый выполнились последовательно за 400 мс.

Конечно, быстрее выполнять работу параллельно, в N «обработчиков». Тогда логика будет такой:

- если есть свободный обработчик — отдать запрос ему;
- иначе ждать, пока кто-нибудь освободится.

На шаге про [N обработчиков](#) мы уже решали похожую задачу. Вспомним принцип:

- создаем канал с буфером размера N и заполняем его «токенами» (любыми значениями);
- перед запуском обработчика забираем токен из канала;
- по завершении обработчика возвращаем токен в канал.

Сделаем обертку `withWorkers(n, fn)`, которая обеспечивает одновременное выполнение. Для этого заведем канал `free` и будем следить, чтобы запускались не более `n` функций `fn()` одновременно:

```
func withWorkers(n int, fn func()) (handle func(), wait func()) {
    // канал с токенами
    free := make(chan struct{}, n)
    for i := 0; i < n; i++ {
        free <- struct{}{}
    }

    // выполняет fn, но не более n одновременно
    handle = func() {
        <-free
        go func() {
```



```

        fn()
        free <- struct{}{}
    }()
}

// ожидает, пока все запущенные fn отработают
wait = func() {
    for i := 0; i < n; i++ {
        <-free
    }
}

return handle, wait
}

```

Теперь клиент вызывает функцию `work()` не напрямую, а через обертку:

```

func main() {
    work := func() {
        time.Sleep(100 * time.Millisecond)
    }

    handle, wait := withWorkers(2, work)

    start := time.Now()

    handle()
    handle()
    handle()
    handle()
    wait()

    fmt.Println("4 calls took", time.Since(start))
}

```

4 calls took 200ms

Получилось так:

- первый и второй вызовы сразу пошли в обработку;
- третий и четвертый ждали, пока предыдущие два закончат обрабатываться.

В результате при двух обработчиках 4 вызова выполняются за 200 мс.

Схема обработки с ожиданием отлично подходит, когда степень параллелизма `n` и индивидуальное время работы `work()` примерно соответствуют интенсивности, с которой вызывают `handle()`. Тогда у каждого вызова есть хороший шанс сразу пойти в обработку.

Если же вызовов сильно больше, чем способны «прожевать» обработчики — система начнет «тормозить». Каждый отдельный `work()` по-прежнему будет отрабатывать за 100 мс. Но вызовы `handle()` будут подвисать, ведь каждому придется ждать свободного токена. Для обработки данных в конвейере это не страшно, а вот для онлайн-запросов может быть не слишком хорошо.

Бывает, что клиент предпочел бы сразу получить ошибку в ответ на `handle()`, если все обработчики заняты. Тут схема с ожиданием уже не подойдет.

[песочница](#)

10

Ограничитель скорости

Популярный способ защитить сервер от перегрузки запросами — *ограничитель скорости* (rate limiter):

1. Настраиваем, скажем, что сервер готов обрабатывать 10 запросов секунду.
2. Теперь сколько бы запросов не прислал клиент, за секунду обработаются только 10 штук. Остальным придется ждать.

Сделайте функцию-обертку `withRateLimit(limit, fn)`, которая следит, чтобы функция `fn` выполнялась не более `limit` раз в секунду:

```
func withRateLimit(limit int, fn func()) (handle func() error, cancel
func()) {
    // ...
}

func main() {
    work := func() {
        fmt.Print(".")
    }

    handle, cancel := withRateLimit(5, work)
    defer cancel()

    start := time.Now()
    const n = 10
    for i := 0; i < n; i++ {
```

```

    handle()
}
fmt.Println()
fmt.Printf("%d queries took %v\n", n, time.Since(start))
}

```

`handle()` вызывается без задержек 10 раз подряд, но благодаря `withRateLimit()` выполнение растягивается на 2 секунды (5 запросов в секунду).

```

.....
10 queries took 2s

```

Если подряд сделаны несколько вызовов `handle()`, ограничитель должен выполнять `fn` не одновременно, а равномерно:

- лимит 2/сек — интервал между запусками `fn()` 500 мс;
- лимит 5/сек — интервал 200 мс;
- лимит 10/сек — интервал 100 мс;
- и так далее.

Ограничитель должен учитывать только количество вызовов, но не время работы `fn`. Например, установлен лимит 10/сек, а `handle()` вызывается постоянно. Тогда:

- если `fn` занимает 1 мс, то каждую секунду должны запускаться 10 `fn()`;
- если `fn` занимает 5 сек, то каждую секунду должны запускаться все те же 10 `fn()`.

Обратите внимание на нюансы:

- `cancel()` должна освободить ресурсы, занятые ограничителем.
- `handle()` может быть вызвана после того, как завершилась `cancel()`. Тогда она должна вернуть ошибку `ErrCanceled`.
- `cancel()` может быть вызвана несколько раз. Тогда в первый раз она должна остановить ограничитель, а в последующие — ничего не делать.

Полный код — в песочнице. Отправляйте в качестве решения только фрагмент, отмеченный комментариями «начало решения» и «конец решения».

[песочница](#)

Sample Input:

Sample Output:

PASS

```
package main

import (
    "errors"
    "fmt"
    "time"
)

var ErrCanceled error = errors.New("canceled")

// начало решения
/*
func withRateLimit(limit int, fn func()) (handle func() error, cancel
func()) {
    dur := time.Duration(1000/limit) * time.Millisecond

    ticker := time.NewTicker(dur)
    cancel_ch := make(chan struct{})
    done := make(chan struct{})

    cancel_fn := func() {
        select {
            case <-cancel_ch: // когда канал закрыт, эта ветка всегда
исполняется (yot так yot...)
                return
            default:
                ticker.Stop()
                close(cancel_ch)
        }
    }

    tick := func() {
        select {
            case <-cancel_ch:
                return
            case <-ticker.C:
                go fn()
                done <- struct{}{}
        }
    }

    handle_fn := func() error {
        go tick()
    }
}
```

```

        select {
        case <-cancel_ch:
            return ErrCanceled
        case <-done:
            return nil
        }
    }

    return handle_fn, cancel_fn
}
*/
// конец решения

func withRateLimit(limit int, fn func()) (handle func() error, cancel
func()) {
    ticker := time.NewTicker(time.Second / time.Duration(limit))
    canceled := make(chan struct{})

    handle = func() error {
        select {
        case <-ticker.C:
            go fn()
            return nil
        case <-canceled:
            return ErrCanceled
        }
    }

    cancel = func() {
        select {
        case <-canceled:
        default:
            ticker.Stop()
            close(canceled)
        }
    }

    return handle, cancel
}

func main() {
    work := func() {

```

```

    fmt.Print(".")
    time.Sleep(1000 * time.Millisecond) // не важно, сколько
    выполняется, запуск по расписанию в параллель
}

handle, cancel := withRateLimit(5, work)
defer cancel()

start := time.Now()
const n = 10
for i := 0; i < n; i++ {
    handle()
}
fmt.Println()
fmt.Printf("%d queries took %v\n", n, time.Since(start))
}

```

9

Планировщик

Напишите функцию `schedule()`, которая запускает регулярное выполнение задачи:

- Принимает на входе интервал времени `dur` и функцию `fn`.
- Возвращает функцию отмены `cancel`.
- После запуска каждые `dur` времени выполняет `fn`.
- Если клиент вызвал `cancel()` — перестает выполнять `fn`.

```

func schedule(dur time.Duration, fn func()) func() {
    // ...
}

func main() {
    work := func() {
        at := time.Now()
        fmt.Printf("%s: work done\n", at.Format("15:04:05.000"))
    }

    cancel := schedule(50*time.Millisecond, work)
    defer cancel()

    // хватит на 5 тиков
}

```

```
    time.Sleep(260 * time.Millisecond)
}
```

Запускаем задачу каждые 50 мс:

```
19:09:25.083: work done
19:09:25.133: work done
19:09:25.183: work done
19:09:25.233: work done
19:09:25.283: work done
```

Полный код — в песочнице. Отправляйте в качестве решения только фрагмент, отмеченный комментариями «начало решения» и «конец решения».

[песочница](#)

Sample Input:

Sample Output:

PASS

```
package main

import (
    "fmt"
    "time"
)

// начало решения

func schedule(dur time.Duration, fn func()) func() {
    ticker := time.NewTicker(dur)
    cancel_ch := make(chan struct{}, 1)

    cancel_fn := func() {
        ticker.Stop()
        if len(cancel_ch) == 0 {
            cancel_ch <- struct{}{}
        }
    }

    go func() {
        defer close(cancel_ch)
        for {
```

```

        select {
        case <-cancel_ch:
            return
        case <-ticker.C:
            fn()
        }
    }
}()

return cancel_fn
}

// конец решения

/*Реализуем schedule() поверх тикера:

Создаем тикер с периодом dur, и канал canceled.
Запускаем горутину tick, которая через for+select либо начитывает тики и
выполняет fn, либо выходит по canceled.
Возвращаем функцию cancel, которая при вызове закрывает одноименный канал.*/
// func schedule(dur time.Duration, fn func()) func() {
//     ticker := time.NewTicker(dur)
//     canceled := make(chan struct{})

//     tick := func() {
//         for {
//             select {
//             case <-ticker.C:
//                 fn()
//             case <-canceled:      // В канал отмены незначем
что-то записывать, достаточно его закрыть. После этого всегда будет
срабатывать эта ветка
//                 return
//             }
//         }
//     }

//     cancel := func() {
//         select {
//         case <-canceled:
//             return
//         default:
//             ticker.Stop()

```



```
//          close(canceled)
//      }
//  }

//  go tick()
//  return cancel
// }

func main() {
    work := func() {
        at := time.Now()
        time.Sleep(50 * time.Millisecond)
        fmt.Printf("%s: work done\n", at.Format("15:04:05.000"))
    }

    cancel := schedule(10*time.Millisecond, work)
    defer func() { cancel(); cancel(); cancel() }()

    time.Sleep(120 * time.Millisecond)
}
```

8

Тикер

Бывает, хочется выполнять какое-то действие с определенной периодичностью. В Go и для этого есть инструмент — *тикер* (ticker). Тикер похож на таймер, только срабатывает не один раз, а регулярно, пока не остановите:

```
func main() {
    work := func(at time.Time) {
        fmt.Printf("%s: work done\n", at.Format("15:04:05.000"))
    }

    ticker := time.NewTicker(50 * time.Millisecond)
    defer ticker.Stop()

    go func() {
        for {
            at := <-ticker.C
            work(at)
        }
    }()
}
```

```
// хватит на 5 тиков
time.Sleep(260 * time.Millisecond)
}
```

`NewTicker(d)` создает тикер, который с периодичностью `d` отправляет текущее время в канал `C`. Тикер обязательно надо рано или поздно остановить через `Stop()`, чтобы он освободил занятые ресурсы.

В нашем случае период 50 мс, так что хватит на 5 тиков:

```
18:59:34.735: work done
18:59:34.785: work done
18:59:34.835: work done
18:59:34.885: work done
18:59:34.935: work done
```

[песочница](#)

Если читатель канала не успевает за тикером, тот будет пропускать «тики»:

```
func main() {
    work := func(at time.Time) {
        // ...
        time.Sleep(100 * time.Millisecond)
    }

    ticker := time.NewTicker(50 * time.Millisecond)
    defer ticker.Stop()

    go func() {
        for {
            at := <-ticker.C
            work(at)
        }
    }()

    // хватит на 3 тика из-за медленной work()
    time.Sleep(260 * time.Millisecond)
}
```

Здесь получатель «опаздывает» начиная со второго тика:

```
19:01:13.784: work done
19:01:13.834: work done
19:01:13.934: work done
```

Как видите, тики не «накапливаются», а подстраиваются под медленного получателя.

[песочница](#)

7

Аналог `time.AfterFunc()`

Напишите функцию `delay()` — аналог `time.AfterFunc()`:

- Принимает на входе интервал времени `dur` и функцию `fn`.
- Возвращает функцию отмены `cancel`.
- После запуска ждет в течение `dur` времени, после чего выполняет `fn`.
- Если клиент вызвал `cancel()` до истечения `dur` — не выполняет `fn`.
- Если клиент вызвал `cancel()` после истечения `dur` — ничего не делает.

```
func delay(dur time.Duration, fn func()) func() {
    // ...
}

func main() {
    rand.Seed(time.Now().Unix())

    work := func() {
        fmt.Println("work done")
    }

    cancel := delay(100*time.Millisecond, work)

    time.Sleep(10 * time.Millisecond)
    if rand.Float32() < 0.5 {
        cancel()
        fmt.Println("delayed function canceled")
    }
    time.Sleep(100 * time.Millisecond)
}
```

В примере мы отменяем запуск с 50% вероятностью:

```
work done
delayed function canceled
work done
work done
```

Клиент может вызвать `cancel()` несколько раз. Отмена должна произойти по первому вызову, при повторных не должно быть паники.

Полный код — в песочнице. Отправляйте в качестве решения только фрагмент, отмеченный комментариями «начало решения» и «конец решения».

[песочница](#)

Sample Input:

Sample Output:

PASS

```
package main

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

// начало решения аналог time.AfterFunc
/*в лоб*/
// func delay(dur time.Duration, fn func()) func() {
//     cancel_ch := make(chan struct{}, 1)

//     cancel_fn := func() {
//         if len(cancel_ch) == 0 {
//             cancel_ch <- struct{}{}
//         }
//     }

//     go func() {
//         time.Sleep(dur)
//         select {
//             case <- cancel_ch:
//                 return
//             default:
//                 fn()
//         }
//     }()

//     return cancel_fn
}
```

```
// }
```

```
// конец решения
```

*/*Можно реализовать delay() с помощью таймера. Но я решил обойтись только каналами.*

Идея следующая:

Создаем два канала – done для успешного срабатывания и canceled для отмены. Запускаем горутину wait(), которая ждет dur времени, после чего сигнализирует done.

Запускаем горутину tick(), которая через select выбирает первый сработавший канал (done или canceled). Если done – выполняет отложенную функцию fn.

Возвращаем функцию cancel, которая при вызове закрывает одноименный канал.

Предотвращаем повторное закрытие канала отмены через select./**

```
// func delay(dur time.Duration, fn func()) func() {
```

```
//     done := make(chan struct{})
```

```
//     canceled := make(chan struct{})
```

```
//     // ждет dur времени, после чего сигнализирует done
```

```
//     wait := func() {
```

```
//         time.Sleep(dur)
```

```
//         close(done)
```

```
//     }
```

```
//     // выполняет fn по готовности, либо выходит по отмене
```

```
//     tick := func() {
```

```
//         select {
```

```
//             case <-done:
```

```
//                 fn()
```

```
//             case <-canceled:
```

```
//                 return
```

```
//         }
```

```
//     }
```

```
//     // отменяет запуск
```

```
//     cancel := func() {
```

```
//         select {
```

```
//             case <-canceled:
```

```
//                 default:
```

```
//                     close(canceled)
```

```
//         }
```

```
//    }

//    go tick()
//    go wait()

//    return cancel
// }
```

*/*Реализация delay() на базе таймера:*

Создаем таймер и канал отмены.

Запускаем горутину, которая через select выбирает из канала таймера или отмены, смотря что сработает раньше. Если канал таймера – выполняет отложенную функцию fn.

Возвращаем функцию cancel, которая при вызове останавливает таймер и закрывает одноименный канал./*

```
func delay(duration time.Duration, fn func()) func() {
    canceled := make(chan struct{})

    timer := time.NewTimer(duration)
    go func() {
        select {
            case <-timer.C:
                fn()
            case <-canceled:
        }
    }()

    cancel := func() {
        if !timer.Stop() {
            return
        }
        close(canceled)
    }
    return cancel
}
```

```
func main() {
    rand.Seed(time.Now().Unix())

    work := func() {
        fmt.Println("work done")
    }
```

```

}

for i := 0; i < 25; i++ {
    cancel := delay(100*time.Millisecond, work)

    time.Sleep(10 * time.Millisecond)
    if rand.Float32() < 0.5 {
        cancel()
        cancel()
        cancel()    // множественный вызов не должен ломать логику
    }
    fmt.Println("delayed function canceled")
}
time.Sleep(100 * time.Millisecond)
}

```

отмены

6

Таймер

Бывает, хочется выполнить действие не прямо сейчас, а через какое-то время. В Go для этого предусмотрен инструмент *таймер* (timer):

```

func main() {
    work := func() {
        fmt.Println("work done")
    }

    var eventTime time.Time

    start := time.Now()
    timer := time.NewTimer(100 * time.Millisecond)    // (1)
    go func() {
        eventTime = <-timer.C                        // (2)
        work()
    }()

    // достаточно времени, чтобы сработал таймер
    time.Sleep(150 * time.Millisecond)
    fmt.Printf("delayed function started after %v\n", eventTime.Sub(start))
}

```

`time.NewTimer()` создает новый таймер ❶, который сработает через указанный промежуток времени. Таймер — это структура с каналом `C`, в который он запишет текущее время, когда сработает ❷. Благодаря этому, функция `work()` выполнится только после того, как таймер сработает.

```
work done
delayed function started after 100ms
```

[песочница](#)

Таймер можно остановить, тогда в канал `C` значение не придет, и `work()` не запустится:

```
func main() {
    // ...

    start := time.Now()
    timer := time.NewTimer(100 * time.Millisecond)
    go func() {
        <-timer.C
        work()
    }()

    time.Sleep(10 * time.Millisecond)
    fmt.Println("10ms has passed...")
    // таймер еще не успел сработать
    if timer.Stop() {
        fmt.Printf("delayed function canceled after %v\n",
time.Since(start))
    }
}
```

`Stop()` останавливает таймер и возвращает `true`, если он еще не успел сработать. Поскольку мы остановили таймер уже через 10 мс, будет `true`:

```
10ms has passed...
delayed function canceled after 10ms
```

Возможно, вы заметили неприятный побочный эффект: поскольку в `timer.C` никогда не придет значение, наша горутина зависнет. Исправить это можно через `select`, либо библиотечной функцией, о которой поговорим ниже.

[песочница](#)

Если остановить таймер слишком поздно, `Stop()` вернет `false`:


```
func main() {
    // ...

    timer := time.NewTimer(100 * time.Millisecond)
    go func() {
        <-timer.C
        work()
    }()

    time.Sleep(150 * time.Millisecond)
    fmt.Println("150ms has passed...")
    // слишком поздно, таймер уже сработал
    if !timer.Stop() {
        fmt.Println("too late to cancel")
    }
}
```

```
work done
150ms has passed...
too late to cancel
```

[песочница](#)

Для отложенного запуска функции не обязательно вручную возиться с созданием таймера и считыванием из канала. Есть удобная обертка `[time.AfterFunc()]` (<https://pkg.go.dev/time#AfterFunc>):

```
func main() {
    work := func() {
        fmt.Println("work done")
    }

    time.AfterFunc(100*time.Millisecond, work)

    // достаточно времени, чтобы сработал таймер
    time.Sleep(150 * time.Millisecond)
}
```

```
work done
```

`AfterFunc(d, f)` ждет в течение времени `d`, после чего выполняет функцию `f`. Возвращает уже знакомый нам таймер, через который можно отменить выполнение, пока оно не началось:

```
func main() {
    work := func() {
```

```

    fmt.Println("work done")
}

timer := time.AfterFunc(100*time.Millisecond, work)

time.Sleep(10 * time.Millisecond)
fmt.Println("10ms has passed...")
// таймер еще не успел сработать
if timer.Stop() {
    fmt.Println("execution canceled")
}
}

```

```

10ms has passed...
execution canceled

```

В этом случае при отмене выполнения через `timer.Stop()` никакие горютины не зависают (лишний довод использовать библиотечные функции вместо собственных реализаций).

[песочница](#)

В работе таймеров много нюансов, так что если будете использовать их в каких-то сложных сценариях — сверьтесь с [документацией](#).

5

Аналог `time.After()`

Напишите функцию `after()` — аналог `time.After()`:

1. Принимает интервал времени `dur`.
2. Возвращает канал типа `time.Time`.
3. Через промежуток `dur` отправляет в канал значение текущего времени.

Используется аналогично `time.After()` ❶:

```

func withTimeout(fn func() int, timeout time.Duration) (int, error) {
    var result int

    done := make(chan struct{})
    go func() {
        result = fn()
        close(done)
    }()
}

```

```

select {
case <-done:
    return result, nil
case <-after(timeout):    // (1)
    return 0, errors.New("timeout")
}
}

```

Полный код — в песочнице. Отправляйте в качестве решения только фрагмент, отмеченный комментариями «начало решения» и «конец решения».

[песочница](#)

Sample Input:

Sample Output:

PASS

```

package main

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

// выполняет какую-то операцию,
// обычно быстро, но иногда медленно
func work() int {
    if rand.Intn(10) < 8 {
        time.Sleep(10 * time.Millisecond)
    } else {
        time.Sleep(200 * time.Millisecond)
    }
    return 42
}

// выполняет функцию fn() с таймаутом timeout и возвращает результат
// если в течение timeout функция не вернула ответ - возвращает ошибку
func withTimeout(fn func() int, timeout time.Duration) (int, error) {
    var result int

```

```

done := make(chan struct{})
go func() {
    result = fn()
    close(done)
}()

select {
case <-done:
    return result, nil
case <-after(timeout):
    return 0, errors.New("timeout")
}
}

// начало решения

// возвращает канал, в котором появится значение
// через промежуток времени dur
// func after(dur time.Duration) <-chan time.Time {
//     ch := make(chan time.Time, 1)
//     go func() {
//         time.Sleep(dur)
//         ch <- time.Now()
//     }()
//     return ch
// }

func after(dur time.Duration) <-chan time.Time {
    // канал без буферизации (встроенный time.After с буфером)
    timeChan := make(chan time.Time)

    go func() {
        time.Sleep(dur)
        select {
        case timeChan <- time.Now():
        default:
            return
        }
    }()

    return timeChan
}

```

```

}

// конец решения

func main() {
    for i := 0; i < 10; i++ {
        start := time.Now()
        timeout := 50 * time.Millisecond
        if answer, err := withTimeout(work, timeout); err != nil {
            fmt.Printf("Took %v. Error: %v\n", time.Since(start), err)
        } else {
            fmt.Printf("Took %v. Result: %v\n", time.Since(start), answer)
        }
    }
}

```

4

Таймаут операции

Есть функция, которая обычно отрабатывает за 10 мс, но в 20% случаев занимает 200 мс:

```

func work() int {
    if rand.Intn(10) < 8 {
        time.Sleep(10 * time.Millisecond)
    } else {
        time.Sleep(200 * time.Millisecond)
    }
    return 42
}

```

Мы в принципе не хотим ждать дольше, скажем, 50 мс. Поэтому установим *таймаут* (timeout) — максимальное время, в течение которого готовы ждать ответ. Если операция не уложилась в таймаут, будем считать это ошибкой.

Сделаем обертку, которая выполняет переданную функцию с указанным таймаутом, и будем вызывать ее вот так:

```

func withTimeout(fn func() int, timeout time.Duration) (int, error) {
    // ...
}

func main() {
    for i := 0; i < 10; i++ {
        start := time.Now()

```

```

        timeout := 50 * time.Millisecond
        if answer, err := withTimeout(work, timeout); err != nil {
            fmt.Printf("Took longer than %v. Error: %v\n",
time.Since(start), err)
        } else {
            fmt.Printf("Took %v. Result: %v\n", time.Since(start), answer)
        }
    }
}

```

```

Took 10ms. Result: 42
Took 10ms. Result: 42
Took 10ms. Result: 42
Took longer than 50ms. Error: timeout
Took 10ms. Result: 42
Took longer than 50ms. Error: timeout
Took 10ms. Result: 42
Took 10ms. Result: 42
Took 10ms. Result: 42
Took 10ms. Result: 42

```

Идея работы `withTimeout()` следующая:

- Запускаем переданную `fn()` в отдельной горутине.
- Ждем в течение `timeout` времени.
- Если `fn()` вернула ответ — возвращаем его.
- Если не успела — возвращаем ошибку.

Вот как можно это реализовать:

```

func withTimeout(fn func() int, timeout time.Duration) (int, error) {
    var result int

    done := make(chan struct{})
    go func() {
        result = fn()
        close(done)
    }()

    select {
    case <-done:
        return result, nil
    case <-time.After(timeout):
        return 0, errors.New("timeout")
    }
}

```

```
}  
}
```

Здесь все знакомо, кроме `[time.After()](https://pkg.go.dev/time#After)`. Эта библиотечная функция возвращает канал, который изначально пуст, а через `timeout` времени отправляет в него значение. Благодаря этому `select` выберет нужный вариант:

- ветку `<-done`, если `fn()` успела до таймаута (вернет ответ);
- ветку `<-time.After()`, если не успела (вернет ошибку).

[песочница](#)

3

Очередь с блокировкой и без

Релизуйте тип `Queue` — очередь фиксированного размера на `n` элементов. Поддерживает две операции:

```
Get(block bool) (int, error)  
Put(val int, block bool) error
```

`Put()` помещает значение `val` в очередь:

- Если в очереди есть место — помещает `val` в очередь и возвращает `nil`.
- Если очередь заполнена и `block = true` — блокируется, пока не освободится место. Затем помещает `val` в очередь и возвращает `nil`.
- Если очередь заполнена и `block = false` — возвращает ошибку `ErrFull`.

`Get()` выбирает значение из очереди:

- Если в очереди есть значения — выбирает очередное значение, возвращает его и `nil`.
- Если очередь пуста и `block = true` — блокируется, пока не появится значение. Затем выбирает и возвращает его и `nil`.
- Если очередь пуста и `block = false` — возвращает нулевое значение и ошибку `ErrEmpty`.

Очередь работает по принципу FIFO (first in — first out): какое значение добавили раньше, такое раньше и вернется.

Полный код — в песочнице. Отправляйте в качестве решения только фрагмент, отмеченный комментариями «начало решения» и «конец решения».

Подсказка

Если запускаете горутин и выделяете отдельный канал с токенами — вы на ложном пути. Прочитайте задание: здесь нет обработчиков, а есть очередь, в которую надо складывать и

доставать значения. Постарайтесь решить ровно ту задачу, что описана в постановке, обойдясь минимальным количеством средств. Не усложняйте.

[песочница](#)

Sample Input:

Sample Output:

PASS

```
package main

import (
    "errors"
    "fmt"
)

var ErrFull = errors.New("Queue is full")
var ErrEmpty = errors.New("Queue is empty")

/*
// начало решения

// Queue - FIFO-очередь на n элементов
// type queue interface {
//     MakeQueue() Queue
//     Get() any
//     Put() any
// }

type Queue struct {
    size int
    fifo chan int
}

// Get возвращает очередной элемент.
// Если элементов нет и block = false -
// возвращает ошибку.
func (q Queue) Get(block bool) (int, error) {
    if len(q.fifo) != 0 {
        return <-q.fifo, nil
    } else {
        if block {
```



```

        return <-q.fifo, nil
    } else {
        return 0, ErrEmpty
    }
}

// Put помещает элемент в очередь.
// Если очередь заполнения и block = false -
// возвращает ошибку.
func (q Queue) Put(val int, block bool) error {
    if len(q.fifo) < q.size {
        q.fifo <- val
        return nil
    } else {
        if block {
            q.fifo <- val
            return nil
        } else {
            return ErrFull
        }
    }
}

// MakeQueue создает новую очередь (конструктор)
func MakeQueue(n int) Queue {
    return Queue{
        size: n,
        fifo: make(chan int, n),
    }
}

// конец решения
*/

/*
// Queue - FIFO-очередь на n элементов
type Queue struct {
    values chan int
}

// Get возвращает очередной элемент.
// Если элементов нет и block = false -

```

```

// возвращает ошибку.
func (q Queue) Get(block bool) (int, error) {
    if block {
        return <-q.values, nil
    }
    select {
    case val := <-q.values:
        return val, nil
    default:
        return 0, ErrEmpty
    }
}

// Put помещает элемент в очередь.
// Если очередь заполнения и block = false -
// возвращает ошибку.
func (q Queue) Put(val int, block bool) error {
    if block {
        q.values <- val
        return nil
    }
    select {
    case q.values <- val:
        return nil
    default:
        return ErrFull
    }
}

// MakeQueue создает новую очередь
func MakeQueue(n int) Queue {
    ch := make(chan int, n)
    return Queue{ch}
}
*/

// начало решения

// Queue - FIFO-очередь на n элементов
type Queue chan int

// Get возвращает очередной элемент.

```

```

// Если элементов нет и block = false -
// возвращает ошибку.
func (q Queue) Get(block bool) (int, error) {
    if len(q) > 0 || block {
        return <-q, nil
    }

    return 0, ErrEmpty
}

// Put помещает элемент в очередь.
// Если очередь заполнения и block = false -
// возвращает ошибку.
func (q Queue) Put(val int, block bool) error {
    if len(q) < cap(q) || block {
        q <- val
        return nil
    }

    return ErrFull
}

// MakeQueue создает новую очередь
func MakeQueue(n int) Queue {
    return make(Queue, n)
}

// конец решения

func main() {
    q := MakeQueue(2)

    err := q.Put(1, false)
    fmt.Println("put 1:", err)

    err = q.Put(2, false)
    fmt.Println("put 2:", err)

    err = q.Put(3, false)
    fmt.Println("put 3:", err)

    res, err := q.Get(false)
    fmt.Println("get:", res, err)
}

```

```
res, err = q.Get(false)
fmt.Println("get:", res, err)

res, err = q.Get(false)
fmt.Println("get:", res, err)
}
```

2

Обработка без ожидания

Изменим логику `withWorkers()`:

- если есть свободный токен — выполнить функцию;
- иначе сразу вернуть ошибку.

Так клиенту не придется ждать «подвисшего» вызова.

Нам в очередной раз поможет инструкция `select`:

```
// выполняет fn, но не более n одновременно
handle = func() error {
    select {
    case <- free:
        go func() {
            fn()
            free <- struct{}{}
        }()
        return nil
    default:
        return errors.New("busy")
    }
}
```

Вспомним алгоритм работы селекта:

1. Проверяет, какие ветки не заблокированы.
2. Если таких веток несколько, выбирает одну из них случайным образом и выполняет ее.
3. Если все ветки заблокированы, блокирует выполнение, пока хотя бы одна ветка не разблокируется.

На самом деле, третий пункт разбивается на два:

- Если все ветки заблокированы *и нет блока default* — блокирует выполнение, пока хотя бы одна ветка не разблокируется.
- Если все ветки заблокированы *и есть блок default* — выполняет его.

`default` идеально подходит для нашей ситуации:

- если есть свободный токен в канале `free` — запускаем `fn`;
- иначе ничего не ждем и возвращаем ошибку `busy`.

Посмотрим на клиента:

```
func main() {
    work := func() {
        time.Sleep(100 * time.Millisecond)
    }

    handle, wait := withWorkers(2, work)

    start := time.Now()

    err := handle()
    fmt.Println("1st call, error:", err)

    err = handle()
    fmt.Println("2nd call, error:", err)

    err = handle()
    fmt.Println("3rd call, error:", err)

    err = handle()
    fmt.Println("4th call, error:", err)

    wait()

    fmt.Println("4 calls took", time.Since(start))
}
```

```
1st call, error: <nil>
2nd call, error: <nil>
3rd call, error: busy
4th call, error: busy
4 calls took 100ms
```

Первые два вызова выполнились одновременно (по 100 мс каждый), а третий и четвертый моментально получили ошибку. В итоге все вызовы обработались за 100 мс.

Конечно, при таком подходе требуется некоторая «сознательность» клиента. Клиент должен понять, что ошибка «busy» означает перегруз, и отложить дальнейшие вызовы `handle()` на некоторое время, или уменьшить их частоту.

[песочница](#)

10

Ограничитель скорости

Популярный способ защитить сервер от перегрузки запросами — *ограничитель скорости* (rate limiter):

1. Настраиваем, скажем, что сервер готов обрабатывать 10 запросов секунду.
2. Теперь сколько бы запросов не прислал клиент, за секунду обработаются только 10 штук. Остальным придется ждать.

Сделайте функцию-обертку `withRateLimit(limit, fn)`, которая следит, чтобы функция `fn` выполнялась не более `limit` раз в секунду:

```
func withRateLimit(limit int, fn func()) (handle func() error, cancel
func()) {
    // ...
}

func main() {
    work := func() {
        fmt.Print(".")
    }

    handle, cancel := withRateLimit(5, work)
    defer cancel()

    start := time.Now()
    const n = 10
    for i := 0; i < n; i++ {
        handle()
    }
    fmt.Println()
    fmt.Printf("%d queries took %v\n", n, time.Since(start))
}
```

`handle()` вызывается без задержек 10 раз подряд, но благодаря `withRateLimit()` выполнение растягивается на 2 секунды (5 запросов в секунду).

.....

10 queries took 2s

Если подряд сделаны несколько вызовов `handle()`, ограничитель должен выполнять `fn` не одновременно, а равномерно:

- лимит 2/сек — интервал между запусками `fn()` 500 мс;
- лимит 5/сек — интервал 200 мс;
- лимит 10/сек — интервал 100 мс;
- и так далее.

Ограничитель должен учитывать только количество вызовов, но не время работы `fn`. Например, установлен лимит 10/сек, а `handle()` вызывается постоянно. Тогда:

- если `fn` занимает 1 мс, то каждую секунду должны запускаться 10 `fn()`;
- если `fn` занимает 5 сек, то каждую секунду должны запускаться все те же 10 `fn()`.

Обратите внимание на нюансы:

- `cancel()` должна освободить ресурсы, занятые ограничителем.
- `handle()` может быть вызвана после того, как выполнялась `cancel()`. Тогда она должна вернуть ошибку `ErrCanceled`.
- `cancel()` может быть вызвана несколько раз. Тогда в первый раз она должна остановить ограничитель, а в последующие — ничего не делать.

Полный код — в песочнице. Отправляйте в качестве решения только фрагмент, отмеченный комментариями «начало решения» и «конец решения».

[песочница](#)

Sample Input:

Sample Output:

PASS

```
package main

import (
    "errors"
    "fmt"
    "time"
)

var ErrCanceled error = errors.New("canceled")
```

```

// начало решения (куча лишнего, ниже правильно)
/*
func withRateLimit(limit int, fn func()) (handle func() error, cancel
func()) {
    dur := time.Duration(1000/limit) * time.Millisecond

    ticker := time.NewTicker(dur)
    cancel_ch := make(chan struct{})
    done := make(chan struct{})

    cancel_fn := func() {
        select {
            case <-cancel_ch: // когда канал закрыт, эта ветка всегда
исполняется (уот так уот...)
                return
            default:
                ticker.Stop()
                close(cancel_ch)
        }
    }

    tick := func() {
        select {
            case <-cancel_ch:
                return
            case <-ticker.C:
                go fn()
                done <- struct{}{}
        }
    }

    handle_fn := func() error {
        go tick()
        select {
            case <-cancel_ch:
                return ErrCanceled
            case <-done:
                return nil
        }
    }

    return handle_fn, cancel_fn
}

```



```

}
*/
// конец решения

func withRateLimit(limit int, fn func()) (handle func() error, cancel
func()) {
    ticker := time.NewTicker(time.Second / time.Duration(limit))
    canceled := make(chan struct{})

    handle = func() error {
        select {
        case <-ticker.C:
            go fn()
            return nil
        case <-canceled:
            return ErrCanceled
        }
    }

    cancel = func() {
        select {
        case <-canceled:
        default:
            ticker.Stop()
            close(canceled)
        }
    }

    return handle, cancel
}

func main() {
    work := func() {
        fmt.Print(".")
        time.Sleep(1000 * time.Millisecond) // не важно, сколько
        выполняется, запуск по расписанию в параллель
    }

    handle, cancel := withRateLimit(5, work)
    defer cancel()

    start := time.Now()
    const n = 10

```

```
for i := 0; i < n; i++ {
    handle()
}
fmt.Println()
fmt.Printf("%d queries took %v\n", n, time.Since(start))
}
```

Многозадачность 6. Контекст

Многозадачность 6. Контекст

1

В программировании *контекст* (context) — это информация о среде, в которой существует объект или выполняется функция. В Go под контекстом обычно имеют в виду интерфейс `Context` из пакета `context`. Его придумали, чтобы облегчить работу с HTTP-запросами, так что мы еще встретим контексты в следующем модуле, посвященном стандартной библиотеке. Но контексты можно использовать и в обычном многозадачном коде. Давайте посмотрим, как именно.

Отмена операции через канал

Рассмотрим функцию `execute()`, которая умеет запустить переданную функцию и поддерживает отмену:

```
// выполняет функцию fn
func execute(cancel <-chan struct{}, fn func() int) (int, error) {
    ch := make(chan int, 1)

    go func() {
        ch <- fn()
    }()

    select {
    case res := <-ch:
        return res, nil
    case <-cancel:
        return 0, errors.New("canceled")
    }
}
```

Здесь все знакомо:

- функция принимает канал, через который может получить сигнал отмены;
- запускает `fn()` в отдельной горутине;

- через `select` дожидается выполнения `fn()` либо прерывается по отмене, смотря что наступит раньше.

Напишем клиента, который отменяет операцию с 50% вероятностью:

```
func main() {
    rand.Seed(time.Now().Unix())

    // работает в течение 100 мс
    work := func() int {
        time.Sleep(100 * time.Millisecond)
        fmt.Println("work done")
        return 42
    }

    // ждет 50 мс, после этого
    // с вероятностью 50% отменяет работу
    maybeCancel := func(cancel chan struct{}) {
        time.Sleep(50 * time.Millisecond)
        if rand.Float32() < 0.5 {
            close(cancel)
        }
    }

    cancel := make(chan struct{})

    go maybeCancel(cancel)

    res, err := execute(cancel, work)
    fmt.Println(res, err)
}
```

Запустим несколько раз:

```
0 canceled
```

```
work done
```

```
42 <nil>
```

```
0 canceled
```

```
work done
```

```
42 <nil>
```

Без неожиданностей.

А теперь сделаем то же самое через контекст.

[песочница](#)

8

Контекст со значениями

Основное назначение контекста в Go — отмена операций. Но есть и еще одно — передача дополнительной информации о вызове. За это отвечает `context.WithValue()`, которая создает контекст со значением по ключу:

```
type contextKey string

var requestIdKey = contextKey("id")
var userKey = contextKey("user")

func main() {
    work := func() int {
        return 42
    }

    // контекст с идентификатором запроса
    ctx := context.WithValue(context.Background(), requestIdKey, 1234)
    // и пользователем
    ctx = context.WithValue(ctx, userKey, "admin")
    res := execute(ctx, work)
    fmt.Println(res)

    // пустой контекст
    ctx = context.Background()
    res = execute(ctx, work)
    fmt.Println(res)
}
```

В качестве ключей принято использовать не строки или числа, а отдельные типы (`contextKey` в нашем примере). Так не возникнет конфликтов, если один и тот же контекст модифицируется в двух пакетах, и оба решат добавить значение с ключом `"user"`.

Чтобы достать значение по ключу, используют метод контекста `Value()`:

```
// выполняет функцию fn с учетом контекста ctx
func execute(ctx context.Context, fn func() int) int {
    reqId := ctx.Value(requestIdKey)
```

```

if reqId != nil {
    fmt.Printf("Request ID = %d\n", reqId)
} else {
    fmt.Println("Request ID unknown")
}

user := ctx.Value(userKey)
if user != nil {
    fmt.Printf("Request user = %s\n", user)
} else {
    fmt.Println("Request user unknown")
}
return fn()
}

```

```

Request ID = 1234
Request user = admin
42
Request ID unknown
Request user unknown
42

```

И `context.WithValue()`, и `ctx.Value()` оперируют значениями типа `any`:

```

func WithValue(parent Context, key, val any) Context

type Context interface {
    // ...
    Value(key any) any
}

```

Эту уродливую нетипизированную парочку применяют в обработке HTTP-запросов от безысходности: там нет нормального способа передать метаданные запроса, кроме как сложить их в контекст. Но нет ни единой причины использовать `WithValue()` в обычном коде. Даже задачу предлагать не буду (_)

[песочница](#)

7

Конвейер с контекстом

На одном из предыдущих уроков мы занимались функцией, которая считает количество цифр в словах:

```

func countDigitsInWords(words []string) counter {
    pending := submitWords(words)
    counted := countWords(pending)
    return fillStats(counted)
}

// ...

func main() {
    phrase := "One two thr33 4068"
    words := strings.Fields(phrase)
    stats := countDigitsInWords(words)
    fmt.Println(stats)
}

```

```
map[0ne:1 two:1 4068:4 thr33:2]
```

Добавьте возможность отмены `countDigitsInWords()` и всех дочерних функций через контекст. Передавайте контекст первым аргументом функции.

Если заметите какие-то проблемы по коду, тоже исправьте.

Полный код — в песочнице. Отправляйте в качестве решения только фрагмент, отмеченный комментариями «начало решения» и «конец решения».

[песочница](#)

Sample Input:

Sample Output:

PASS

```

package main

import (
    "context"
    "fmt"
    "strings"
    "unicode"
)

// информация о количестве цифр в каждом слове
type counter map[string]int

```

```
// слово и количество цифр в нем
type pair struct {
    word  string
    count int
}

// начало решения

// считает количество цифр в словах
func countDigitsInWords(ctx context.Context, words []string) counter {

    ctx, cancel := context.WithCancel(ctx)
    defer cancel()

    pending := submitWords(ctx, words)
    counted := countWords(ctx, pending)
    return fillStats(ctx, counted)
}

// отправляет слова на подсчет
func submitWords(ctx context.Context, words []string) <-chan string {
    out := make(chan string, 10)

    go func() {
        defer close(out)
        for _, word := range words {
            select {
            case out <- word:
            case <-ctx.Done():
                return
            }
        }
    }()

    return out
}

// считает цифры в словах
func countWords(ctx context.Context, in <-chan string) <-chan pair {
    out := make(chan pair, 10)

    go func() {
        defer close(out)
```

```

        for word := range in {
            count := countDigits(word)
            select {
            case out <- pair{word, count}:
            case <-ctx.Done():
                return
            }
        }
    }()

    return out
}

// ГОТОВИТ ИТОГОВУЮ СТАТИСТИКУ
func fillStats(ctx context.Context, in <-chan pair) counter {
    /*Но в данном случае контекст для отмены не нужен, т.к. функция не пишет
    в каналы и без
    горутинг, когда закончится канал, закончится и цикл*/
    stats := counter{}

    for p := range in {
        stats[p.word] = p.count
        select {
        case <-ctx.Done():
            return stats
        default:
            stats[p.word] = p.count
        }
        // ну или так тоже сойдет...
        // if len(ctx.Done()) > 0 {
        //     return stats
        // }
    }

    return stats
}

// конец решения

// считает количество цифр в слове
func countDigits(str string) int {
    count := 0
    for _, char := range str {

```



```

        if unicode.IsDigit(char) {
            count++
        }
    }
    return count
}

func main() {
    phrase := "One two thr33 4068"
    words := strings.Fields(phrase)

    ctx := context.Background()
    stats := countDigitsInWords(ctx, words)
    fmt.Println(stats)
}

```

6

Дедлайн

Кроме таймаута, контекст поддерживает *дедлайн* (deadline) — это когда операция отменяется не через N секунд, а в конкретный момент времени:

```

// выполняет функцию fn с учетом контекста ctx
func execute(ctx context.Context, fn func() int) (int, error) {
    // без изменений
}

func main() {
    // ...

    // работает в течение 100 мс
    work := func() int {
        // ...
    }

    // возвращает случайный аргумент из переданных
    randomChoice := func(arg ...int) int {
        // ...
    }

    // случайный дедлайн - +50 мс либо +150 мс
    // от текущего времени
    timeout := time.Duration(randomChoice(50, 150)) * time.Millisecond
}

```

```

    deadline := time.Now().Add(timeout)
    ctx, cancel := context.WithDeadline(context.Background(), deadline) //
(1)
    defer cancel()

    res, err := execute(ctx, work)
    fmt.Println(res, err)
}

```

work done

42 <nil>

0 context deadline exceeded

0 context deadline exceeded

work done

42 <nil>

[песочница](#)

Как видите, `context.WithDeadline()` ^❶ ведет себя точно так же, как `context.WithTimeout()` — только принимает значение `time.Time` вместо `time.Duration`.

Больше того, на самом деле `WithTimeout()` — это просто обертка над `WithDeadline()`:

// фрагмент кода стандартной библиотеки

```

func WithTimeout(parent Context, timeout time.Duration) (Context,
CancelFunc) {
    return WithDeadline(parent, time.Now().Add(timeout))
}

```

Внутри контекст всегда оперирует конкретным дедлайном. Он доступен через метод `Deadline()`:

```

now := time.Now()
fmt.Println(now)
// 2009-11-10 23:00:00

ctx, _ := context.WithTimeout(context.Background(), 5*time.Second)
deadline, ok := ctx.Deadline()
fmt.Println(deadline, ok)
// 2009-11-10 23:00:05 true

plus5s := now.Add(5 * time.Second)
ctx, _ = context.WithDeadline(context.Background(), plus5s)

```

```
deadline, ok = ctx.Deadline()
fmt.Println(deadline, ok)
// 2009-11-10 23:00:05 true
```

Второе значение на выходе `Deadline()` — признак того, что дедлайн установлен:

- для контекстов, созданных через `WithTimeout()` и `WithDeadline()`, признак равен `true`;
- для `WithCancel()` и `Background()` — `false`.

```
ctx, _ := context.WithCancel(context.Background())
deadline, ok := ctx.Deadline()
fmt.Println(deadline, ok)
// 0001-01-01 00:00:00 false
```

```
ctx = context.Background()
deadline, ok = ctx.Deadline()
fmt.Println(deadline, ok)
// 0001-01-01 00:00:00 false
```

[песочница](#)

5

Родительский и дочерний таймауты

Допустим, у нас есть все та же функция `execute()` и две функции, которые она может выполнить — быстрая `work()` и медленная `slow()`:

```
// выполняет функцию fn с учетом контекста ctx
func execute(ctx context.Context, fn func() int) (int, error) {
    ch := make(chan int, 1)

    go func() {
        ch <- fn()
    }()

    select {
    case res := <-ch:
        return res, nil
    case <-ctx.Done():
        return 0, ctx.Err()
    }
}
```

```
// работает в течение 100 мс
work := func() int {
    time.Sleep(100 * time.Millisecond)
    return 42
}

// работает в течение 300 мс
slow := func() int {
    time.Sleep(300 * time.Millisecond)
    return 13
}
```

Пусть таймаут по умолчанию составляет 200 мс:

```
// возвращает контекст
// с умолчательным таймаутом 200 мс
getDefaultCtx := func() (context.Context, context.CancelFunc) {
    const timeout = 200 * time.Millisecond
    return context.WithTimeout(context.Background(), timeout)
}
```

Тогда `work()` с умолчательным контекстом успеет выполниться:

```
// таймаут 200 мс
ctx, cancel := getDefaultCtx()
defer cancel()

// успеет выполниться
res, err := execute(ctx, work)
fmt.Println(res, err)
// 42 <nil>
```

А `slow()` — не успеет:

```
// таймаут 200 мс
ctx, cancel := getDefaultCtx()
defer cancel()

// НЕ успеет выполниться
res, err := execute(ctx, slow)
fmt.Println(res, err)
// 0 context deadline exceeded
```

Мы можем создать дочерний контекст, чтобы задать более жесткий таймаут. Тогда применится именно он, а не родительский:

```
// родительский контекст с таймаутом 200 мс
parentCtx, cancel := getDefaultCtx()
defer cancel()

// дочерний контекст с таймаутом 50 мс
childCtx, cancel := context.WithTimeout(parentCtx, 50*time.Millisecond)
defer cancel()

// теперь work НЕ успеет выполниться
res, err := execute(childCtx, work)
fmt.Println(res, err)
// 0 context deadline exceeded
```

А если создать дочерний контекст с более мягким ограничением — он окажется бесполезен. Таймаут родительского контекста сработает раньше:

```
// родительский контекст с таймаутом 200 мс
parentCtx, cancel := getDefaultCtx()
defer cancel()

// дочерний контекст с таймаутом 500 мс
childCtx, cancel := context.WithTimeout(parentCtx, 500*time.Millisecond)
defer cancel()

// slow все равно НЕ успеет выполниться
res, err := execute(childCtx, slow)
fmt.Println(res, err)
// 0 context deadline exceeded
```

Получается вот что:

- Из таймаутов, наложенных родительским и дочерним контекстами, всегда срабатывает более жесткий.
- Дочерние контексты могут только ужесточить таймаут родительского, но не ослабить его.

[песочница](#)

4

Таймаут

Настоящая сила контекста в том, что его можно использовать как для ручной отмены, так и для отмены по таймауту. Следите за руками:

```

// выполняет функцию fn с учетом контекста ctx
func execute(ctx context.Context, fn func() int) (int, error) {
    // код не меняется
}

func main() {
    // ...

    // работает в течение 100 мс
    work := func() int {
        // ...
    }

    // возвращает случайный аргумент из переданных
    randomChoice := func(arg ...int) int {
        i := rand.Intn(len(arg))
        return arg[i]
    }

    // случайный таймаут - 50 мс либо 150 мс
    timeout := time.Duration(randomChoice(50, 150)) * time.Millisecond
    ctx, cancel := context.WithTimeout(context.Background(), timeout)    //
(1)
    defer cancel()

    res, err := execute(ctx, work)
    fmt.Println(res, err)
}

```

Функция `execute()` вообще не изменилась, а в `main()` вместо `context.WithCancel()` теперь `context.WithTimeout()` ❶. Этого достаточно, чтобы `execute()` теперь отваливалась по таймауту в половине случаев (ошибка `context.DeadlineExceeded`):

```

work done
42 <nil>

0 context deadline exceeded

0 context deadline exceeded

work done
42 <nil>

```

Благодаря контексту, функции `execute()` больше не нужно знать, чем вызвана отмена — ручным действием или таймаутом. Все, что от нее требуется — слушать сигнал отмены на канале `ctx.Done()`.

Удобно!

[песочница](#)

3

Отмена генератора

Есть функция `generate()`, которая генерит числа:

```
// генерит целые числа от start и до бесконечности
func generate(cancel <-chan struct{}, start int) <-chan int {
    out := make(chan int)
    go func() {
        defer close(out)
        for i := start; ; i++ {
            select {
            case out <- i:
            case <-cancel:
                return
            }
        }
    }()
    return out
}
```

Функция использует канал отмены. Переделайте на контекст:

```
func main() {
    ctx, cancel := context.WithCancel(context.Background())
    defer cancel()

    generated := generate(ctx, 11)
    for num := range generated {
        fmt.Print(num, " ")
        if num > 14 {
            break
        }
    }
}
```

```
    fmt.Println()
}
```

11 12 13 14 15

Полный код — в песочнице. Отправляйте в качестве решения только фрагмент, отмеченный комментариями «начало решения» и «конец решения».

[песочница](#)

Sample Input:

Sample Output:

PASS

```
package main

import (
    "context"
    "fmt"
)

// начало решения

// генерит целые числа от start и до бесконечности
func generate(ctx context.Context, start int) <-chan int {
    out := make(chan int)

    go func() {
        defer close(out)
        for i := start; ; i++ {
            select {
            case out <- i:
            case <-ctx.Done():           // defer cancel() напишет в
ctx.Done()
                return
            }
        }
    }()

    return out
}

// конец решения
```



```
func main() {
    ctx, cancel := context.WithCancel(context.Background())
    defer cancel()

    generated := generate(ctx, 11)
    for num := range generated {
        fmt.Print(num, " ")
        if num > 14 {
            break
        }
    }
    fmt.Println()
}
```

2

Отмена операции через контекст

Основное назначение контекста в Go — отмена операций.

Повторим через контекст то, что мы только что сделали через канал отмены. Функция `execute()` будет принимать контекст `ctx` вместо канала `cancel`:

```
// выполняет функцию fn с учетом контекста ctx
func execute(ctx context.Context, fn func() int) (int, error) {
    ch := make(chan int, 1)

    go func() {
        ch <- fn()
    }()

    select {
    case res := <-ch:
        return res, nil
    case <-ctx.Done(): // (1)
        return 0, ctx.Err() // (2)
    }
}
```

Код почти не изменился:

- вместо канала `cancel` сигнал об отмене может прийти из канала `ctx.Done()` ❶
- вместо ручного создания ошибки при отмене возвращаем `ctx.Err()` ❷

Клиент тоже несколько меняется:

```
func main() {
    // ...

    // работает в течение 100 мс
    work := func() int {
        // ...
    }

    // ждет 50 мс, после этого
    // с вероятностью 50% отменяет работу
    maybeCancel := func(cancel func()) {
        time.Sleep(50 * time.Millisecond)
        if rand.Float32() < 0.5 {
            cancel()
        }
    }

    ctx := context.Background()           // (1)
    ctx, cancel := context.WithCancel(ctx) // (2)
    defer cancel()                         // (3)

    go maybeCancel(cancel)                 // (4)

    res, err := execute(ctx, work)        // (5)
    fmt.Println(res, err)
}
```

Вот что здесь происходит:

- ❶ через `context.Background()` создаем пустой контекст;
- ❷ через `context.WithCancel()` на базе пустого контекста создаем новый, с возможностью ручной отмены;
- ❸ планируем отложенную отмену контекста при выходе из `main()`;
- ❹ отменяем контекст с 50% вероятностью;
- ❺ передаем контекст в функцию `execute()`.

`context.WithCancel()` возвращает сам контекст и функцию `cancel` для его отмены. Вызов `cancel()` освобождает занятые контекстом ресурсы и закрывает канал `ctx.Done()` — этот эффект мы и используем для прерывания `execute()`. Если контекст отменен, то `ctx.Err()` возвращает ошибку с причиной отмены (`context.Canceled` в нашем случае).

Все вместе работает точно так же, как предыдущая версия с каналом отмены:

```
work done
```

```
42 <nil>
```

```
0 context canceled
```

```
0 context canceled
```

```
work done
```

```
42 <nil>
```

Пара нюансов, которых не было с каналом отмены:

Контекст — это матрешка. Объект контекста неизменяемый. Чтобы добавить контексту новые свойства, создают новый контекст («дочерний») на основе старого («родительского»). Поэтому мы сначала создали пустой контекст, а затем новый (с возможностью отмены) на его основе:

```
// родительский контекст
ctx := context.Background()
```

```
// дочерний
ctx, cancel := context.WithCancel(ctx)
```

Если отменить родительский контекст — отменятся и все дочерние (но не наоборот).

```
// родительский контекст
parentCtx, parentCancel := context.WithCancel(context.Background())
```

```
// дочерний контекст
childCtx, childCancel := context.WithCancel(parentCtx)
```

```
// parentCancel() отменит parentCtx и childCtx
// childCancel() отменит только childCtx
```

Многократная отмена безопасна. Если два раза вызвать `close()` на канале, получим панику. А вот вызывать `cancel()` контекста можно сколько угодно. Первая отмена сработает, а остальные будут проигнорированы. Это удобно, потому что можно сразу после создания контекста запланировать отложенный `cancel()`, плюс явно отменить контекст при необходимости (как мы сделали в функции `maybeCancel`). С каналом бы так не получилось.

[песочница](#)

Стандартная библиотека 1. Текст

На этом уроке:

- операции над строками;
- преобразование типа;
- немного юникода;
- построитель строк;
- регулярные выражения;
- шаблонизатор;
- соревнование!

Операции над строками

Пакет `strings` сосредоточил великое множество функций работы со строками, которые наверняка знакомы вам по другим языкам.

Поиск по строке

```
s := "go is awesome"

fmt.Println(strings.Contains(s, "go"))
// true

fmt.Println(strings.HasPrefix(s, "go"))
// true

fmt.Println(strings.HasSuffix(s, "some"))
// true

fmt.Println(strings.Index(s, "is"))
// 3

fmt.Println(strings.LastIndex("go no go", "go"))
// 6

fmt.Println(strings.Count("go no go", "go"))
// 2
```

Замена

```
s := "go go go"

fmt.Println(strings.ReplaceAll(s, "go", "no"))
```

```
// no no no

fmt.Println(strings.Replace(s, "go", "no", 2))
// no no go
```

`ReplaceAll()` заменяет все вхождения искомой подстроки, а `Replace()` — только указанное количество.

Разделить / соединить

```
fmt.Println(strings.Split("go is awesome", " "))
// [go is awesome]

fmt.Println(strings.Fields("go is awesome"))
// [go is awesome]

before, after, found := strings.Cut("go is awesome", " is ")
fmt.Println(before, after, found)
// go awesome true

fmt.Println(strings.Join([]string{"go", "is", "awesome"}, " "))
// go is awesome
```

`Split()` бьет по указанному разделителю, а `Fields()` — только по последовательностям пробельных символов.

`Cut()` делит строку на две части — до и после указанной подстроки.

Обрезка

```
s := "--=go is awesome=--"

fmt.Println(strings.Trim(s, "-="))
// go is awesome

fmt.Println(strings.TrimLeft(s, "-="))
// go is awesome=--

fmt.Println(strings.TrimRight(s, "-="))
// --=go is awesome

fmt.Println(strings.TrimSpace(" go is awesome \n"))
// go is awesome
```

`Trim()`, `TrimLeft()` и `TrimRight()` вырезают не указанную подстроку, а любые входящие в нее символы.

Преобразование регистра

```
fmt.Println(strings.ToUpper("go is awesome"))  
// GO IS AWESOME  
  
fmt.Println(strings.ToLower("GO IS AWESOME"))  
// go is awesome
```

Повтор

```
fmt.Println(strings.Repeat("go", 3))  
// gogogo
```

[песочница](#)

2

Безопасный заголовок

Допустим, вы работаете в новостном издании и хотите генерить человекочитаемые фрагменты URL статей на основе заголовков:

```
// slugify возвращает "безопасный" вариант заголовка:  
// только латиница, цифры и дефис  
func slugify(src string) string {  
    // ...  
}  
  
func main() {  
    phrase := "Go Is Awesome!"  
    fmt.Println(slugify(phrase))  
    // go-is-awesome  
  
    phrase = "Tabs are all we've got"  
    fmt.Println(slugify(phrase))  
    // tabs-are-all-we-ve-got  
}
```

Требования:

- допустимыми символами исходной строки считаются латинские буквы `a-z` и `A-Z`, цифры `0-9` и дефис `-`
- последовательности допустимых символов образуют «слова»;
- в результирующую строку должны попасть слова, объединенные через дефис;
- при этом буквы `A-Z` должны быть приведены к нижнему регистру;

- других символов в результирующей строке быть не должно.

Реализуйте функцию `slugify()`.

Полный код — в песочнице. Отправляйте в качестве решения только фрагмент, отмеченный комментариями «начало решения» и «конец решения».

P.S. Задачу можно решить только с использованием функций из предыдущего шага. Но, возможно, другие функции пакета `[strings](https://pkg.go.dev/strings)` тоже смогут вас заинтересовать.

[песочница](#)

Sample Input:

Sample Output:

PASS

```
package main

// go test ./text_slugify_test.go -v

import (
    // "fmt"
    "strings"
    "testing"
)

// начало решения

// НИКОГДА не использовать сплит, только филд! сплит "чето там" ->
// ["чето", "", "", "", "там"]

// slugify возвращает "безопасный" вариант заголовка:
// только латиница, цифры и дефис
/*func slugify(src string) string {
    var runes []byte
    for i, r := range src {
        if (r >= 'a' && r <= 'z') || (r >= 'A' && r <= 'Z') || (r >= '0' &&
r <= '9') || (r == ' ') || (r == '-') {
            runes = append(runes, src[i])
        } else {
            runes = append(runes, ' ')
        }
    }
}
```

```

    }
    result := string(runes)
    fmt.Printf("%s\n", result)

    var arr []string
    for _, s := range strings.Split(result, " ") {
        if s != "" {
            arr = append(arr, s)
        }
    }

    fmt.Printf("%s\n", arr)
    result = strings.Join(arr, "-")

    result = strings.ToLower(result)
    return string(result)

}*/

// конец решения

// кратко
/*func slugify(src string) string {
    src = strings.ToLower(src)
    words := strings.FieldsFunc(src, func(r rune) bool {
        return (r < 'a' || r > 'z') && (r < 'A' || r > 'Z') && (r < '0' || r
> '9') && r != '-'
    })
    return strings.Join(words, "-")
}*/

// средне
func slugify(src string) string {
    res := strings.ToLower(src)
    res = strings.Map(purifyChar, res)
    words := strings.Fields(res)
    return strings.Join(words, "-")
}

// purifyChar преобразует недопустимые символы в пробелы
func purifyChar(r rune) rune {
    const validChars string = "abcdefghijklmnopqrstuvwxyz01234567890- "
    if strings.IndexRune(validChars, r) == -1 {

```



```

        return ' '
    }
    return r
}

func Test(t *testing.T) {
    const phrase = "Go 1.18 is released!"
    const want = "go-1-18-is-released"
    got := slugify(phrase)

    if got != want {
        t.Errorf("%s: got %#v, want %#v", phrase, got, want)
    }
}

func main() {
    Test(&testing.T{})
}

```

3

Преобразование типа

Пакет `strconv` преобразует строки в примитивные типы и обратно.

`string` ⇔ `int`

```

n, err := strconv.Atoi("42")
fmt.Printf("%T %v %v\n", n, n, err)
// int 42 <nil>

```

```

s := strconv.Itoa(42)
fmt.Printf("%T %v\n", s, s)
// string 42

```

`string` ⇔ `float`

```

f, err := strconv.ParseFloat("12.34", 64)
fmt.Printf("%T %v %v\n", f, f, err)
// float64 12.34 <nil>

```

```

s := strconv.FormatFloat(12.34, 'g', -1, 64)
fmt.Printf("%T %v\n", s, s)
// string 12.34

```

`ParseFloat()` вторым параметром принимает разрядность числа (32 / 64).

`FormatFloat()` принимает три дополнительных параметра:

- формат строки (`f` — десятичная дробь, `e` — с экспонентой, `g` — выбрать автоматически);
- количество цифр после разделителя (-1 — автоматически);
- разрядность числа (32 / 64).

string ⇔ bool

```
b, err := strconv.ParseBool("true")
fmt.Printf("%T %v %v\n", b, b, err)
// bool true <nil>
```

```
s := strconv.FormatBool(false)
fmt.Printf("%T %v\n", s, s)
// string false
```

quote / unquote

```
s := strconv.Quote("go лучше всех")
fmt.Println(s)
// "go лучше всех"
```

```
s = strconv.QuoteToASCII("go лучше всех")
fmt.Println(s)
// "go \u0043\u0044\u0047\u0048\u0035 \u0032\u0041\u0035\u0045"
```

```
s, err := strconv.Unquote(`"go \u0043\u0044\u0047\u0048\u0035
\u0032\u0041\u0035\u0045"`)
fmt.Println(s, err)
// go лучше всех <nil>
```

`Quote()` принимает строку, а возвращает ее же в кавычках. `QuoteToASCII()` дополнительно заменяет все не-однобайтные символы на escape-последовательности `\uxxxx`. `Unquote()` превращает escape-последовательности обратно в символы и возвращает строку без кавычек.

[песочница](#)

4

Юникод

Пакет `unicode` проверяет, относится ли символ к одному из распространенных классов.

```
// цифра
fmt.Println(unicode.IsDigit('9'))
// true
```

```
// буква
fmt.Println(unicode.IsLetter('ы'))
// true

// знак пунктуации
fmt.Println(unicode.IsPunct('!'))
// true

// пробельный символ
fmt.Println(unicode.IsSpace(' '))
// true

// в нижнем регистре
fmt.Println(unicode.IsLower('ы'))
// true

// в верхнем регистре
fmt.Println(unicode.IsUpper('Ы'))
// true
```

Кроме того, есть пара полезных функций в пакете `unicode/utf8`:

```
// количество символов в строке
fmt.Println(utf8.RuneCountInString("go лучше всех"))
// 13

// первая руна в строке и ее размер в байтах
char, size := utf8.DecodeRuneInString("го")
fmt.Println(char, size)
// 1075 2
```

[песочница](#)

5

Длина маршрута

Есть навигатор, который выдает последовательность команд:

```
100m to intersection
turn right
straight 300m
enter motorway
straight 5km
```

```
exit motorway
500m straight
turn sharp left
continue 100m to destination
```

Реализуйте функцию `calcDistance()`, которая получает срез таких команд, а возвращает общую длину маршрута в метрах:

```
func calcDistance(directions []string) int {
    // ...
}
```

Для маршрута выше получится `6000`.

Особенности:

- в отдельной команде расстояние фигурирует не более одного раза;
- расстояние всегда имеет формат `Nm` либо `Xkm`;
- `N` — целое число, `X` — либо целое, либо десятичная дробь;
- точка используется только в качестве десятичного разделителя;
- других чисел, помимо расстояний, в командах не встречается (никаких `3rd exit` и тому подобного).

Полный код — в песочнице. Отправляйте в качестве решения только фрагмент, отмеченный комментариями «начало решения» и «конец решения».

[песочница](#)

Sample Input:

Sample Output:

PASS

```
package main
```

```
import (
    "strconv"
    "strings"
    "testing"
)
```

```
// начало решения
```

```
// calcDistance возвращает общую длину маршрута в метрах
```

```

func calcDistance(directions []string) int {
    distance := 0
    for _, dir := range directions {
        word := getDigits(dir)
        distance = distance + parseDigits(word)
    }
    return distance
}

func getDigits(str string) string {
    for _, s := range strings.Fields(str) {
        if strings.ContainsAny(s, "0123456789") {
            return s
        }
    }
    return ""
}

func parseDigits(str string) int {
    if strings.HasSuffix(str, "km") {
        str = strings.Split(str, "km")[0]
        f, _ := strconv.ParseFloat(str, 64)
        return int(1000 * f)
    } else {
        str = strings.Split(str, "m")[0]
        i, _ := strconv.Atoi(str)
        return i
    }
}

// конец решения

/* Вариант
// calcDistance возвращает общую длину маршрута в метрах
func calcDistance(directions []string) int {
    total := 0
    for _, dir := range directions {
        total += extractDistance(dir)
    }
    return total
}

```

```

// extractDistance извлекает из строки расстояние
// в метрах
func extractDistance(s string) int {
    for _, word := range strings.Fields(s) {
        char, _ := utf8.DecodeRuneInString(word)
        if !unicode.IsDigit(char) {
            continue
        }
        if strings.HasSuffix(word, "km") {
            return parseDistance(word[:len(word)-2], 1000)
        }
        return parseDistance(word[:len(word)-1], 1)
    }
    return 0
}

// parseDistance преобразует строковое расстояние
// в целое число с учетом мультипликатора
func parseDistance(distance string, multiplier int) int {
    num, _ := strconv.ParseFloat(distance, 64)
    return int(num * float64(multiplier))
}*/

func Test(t *testing.T) {
    directions := []string{
        "100m to intersection",
        "turn right",
        "straight 300m",
        "enter motorway",
        "straight 5.12km",
        "exit motorway",
        "500m straight",
        "turn sharp left",
        "continue 100m to destination",
    }
    const want = 6120
    got := calcDistance(directions)
    if got != want {
        t.Errorf("%v: got %v, want %v", directions, got, want)
    }
}

```

```
func main() {  
    Test(&testing.T{})  
}
```

6

Строитель строк

Допустим, мы решили написать функцию `formatList()`, которая форматирует срез строк в виде нумерованного списка:

```
func main() {  
    list := []string{  
        "go is awesome",  
        "cats are cute",  
        "rain is wet",  
    }  
    s := formatList(list)  
    fmt.Print(s)  
}
```

- 1) go is awesome
- 2) cats are cute
- 3) rain is wet

Как бы это сделать?

Первое, что приходит в голову — для каждого элемента среза отформатировать строку и сложить такие строки все вместе:

```
func formatList1(items []string) string {  
    var str string  
    for idx, item := range items {  
        str += fmt.Sprintf("%d) %s\n", idx+1, item)  
    }  
    return str  
}
```

Наверно, с точки зрения использования памяти это не слишком эффективно. Строки в Go неизменны, так что на каждой итерации цикла придется выделять память под новую, увеличенную строку. Давайте переделаем на срез строк в сочетании со `strings.Join()`:

```
func formatList2(items []string) string {  
    strs := make([]string, len(items))  
    for idx, item := range items {
```

```

    strs[idx] = fmt.Sprintf("%d) %s", idx+1, item)
}
return strings.Join(strs, "\n")
}

```

Выполним сравнительный бенчмарк (тут я взял список побольше, из 10 пунктов):

Benchmark_formatList1-8	976683	1188 ns/op
1456 B/op	29 allocs/op	
Benchmark_formatList2-8	1235349	972 ns/op
712 B/op	22 allocs/op	

Действительно, срез расходует в 2 раза меньше памяти и выполняется на 20% быстрее. Но из-за `fmt.Sprintf()` на каждой итерации память выделяется 22 раза — кажется, многовато.

В пакете `strings` есть специальный тип для эффективного сбора длинных строк — `Builder`. Он минимизирует копирование памяти. Смотрите:

```

func formatList3(items []string) string {
    var b strings.Builder
    for idx, item := range items {
        b.WriteString(strconv.Itoa(idx + 1))
        b.WriteString(" ")
        b.WriteString(item)
        b.WriteRune('\n')
    }
    return b.String()
}

```

Benchmark_formatList1-8	976683	1188 ns/op
1456 B/op	29 allocs/op	
Benchmark_formatList2-8	1235349	972 ns/op
712 B/op	22 allocs/op	
Benchmark_formatList3-8	5599970	216 ns/op
504 B/op	6 allocs/op	

В 4.5 раза быстрее среза, а расходует на 30% меньше памяти. И всего 6 аллокаций памяти!

Когда собираете строку из большого числа кусочков — используйте `strings.Builder`.

[песочница](#)

По умолчанию Go печатает карты так:

```
m := map[string]int{"one": 1, "two": 2, "three": 3}
fmt.Println(m)
// map[one:1 three:3 two:2]
```

Реализуйте функцию `prettyfy()`, которая возвращает красивое строковое представление карты:

```
m := map[string]int{"one": 1, "two": 2, "three": 3}
fmt.Println(prettyfy(m))
```

```
{
    one: 1,
    three: 3,
    two: 2,
}
```

Для карт с двумя и более записями:

- фигурные скобки в начале и в конце;
- каждый пункт на отдельной строке с отступом в 4 пробела;
- порядок пунктов — по алфавиту;
- без кавычек у строк.

Для карт с одной записью или вовсе без записей функция должна вернуть однострочник:

```
{ one: 1 }
{ }
```

Полный код — в песочнице. Отправляйте в качестве решения только фрагмент, отмеченный комментариями «начало решения» и «конец решения».

Подсказка

С сортировкой поможет `[sort.Strings](https://pkg.go.dev/sort#Strings)`.

[песочница](#)

Sample Input:

Sample Output:

PASS

```
package main
```

```

import (
    "sort"
    "strconv"
    "strings"
    "testing"
)

// начало решения

// prettify возвращает отформатированное
// строковое представление карты
func prettify(m map[string]int) string {
    if len(m) == 0 {
        return "{}"
    }

    prefix := "{\n"
    suffix := "}"
    if len(m) == 1 {
        prefix = "{ "
        suffix = " }"
    }

    var b strings.Builder

    keys := make([]string, 0, len(m))
    for k := range m {
        keys = append(keys, k)
    }
    sort.Strings(keys)

    for _, key := range keys {
        if len(m) > 1 {
            b.WriteString("    ")
        }
        b.WriteString(key)
        b.WriteString(": ")
        b.WriteString(strconv.Itoa(m[key]))
        if len(m) > 1 {
            b.WriteString(",\n")
        }
    }
    return prefix + b.String() + suffix
}

```

```

}

// конец решения

/*
// prettify возвращает отформатированное
// строковое представление карты
func prettify(m map[string]int) string {
    if len(m) == 0 {
        return "{}"
    }
    if len(m) == 1 {
        for key, val := range m {
            return fmt.Sprintf("{ %v: %v }", key, val)
        }
    }
    keys := extractKeys(m)
    return asOrdered(keys, m)
}

// extractKeys возвращает упорядоченные
// по алфавиту ключи карты
func extractKeys(m map[string]int) []string {
    keys := make([]string, 0, len(m))
    for key := range m {
        keys = append(keys, key)
    }
    sort.Strings(keys)
    return keys
}

// asOrdered форматирует карту,
// выводя ключи в указанном порядке
func asOrdered(keys []string, m map[string]int) string {
    var b strings.Builder
    b.WriteString("{\n")
    for _, key := range keys {
        b.WriteString("    ")
        b.WriteString(key)
        b.WriteString(": ")
        b.WriteString(strconv.Itoa(m[key]))
        b.WriteString(",\n")
    }
}

```

```

    b.WriteString("{}")
    return b.String()
}*/

func Test(t *testing.T) {
    m := map[string]int{"one": 1, "two": 2, "three": 3}
    const want = "{\n    one: 1,\n    three: 3,\n    two: 2,\n}"
    got := prettify(m)
    if got != want {
        t.Errorf("%v\ngot:\n%v\n\nwant:\n%v", m, got, want)
    }
}

func main() {
    Test(&testing.T{})
}

```

8

Регулярные выражения

[Регулярные выражения](#) — это специальный язык для поиска и замены в тексте по шаблону. В Go за регулярные выражения отвечает пакет `regexp`.

Синтаксис «регулярок» в Go не такой богатый, как в Python или Java. Но зато гарантируется, что поиск обрабатывает за линейное время $O(n)$ — этим другие языки обычно похвастать не могут.

Компиляция

Перед использованием шаблон регулярного выражения следует скомпилировать. Благодаря этому дальнейшие методы поиска и замены будут выполняться быстро, и не будут парсить каждый раз шаблон заново.

```

// одна или более цифр
re := regexp.MustCompile(`\d+`)

```

Для шаблона обычно используют *сырые строки* (raw strings) в ``таких кавычках`` вместо `"двойных кавычек"`, чтобы не экранировать в шаблоне обратный слеш `\` и сами кавычки `"`.

Поиск по шаблону

```

// одна или более цифр
re := regexp.MustCompile(`\d+`)
s := "2050-11-05 is November 5th, 2050"

// подходит ли строка под шаблон?

```

```
ok := re.MatchString(s)
fmt.Println(ok)
// true

// первое совпадение с шаблоном
first := re.FindString(s)
fmt.Println(first)
// 2050

// индекс начала и окончания
// первого совпадения
idx := re.FindStringIndex(s)
fmt.Println(idx)
// [0 4]

// N совпадений с шаблоном (в данном случае 3)
// если указать -1 - вернет все совпадения
three := re.FindAllString(s, 3)
fmt.Println(three)
// [2050 11 05]

// индексы совпадений
indices := re.FindAllStringIndex(s, 3)
fmt.Println(indices)
// [[0 4] [5 7] [8 10]]
```

Группы

```
// сам шаблон описывает дату в формате дddd-мм-гг
// а три группы соответствуют году, месяцу и дню
re := regexp.MustCompile(`(\d\d\d\d)-(\d\d)-(\d\d)`)
s := "2050-11-05 is November 5th, 2050"

// совпадение со всем шаблоном
match := re.FindString(s)
fmt.Println(match)
// 2050-11-05

// совпадения с группами шаблона
groups := re.FindStringSubmatch(s)
fmt.Println(groups)
// [2050-11-05 2050 11 05]

// индексы групп
```

```
indices := re.FindStringSubmatchIndex(s)
fmt.Println(indices)
// [0 10 0 4 5 7 8 10]
```

Разбивка на части

```
// разделителем считаются цифры, окруженные пробелами
re := regexp.MustCompile(`\s*\d+\s*`)
s := "one 01 two 02 three 03"

parts := re.Split(s, -1)
fmt.Printf("%#v\n", parts)
// []string{"one", "two", "three", ""}
```

Замена

```
re := regexp.MustCompile(`(\d\d\d\d)-(\d\d)-(\d\d)`)
src := "2050-11-05 is November 5th, 2050"

// замена по шаблону
res := re.ReplaceAllString(src, "$3.$2.$1")
fmt.Println(res)
// 05.11.2050 is November 5th, 2050

// замена функцией
fn := func(src string) string {
    parts := strings.Split(src, "-")
    reversed := []string{parts[2], parts[1], parts[0]}
    return strings.Join(reversed, ".")
}
res = re.ReplaceAllStringFunc(src, fn)
fmt.Println(res)
// 05.11.2050 is November 5th, 2050
```

[песочница](#)

9

Безопасный заголовок на регулярках

Пишем функцию, которая преобразует заголовок (как в первой задаче урока):

```
// slugify возвращает "безопасный" вариант заголовка:
// только латиница, цифры и дефис
func slugify(src string) string {
    // ...
```

```
}

func main() {
    phrase := "Go Is Awesome!"
    fmt.Println(slugify(phrase))
    // go-is-awesome

    phrase = "Tabs are all we've got"
    fmt.Println(slugify(phrase))
    // tabs-are-all-we-ve-got
}
```

Требования:

- допустимыми символами исходной строки считаются латинские буквы `a-z` и `A-Z`, цифры `0-9` и дефис `-`
- последовательности допустимых символов образуют «слова»;
- в результирующую строку должны попасть слова, объединенные через дефис;
- при этом буквы `A-Z` должны быть приведены к нижнему регистру;
- других символов в результирующей строке быть не должно.

Реализуйте функцию `slugify()` с использованием регулярных выражений.

Полный код — в песочнице. Отправляйте в качестве решения только фрагмент, отмеченный комментариями «начало решения» и «конец решения».

[песочница](#)

Sample Input:

Sample Output:

PASS

```
package main

// go test ./text_slugify_test.go -v

import (
    // "fmt"
    "regexp"
    "strings"
    "testing"
)
```

```

/*
func slugify(src string) string {
    res := strings.ToLower(src)

    sep := regexp.MustCompile(`[^0-9a-z\ -]+`) // 1+ НЕ разрешенных
символов (включая пробел)
    res = sep.ReplaceAllString(res, "-")
    res = strings.Trim(res, "-") // в
начале и конце могут остаться хвосты
    return res
}
*/

// последовательность допустимых символов
var wordRE = regexp.MustCompile(`[a-z0-9\ -]+`)

// slugify возвращает "безопасный" вариант заголовка:
// только латиница, цифры и дефис
func slugify(src string) string {
    words := wordRE.FindAllString(strings.ToLower(src), -1)
    return strings.Join(words, "-")
}

func Test(t *testing.T) {
    const phrase = "Go: 1.18 - is released!!"
    const want = "go-1-18---is-released"
    got := slugify(phrase)

    if got != want {
        t.Errorf("%s: got %#v, want %#v", phrase, got, want)
    }
}

func main() {
    Test(&testing.T{})
}

```

10

О регулярках

Программисты, знакомые с регулярными выражениями, часто начинают использовать их повсюду. Это не лучший путь:

- сложные регулярки тяжело поддерживать;
- регулярки медленные.

Вот две реализации функции `slugify()`. Первая без регулярных выражений, вторая с ними:

```
func slugify(src string) string {
    // 1) привести строку к нижнему регистру
    // 2) заменить недопустимые символы на пробелы БЕЗ регулярок
    // 3) разбить строку по пробелам
    // 4) объединить через дефис
}

func slugifyRegexp(src string) string {
    // 1) привести строку к нижнему регистру
    // 2) заменить недопустимые символы на пробелы через регулярку
    // 3) разбить строку по пробелам
    // 4) объединить через дефис
}
```

(код не привожу, чтобы не спойлерить решения задач)

Отличие только в строке ②, остальные шаги одинаковые.

А вот бенчмарки для этих функций:

Benchmark_slugify-8	1834732	640.4 ns/op
320 B/op	4 allocs/op	
Benchmark_slugifyRegexp-8	566186	2053 ns/op
458 B/op	9 allocs/op	

`slugifyRegexp()` хуже по всем показателям: в 3 раза медленнее, в 1.5 раза больше памяти, в 2 раза больше аллокаций.

Поэтому имеет смысл использовать регулярки только там, где выгода от них заметно перевешивает недостатки.

11

Шаблонизатор

Шаблонизатор — это штука, которая форматирует данные в соответствии с шаблоном. В Go их аж два — `text/template` и `html/template`.

Простейший шаблон подставляет в текст значение переменной:

```
const txt = "Алиса: - {{.}}\n"

tpl := template.New("value")
tpl = template.Must(tpl.Parse(txt))

tpl.Execute(os.Stdout, "Привет!")
tpl.Execute(os.Stdout, "Как дела?")
tpl.Execute(os.Stdout, "Пока!")
```

```
Алиса: - Привет!
Алиса: - Как дела?
Алиса: - Пока!
```

Это выглядит избыточным и не сильно отличается от обычного `fmt.Printf()`. Интереснее становится, если выводить в шаблоне поля структуры и использовать условную логику:

```
const txt = `Сейчас {{.Time}}, {{.Day}}.
{{if .Sunny -}} Солнечно! {{- else -}} Пасмурно :-/ {{- end}}
`

tpl := template.New("greeting")
tpl = template.Must(tpl.Parse(txt))

type State struct {
    Time  string
    Day   string
    Sunny bool
}

state := State{"9:00", "четверг", true}
tpl.Execute(os.Stdout, state)

fmt.Println()

state = State{"21:00", "пятница", false}
tpl.Execute(os.Stdout, state)
```

```
Сейчас 9:00, четверг.
Солнечно!
```

```
Сейчас 21:00, пятница.
Пасмурно :-/
```

Или форматировать списки:

```
const txt = "{{range .}}- {{.}}\n{{end}}"

tpl := template.New("list")
tpl = template.Must(tpl.Parse(txt))

list := []string{"Купить молоко", "Погладить кота", "Вынести мусор"}
tpl.Execute(os.Stdout, list)
```

```
- Купить молоко
- Погладить кота
- Вынести мусор
```

У шаблонов навороченный синтаксис, так что если они вам понадобятся — загляните в [документацию](#).

[песочница](#)

12

Сообщение о балансе

В некоторой системе хранится информация о балансе лицевого счета каждого пользователя. Система хочет отправлять пользователям сообщения вида:

Алиса, добрый день! Ваш баланс - 1000₽. Все в порядке.

В сообщении указывается имя пользователя и баланс. Кроме того, последнее предложение меняется в зависимости от баланса:

- баланс ≥ 100 ₽:
Алиса, добрый день! Ваш баланс - 1234₽. Все в порядке.
- баланс больше 0₽, но меньше 100₽:
Алиса, добрый день! Ваш баланс - 77₽. Пора пополнить.
- баланс равен 0:
Алиса, добрый день! Ваш баланс - 0₽. Доступ заблокирован.

Составьте текст шаблона, который позволит отправлять такие сообщения.

```
var templateText = `...`

type User struct {
    Name    string
    Balance int
}
```

```
func main() {
    tpl := template.New("message")
    tpl = template.Must(tpl.Parse(templateText))
    user := User{"Алиса", 500}
    tpl.Execute(os.Stdout, user)
}
```

Полный код — в песочнице. Отправляйте в качестве решения только фрагмент, отмеченный комментариями «начало решения» и «конец решения».

[песочница](#)

Sample Input:

Sample Output:

PASS

13

Соревнование!

Пишем функцию, которая преобразует заголовок (как уже делали дважды на предыдущих шагах):

```
func slugify(src string) string {
    // ...
}

func main() {
    const phrase = "A 100x Investment (2019)"
    slug := slugify(phrase)
    fmt.Println(slug)
    // a-100x-investment-2019
}
```

[песочница](#)

Требования:

- допустимыми символами исходной строки считаются латинские буквы `a-z` и `A-Z`, цифры `0-9` и дефис `-`
- последовательности допустимых символов образуют «слова»;
- в результирующую строку должны попасть слова, объединенные через дефис;
- при этом буквы `A-Z` должны быть приведены к нижнему регистру;

- других символов в результирующей строке быть не должно.

Кешировать ответы и использовать многозадачность нельзя.

Правила

В соревновании побеждает тот, чья функция `slugify()` работает быстрее всех и потребляет меньше всего памяти.

1. Есть решение-чемпион.
2. Вы отправляете свое решение, оно играет против чемпиона.
3. Если ваше побеждает — оно становится новым чемпионом, а вы проходите задание.
4. (новый чемпион устанавливается вручную, так что проходит какое-то время)
5. Другие игроки играют уже против нового чемпиона.

Чтобы победить чемпиона, решение должно быть лучше хотя бы на 5% по времени работы или использованию памяти, и не уступать по остальным показателям. Чемпион очень сильный, поэтому чтобы пройти задание, достаточно сыграть с ним вничью или даже немного уступить.

Чтобы получить «зачет», достаточно использовать только те инструменты, которые мы разбирали на курсе, без `unsafe` и прочей магии. Призовое место вы так не получите, но задание пройдете.

У вас есть 5 попыток. Если за 5 попыток пройти задание не удастся, то все равно откроется раздел с решениями.

Отправка решения

Полный код — в песочнице. Отправляйте в качестве решения только фрагмент, отмеченный комментариями «начало решения» и «конец решения».

К сожалению, механизм проверки заданий у «Степика» медленный и нестабильный, так что может давать очень разные результаты для одного и того же кода. Именно поэтому у вас есть пять попыток, а не одна. Отправляйте код, только когда уверены, что он хорош. До этого пользуйтесь кнопкой «Запустить код».

Поскольку «Степик» ненадежен, результаты победителей я буду перепроверять вручную.

В проверочном коде импортированы основные пакеты, но если вам каких-то не хватает — напишите в комментариях, добавлю.

Действующий чемпион

[Aleksandr Panfilov](#)

Sample Input:

Sample Output:

PASS

```
package main

import (
    "fmt"
    "regexp"
    "strings"
    "testing"
    "unsafe"
)

// начало решения

var sepRE = regexp.MustCompile(`^[^0-9a-z\-\-]+`) // 1+ НЕ разрешенных символов
(включая пробел)

func slugify1(src string) string {
    res := strings.ToLower(src)
    res = sepRE.ReplaceAllString(res, "-")
    res = strings.Trim(res, "-") // в начале и конце могут остаться хвосты
    return res
}

var wordRE = regexp.MustCompile(`[a-z0-9\-\-]+`)

func slugify2(src string) string {
    words := wordRE.FindAllString(strings.ToLower(src), -1)
    return strings.Join(words, "-")
}

// средне
func slugify3(src string) string {
    res := strings.ToLower(src)
    res = strings.Map(purifyChar, res)
    words := strings.Fields(res)
    return strings.Join(words, "-")
}

// purifyChar преобразует недопустимые символы в пробелы
func purifyChar(r rune) rune {
```

```

const validChars string = "abcdefghijklmnopqrstuvwxyz01234567890- "
if strings.IndexRune(validChars, r) == -1 {
    return ' '
}
return r
}

// длинно
func slugify4(src string) string {
    var runes []byte
    for i, r := range src {
        if (r >= 'a' && r <= 'z') || (r >= 'A' && r <= 'Z') || (r >= '0' &&
r <= '9') || (r == ' ') || (r == '-') {
            runes = append(runes, src[i])
        } else {
            runes = append(runes, ' ')
        }
    }
    result := string(runes)

    var arr []string
    for _, s := range strings.Split(result, " ") {
        if s != "" {
            arr = append(arr, s)
        }
    }

    result = strings.Join(arr, "-")
    result = strings.ToLower(result)
    return string(result)
}

// длинно
func slugify5(src string) string {
    runes := make([]byte, len(src))
    prev_bad := true

    ptr := 0
    for _, r := range src {
        if (r >= 'a' && r <= 'z') || (r >= 'A' && r <= 'Z') || (r >= '0' &&
r <= '9') || (r == '-') {
            if r >= 'A' && r <= 'Z' {

```

```

        r += 32 // ToLower
    }
    runes[ptr] = byte(r)
    prev_bad = false
    ptr++
} else {
    if prev_bad {
        continue
    }
    runes[ptr] = byte('-')
    prev_bad = true
    ptr++
}
}

if ptr > 1 && prev_bad && runes[ptr-1] == '-' {
    return string(runes[:ptr-1])
} else if ptr > 0 {
    return string(runes[:ptr])
} else {
    return ""
}
}

// unsafe
func BytesToString(b []byte) string {
    return *(*string)(unsafe.Pointer(&b))
}

func StringToBytes(s string) []byte {
    return *(*[]byte)(unsafe.Pointer(&s))
}

func slugify6(src string) string {
    src_bytes := *(*[]byte)(unsafe.Pointer(&src))
    runes := make([]byte, len(src))
    prev_bad := true
    ptr := 0
    for _, r := range src_bytes {
        // a-z 97-122 A-Z 65-90 0-9 48-57 - 45
        if r > 64 && r < 91 { // A-Z
            r += 32 // ToLower
        }
    }
}

```



```

        if (r == 45) || (r > 47 && r < 58) || (r > 96 && r < 123) {
            runes[ptr] = r
            prev_bad = false
            ptr++
        } else {
            if prev_bad {
                continue
            }
            runes[ptr] = 45
            prev_bad = true
            ptr++
        }
    }

    if ptr > 1 && prev_bad && runes[ptr-1] == 45 {
        runes = runes[:ptr-1]
    } else if ptr > 0 {
        runes = runes[:ptr]
    } else {
        runes = nil
    }

    return *(*string)(unsafe.Pointer(&runes))
}

const phrase = "? ? A 100x Investment (2019) ! Go 1.18 is released! Go - 1.18 is - released! !"

func BenchmarkMatchSlugify1(b *testing.B) {
    for n := 0; n < b.N; n++ {
        slugify1(phrase)
    }
}

func BenchmarkMatchSlugify2(b *testing.B) {
    for n := 0; n < b.N; n++ {
        slugify2(phrase)
    }
}

func BenchmarkMatchSlugify3(b *testing.B) {
    for n := 0; n < b.N; n++ {
        slugify3(phrase)
    }
}

```

```

    }
}

func BenchmarkMatchSlugify4(b *testing.B) {
    for n := 0; n < b.N; n++ {
        slugify4(phrase)
    }
}

func BenchmarkMatchSlugify5(b *testing.B) {
    for n := 0; n < b.N; n++ {
        slugify5(phrase)
    }
}

func BenchmarkMatchSlugify6(b *testing.B) {
    for n := 0; n < b.N; n++ {
        slugify6(phrase)
    }
}

func Test(t *testing.T) {
    // в го есть еще много таких же извращенских способов проверки равных
    // значений,
    // но нормального в стандартной библиотеке нет
    allEqual := len(map[string]bool{
        slugify1(phrase): true,
        slugify2(phrase): true,
        slugify3(phrase): true,
        slugify4(phrase): true,
        slugify5(phrase): true,
        slugify6(phrase): true,
    }) == 1

    fmt.Println(slugify1(phrase))
    fmt.Println(slugify2(phrase))
    fmt.Println(slugify3(phrase))
    fmt.Println(slugify4(phrase))
    fmt.Println(slugify5(phrase))
    fmt.Println(slugify6(phrase))

    fmt.Println(map[string]bool{
        slugify1(phrase): true,

```

```

        slugify2(phrase): true,
        slugify3(phrase): true,
        slugify4(phrase): true,
        slugify5(phrase): true,
        slugify6(phrase): true,
    })
    fmt.Printf("All equal: %v\n", allEqual)
    if !allEqual {
        t.Errorf("One or more variants work wrong")
    }
}

```

```

$ go test ./text_slugifyfast_test.go -bench=. -benchmem
All equal: true
goos: linux
goarch: amd64
cpu: Intel(R) Core(TM) i5-4440 CPU @ 3.10GHz
BenchmarkMatchSlugify1-4          184717          6239 ns/op
370 B/op          7 allocs/op
BenchmarkMatchSlugify2-4          182656          6643 ns/op
916 B/op          20 allocs/op
BenchmarkMatchSlugify3-4          461876          2533 ns/op
528 B/op          4 allocs/op
BenchmarkMatchSlugify4-4          488532          3304 ns/op
1512 B/op          14 allocs/op
BenchmarkMatchSlugify5-4          4063060          292.8 ns/op
176 B/op          2 allocs/op
PASS
ok      command-line-arguments  9.772s

```

Победитель с одной аллокацией и в 2 раза быстрее, ну и молодец.

Failed. Wrong answer

This is a sample **test** from the problem statement!

Test input:

Correct output:

PASS

Your code output:

contender: 292 ns 91 B 2 allocs

champion: 126 ns 46 B 1 allocs

Чемпион вас сокрушил 😊 Попробуйте еще раз

Добавили битардовства

BenchmarkMatchSlugify1-4	225655	4771 ns/op
370 B/op	7 allocs/op	
BenchmarkMatchSlugify2-4	214194	5034 ns/op
914 B/op	20 allocs/op	
BenchmarkMatchSlugify3-4	637558	1860 ns/op
528 B/op	4 allocs/op	
BenchmarkMatchSlugify4-4	558219	2129 ns/op
1512 B/op	14 allocs/op	
BenchmarkMatchSlugify5-4	4696406	247.3 ns/op
176 B/op	2 allocs/op	
BenchmarkMatchSlugify6-4	5429486	190.8 ns/op
96 B/op	1 allocs/op	
PASS		

contender: 257 ns 46 B 1 allocs

champion: 172 ns 46 B 1 allocs

Вы уступили чемпиону, но результат неплохой

PASS

В общем, думаю сойдет...

```
/*
ЧЕМПИОНСКИЕ РЕШЕНИЯ
*/

/* 1 место Aleksandr Panfilov
var charType = [256]int8{}

const (
    upper = 1
    lower = 2
    digit = 4
    dash  = 8
)

func init() {
    for i := 'A'; i <= 'Z'; i++ {
        charType[i] = upper
    }
    for i := 'a'; i <= 'z'; i++ {
```

```

        charType[i] = lower
    }
    for i := '0'; i <= '9'; i++ {
        charType[i] = digit
    }
    charType['-'] = dash
}

// Let's go black unsafe magic 🐱

func ptrAdd(p *byte, n int) *byte {
    return (*byte)(unsafe.Add(unsafe.Pointer(p), n))
}

func ptrSub(p1, p2 *byte) int {
    return int(uintptr(unsafe.Pointer(p1)) - uintptr(unsafe.Pointer(p2)))
}

func ptrGet(p *byte) (byte, *byte) {
    return *p, (*byte)(unsafe.Add(unsafe.Pointer(p), 1))
}

func ptrSet(p *byte, v byte) *byte {
    *p = v
    return (*byte)(unsafe.Add(unsafe.Pointer(p), 1))
}

func slugify(s string) string {
    if len(s) == 0 {
        return ""
    }

    buf := make([]byte, len(s))

    // See `stringStruct` and `slice` in GOROOT/runtime/string.go and
    GOROOT/runtime/slice.go.
    src := (**byte)(unsafe.Pointer(&s))
    dst := (**byte)(unsafe.Pointer(&buf))

    srcEnd := ptrAdd(src, len(s))
    dstStart := dst
    ch := byte(0)

```

```

mainLoop:
    for {
        for {
            if src == srcEnd {
                ch = 0
                break mainLoop
            }

            ch, src = ptrGet(src)

            if t := charType[ch]; t&upper != 0 {
                ch += 32
                break
            } else if t&(lower|digit|dash) != 0 {
                break
            }
        }

        for {
            dst = ptrSet(dst, ch)

            if src == srcEnd {
                break mainLoop
            }

            ch, src = ptrGet(src)

            if t := charType[ch]; t&(lower|digit|dash) != 0 {
                //
            } else if t&upper != 0 {
                ch += 32
            } else {
                break
            }
        }

        dst = ptrSet(dst, '-')
    }

    count := ptrSub(dst, dstStart)

    if count == 0 {
        return ""
    }

```

```

}

if ch == 0 {
    count--
}

// buf = buf[:count]
(*struct {
    p    *byte
    len int
})(unsafe.Pointer(&buf)).len = count

return *(*string)(unsafe.Pointer(&buf))
}
*/

/* самое быстрое БЕЗ УКАЗАТЕЛЕЙ (автор курса)
func slugify(src string) string {
    dst := new(strings.Builder)
    dst.Grow(len(src) + 1)

    for j := 0; j < len(src); {
        for ; j < len(src) && !isValid(src[j]); j++ {
        }
        if j == len(src) {
            break
        }
        dst.WriteByte('-')
        for ; j < len(src) && isValid(src[j]); j++ {
            dst.WriteByte(toLower(src[j]))
        }
    }

    if dst.Len() == 0 {
        return ""
    }
    return dst.String()[1:]
}

func isValid(ch byte) bool {
    return ch >= 'a' && ch <= 'z' || ch >= 'A' && ch <= 'Z' || ch >= '0' &&
ch <= '9' || ch == '-'
}

```

```
func toLower(ch byte) byte {
    if ch >= 'A' && ch <= 'Z' {
        return ch + 32
    }
    return ch
}
*/
```

Стандартная библиотека 2. Дата и время

1

Работа с датой и временем почему-то особенно не дается авторам языков программирования. Популярные языки, созданные в прошлом веке, полностью провалили эту часть. Java потребовалось три попытки, чтобы сделать нормальное API. В Python получилась неудобная конструкция с пятью разновидностями дат. В JavaScript интерфейс работы с датами простой, но нефункциональный.

К счастью, в Go все сделали сразу хорошо (ну почти).

На этом уроке:

- момент времени;
- часовой пояс;
- продолжительность и арифметика;
- парсинг и форматирование;
- юникс-время;
- монотонное время.

Момент времени

В Go за работу с датами отвечает пакет `time`. Тип `time.Time` описывает конкретный *момент времени* (instant in time) с наносекундной точностью. Чтобы получить текущий момент времени (далее для краткости вместо «момент времени» буду писать просто «время»), используют функцию `time.Now()`:

```
t := time.Now()
fmt.Println(t)
// 2022-05-24 17:45:22.951205 +0300 MSK m=+0.000000001
```

Значение `t` включает дату (2022-05-24), время (17:45:22.951205), часовой пояс (+0300 MSK), а также монотонное время (о нем поговорим отдельно).

Для каждого компонента времени предусмотрен отдельный метод:

```
fmt.Println(t.Year())  
// 2022  
fmt.Println(int(t.Month()))  
// 5  
fmt.Println(t.Day())  
// 24  
fmt.Println(t.Hour())  
// 17  
fmt.Println(t.Minute())  
// 45  
fmt.Println(t.Second())  
// 22  
fmt.Println(t.Nanosecond())  
// 951205000
```

Метод `Month()` возвращает значение типа `time.Month`, образованного от `int`:

```
fmt.Println(time.January == time.Month(1))  
// true  
fmt.Println(time.December == time.Month(12))  
// true
```

Еще несколько полезных методов:

- порядковый номер дня в году

```
•   fmt.Println(t.YearDay())  
    // 144
```

- год и порядковый номер недели

```
•   year, week := t.ISOWeek()  
    fmt.Println(year, week)  
    // 2022 21
```

- порядковый номер дня недели (0...6 — воскресенье...суббота)

```
•   fmt.Println(int(t.Weekday()))  
    // 2
```

- часы, минуты, секунды

```
•   hour, min, sec := t.Clock()  
    fmt.Println(hour, min, sec)  
    // 17 45 22
```

Конкретное время в прошлом или будущем можно получить через функцию `time.Date()`. Придется перечислить все компоненты от года до наносекунд, а также часовой пояс:

```
t := time.Date(2022, 5, 24, 17, 45, 22, 951205000, time.Local)
fmt.Println(t)
// 2022-05-24 17:45:22.951205 +0300 MSK
```

[песочница](#)

2

Високосный год

Напишите функцию `isLeapYear()`, которая определяет, високосный год или нет. Високосный — это тот, в котором есть 29 февраля.

```
func isLeapYear(year int) bool {
    // ...
}

func main() {
    fmt.Println(isLeapYear(2020))
    // true

    fmt.Println(isLeapYear(2022))
    // false
}
```

Полный код — в песочнице. Отправляйте в качестве решения только фрагмент, отмеченный комментариями «начало решения» и «конец решения».

[песочница](#)

Sample Input:

Sample Output:

PASS

```
package main

import (
    "testing"
    "time"
)
```

// начало решения

```
func isLeapYear(year int) bool {  
    d := time.Date(year, 2, 29, 0, 0, 0, 0, time.UTC)  
    // в невисокосном году дата распарсится как 01 марта  
    return d.Month() == time.February  
  
    // или  
    // return time.Date(year, 2, 29, 12, 0, 0, 0, time.Local).Day() == 29  
  
    // или классический вариант  
    // return year%4 == 0 && (year%100 != 0 || year%400 == 0)  
}
```

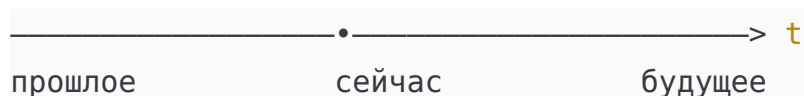
// конец решения

```
func Test(t *testing.T) {  
    if !isLeapYear(2020) {  
        t.Errorf("2020 is a leap year")  
    }  
    if isLeapYear(2022) {  
        t.Errorf("2022 is NOT a leap year")  
    }  
}  
  
func main() {  
    Test(&testing.T{})  
}
```

3

Часовой пояс

Представим время как числовую ось из прошлого в будущее, общую для всего мира.



Одна и та же точка на этой оси соответствует разному локальному времени, в зависимости от географического положения. Например, если 24 мая в Нью-Йорке 9:00, то:

- в Лондоне в это же время 14:00,

- в Париже 15:00,
- в Москве 16:00.

Часовой пояс (timezone) описывает смещение локального времени относительно нулевой точки отсчета — всемирного координированного времени (UTC). При этом некоторые города живут в фиксированном часовом поясе (Москва, UTC+3), а некоторые летом живут по одному времени, зимой по другому (Париж, летом UTC+2, зимой UTC+1).

В любом случае, чтобы корректно сравнивать время из разных точек Земли, приходится учитывать, к какой локации относится это время. В Go за локации отвечает тип `time.Location`.

Проще всего считать время по UTC — для него предусмотрена готовая локация `time.UTC`:

```
t1 := time.Date(2022, 5, 24, 0, 0, 0, 0, time.UTC)
```

`time.FixedZone()` создает часовой пояс с фиксированным смещением в секундах относительно UTC:

```
utc := time.FixedZone("UTC", 0)
t1 := time.Date(2022, 5, 24, 0, 0, 0, 0, time.UTC)
t2 := time.Date(2022, 5, 24, 0, 0, 0, 0, utc)
fmt.Println(t1.Equal(t2))
// true
```

Обратите внимание — для проверки времени на равенство мы используем метод `Equal()` вместо `==`. Он понимает, что `time.UTC` и `time.FixedZone("UTC", 0)` дают одинаковое смещение, и возвращает `true`. Сравнение через `==` показало бы `false`.

Еще пример:

```
offset_sec := 3 * 3600
utc_3 := time.FixedZone("UTC+3", offset_sec)

// я в Москве, так что time.Local = UTC+3
t1 := time.Date(2022, 5, 24, 0, 0, 0, 0, time.Local)
t2 := time.Date(2022, 5, 24, 0, 0, 0, 0, utc_3)
fmt.Println(t1.Equal(t2))
// true
```

Часовой пояс можно получить по наименованию через `time.LoadLocation()`. Такая локация корректно учитывает переход на летнее время и другие особенности поведения времени в конкретной географической зоне:

```
paris, _ := time.LoadLocation("Europe/Paris")
utc_2 := time.FixedZone("UTC+2", 2*3600)
```

```
t1 := time.Date(2022, 5, 24, 0, 0, 0, 0, paris)
t2 := time.Date(2022, 5, 24, 0, 0, 0, 0, utc_2)
fmt.Println(t1.Equal(t2))
// true
```

Чтобы перевести время из одного часового пояса в другой, используют метод `In()`:

```
t := time.Date(2022, 5, 24, 0, 0, 0, 0, time.UTC)
fmt.Println(t)
// 2022-05-24 00:00:00 +0000 UTC

paris, _ := time.LoadLocation("Europe/Paris")
fmt.Println(t.In(paris))
// 2022-05-24 02:00:00 +0200 CEST

ny, _ := time.LoadLocation("America/New_York")
fmt.Println(t.In(ny))
// 2022-05-23 20:00:00 -0400 EDT
```

Список локаций можно посмотреть в [IANA Time Zone Database](https://iana.org/time-zones/).

Локальное время — сложная штука. Поэтому старайтесь избегать часовых поясов и работать только с `time.UTC`.

[песочница](#)

4

Сравнение

Как мы уже выяснили, для проверки дат на равенство используют метод `Equal()`:

```
t1 := time.Date(2022, 5, 24, 17, 45, 22, 0, time.UTC)
t2 := time.Date(2022, 5, 24, 20, 45, 22, 0, time.FixedZone("UTC+3",
3*60*60))
fmt.Println(t1 == t2)
// false
fmt.Println(t1.Equal(t2))
// true
```

За сравнение «раньше» и «позже» отвечают методы `Before()` и `After()`:

```
t1 := time.Date(2022, 5, 25, 0, 0, 0, 0, time.UTC)
t2 := time.Date(2022, 5, 25, 17, 45, 22, 0, time.UTC)
fmt.Println(t2.After(t1))
```

```
// true
fmt.Println(t1.Before(t2))
// true
```

Нулевое значение `time.Time` — 01.01.0001 00:00:00 UTC. Его проверяют через метод `IsZero()`:

```
var t time.Time
fmt.Println(t)
// 0001-01-01 00:00:00 +0000 UTC
fmt.Println(t.IsZero())
// true
```

[песочница](#)

5

Время дня

Реализуйте тип `TimeOfDay`, который описывает время без года, месяца, дня и прочих подробностей. Вот его интерфейс:

```
// Hour возвращает часы в пределах дня
Hour() int

// Minute возвращает минуты в пределах часа
Minute() int

// Second возвращает секунды в пределах минуты
Second() int

// String возвращает строковое представление времени
// в формате чч:мм:сс TZ (например, 12:34:56 UTC)
String()

// Equal сравнивает одно время с другим.
// Если у t и other разные локации - возвращает false.
Equal(other TimeOfDay) bool

// Before возвращает true, если время t предшествует other.
// Если у t и other разные локации - возвращает ошибку.
Before(other TimeOfDay) (bool, error)

// After возвращает true, если время t идет после other.
// Если у t и other разные локации - возвращает ошибку.
After(other TimeOfDay) (bool, error)
```

Пример использования:

```
t1 := MakeTimeOfDay(17, 45, 22, time.UTC)
t2 := MakeTimeOfDay(20, 3, 4, time.UTC)

fmt.Println(t1.Hour(), t1.Minute(), t1.Second())
// 17 45 22

fmt.Println(t1)
// 17:45:22 UTC

fmt.Println(t1.Equal(t2))
// false

before, err := t1.Before(t2)
fmt.Println(before, err)
// true <nil>

after, err := t1.After(t2)
fmt.Println(after, err)
// false <nil>
```

Два времени можно сравнивать, только если у них одинаковая локация. В противном случае `Equal()` должен возвращать `false`, а `Before()` и `After()` — ошибку. Тип и текст ошибки на ваше усмотрение.

Для простоты будем считать, что локации можно сравнить по названию, метод `[Location.String()](https://pkg.go.dev/time#Location.String)`.

Полный код — в песочнице. Отправляйте в качестве решения только фрагмент, отмеченный комментариями «начало решения» и «конец решения».

[песочница](#)

Sample Input:

Sample Output:

PASS

```
package main

import (
    "errors"
    "fmt"
```

```

    "testing"
    "time"
)

// начало решения

// TimeOfDay описывает время в пределах одного дня
type TimeOfDay struct {
    wrapped time.Time
}

// Hour возвращает часы в пределах дня
func (t TimeOfDay) Hour() int {
    return t.wrapped.Hour()
}

// Minute возвращает минуты в пределах часа
func (t TimeOfDay) Minute() int {
    return t.wrapped.Minute()
}

// Second возвращает секунды в пределах минуты
func (t TimeOfDay) Second() int {
    return t.wrapped.Second()
}

// String возвращает строковое представление времени
// в формате чч:мм:сс TZ (например, 12:34:56 UTC)
func (t TimeOfDay) String() string {
    return fmt.Sprintf("%02d:%02d:%02d %s", t.Hour(), t.Minute(),
t.Second(), t.wrapped.Location())
}

// Equal сравнивает одно время с другим.
// Если у t и other разные локации - возвращает false.
func (t TimeOfDay) Equal(other TimeOfDay) bool {
    return (t.Hour() == other.Hour()) && (t.Minute() == other.Minute()) &&
(t.Second() == other.Second()) && (t.wrapped.Location().String() ==
other.wrapped.Location().String())
}

var ErrDifferentLocations = errors.New("different locations")

```



```

// Before возвращает true, если время t предшествует other.
// Если у t и other разные локации - возвращает ошибку.
func (t TimeOfDay) Before(other TimeOfDay) (bool, error) {
    if t.wrapped.Location().String() != other.wrapped.Location().String() {
        return false, ErrDifferentLocations
    }

    before := false
    if t.Hour() < other.Hour() {
        before = true
    } else if (t.Hour() == other.Hour()) && (t.Minute() < other.Minute()) {
        before = true
    } else if (t.Minute() == other.Minute()) && (t.Second() <
other.Second()) {
        before = true
    }

    return before, nil
}

// After возвращает true, если время t идет после other.
// Если у t и other разные локации - возвращает ошибку.
func (t TimeOfDay) After(other TimeOfDay) (bool, error) {
    if t.wrapped.Location().String() != other.wrapped.Location().String() {
        return false, ErrDifferentLocations
    }

    after := false
    if t.Hour() > other.Hour() {
        after = true
    } else if (t.Hour() == other.Hour()) && (t.Minute() > other.Minute()) {
        after = true
    } else if (t.Minute() == other.Minute()) && (t.Second() >
other.Second()) {
        after = true
    }

    // ... или просто использовать wrapped.After(), если полагаемся на
    равенство года/месяца/числа

    return after, nil
}

```

```
// MakeTimeOfDay создает время в пределах дня
func MakeTimeOfDay(hour, min, sec int, loc *time.Location) TimeOfDay {
    return TimeOfDay{time.Date(0, 0, 0, hour, min, sec, 0, loc)}
}

// конец решения

func Test(t *testing.T) {
    t1 := MakeTimeOfDay(17, 45, 22, time.UTC)
    t2 := MakeTimeOfDay(20, 3, 4, time.UTC)

    fmt.Println(t1.String())
    if t1.Equal(t2) {
        t.Errorf("%v should not be equal to %v", t1, t2)
    }

    before, _ := t1.Before(t2)
    if !before {
        t.Errorf("%v should be before %v", t1, t2)
    }

    after, _ := t1.After(t2)
    if after {
        t.Errorf("%v should NOT be after %v", t1, t2)
    }
}
```

6

Продолжительность

Помимо *момента* времени часто используют *продолжительность* (duration) — интервал между двумя моментами времени (10 секунд, 15 минут, 3 часа, ...). В Go за продолжительность отвечает тип `time.Duration`.

Продолжительность можно создать из строкового описания с помощью функции `time.ParseDuration()`:

```
d1, _ := time.ParseDuration("30s")
fmt.Printf("#v\n", d1)
// 300000000000

d2, _ := time.ParseDuration("2h15m30s")
```

```
fmt.Printf("%#v\n", d2)
// 813000000000000
```

Продолжительность измеряется с точностью до наносекунд, но можно получить ее в микро-, милли- или обычных секундах:

```
d, _ := time.ParseDuration("2h15m30s")
fmt.Println(d.Seconds())
// 8130
```

```
fmt.Println(d.Milliseconds())
// 8130000
```

```
fmt.Println(d.Microseconds())
// 8130000000
```

```
fmt.Println(d.Nanoseconds())
// 813000000000000
```

Чтобы не парсить продолжительность, ее часто задают явно, с помощью готовых констант:

```
d1 := 30 * time.Second
fmt.Println(d1)
// 30s
```

```
d2 := 15 * time.Minute
fmt.Println(d2)
// 15m0s
```

```
d3 := 2 * time.Hour
fmt.Println(d3)
// 2h0m0s
```

Самая крупная константа — `time.Hour`. Констант для дня, недели, месяца и года не предусмотрено.

Продолжительности можно складывать и вычитать:

```
d := d1 + d2 + d3
fmt.Println(d)
// 2h15m30s
```

```
d = d3 - d2
fmt.Println(d)
// 1h45m0s
```

А функция `time.Since()` возвращает продолжительность с указанного момента до текущего:

```
before := time.Now()
time.Sleep(time.Second)
elapsed := time.Since(before)
fmt.Println(elapsed)
// 1s
```

[песочница](#)

7

Арифметика

Чтобы прибавить ко времени продолжительность и получить новое время, используют метод

`Add()`:

```
before := time.Date(2022, 5, 24, 0, 0, 0, 0, time.UTC)
after := before.Add(7 * time.Hour)
fmt.Println(after)
// 2022-05-24 07:00:00 +0000 UTC
```

```
after = before.Add(7 * time.Minute)
fmt.Println(after)
// 2022-05-24 00:07:00 +0000 UTC
```

```
after = before.Add(7 * time.Second)
fmt.Println(after)
// 2022-05-24 00:00:07 +0000 UTC
```

```
after = before.Add(100 * time.Millisecond)
fmt.Println(after)
// 2022-05-24 00:00:00.1 +0000 UTC
```

```
after = before.Add(100 * time.Microsecond)
fmt.Println(after)
// 2022-05-24 00:00:00.0001 +0000 UTC
```

```
after = before.Add(100 * time.Nanosecond)
fmt.Println(after)
// 2022-05-24 00:00:00.00000001 +0000 UTC
```

```
before = time.Date(2022, 5, 24, 0, 0, 0, 0, time.UTC)
after = before.Add(-7 * time.Hour)
```

```
fmt.Println(after)
// 2022-05-23 17:00:00 +0000 UTC
```

Метод `AddDate()` добавляет указанное количество лет, месяцев и дней:

```
before = time.Date(2022, 5, 24, 0, 0, 0, 0, time.UTC)
after = before.AddDate(1, 2, 3)
fmt.Println(after)
// 2023-07-27 00:00:00 +0000 UTC
```

Разность двух времен через метод `Sub()` возвращает продолжительность между ними:

```
before := time.Date(2022, 5, 24, 0, 0, 0, 0, time.UTC)
after := time.Date(2022, 5, 24, 17, 45, 22, 0, time.UTC)
diff := after.Sub(before)
fmt.Println(diff)
// 17h45m22s
```

[песочница](#)

8

Парсинг

Допустим, есть строка `24.05.2022`, которую мы хотим превратить в дату. Для этого понадобится еще маска (layout), которая описывает, где в строке год, месяц и число. Вечная проблема с масками — в каждом языке программирования они свои. Например, в Java использовали бы `dd.MM.yyyy`, а в Python — `%d.%m.%Y`. Второй формат более распространен — Питон позаимствовал его из C, и так поступили многие другие языки.

В Go, к сожалению, придумали свой оригинальный способ. В качестве маски Go использует *контрольное время* (reference time):

```
// маска
layout := "02.01.2006"
// значение
value := "24.05.2022"
t, _ := time.Parse(layout, value)
fmt.Println(t)
// 2022-05-24 00:00:00 +0000 UTC
```

Полностью контрольное время выглядит так:

- 2 января 2006 года,
- время 15:04:05,

- часовой пояс MST (UTC-7).

Вот несколько примеров, как использовать контрольное время для парсинга. Маска идет первым параметром `time.Parse()`, исходное значение — вторым:

```
s := "24.05.2022 17:45"
t, _ := time.Parse("02.01.2006 15:04", s)
fmt.Println(t)
// 2022-05-24 17:45:00 +0000 UTC
```

Если часовой пояс не указан, `Parse()` устанавливает его в UTC.

```
s := "17:45:22"
t, _ := time.Parse("15:04:05", s)
fmt.Println(t)
// 0000-01-01 17:45:22 +0000 UTC
```

Дата по умолчанию — 01.01.0000.

```
s := "24.05.2022 17:45:22.951205+03:00"
t, _ := time.Parse("02.01.2006 15:04:05.999999-07:00", s)
fmt.Println(t)
// 2022-05-24 17:45:22.951205 +0300 MSK
```

На мой взгляд, получилось неудобно. Но такой уж подход выбрали авторы языка.

Возможно, вы задаетесь вопросом, как можно было выбрать настолько дикий формат для маски. В этом формате есть некоторая логика. Но работает она, только если вы житель США. В привычном для американца формате контрольное время выглядит как `01/02 03:04:05PM '06 -0700`. То есть мнемонически его можно запомнить как 1-2-3-4-5-6-7. Для остального мира, естественно, мнемоника не работает. Спасибо за заботу, авторы Go.

Чтобы не мучиться каждый раз с маской, в Go предусмотрели несколько predefined форматов. К сожалению, они распространены только в Северной Америке, поэтому для нас по большей части бесполезны. Кроме одного — `time.RFC3339`. Это общепринятый стандарт представления даты и времени, также известный как ISO 8601:

```
s := "2022-05-24T17:45:22.951205+03:00"
t, _ := time.Parse(time.RFC3339, s)
fmt.Println(t)
// 2022-05-24 17:45:22.951205 +0300 MSK
```

[песочница](#)

Форматирование

Форматирование — обратная парсингу операция: преобразует время в строку. Использует метод `Format()` и те же самые маски:

```
t, _ := time.Parse(time.RFC3339, "2022-05-24T17:45:22.951205+03:00")

fmt.Println(t.Format("02.01.2006"))
// 24.05.2022

fmt.Println(t.Format("02.01.2006 15:04"))
// 24.05.2022 17:45

fmt.Println(t.Format("02.01.2006 15:04:05 MST"))
// 24.05.2022 17:45:22 MSK
```

[песочница](#)

9

Журнал задач

Вы прочитали модную книгу по осознанности и так вдохновились, что целый год вели журнал задач. Записи за день выглядят примерно так:

```
15.04.2022
8:00 - 8:30 Завтрак
8:30 - 9:30 Оглаживание кота
9:30 - 10:00 Интернеты
10:00 - 14:00 Напряженная работа
14:00 - 14:45 Обед
14:45 - 15:00 Оглаживание кота
15:00 - 19:00 Напряженная работа
19:00 - 19:30 Интернеты
19:30 - 22:30 Безудержное веселье
22:30 - 23:00 Оглаживание кота
```

Первая строка — дата. За ней на каждой строке задача: время начала, время окончания, название. Как видите, некоторые особо важные задачи повторяются в течение дня.

Записи журнала текстовые (каждый день — текст из нескольких строк), и в таком виде для анализа не пригодны. Напишите функцию-парсер `ParsePage()`, которая принимает на входе текст за день (параметр `src`), а возвращает срез записей типа `Task`:

```
// Task описывает задачу, выполненную в определенный день
type Task struct {
    Date  time.Time
```

```

    Dur    time.Duration
    Title string
}

// ParsePage разбирает страницу журнала
// и возвращает задачи, выполненные за день
func ParsePage(src string) ([]Task, error) {
    // ...
}

func main() {
    page := `...`

    entries, err := ParsePage(page)
    if err != nil {
        panic(err)
    }
    fmt.Println("Мои достижения за", entries[0].Date.Format("2006-01-02"))
    for _, entry := range entries {
        fmt.Printf("- %v: %v\n", entry.Title, entry.Dur)
    }
}

```

Мои достижения за 2022-04-15

- Напряженная работа: 8h0m0s
- Безудержное веселье: 3h0m0s
- Оглаживание кота: 1h45m0s
- Интернеты: 1h0m0s
- Обед: 45m0s
- Завтрак: 30m0s

`ParsePage()` считает общую длительность по каждой задаче и возвращает задачи в порядке убывания длительности. Если какие-то задачи имеют одинаковую продолжительность, их взаимный порядок не имеет значения.

Вы вели журнал не слишком внимательно, так что в тексте могут быть ошибки. Например, неожиданный формат даты или отсутствующее время начала задачи. В таких случаях

`ParsePage()` должна возвращать ошибку. Тип и текст ошибки на ваше усмотрение.

При парсинге времени используйте часовой пояс UTC.

Если решите использовать регулярное выражение для разбора строки журнала, вот подходящий шаблон:


```
(\d+:\d+) - (\d+:\d+) (.+)
```

В сортировке задач поможет функция `[sort.Slice]` (<https://pkg.go.dev/sort#Slice>).

В песочнице вы найдете «скелет» решения. Реализуйте логику функций, не меняя их сигнатуры. Новые функции и типы добавлять можно. Отправляйте в качестве решения только фрагмент, отмеченный комментариями «начало решения» и «конец решения».

[песочница](#)

Sample Input:

Sample Output:

PASS

```
package main

import (
    "errors"
    "fmt"
    "regexp"
    "sort"
    "strings"
    "time"
)

// начало решения

// Task описывает задачу, выполненную в определенный день
type Task struct {
    Date  time.Time
    Dur   time.Duration
    Title string
}

// ParsePage разбирает страницу журнала
// и возвращает задачи, выполненные за день
func ParsePage(src string) ([]Task, error) {
    lines := strings.Split(src, "\n")
    date, err := parseDate(lines[0])

    if err != nil {
        return nil, err
    }
}
```

```

}

tasks, err := parseTasks(date, lines[1:])
sortTasks(tasks)
return tasks, err
}

// parseDate разбирает дату в формате дд.мм.гггг
func parseDate(src string) (time.Time, error) {
    layout := "02.01.2006"
    return time.Parse(layout, src)
}

// parseTasks разбирает задачи из записей журнала
func parseTasks(date time.Time, lines []string) ([]Task, error) {
    time_title_RE := regexp.MustCompile(`(\d+:\d+) - (\d+:\d+) (.+)`)
    tmp := make(map[string]time.Duration)

    for _, line := range lines {
        match := time_title_RE.FindStringSubmatch(line)
        if len(match) < 4 {
            return nil, errors.New("bad string")
        }

        start, err1 := time.Parse("15:04", match[1])
        end, err2 := time.Parse("15:04", match[2])

        if (err1 != nil) || (err2 != nil) {
            return nil, errors.New("wrong time format")
        }
        if !end.After(start) { // автор хочет, чтоб 0 продолжительность
тоже отсекалась
            return nil, errors.New("wrong time format")
        }

        tmp[match[3]] += end.Sub(start)
    }

    v := make([]Task, 0, len(tmp))

    for title, dur := range tmp {
        v = append(v, Task{date, dur, title})
    }
}

```

```

    return v, nil
}

// sortTasks упорядочивает задачи по убыванию длительности
func sortTasks(tasks []Task) {
    sort.Slice(tasks, func(i, j int) bool { return tasks[i].Dur >
tasks[j].Dur })
}

// конец решения

func main() {
    page := `15.04.2022
8:00 - 8:30 Завтрак
8:30 - 9:30 Оглаживание кота
9:30 - 10:00 Интернеты
10:00 - 14:00 Напряженная работа
14:00 - 14:45 Обед
14:45 - 15:00 Оглаживание кота
15:00 - 19:00 Напряженная работа
19:00 - 19:30 Интернеты
19:30 - 22:30 Безудержное веселье
22:30 - 23:00 Оглаживание кота`

    parseTasks(time.Date(0, 0, 0, 0, 0, 0, 0, time.UTC), []string{"11:00 -
12:00 task"})

    entries, err := ParsePage(page)
    if err != nil {
        panic(err)
    }
    fmt.Println("Мои достижения за", entries[0].Date.Format("2006-01-02"))
    for _, entry := range entries {
        fmt.Printf("- %v: %v\n", entry.Title, entry.Dur)
    }

    // ожидаемый результат
    /*
        Мои достижения за 2022-04-15
        - Напряженная работа: 8h0m0s
        - Безудержное веселье: 3h0m0s
        - Оглаживание кота: 1h45m0s
    */
}

```

```
- Интернеты: 1h0m0s
- Обед: 45m0s
- Завтрак: 30m0s

*/
}
```

10

Юникс-время

В программировании существует альтернативная упрощенная концепция времени — *юникс-время* (unix time). Это количество секунд, прошедших с начала *эпохи* (epoch) — 01.01.1970 00:00:00 UTC:

- 31.12.1969 23:00:00 UTC = -3600
- 01.01.1970 00:00:00 UTC = 0
- 01.01.1970 01:00:00 UTC = 3600
- 01.01.1970 02:00:00 UTC = 7200
- ...

Юникс-время не слишком человеко-читаемо, но прекрасно своей простотой: в нем нет часовых поясов, а моменты времени легко сравнивать и вычитать.

Go преобразует обычное время в юникс-время с любой точностью — от секунд до наносекунд:

```
t, _ := time.Parse(time.RFC3339, "2022-05-24T17:45:22.951205+03:00")

fmt.Println(t.Unix())
// 1653403522
fmt.Println(t.UnixMilli())
// 1653403522951
fmt.Println(t.UnixMicro())
// 1653403522951205
fmt.Println(t.UnixNano())
// 1653403522951205000
```

И обратно — из юникс-времени в обычное:

```
// из секунд
t := time.Unix(1653403522, 0)
fmt.Println(t)
// 2022-05-24 17:45:22 +0300 MSK

// из наносекунд
t = time.Unix(0, 1653403522951205123)
```

```

fmt.Println(t)
// 2022-05-24 17:45:22.951205123 +0300 MSK

// из секунд и наносекундного остатка
t = time.Unix(1653403522, 951205123)
fmt.Println(t)
// 2022-05-24 17:45:22.951205123 +0300 MSK

// из миллисекунд
t = time.UnixMilli(1653403522951)
fmt.Println(t)
// 2022-05-24 17:45:22.951 +0300 MSK

// из микросекунд
t = time.UnixMicro(1653403522951205)
fmt.Println(t)
// 2022-05-24 17:45:22.951205 +0300 MSK

```

[песочница](#)

12

Монотонное время

Вернемся к `time.Now()`, с которой мы начали урок:

```

t := time.Now()
fmt.Println(t)
// 2022-05-24 17:45:22.951205 +0300 MSK m=+0.0000000001

```

`m=+...` — это *монотонное время* (monotonic time). Монотонное время — это локальное время процесса в секундах. Сразу после старта процесса оно равно 0, а дальше увеличивается на 0.0000000001 с каждой прошедшей наносекундой:

```

fmt.Println(time.Now())
// ... m=+0.0000000001

time.Sleep(50 * time.Millisecond)
fmt.Println(time.Now())
// ... m=+0.0500000001

time.Sleep(50 * time.Millisecond)
fmt.Println(time.Now())
// ... m=+0.1000000001

```

Программистам не нужно как-то специально работать с монотонным временем: я рассказываю о нем только для того, чтобы `m=+...` не осталось для вас загадкой.

Сам Go использует монотонное время в реализации некоторых методов. Например, `Since()`, который возвращает продолжительность времени с указанного момента до текущего:

```
start := time.Now()
time.Sleep(50 * time.Millisecond)
elapsed := time.Since(start)
fmt.Println(elapsed)
// 50ms
```

Или `Until()`, который возвращает оставшуюся продолжительность времени до «дедлайна»:

```
deadline := time.Now().Add(60 * time.Second)

time.Sleep(time.Second)
fmt.Println(time.Until(deadline))
// 59s

time.Sleep(time.Second)
fmt.Println(time.Until(deadline))
// 58s
```

Монотонная часть существует только у времени, полученного через `time.Now()`. Если создать время через `time.Date()` или `time.Parse()` — ее не будет:

```
t1 := time.Date(2022, 5, 24, 17, 45, 22, 951205000, time.Local)
fmt.Println(t1)
// 2022-05-24 17:45:22.951205 +0300 MSK

t2, _ := time.Parse(time.RFC3339, "2022-05-24T17:45:22.951205+03:00")
fmt.Println(t2)
// 2022-05-24 17:45:22.951205 +0300 MSK
```

Чтобы «отрезать» монотонную часть, используют метод `Round()`:

```
t := time.Now()
fmt.Println(t)
// 2022-05-24 17:45:22.951205 +0300 MSK m=+0.000000001

wall := t.Round(0)
fmt.Println(wall)
// 2022-05-24 17:45:22.951205 +0300 MSK
```

13

Легаси-дата

Вам не повезло интегрироваться с унаследованной системой. Дата в ней хранится в необычном формате: это дробное число секунд от юникс-эпохи, но не float, а строка. Примерно так:

- `"0.0"` → 1970-01-01 00:00:00.000
- `"3600.0"` → 1970-01-01 01:00:00.000
- `"3600.123"` → 1970-01-01 01:00:00.123

Дробная часть может быть любой точности, вплоть до наносекундной:

```
3600.0
3600.123
3600.123456
3600.123456789
```

Корректными считаются только значения с заполненной целой и дробной частью, и точкой в качестве разделителя. Например, эти значения — некорректные:

```
3600
.123
3600,123
```

Все легаси-даты расположены не раньше юникс-эпохи. Отрицательных дат нет.

Напишите функции, которые преобразуют легаси-дату в `time.Time` и обратно:

```
asLegacyDate(t time.Time) string
parseLegacyDate(d string) (time.Time, error)
```

`parseLegacyDate()` должна возвращать ошибку для некорректных исходных значений. Тип и текст ошибки на ваше усмотрение.

Полный код — в песочнице. Отправляйте в качестве решения только фрагмент, отмеченный комментариями «начало решения» и «конец решения».

Sample Input:

Sample Output:

PASS

```
package main

import (
    "errors"
    "fmt"
    "math"
    "regexp"
    "strconv"
    // "strings"
    "testing"
    "time"
)

// начало решения

// asLegacyDate преобразует время в легаси-дату
func asLegacyDate(t time.Time) string {
    s := t.Unix()
    ns := t.Nanosecond()

    rem := ns % 10
    for {
        if (ns != 0) && (rem == 0) {
            ns = ns / 10
            rem = ns % 10
        } else {
            break
        }
    }

    return fmt.Sprintf("%d.%d", s, ns)
}

// parseLegacyDate преобразует легаси-дату во время.
// Возвращает ошибку, если легаси-дата некорректная.
func parseLegacyDate(d string) (time.Time, error) {
    RE := regexp.MustCompile(`(\d+)\.(\d+)`)
    parts := RE.FindStringSubmatch(d)
    if len(parts) != 3 {
        return time.Time{}, errors.New("Bad time format")
    }
}
```



```

s, _ := strconv.Atoi(parts[1])
ns, _ := strconv.Atoi(parts[2])
dig := len(parts[2])

//
t := time.Unix(int64(s), int64(ns*int(math.Pow10(9-dig))))
return t, nil
}

// конец решения

/*
// asLegacyDate преобразует время в легаси-дату
func asLegacyDate(t time.Time) string {
    sec := t.Unix()
    nano := t.UnixNano() - sec*1e9
    if nano == 0 {
        return fmt.Sprintf("%d.0", sec)
    }
    str := fmt.Sprintf("%d.%d", sec, nano)
    return strings.TrimRight(str, "0")
}

// parseLegacyDate преобразует легаси-дату во время.
// Возвращает ошибку, если легаси-дата некорректная.
func parseLegacyDate(d string) (time.Time, error) {
    strSec, strNano, ok := strings.Cut(d, ".")
    if !ok {
        return time.Time{}, fmt.Errorf("invalid date: %v", d)
    }

    sec, err := strconv.ParseInt(strSec, 10, 64)
    if err != nil {
        return time.Time{}, fmt.Errorf("invalid date: %v", d)
    }

    if len(strNano) == 0 {
        return time.Time{}, fmt.Errorf("invalid date: %v", d)
    }
    strNano = padZerosRight(strNano, 9)
    nano, err := strconv.ParseInt(strNano, 10, 64)
    if err != nil {

```

```

        return time.Time{}, fmt.Errorf("invalid date: %v", d)
    }

    return time.Unix(sec, nano), nil
}

// padZerosRight отбивает строку нулями справа до указанной длины
func padZerosRight(str string, length int) string {
    if len(str) >= length {
        return str
    }
    return str + strings.Repeat("0", length-len(str))
}

*/

func Test_asLegacyDate(t *testing.T) {
    samples := map[time.Time]string{
        time.Date(1970, 1, 1, 1, 0, 0, 123456789, time.UTC):
"3600.123456789",
        time.Date(1970, 1, 1, 1, 0, 0, 0, time.UTC):          "3600.0",
        time.Date(1970, 1, 1, 0, 0, 0, 0, time.UTC):          "0.0",
        time.Date(2022, 5, 24, 14, 45, 22, 951, time.UTC):
"1653403522.951",
        time.Date(2022, 5, 24, 14, 45, 22, 951205, time.UTC):
"1653403522.951205",
    }
    for src, want := range samples {
        got := asLegacyDate(src)
        if got != want {
            t.Fatalf("%v: got %v, want %v", src, got, want)
        }
    }
}

func Test_parseLegacyDate(t *testing.T) {
    samples := map[string]time.Time{
        "3600.123456789":    time.Date(1970, 1, 1, 1, 0, 0, 123456789,
time.UTC),
        "3600.0":            time.Date(1970, 1, 1, 1, 0, 0, 0, time.UTC),
        "0.0":                time.Date(1970, 1, 1, 0, 0, 0, 0, time.UTC),
        "1.123456789":       time.Date(1970, 1, 1, 0, 0, 1, 123456789,
time.UTC),
        "1653403522.951205999": time.Date(2022, 5, 24, 14, 45, 22,

```

```

951205999, time.UTC),
    // "1653403522":          error,
}
for src, want := range samples {
    got, err := parseLegacyDate(src)
    if err != nil {
        t.Fatalf("%v: unexpected error", src)
    }
    if !got.Equal(want) {
        t.Fatalf("%v: got %v, want %v", src, got, want)
    }
}
}

```

до чего же убогий язык.....

14

Таймеры и тикеры

Кроме традиционных функций работы со временем, пакет `time` помогает в многозадачных программах. Мы уже разбирали эту часть в модуле «Многозадачность», так что не буду повторяться. Только перечислю кратко:

`[time.Timer](https://pkg.go.dev/time#Timer)` — таймер с каналом, в котором появится значение через продолжительность `d`.

`[time.After(d)](https://pkg.go.dev/time#After)` — канал, в котором появится значение через продолжительность `d`.

`[time.AfterFunc(d, f)](https://pkg.go.dev/time#AfterFunc)` — выполняет функцию `f` через продолжительность `d`.

`[time.Ticker](https://pkg.go.dev/time#Ticker)` — тикер с каналом, в котором появляются значения с периодичностью `d`.

`[time.Tick(d)](https://pkg.go.dev/time#Tick)` — канал, в котором появляются значения с периодичностью `d`.

И, конечно, есть `[time.Sleep(d)](https://pkg.go.dev/time#Sleep)` — думаю, она не требует пояснений.

Стандартная библиотека 3. Чтение и запись

Бета-версия урока. Оставляйте вопросы и замечания в комментариях.

В стандартной библиотеке Go полно инструментов I/O — чтения и записи данных. На этом уроке посмотрим на некоторые из них:

- четыре базовых способа чтения данных;
- пара способов записи;
- интерфейсы, которые лежат в основе I/O;
- пути, файлы и каталоги.

Плохая новость: в финале урока появляются рептилоиды. Хорошая новость: мы держим их под контролем.

os.ReadFile

Самый простой способ прочитать содержимое файла — воспользоваться функцией `os.ReadFile()`. Допустим, есть файл `answer.txt` с таким незамысловатым текстом:

```
42
```

Прочитаем его и напечатаем:

```
data, err := os.ReadFile("answer.txt")
if err != nil {
    panic(err)
}
fmt.Println(len(data), data)
// 2 [52 50]
fmt.Println(string(data))
// 42
```

`os.ReadFile()` возвращает содержимое файла как срез байт: `52` = `'4'`, `50` = `'2'`.

Если что-то пошло не так — вторым значением вернется ошибка:

```
data, err := os.ReadFile("no-such-file.txt")
if err != nil {
    panic(err)
}
fmt.Println(string(data))
```

```
panic: open no-such-file.txt: no such file or directory
```

`os.ReadFile()` подходит для одноразовых скриптов и совсем небольших файлов. В остальных случаях используют другие средства.

2

Строки из файла

Реализуйте функцию `readLines()`, которая читает все строки из файла и возвращает их в виде среза:

```
func readLines(name string) ([]string, error)
```

В файле строки разделены символом перевода строки `\n`, а в срезе должны быть уже без него. Если в файле последняя строка пустая (как принято в линуксе), она не попадает в срез.

Если файла не существует или его не удалось прочитать, возвращается ошибка (тип и текст на ваше усмотрение).

Полный код — в песочнице. Отправляйте в качестве решения только фрагмент, отмеченный комментариями «начало решения» и «конец решения».

[песочница](#)

Sample Input:

Sample Output:

PASS

```
package main

import (
    "fmt"
    "os"
    "strings"
)

// начало решения

// readLines возвращает все строки из указанного файла
func readLines(name string) ([]string, error) {
    data, err := os.ReadFile(name)
    if err != nil {
        return nil, err
    }
}
```

```

    if len(data) < 1 {
        return []string{}, nil
    } else {
        return strings.Split(string(data[:len(data)-1]), "\n"), nil
    }
}

// конец решения

/*
func readLines(name string) ([]string, error) {
    data, err := os.ReadFile(name)
    if err != nil {
        return nil, err
    }
    lines := strings.Split(string(data), "\n")
    if len(lines) > 0 && lines[len(lines)-1] == "" {
        lines = lines[:len(lines)-1]
    }
    return lines, nil
}
*/

func main() {
    lines, err := readLines("/etc/passwd")
    if err != nil {
        panic(err)
    }
    for idx, line := range lines {
        fmt.Printf("%d: %s\n", idx+1, line)
    }
}

```

3

os.File.Read

Обычно файлы считывают не одним куском, а *буфером* фиксированного размера. Для этого подойдет метод `os.File.Read()`.

Допустим, есть файл `awesome.txt` с таким содержимым:

```
go is awesome
```

Прочитаем его буфером размером в 5 байт:

```
file, err := os.Open("awesome.txt")    // (1)
if err != nil {
    panic(err)
}
defer file.Close()                     // (2)

buf := make([]byte, 5)                 // (3)
for {
    n, err := file.Read(buf)           // (4)
    fmt.Println(n, err)
    if err == io.EOF {                 // (5)
        break
    }
    if err != nil {
        panic(err)
    }
    fmt.Printf("read %d bytes: %q\n", n, buf[:n]) // (6)
}
```

`os.Open()` открывает файл для чтения и возвращает объект типа `File` ❶. Когда закончим работать с файлом, его необходимо закрыть, чтобы освободить занятые ресурсы ❷.

Мы используем буфер фиксированного размера в 5 байт ❸. Метод `File.Read()` считывает очередную порцию данных из файла в буфер ❹. Он возвращает количество считанных байт (обычно оно равно размеру буфера) и ошибку.

Если данные в файле закончились, `File.Read()` возвращает особое значение ошибки — `io.EOF` ❺. Мы ориентируемся на него, чтобы выйти из цикла.

Наконец, мы выводим `n` прочитанных байт на экран ❻.

Результат работы программы:

```
5 <nil>
read 5 bytes: "go is"
5 <nil>
read 5 bytes: " awes"
3 <nil>
read 3 bytes: "ome"
0 EOF
```

Как видите, `n` равно размеру буфера (5 байт) до тех пор, пока в файле есть по крайней мере 5 непрочитанных байт. В предпоследнем вызове `File.Read()` осталось только 3 байта (`"ome"`),

поэтому `n = 3`. Последний вызов `File.Read()` возвращает `n = 0` и ошибку `io.EOF`, поэтому буфер мы не печатаем, а выходим из цикла.

`File.Read()` демонстрирует распространенный подход: читаем данные в цикле буфером фиксированного размера, пока не получим `io.EOF`. Но на практике чаще используют не его, а другой похожий инструмент.

4

bufio.Reader

Тип `bufio.Reader` умеет читать данные как `os.File`, но делает это более эффективно. Внутри у него собственный буфер размера 4096 байт, так что `bufio.Reader` всегда сначала начитывает данные из файла в этот буфер, а затем уже отдает в методе `Read()`.

За счет этого получается меньше обращений к диску. Даже если мы читаем файл размера 100 Кб кусочками размером в 1 байт, то фактических обращений к диску получится только $100 * 1024 / 4096 + 1 = 26$ раз (+1 обращение нужно, чтобы отследить конец файла).

```
file, err := os.Open("awesome.txt")
if err != nil {
    panic(err)
}
defer file.Close()

reader := bufio.NewReader(file) // (1)
buf := make([]byte, 5)
for {
    n, err := reader.Read(buf)
    if err == io.EOF {
        break
    }
    if err != nil {
        panic(err)
    }
    fmt.Printf("read %d bytes: %q\n", n, buf[:n])
}
```

Как видите, программа почти не отличается от той, что использовала `os.File.Read()`. Разница только в том, что вместо объекта `os.File` мы используем `bufio.Reader`, созданный на его основе ❶.

Результат тоже не отличается:


```
read 5 bytes: "go is"  
read 5 bytes: " awes"  
read 3 bytes: "ome"
```

Но если `File.Read()` обращался к диску 4 раза, то `Reader.Read()` — только 2 (самый первый и последний, который вернет `io.EOF`). После первого вызова все содержимое файла поместилось во внутренний буфер, и следующие вызовы `Read()` возвращали данные уже оттуда.

`bufio.Reader` отлично подходит для чтения больших файлов. Кроме того, у него много приятных дополнительных возможностей:

- размер буфера настраивается через конструктор `NewReaderSize()`;
- можно подсмотреть следующие `n` байт через метод `Peek()` или пропустить их через `Discard()`;
- можно считать отдельный байт через `ReadByte()` или руну через `ReadRune()`;
- можно считать все байты до указанного разделителя через `ReadBytes()`;
- или аналогично считать строку до указанного разделителя через `ReadString()`.

А вот чего `bufio.Reader` не умеет в сравнении с `os.File` — так это перемещаться в произвольном порядке по файлу с помощью метода `Seek()`. Так что если вам зачем-то понадобится такая возможность — используйте `os.File`.

6

Строки из файла (снова)

Реализуйте функцию `readLines()`, которая читает все строки из файла и возвращает их в виде среза. Точно такую же, как в первом задании. Только на этот раз не используйте `os.ReadFile()`.

```
func readLines(name string) ([]string, error)
```

В файле строки разделены символом перевода строки `\n`, а в срезе должны быть уже без него. Если в файле последняя строка пустая (как принято в линуксе), она не попадает в срез.

Если файла не существует или его не удалось прочитать, возвращается ошибка (тип и текст на ваше усмотрение).

Полный код — в песочнице. Отправляйте в качестве решения только фрагмент, отмеченный комментариями «начало решения» и «конец решения».

[песочница](#)

Sample Input:

Sample Output:

7

io.Reader

На предыдущих шагах мы рассмотрели два инструмента порционного чтения данных в пакете `bufio`: `Reader` и `Scanner`. Глядя на текст примеров, можно предположить, что их конструкторы выглядят так:

```
func NewReader(file os.File) *Reader
func NewScanner(file os.File) *Scanner
```

А вот как они выглядят на самом деле:

```
func NewReader(rd io.Reader) *Reader
func NewScanner(rd io.Reader) *Scanner
```

И ридер, и сканер принимают на входе не файл, а нечто типа `io.Reader`. Это интерфейс с единственным методом `Read()`:

```
type Reader interface {
    Read(p []byte) (n int, err error)
}
```

`io.Reader` — это любая штука, которая умеет прочитать очередную порцию данных в переданный буфер:

- файл (как в предыдущих примерах);
- данные из сети;
- строка в памяти;
- что угодно еще.

Представим, например, что вместо файла `people.txt` с именами людей у нас есть строка с этими же данными:

```
text := "Дарья Борис Елена Ксения Леонид"
```

Если бы `bufio.Scanner` был жестко привязан к `os.File`, мы не смогли бы считать данные сканером из `text`. А благодаря зависимости от универсального интерфейса `io.Reader` — сможем:

```
text := "Дарья Борис Елена Ксения Леонид"

reader := strings.NewReader(text)    // (1)
scanner := bufio.NewScanner(reader)
```

```
scanner.Split(bufio.ScanWords)
for scanner.Scan() {
    word := scanner.Text()
    fmt.Println(word)
}
if err := scanner.Err(); err != nil {
    panic(err)
}
```

Все, что потребовалось — создать экземпляр `strings.Reader` ^❶, который предоставляет метод `Read()` для чтения из строки вместо файла. Остальной код остался без изменений. Удобно!

В Go множество конкретных типов, которые реализуют интерфейс `io.Reader`:

- `bytes.Reader` читает из среза байт;
- `gzip.Reader` читает из источника со сжатыми данными;
- `strings.Reader` читает из строки;
- ...

А также вспомогательных функций, которые возвращают `io.Reader`:

- `io.LimitedReader()` возвращает ридер, который читает не более N байт;
- `io.MultiReader()` возвращает ридер, который комбинирует несколько источников в один;
- ...

Если пишете собственные функции для чтения данных, принимайте на входе `io.Reader` вместо конкретного типа вроде `os.File` — так функцией можно будет воспользоваться для любого источника данных, а не только файла.

Что выбрать

Напоследок несколько эвристик, которые помогут выбрать между разными инструментами:

- `os.ReadFile()` — для маленьких файлов, которые хочется прочитать одним куском.
- `os.File` — если нужно перемещаться по файлу в произвольном порядке.
- `bufio.Reader` — для бинарных данных и текстовых данных сложной структуры.
- `bufio.Scanner` — для данных, значения в которых идут через разделитель.

Кроме того, `bufio.Reader` походит как базовый ридер для более сложных ридеров. `bufio.Scanner` не подходит, потому что не реализует `io.Reader`.

Бывает нижний регистр (`сорок два`), бывает верхний регистр (`СОРОК ДВА`), а бывает *регистр заголовка* (title case): `Сорок Два`. В этом задании работаем с ним.

Напишите программу, которая читает строчку из стандартного ввода (`os.Stdin`), приводит ее к регистру заголовка и печатает на экран. Слова в строке всегда разделены пробелами.

`os.Stdin` реализует `io.Reader`, так что вы знаете, как с ним обращаться.

В этом задании отправляйте программу целиком.

Если будете отлаживать локально, передать строку на вход можно через `echo` и пайп. Например, если исходный файл называется `title.go`:

```
echo "go is awesome" | go run title.go
```

Sample Input:

go is awesome

Sample Output:

Go Is Awesome

```
package main

import (
    "fmt"
    "strings"
    "bufio"
    "os"
)

func titleCase(s string) string {
    return strings.Title(strings.ToLower(s))
}

func main () {
    reader := bufio.NewReader(os.Stdin)
    scanner := bufio.NewScanner(reader)
    scanner.Split(bufio.ScanWords)                                // по пробелу

    for scanner.Scan() {
        word := scanner.Text()
        fmt.Printf("%s ", titleCase(word))
    }
}
```

```

    if err := scanner.Err(); err != nil {
        panic(err)
    }
}

/* // как вариант
func main() {
    r := bufio.NewReader(os.Stdin)
    s, _ := r.ReadString('\n')
    fmt.Print(strings.Title(strings.ToLower(s)))
}

func main() {
    scanner := bufio.NewScanner(os.Stdin)
    scanner.Split(bufio.ScanWords)
    for scanner.Scan() {
        rns := []rune(scanner.Text())
        rns[0] = rune(unicode.ToUpper(rns[0]))

        fmt.Printf("%v%v ", string(rns[0]), strings.ToLower(string(rns[1:])))
    }
}
*/

```

9

Случайный читатель

Реализуйте функцию `RandomReader()`:

```
func RandomReader(max int) io.Reader
```

Функция создает читателя, который возвращает случайные байты, но не более `max` штук.

Пример использования:

```

func main() {
    rand.Seed(0)

    rnd := RandomReader(5)
    rd := bufio.NewReader(rnd)
    for {
        b, err := rd.ReadByte()
        if err == io.EOF {

```

```

        break
    }
    if err != nil {
        panic(err)
    }
    fmt.Printf("%d ", b)
}
fmt.Println()
// 1 148 253 194 250
}

```

Для генерации случайных байт используйте `[rand.Read()]`
[\(https://pkg.go.dev/math/rand#Read\)](https://pkg.go.dev/math/rand#Read) .

Полный код — в песочнице. Отправляйте в качестве решения только фрагмент, отмеченный комментариями «начало решения» и «конец решения».

[песочница](#)

Sample Input:

Sample Output:

PASS

```

package main

import (
    "bufio"
    "fmt"
    "io"
    "math/rand"
    // "strings"
)

// начало решения
type rndReader struct {
    src []byte
    max int
    cur int
}

// Read implements the io.Reader interface.
func (r *rndReader) Read(b []byte) (n int, err error) {

```

```

    if r.cur >= r.max {
        return 0, io.EOF
    }
    n = copy(b, r.src[r.cur:])
    r.cur += n
    return
}

// RandomReader создает читателя, который возвращает случайные байты,
// но не более max штук
func RandomReader(max int) io.Reader {
    src := make([]byte, max)
    rand.Read(src)
    return &rndReader{src: src, max: max, cur: 0}
}

// конец ВООБЩЕ НЕПРАВИЛЬНОГО (?), но решения

/*
type randomReader struct{}

func (r *randomReader) Read(p []byte) (n int, err error) {
    return rand.Read(p)
}

func RandomReader(max int) io.Reader {
    rd := &randomReader{}
    return io.LimitReader(rd, int64(max))
}
*/

/*
type randomReader struct {
    max int
    n    int
}

func (r *randomReader) Read(p []byte) (n int, err error) {
    if r.n >= r.max {
        return 0, io.EOF
    }
    if len(p) > r.max {

```

```

        p = p[:r.max]
    }
    n, err = rand.Read(p)
    r.n += n
    return
}

func RandomReader(max int) io.Reader {
    return &randomReader{max: max, n: 0}
}
*/

func main() {
    rand.Seed(0)

    rnd := RandomReader(5)
    rd := bufio.NewReader(rnd)
    for {
        b, err := rd.ReadByte()
        if err == io.EOF {
            break
        }
        if err != nil {
            panic(err)
        }
        fmt.Printf("%d ", b)
    }
    fmt.Println()
    // 1 148 253 194 250
}

```

10

Запись данных

С записью данных ситуация похожа на чтение, поэтому изложу кратко.

os.WriteFile

Если данных немного, записать их можно в один заход с помощью `os.WriteFile()`:

```

people := []string{"Дарья", "Борис", "Елена", "Ксения", "Леонид"}
text := strings.Join(people, "\n")

err := os.WriteFile("people.txt", []byte(text), 0644)

```



```
if err != nil {
    panic(err)
}
```

Третий аргумент в `os.WriteFile()` задает права на доступ к файлу. `0644` означает, что записывать в этот файл сможет только владелец, а остальные пользователи — только читать. Распространенные наборы прав:

- `0600` — владелец читает и пишет, остальные не имеют доступа;
- `0644` — владелец читает и пишет, остальные только читают;
- `0666` — все читают и пишут.

bufio.Writer

Для порционной записи предусмотрен интерфейс `io.Writer` с методом `Write()`:

```
type Writer interface {
    Write(p []byte) (n int, err error)
}
```

Есть конкретные типы, которые его реализуют: `bufio.Writer`, `strings.Builder` (знакомый вам по уроку «Текст»), `bytes.Buffer` и другие.

Для записи в файл удобно использовать `bufio.Writer`. Он подходит для бинарных и текстовых данных:

```
file, err := os.Create("people.txt") // (1)
if err != nil {
    panic(err)
}

people := []string{"Дарья", "Борис", "Елена", "Ксения", "Леонид"}
writer := bufio.NewWriter(file)
for _, p := range people {
    writer.WriteString(p)           // (2)
    writer.WriteByte('\n')          // (3)
}

err = writer.Flush()               // (4)
if err != nil {
    file.Close()                   // (5)
    panic(err)
}

err = file.Close()                 // (6)
```

```
if err != nil {
    panic(err)
}
```

`os.Create()` создает файл для записи ❶.

Помимо стандартного метода `Write()`, у `bufio.Writer` есть методы для записи строк `WriteString()` ❷, отдельных байт `WriteByte()` ❸ и рун `WriteRune()`.

Метод `Flush()` ❹ принудительно записывает данные из внутреннего буфера на диск. Мы вызываем его, поскольку закончили работать с файлом. При ошибке закрываем файл, прежде чем паниковать ❺.

`file.Close()` может вернуть ошибку ❻. При записи в файл игнорировать ее не стоит — такая ошибка может сигнализировать, что часть данных не были записаны. Поэтому паникуем, если что-то пошло не так.

Чтобы дописать данные в существующий файл, используют `os.OpenFile()` с подходящими параметрами. Код для записи данных при этом не меняется:

```
file, err := os.OpenFile("people.txt", os.O_APPEND|os.O_CREATE|os.O_WRONLY,
0644)
if err != nil {
    panic(err)
}

people := []string{"Марина", "Иван", "Вероника"}
writer := bufio.NewWriter(file)
// дальше без изменений
```

`os.O_WRONLY` устанавливает, что в файл будем только писать, но не читать.

`os.O_APPEND|os.O_CREATE` говорят, что в файл следует дописывать, если он существует, иначе создать.

Аналога `bufio.Scanner` для записи данных нет, потому что `bufio.Writer` и так отлично справляется.

11

Писатель в никуда

Разработайте тип `AbyssWriter`, который:

- реализует интерфейс `io.Writer`,
- пишет данные в никуда,

- но при этом считает количество записанных байт.

Интерфейс:

```
// Write пишет данные в никуда
Write(p []byte) (n int, err error)

// Total возвращает общее количество записанных байт
Total() int
```

Пример использования:

```
func main() {
    r := strings.NewReader("go is awesome")
    w := NewAbyssWriter()
    written, err := io.Copy(w, r)
    if err != nil {
        panic(err)
    }
    fmt.Printf("written %d bytes\n", written)
    fmt.Println(written == int64(w.Total()))
}
```

Полный код — в песочнице. Отправляйте в качестве решения только фрагмент, отмеченный комментариями «начало решения» и «конец решения».

[песочница](#)

Sample Input:

Sample Output:

PASS

```
package main

import (
    "fmt"
    "io"
    "io/ioutil"
    "strings"
)

// начало решения

// AbyssWriter пишет данные в никуда,
```

```

// но при этом считает количество записанных байт
type AbyssWriter struct {
    w io.Writer
    tot int
}

// Writer is the interface that wraps the basic Write method.
func (e *AbyssWriter) Write(p []byte) (int, error) {
    n, err := e.w.Write(p)
    e.tot += n
    if err != nil {
        return n, err
    }
    if n != len(p) {
        return n, io.ErrShortWrite
    }
    return n, nil
}

// Total возвращает общее количество записанных байт
func (e *AbyssWriter) Total() int {
    return e.tot
}

// NewAbyssWriter создает новый AbyssWriter
func NewAbyssWriter() *AbyssWriter {
    return &AbyssWriter{w: ioutil.Discard, tot: 0}
}

// конец НЕКОРРЕКТНОГО решения

/*
// AbyssWriter пишет данные в никуда,
// но при этом считает количество записанных байт
type AbyssWriter struct {
    total int
}

// Write пишет данные в никуда
func (w *AbyssWriter) Write(p []byte) (n int, err error) {
    w.total += len(p)
    return len(p), nil
}

```

```

// Total возвращает общее количество записанных байт
func (w *AbyssWriter) Total() int {
    return w.total
}

// NewAbyssWriter создает новый AbyssWriter
func NewAbyssWriter() *AbyssWriter {
    return &AbyssWriter{}
}
*/
func main() {
    r := strings.NewReader("go is awesome")
    w := NewAbyssWriter()
    written, err := io.Copy(w, r)
    if err != nil {
        panic(err)
    }

    fmt.Printf("written %d bytes\n", written)
    fmt.Println(written == int64(w.Total()))
}

```

12

Фильтр токенов

Есть пара интерфейсов:

- `TokenReader` читает токены (строковые значения),
- `TokenWriter` пишет токены.

```

// TokenReader начитывает токены из источника
type TokenReader interface {
    // ReadToken считывает очередной токен
    // Если токенов больше нет, возвращает ошибку io.EOF
    ReadToken() (string, error)
}

// TokenWriter записывает токены в приемник
type TokenWriter interface {
    // WriteToken записывает очередной токен
    WriteToken(s string) error
}

```

Реализуйте функцию `FilterTokens()`, которая считывает все токены из источника и записывает в приемник тех из них, кто проходит проверку:

```
// FilterTokens читает все токены из src и записывает в dst тех,  
// кто проходит проверку predicate  
func FilterTokens(dst TokenWriter, src TokenReader, predicate func(s string)  
bool) (int, error)
```

Пример использования:

```
func main() {  
    // Для проверки придется создать конкретные типы,  
    // которые реализуют интерфейсы TokenReader и TokenWriter.  
  
    // Ниже для примера используются NewWordReader и NewWordWriter,  
    // но вы можете сделать любые на свое усмотрение.  
  
    r := NewWordReader("go is awesome")  
    w := NewWordWriter()  
    predicate := func(s string) bool {  
        return s != "is"  
    }  
    n, err := FilterTokens(w, r, predicate)  
    if err != nil {  
        panic(err)  
    }  
    fmt.Printf("%d tokens: %v\n", n, w.Words())  
    // 2 tokens: [go awesome]  
}
```

Полный код — в песочнице. Отправляйте в качестве решения только фрагмент, отмеченный комментариями «начало решения» и «конец решения».

[песочница](#)

Sample Input:

Sample Output:

PASS

```
/*Читаем токены из src, пока не закончатся.  
Если токен не проходит predicate, игнорируем его.  
Если проходит — пишем в dst.  
Попутно считаем количество записанных токенов.
```

```
func FilterTokens(dst TokenWriter, src TokenReader, predicate func(s string)
bool) (int, error) {
    total := 0
    for {
        token, err := src.ReadToken()
        if err == io.EOF {
            break
        }
        if err != nil {
            return total, err
        }

        if !predicate(token) {
            continue
        }

        err = dst.WriteToken(token)
        if err != nil {
            return total, err
        }
        total++
    }
    return total, nil
}*/
```

13

Пути

В путях файловой системы Windows использует обратные слешы (`Users\anton\hello.txt`), а остальные операционные системы — прямые (`Users/anton/hello.txt`). Пакет `path/filepath` помогает абстрагироваться от этих различий, чтобы писать кросс-платформенные программы.

`Join()` собирает путь из кусочков:

```
p := filepath.Join("Users", "anton", "hello.txt")
fmt.Println(p)
// Linux:   Users/anton/hello.txt
// Windows: Users\anton\hello.txt
```

Альтернативный вариант — всегда использовать прямой слеш и функцию `FromSlash()`:

```
p := filepath.FromSlash("Users/anton/hello.txt")
fmt.Println(p)
// Linux:  Users/anton/hello.txt
// Windows: Users\anton\hello.txt
```

Есть функции, которые получают отдельные участки пути:

```
dir := filepath.Dir(p)
fmt.Println(dir)
// Users/anton

base := filepath.Base(p)
fmt.Println(base)
// hello.txt

ext := filepath.Ext(p)
fmt.Println(ext)
// .txt

dir, file := filepath.Split(p)
fmt.Println(dir, file)
// Users/anton/ hello.txt
```

Есть функции, которые переводят относительный путь в абсолютный, и наоборот:

```
abs, err := filepath.Abs("hello.txt")
fmt.Println(abs, err)
// /Users/anton/hello.txt

rel, err := filepath.Rel("/Users/anton/", abs)
fmt.Println(rel, err)
// hello.txt
```

`Match()` проверяет, подходит ли путь под шаблон:

```
matched, err := filepath.Match("*.txt", "people.txt")
fmt.Println(matched, err)
// true <nil>
```

`Glob()` возвращает все пути, подходящие под шаблон:

```
paths, err := filepath.Glob("*.txt")
fmt.Println(paths, err)
// [answer.txt awesome.txt people.txt] <nil>
```

`WalkDir()` рекурсивно обходит файлы и каталоги, подчиненные заданному:


```

filepath.WalkDir(".", func(path string, d fs.DirEntry, err error) error {
    if err != nil {
        return err
    }
    if d.IsDir() {
        fmt.Println("d", path)
    } else {
        fmt.Println("f", path)
    }
    return nil
})

```

```

d .
f answer.txt
f awesome.txt
d code
f code/path-1.go
f code/path-2.go
f code/path-3.go
f code/path-4.go
f people.txt

```

14

Файлы и каталоги

Функции для управления файлами и каталогами находятся в пакете `os`.

Создать каталог

```

// создать каталог
os.Mkdir("stuff", 0755)

// в нем вложенные каталоги
p := filepath.Join("stuff", "deep", "ocean")
os.MkdirAll(p, 0755)

// и три файла
touch := func(path string) {
    p := filepath.FromSlash(path)
    data := []byte{}
    os.WriteFile(p, data, 0644)
}

touch("stuff/file-1.txt")

```

```
touch("stuff/file-2.txt")
touch("stuff/file-3.txt")
```

Прочитать содержимое каталога

```
// перейти в каталог
os.Chdir("stuff")

// прочитать содержимое каталога
entries, err := os.ReadDir(".")
if err != nil {
    panic(err)
}
for _, entry := range entries {
    fmt.Println(entry.Name())
}
```

```
deep
file-1.txt
file-2.txt
file-3.txt
```

Переименовать или переместить

```
// переименовать один из файлов
os.Rename("file-1.txt", "useless.txt")

// а другой переместить
os.Rename("file-2.txt", filepath.Join("deep", "file-2.txt"))
```

Удалить файл или каталог

```
// удалить один файл
os.Remove("useless.txt")

// удалить каталог со всем содержимым
os.RemoveAll("deep")
```

Временный файл

```
// каталог операционной системы для временных файлов
dir := os.TempDir()
fmt.Println(dir)

// создать временный файл по указанной маске
f, err := os.CreateTemp(dir, "file-*.")
if err != nil {
```

```
    panic(err)
}
fmt.Println(f.Name())

// работаем с файлом...

// удалить файл
os.Remove(f.Name())

/tmp/
/tmp/file-37870885
```

15

Перепись рептилоидов

Наконец-то ответственное задание! Предстоит переписать все население планеты Нибиру.

Для начала некоторые особенности жизни на Нибиру:

- у каждого рептилоида есть имя;
- имена могут повторяться;
- имена содержат только символы рептилоидного алфавита в нижнем регистре;
- алфавит состоит из букв `aeiourtnsl`.

Теперь правила переписи:

- Записи о рептилоидах хранятся в каталоге `census`, в отдельных файлах.
- Создается по одному файлу для каждой буквы алфавита.
- В каждом файле перечислены рептилоиды, чьи имена начинаются на соответствующую букву.
- Каждый рептилоид записан в отдельной строке.
- Если рептилоидов с одним и тем же именем несколько — в файле будет несколько одинаковых строк.

Ваша задача — реализовать логику переписи в типе `Census` в соответствии с правилами.

Вот как будет вызываться ваш код:

```
func main() {
    rand.Seed(0)
    census := NewCensus()
    defer census.Close()
    for i := 0; i < 1024; i++ {
        reptoid := randomName(5)
    }
}
```

```

        census.Add(reptoid)
    }
    fmt.Println(census.Count())
}

```

Порядок рептилоидов в файлах должен соответствовать порядку вызова `Add()`: если условный `aerio` был добавлен после условного `aarrt`, то и в файле `a.txt` он будет записан после.

Поскольку Степик не разрешает создавать файлы и каталоги, задачу придется решать локально. Шаблон решения — в песочнице. В качестве ответа укажите через пробел:

- количество записей в файле `census/n.txt` (рептилоиды на букву n),
- имя 42-го рептилоида в этом файле (считая с 1).

Например, если в `census/n.txt` 100 записей, а 42-го рептилоида зовут `nasir`, то ответ будет такой:

```
100 nasir
```

[песочница](#)

```

package main

import (
    "fmt"
    "math/rand"
    "os"
    "path/filepath"
    "bufio"
)

// алфавит планеты Нибиру
const alphabet = "aeiourtnsl"

// Census реализует перепись населения.
// Записи о рептилоидах хранятся в каталоге census, в отдельных файлах,
// по одному файлу на каждую букву алфавита.
// В каждом файле перечислены рептилоиды, чьи имена начинаются
// на соответствующую букву, по одному рептилоиду на строку.
type Census struct {
    files map[rune]string
}

// Count возвращает общее количество переписанных рептилоидов.
func (c *Census) Count() int {

```

```

    return 0
}

// Add записывает сведения о рептилоиде.
func (c *Census) Add(name string) {
    first_letter := []rune(name)[0]
    p := c.files[first_letter]

    file, err := os.OpenFile(p, os.O_APPEND|os.O_CREATE|os.O_WRONLY, 0644)
    if err != nil {
        panic(err)
    }

    writer := bufio.NewWriter(file)
    writer.WriteString(name)
    writer.WriteByte('\n')

    err = writer.Flush()
    if err != nil {
        file.Close()
        panic(err)
    }

    err = file.Close()
    if err != nil {
        panic(err)
    }
}

// Close закрывает файлы, использованные переписью.
func (c *Census) Close() {
    // os.RemoveAll("census")
}

// NewCensus создает новую перепись и пустые файлы
// для будущих записей о населении.
func NewCensus() *Census {
    os.Mkdir("census", 0755)

    touch := func(path string) {
        p := filepath.FromSlash(path)
        data := []byte{}
    }

```

```

    os.WriteFile(p, data, 0644)
}

files := make(map[rune]string)
for _, rune := range alphabet {
    p := filepath.Join("census", string(rune)+".txt")
    touch(p)
    files[rune] = p
}

return &Census{files: files}
}

// 

не меняйте код ниже этой строки


//

// randomName возвращает имя очередного рептилоида.
func randomName(n int) string {
    chars := make([]byte, n)
    for i := range chars {
        chars[i] = alphabet[rand.Intn(len(alphabet))]
    }
    return string(chars)
}

func main() {
    rand.Seed(0)
    census := NewCensus()
    defer census.Close()
    for i := 0; i < 1024; i++ {
        reptoid := randomName(5)
        census.Add(reptoid)
    }
    fmt.Println(census.Count())
}

```

96 niuir

Стандартная библиотека 4. JSON, XML, CSV

JSON как формат обмена данными стал популярен благодаря человекочитаемости и чрезвычайной простоте. Он поддерживает четыре примитивных типа (строка, число, true/false и null) и два составных (массив и объект).

В языках с динамической типизацией, вроде JavaScript и Python, работать с JSON одно удовольствие. В языках со статической типизацией, вроде Java и C# — сильно сложнее. Go, хоть он и статически типизирован, находится где-то посередине.

На этом уроке:

- как кодировать объекты в json и обратно;
- как использовать структурные теги;
- как извлечь из огромного json-документа только то, что интересно;
- как читать json-файлы поточно, не загружая целиком в память;
- как преобразовать xml в csv, не привлекая внимания санитаров.

Кодирование в JSON

Кодирование — это преобразование «родных» структур данных языка в строку (набор байт). В Go для кодирования используют термин «маршалинг» (marshal), а для декодирования — «анмаршалинг» (unmarshaling). Я буду использовать «кодирование» и «декодирование», чтобы не ломать язык.

За кодирование и декодирование JSON в Go отвечает пакет `encoding/json`.

Допустим, есть тип, который описывает человека:

```
type Person struct {
    Name      string
    Age       int
    Weight    float64
    IsAwesome bool
    secret    string
}
```

Преобразуем объект типа `Person` в JSON:

```
alice := Person{
    Name:      "Alice",
    Age:       25,
    Weight:    55.5,
    IsAwesome: true,
    secret:    "42",
}
```

```

}

b, err := json.Marshal(alice)
if err != nil {
    panic(err)
}
fmt.Println(string(b))
// {"Name": "Alice", "Age": 25, "Weight": 55.5, "IsAwesome": true}

```

`json.Marshal()` принимает значение произвольного типа `any`, кодирует его в JSON, и возвращает срез байт.

Структура кодируется в JSON-объект. При этом типы полей Go преобразует автоматически:

- поля типа `string` превращаются в строку;
- `int` и `float64` — в число;
- `bool` — в `true/false`.

Go игнорирует приватные поля структуры — те, что написаны со строчной буквы. Поэтому `secret` на выходе отсутствует.

Чтобы получить красивый JSON с отступами, используют `json.MarshalIndent()`:

```

b, err := json.MarshalIndent(alice, "", " ")
if err != nil {
    panic(err)
}
fmt.Println(string(b))

```

```

{
  "Name": "Alice",
  "Age": 25,
  "Weight": 55.5,
  "IsAwesome": true
}

```

На практике, конечно, в структурах встречаются не только строки, числа и логические значения. Сейчас посмотрим на более сложные случаи.

[песочница](#)

2

Определяемые типы

Добавим человеку дату рождения:

```
type Person struct {
    Name      string
    BirthDate time.Time
}
```

Проверим, как она закодируется:

```
date, _ := time.Parse("2006-01-02", "2000-05-25")
alice := Person{
    Name:      "Alice",
    BirthDate: date,
}

b, err := json.Marshal(alice)
fmt.Println(err, string(b))
// <nil> {"Name":"Alice","BirthDate":"2000-05-25T00:00:00Z"}
```

Вообще говоря, для типа `time.Time` нет соответствия в JSON. Несмотря на это, он вполне прилично закодировался — в строку по стандарту RFC 3339 (ISO 8601). Дело в том, что `time.Time` «знает», как преобразовать время в строку. Для этого он реализует интерфейс `Marshaler`:

```
// интерфейс
type Marshaler interface {
    MarshalJSON() ([]byte, error)
}

// реализация в time.Time
func (t Time) MarshalJSON() ([]byte, error) {
    // ...
    b := make([]byte, 0, len(RFC3339Nano)+2)
    b = append(b, '"')
    b = t.AppendFormat(b, RFC3339Nano)
    b = append(b, '"')
    return b, nil
}
```

`json.Marshal()` для каждого встреченного значения проверяет — реализует ли оно интерфейс `Marshaler`? Если да, преобразует с помощью `MarshalJSON()`. Так что если захотите какое-то хитрое преобразование в самописном типе — достаточно реализовать этот метод.

Карты и срезы

Срез автоматически превращается в JSON-массив:

```
nums := []int{1, 3, 5}
b, err := json.Marshal(nums)
fmt.Println(err, string(b))
// <nil> [1,3,5]
```

Карта — в JSON-объект:

```
m := map[string]int{
    "one": 1,
    "three": 3,
    "five": 5,
}
b, err := json.Marshal(m)
fmt.Println(err, string(b))
// <nil> {"five":5,"one":1,"three":3}
```

Неподдерживаемые типы

Некоторые типы — например, функции и каналы — вовсе не получится закодировать:

```
ch := make(chan int)
_, err := json.Marshal(ch)
fmt.Println(err)
// json: unsupported type: chan int

fn := func() int { return 42 }
_, err := json.Marshal(fn)
fmt.Println(err)
// json: unsupported type: func() int
```

[песочница](#)

3

Составные значения

Добавим человеку адрес:

```
type Address struct {
    Country string
    City    string
}

type Person struct {
```

```
    Name      string
    Residence Address
}
```

Проверим, как он закодируется:

```
paris := Address{"France", "Paris"}

alice := Person{
    Name:      "Alice",
    Residence: paris,
}

b, _ := json.MarshalIndent(alice, "", "    ")
fmt.Println(string(b))

{
    "Name": "Alice",
    "Residence": {
        "Country": "France",
        "City": "Paris"
    }
}
```

Логично: вложенная структура превратилась во вложенный JSON-объект.

[песочница](#)

Указатели

А вот вложенный указатель на структуру:

```
type Person struct {
    Name      string
    Residence *Address
}
```

Он дает на выходе точно такой же JSON, потому что `json.Marshal()` проходит по указателю и кодирует полученное значение:

```
{
    "Name": "Alice",
    "Residence": {
        "Country": "France",
        "City": "Paris"
    }
}
```

```
}  
}
```

Пустой указатель (`nil`) на выходе превращается в `null`:

```
emma := Person{  
    Name: "Emma",  
}  
b, _ = json.Marshal(emma)  
fmt.Println(string(b))  
// {"Name":"Emma","Residence":null}
```

Разыменованние указателей работает и в более сложных случаях. Например, если добавить человеку друзей:

```
type Person struct {  
    Name    string  
    Friends []*Person  
}
```

То они превратятся в массив объектов:

```
emma := Person{Name: "Emma"}  
grace := Person{Name: "Grace"}  
  
alice := Person{  
    Name:    "Alice",  
    Friends: []*Person{&emma, &grace},  
}  
  
b, _ := json.MarshalIndent(alice, "", "  ")  
fmt.Println(string(b))
```

```
{  
  "Name": "Alice",  
  "Friends": [  
    {  
      "Name": "Emma",  
      "Friends": null  
    },  
    {  
      "Name": "Grace",  
      "Friends": null  
    }  
  ]  
}
```

```
]
}
```

[песочница](#)

Думаю, общую логику вы уловили. Если понадобятся нюансы — они в документации на `[json.Marshal()]` (<https://pkg.go.dev/encoding/json#Marshal>).

4

Фильм в JSON

Есть структура с информацией о фильме:

```
type Movie struct {
    // ...
}
```

Требуется преобразовать ее в JSON такого вида:

```
{
  "Title": "Sully",
  "Year": 2016,
  "Director": "Clint Eastwood",
  "Genres": ["Drama", "History"],
  "Duration": "1h36m",
  "Rating": "★★★★☆"
}
```

- название фильма — строка;
- год выхода — число;
- режиссер — строка;
- жанры — массив строк;
- продолжительность — строка вида XhYm;
- рейтинг — строка с количеством звезд от 0 до 5.

Напишите функцию, которая кодирует фильмы в JSON-массив (каждый элемент массива соответствует одному фильму):

```
// MarshalMovies кодирует фильмы в JSON.
// - если indent = 0 - использует json.Marshal
// - если indent > 0 - использует json.MarshalIndent
//   с отступом в указанное количество пробелов.
func MarshalMovies(indent int, movies ...Movie) (string, error) {
```

```
// ...  
}
```

Поля в структуре `Movie` назовите так же, как они называются в JSON. Используйте предложенные типы `Duration` и `Rating` для продолжительности и рейтинга фильма.

Правила кодирования продолжительности фильма:

- строка вида `XhYm`, где `X` — количество часов, а `Y` — количество минут (`2h15m`);
- если часов 0, то они опускаются (не `0h45m`, а `45m`);
- если минут 0, то они опускаются (не `2h0m`, а `2h`).

Правила кодирования рейтинга:

- рейтинг принимает значения от 0 до 5;
- выходная строка всегда состоит из 5 звезд;
- из них заполнены `X` звезд, где `X` — значение рейтинга (`3 = ★★★☆☆`).

Полный код — в песочнице. Отправляйте в качестве решения только фрагмент, отмеченный комментариями «начало решения» и «конец решения».

[песочница](#)

Sample Input:

Sample Output:

PASS

```
package main  
  
import (  
    "encoding/json"  
    "fmt"  
    "time"  
)  
  
// начало решения  
  
// Duration описывает продолжительность фильма  
type Duration time.Duration  
  
func (d Duration) MarshalJSON() ([]byte, error) {  
    d1 := time.Duration(d)  
    h := int64(d1.Hours())
```

```

m := int64((d1 - time.Duration(h*int64(time.Hour))).Minutes())

if h == 0 {
    return []byte(fmt.Sprintf(`"%dm"`, m)), nil
} else if m == 0 {
    return []byte(fmt.Sprintf(`"%dh"`, h)), nil
} else {
    return []byte(fmt.Sprintf(`"%dh%dm"`, h, m)), nil
}
}

// Rating описывает рейтинг фильма
type Rating int

func (r Rating) MarshalJSON() ([]byte, error) {
    // ★★
    rt := []rune{}
    for i := 0; i < 5; i++ {
        if i < int(r) {
            rt = append(rt, '★')
        } else {
            rt = append(rt, '☆')
        }
    }
    return []byte(fmt.Sprintf(`"%s"`, string(rt))), nil
}

// Movie описывает фильм
type Movie struct {
    Title    string    // "Interstellar",
    Year     int       // 2014,
    Director string    // "Christopher Nolan",
    Genres   []string  // []string{"Adventure", "Drama", "Science Fiction"},
    Duration Duration // Duration(2*time.Hour + 49*time.Minute),
    Rating   Rating    // 5,
}

// MarshalMovies кодирует фильмы в JSON.
// - если indent = 0 - использует json.Marshal
// - если indent > 0 - использует json.MarshalIndent
//   с отступом в указанное количество пробелов.
func MarshalMovies(indent int, movies ...Movie) (string, error) {
    m := []Movie{}

```

```

for _, mv := range movies {
    m = append(m, mv)
}

if indent == 0 {
    b, err := json.Marshal(m)
    return string(b), err
} else {
    ind := []byte{}
    for i := 0; i < indent; i++ {
        ind = append(ind, ' ')
    }
    b, err := json.MarshalIndent(m, "", string(ind))
    return string(b), err
}
}

// конец решения

/*
// Duration описывает продолжительность фильма
type Duration time.Duration

func (d Duration) MarshalJSON() ([]byte, error) {
    str := time.Duration(d).String()
    if strings.HasSuffix(str, "m0s") {
        str = str[:len(str)-2]
    }
    if strings.HasSuffix(str, "h0m") {
        str = str[:len(str)-2]
    }
    b := make([]byte, 0, len(str)+2)
    b = append(b, '"')
    b = append(b, []byte(str)...)
    b = append(b, '"')
    return b, nil
}

// Rating описывает рейтинг фильма
type Rating int

func (r Rating) MarshalJSON() ([]byte, error) {
    str := strings.Repeat("★", int(r)) + strings.Repeat("☆", 5-int(r))

```



```

    b := make([]byte, 0, 7)
    b = append(b, '"')
    b = append(b, []byte(str)...)
    b = append(b, '"')
    return b, nil
}

// Movie описывает фильм
type Movie struct {
    Title    string
    Year     int
    Director string
    Genres   []string
    Duration Duration
    Rating   Rating
}

// MarshalMovies кодирует фильмы в JSON.
// Если indent > 0 - форматирует с отступом в указанное количество пробелов.
func MarshalMovies(indent int, movies ...Movie) (string, error) {
    var b []byte
    var err error
    if indent <= 0 {
        b, err = json.Marshal(movies)
    } else {
        padding := strings.Repeat(" ", indent)
        b, err = json.MarshalIndent(movies, "", padding)
    }
    if err != nil {
        return "", nil
    }
    return string(b), nil
}

*/

func main() {
    m1 := Movie{
        Title:    "Interstellar",
        Year:     2014,
        Director: "Christopher Nolan",
        Genres:   []string{"Adventure", "Drama", "Science Fiction"},
        Duration: Duration(0*time.Hour + 49*time.Minute),
        Rating:   5,
    }
}

```

```

}
m2 := Movie{
    Title:    "Sully",
    Year:     2016,
    Director: "Clint Eastwood",
    Genres:   []string{"Drama", "History"},
    Duration: Duration(time.Hour + 0*time.Minute),
    Rating:   4,
}

```

```

b, err := MarshalMovies(4, m1, m2)
fmt.Println(err)
// nil
fmt.Println(string(b))
/*
    [
        {
            "Title": "Interstellar",
            "Year": 2014,
            "Director": "Christopher Nolan",
            "Genres": [
                "Adventure",
                "Drama",
                "Science Fiction"
            ],
            "Duration": "2h49m",
            "Rating": "*****"
        },
        {
            "Title": "Sully",
            "Year": 2016,
            "Director": "Clint Eastwood",
            "Genres": [
                "Drama",
                "History"
            ],
            "Duration": "1h36m",
            "Rating": "★★★★☆"
        }
    ]
*/
}

```

Теги

Как видите, Go справляется с кодированием без какой-либо настройки: передали структуру `Person` в `json.Marshal()` — получили в ответ JSON с полями как у структуры:

```
{
  "Name": "Alice",
  "Age": 25,
  "Weight": 55.5,
  "IsAwesome": true
}
```

Но что делать, если хочется поменять состав или написание полей, не трогая исходный тип? Для этого в Go предусмотрены *теги* (tag), в которых указывают всякую дополнительную информацию о полях структуры:

```
type Person struct {
    Name      string `json:"name"`
    Age       int    `json:"age"`
    Weight    float64 `json:"- "`
    IsAwesome bool   `json:"is_awesome"`
}
```

Если создать объект типа `Person`, то прямого доступа к его тегам не будет. Так что в «обычном» коде теги вам вряд ли пригодятся.

Но теги можно получить через механизм рефлексии (пакет `reflect` дает полный доступ к типам объектов), что и делает `json.Marshal()`. Кодируя поле, он смотрит на тег с названием `json` и меняет результат в соответствии с тегом. В нашем случае:

- `Name` превращается в `name`,
- `Age` превращается в `age`,
- `Weight` вообще игнорируется,
- `IsAwesome` превращается в `is_awesome`.

```
alice := Person{
    Name:      "Alice",
    Age:       25,
    Weight:    55.5,
    IsAwesome: true,
}
```

```
b, err := json.MarshalIndent(alice, "", "    ")
if err != nil {
    panic(err)
}
fmt.Println(string(b))
```

```
{
    "name": "Alice",
    "age": 25,
    "is_awesome": true
}
```

Удобно!

[песочница](#)

6

Декодирование JSON

Декодирование похоже на кодирование, только функция отличается.

Допустим, есть у нас исходный JSON:

```
{
    "name": "Alice",
    "is_awesome": true,
    "residence": {
        "country": "France",
        "city": "Paris"
    },
    "friends": [
        { "name": "Emma" },
        { "name": "Grace" }
    ]
}
```

Проверим корректность с помощью `json.Valid()`:

```
src := `{...}`
fmt.Println(json.Valid([]byte(src)))
// true
```

Создадим подходящие типы:

```

type Address struct {
    Country string
    City    string
}

type Person struct {
    Name      string
    IsAwesome bool `json:"is_awesome"`
    Residence Address
    Friends   []*Person
}

```

И декодируем в них:

```

src := `{...}`

var alice Person
err := json.Unmarshal([]byte(src), &alice)

fmt.Println(err, alice)
// <nil> {Alice true {France Paris} [0x1400010c0a0 0x1400010c0f0]}

fmt.Println(alice.Friends[0])
// &{Emma { } []}

fmt.Println(alice.Friends[1])
// &{Grace { } []}

```

`json.Unmarshal()` принимает срез байт с исходным JSON и ссылку на переменную, в которую поместит результат. Как видите, поля-значения `Name`, `IsAwesome` и `Residence` заполнились подходящими значениями, а срез `Friends` — указателями на объекты `Person`.

Хотя в исходном JSON названия полей в нижнем регистре, нам не пришлось прописывать теги — за исключением `IsAwesome`, которое отличается сильнее. `json.Unmarshal()` достаточно умна, чтобы сопоставить поля и так.

[песочница](#)

Определяемые типы

Чтобы задать собственную логику декодирования из JSON, тип должен реализовать интерфейс

`json.Unmarshaler`:

```

type Unmarshaler interface {
    UnmarshalJSON([]byte) error
}

```

```
}
```

Допустим, мы решили сделать тип `AncientNumber`, который записывает число как строку по заветам мудрых предков:

```
var n AncientNumber

err := json.Unmarshal([]byte("1"), &n)
fmt.Println(err, n)
// <nil> one

err = json.Unmarshal([]byte("2"), &n)
fmt.Println(err, n)
// <nil> two

err = json.Unmarshal([]byte("42"), &n)
fmt.Println(err, n)
// <nil> many

err = json.Unmarshal([]byte("-1"), &n)
fmt.Println(err, n)
// <nil> impossible!
```

Логика декодирования может быть такой:

```
type AncientNumber string

func (an *AncientNumber) UnmarshalJSON(data []byte) error {
    // Go рекомендует игнорировать значения null
    if string(data) == "null" {
        return nil
    }
    // декодируем исходное число
    var n int
    err := json.Unmarshal(data, &n)
    if err != nil {
        return err
    }
    // преобразуем в значение типа AncientNumber
    switch {
    case n <= 0:
        *an = "impossible!"
    case n == 1:
        *an = "one"
    }
```

```
case n == 2:
    *an = "two"
case n > 2:
    *an = "many"
}
return nil
}
```

[песочница](#)

7

Выборочные поля

Бывает, получили вы ответ от какого-нибудь внешнего API, а там объект на 200 полей, из которых вам интересны только 2. Это что же теперь, все двести полей в структуру заводить ради двух значений? К счастью, нет — достаточно перечислить только интересные вам поля:

```
type Person struct {
    Name string
}

src := `{
    "name": "Alice",
    "is_awesome": true,
    "residence": {
        "country": "France",
        "city": "Paris"
    },
    "friends": [
        { "name": "Emma" },
        { "name": "Grace" }
    ]
}`

var alice Person
err := json.Unmarshal([]byte(src), &alice)
fmt.Println(err, alice)
// <nil> {Alice}
```

`json.Unmarshal()` игнорирует все поля, кроме перечисленных в структуре. Очень удобно в работе с развесистыми JSON-объектами.

[песочница](#)

Декодирование в карту

Можно обойтись и вовсе без отдельного типа. Если передать в `json.Unmarshal()` ссылку на значение типа `any` — оно заполнится картой, структура которой в точности соответствует исходному JSON (или срезом, если на входе был массив):

```
src := `{...}`

var alice any
err := json.Unmarshal([]byte(src), &alice)

fmt.Printf("%v %T\n", err, alice)
// <nil> map[string]interface {}

fmt.Println(alice)
// map[friends:[map[name:Emma] map[name:Grace]] is_awesome:true name:Alice
residence:map[city:Paris country:France]]
```

Может показаться, что это очень удобно. На самом деле — нет. Работая с картой, придется на каждом шаге заниматься преобразованием типов:

```
m := alice.(map[string]any)

name := m["name"].(string)
fmt.Printf("Name '%s' is %d letters long\n", name, len(name))
// Name 'Alice' is 5 letters long

if m["is_awesome"].(bool) {
    fmt.Println("Alice is awesome")
} else {
    fmt.Println("Alice is okay")
}
// Alice is awesome

friends := m["friends"].([]any)
for idx, friend := range friends {
    m := friend.(map[string]any)
    name := m["name"].(string)
    fmt.Printf("- friend #%d: %s\n", idx+1, name)
}
// - friend #1: Emma
// - friend #2: Grace
```

Так что я за конкретные типы вместо `map[string]any`.

8

Фильм из JSON

Есть JSON-объект с информацией о фильме:

```
{
  "name": "Interstellar",
  "released_at": 2014,
  "director": "Christopher Nolan",
  "tags": [
    { "name": "Adventure" },
    { "name": "Drama" },
    { "name": "Science Fiction" }
  ],
  "duration": "2h49m",
  "rating": "*****"
}
```

Требуется извлечь его в структуру:

```
// Genre описывает жанр фильма
type Genre string

// Movie описывает фильм
type Movie struct {
  Title  string
  Year   int
  Genres []Genre
}
```

Дополните описание типов и добавьте недостающую логику, чтобы `json.Unmarshal()` корректно декодировал JSON-объект в значение типа `Movie`:

```
func main() {
  const src = `{
    "name": "Interstellar",
    "released_at": 2014,
    "director": "Christopher Nolan",
    "tags": [
      { "name": "Adventure" },
      { "name": "Drama" },
      { "name": "Science Fiction" }
    ]
  }`
```

```

    ],
    "duration": "2h49m",
    "rating": "*****"
  }`

  var m Movie
  err := json.Unmarshal([]byte(src), &m)
  fmt.Println(err)
  // nil
  fmt.Println(m)
  // {Interstellar 2014 [Adventure Drama Science Fiction]}
}

```

Обратите внимание, что `Genre` — не структура, а производный от строки тип. Не меняйте типы и названия полей в `Movie`. Не добавляйте новые поля.

Полный код — в песочнице. Отправляйте в качестве решения только фрагмент, отмеченный комментариями «начало решения» и «конец решения».

[песочница](#)

Sample Input:

Sample Output:

PASS

```

package main

import (
    "encoding/json"
    "fmt"
)

// начало решения

// Genre описывает жанр фильма
type Genre string

// Movie описывает фильм
type Movie struct {
    Title  string `json:"name"`
    Year   int    `json:"released_at"`
    Genres []Genre `json:"tags"`
}

```

```

}

func (g *Genre) UnmarshalJSON(data []byte) error {
    // Go рекомендует игнорировать значения null
    if string(data) == "null" {
        return nil
    }

    // декодируем исходную строку через карту с приведением типа
    var gnr any
    err := json.Unmarshal(data, &gnr)
    if err != nil {
        fmt.Println("err")
        return err
    }

    m := gnr.(map[string]any)

    name := m["name"].(string)
    *g = Genre(name)
    return nil
}

```

// конец решения

/ // более красиво*

```

func (g *Genre) UnmarshalJSON(data []byte) error {
    if string(data) == "null" {
        return nil
    }
    var obj map[string]string
    if err := json.Unmarshal(data, &obj); err != nil {
        return err
    }
    if val, ok := obj["name"]; ok {
        *g = Genre(val)
    }
    return nil
}
*/

```

```

func main() {

```

```
const src = `{
    "name": "Interstellar",
    "released_at": 2014,
    "director": "Christopher Nolan",
    "tags": [
        { "name": "Adventure" },
        { "name": "Drama" },
        { "name": "Science Fiction" }
    ],
    "duration": "2h49m",
    "rating": "*****"
}`

var m Movie
err := json.Unmarshal([]byte(src), &m)
fmt.Println(err)
// nil
fmt.Println(m)
// {Interstellar 2014 [Adventure Drama Science Fiction]}
```

9

Поточное декодирование

В логах и анализе данных встречается формат [JSON Lines](#) — это когда на каждой строке записан полноценный JSON-объект:

```
{ "name": "Alice", "age": 25 }
{ "name": "Emma", "age": 23 }
{ "name": "Grace", "age": 27 }
```

Формат особенно удобен для больших файлов: можно читать записи по одной, не загружая все содержимое файла в память.

Чтобы преобразовать строки в структуры Go, можно пройти по файлу каким-нибудь `bufio.Scanner` и декодировать каждую строку через `json.Unmarshal()`. А можно использовать готовый инструмент — `json.Decoder`:

```
type Person struct {
    Name string
    Age  int
}

f, err := os.Open("people.json")
```

```

if err != nil {
    panic(err)
}
defer f.Close()

r := bufio.NewReader(f)
dec := json.NewDecoder(r) // (1)
for {
    var person Person
    err := dec.Decode(&person) // (2)
    if err == io.EOF {
        break
    }
    if err != nil {
        panic(err)
    }
    fmt.Println(person) // (3)
}

```

```

{Alice 25}
{Emma 23}
{Grace 27}

```

Конструктор `json.NewDecoder()` принимает на входе источник вида `io.Reader` и возвращает декодер ❶. Метод `Decode()` считывает очередное JSON-значение из источника и записывает его по переданному указателю ❷. В результате на каждой итерации цикла в `person` попадает очередной объект ❸.

Поскольку декодер создается на основе `io.Reader`, он может использовать любые источники — строки, stdin, файлы, сеть. Этим он отличается от `json.Unmarshal()`, который работает только со срезом байт. Но как мы увидим дальше, это не единственное отличие.

[песочница](#)

Декодирование JSON-документа

Если бы данные всегда приходили в формате JSON Lines, работать с ними было бы легко и приятно. Но чаще встречаются обычные JSON-документы.

Например, такой файл:

```

[
  {
    "name": "Alice",
    "age": 25
  }
]

```

```
    },  
    {  
        "name": "Emma",  
        "age": 23  
    },  
    {  
        "name": "Grace",  
        "age": 27  
    }  
]  
]
```

Его уже не прочитаешь построчно: интересующие нас объекты «размазаны» по всему файлу. Но ведь декодер должен с этим справиться? Давайте проверим:

```
f, err := os.Open("people.json")  
// ...  
  
r := bufio.NewReader(f)  
dec := json.NewDecoder(r)  
for {  
    var person Person  
    err := dec.Decode(&person)  
    // ...  
    fmt.Println(person)  
}
```

Результат — ошибка:

```
panic: json: cannot unmarshal array into Go value of type main.Person
```

Декодер не в состоянии прочитать элементы массива (объекты `Person`). Только весь массив целиком:

```
f, err := os.Open("people.json")  
// ...  
  
r := bufio.NewReader(f)  
dec := json.NewDecoder(r)  
for {  
    var val any  
    err := dec.Decode(&val)  
    // ...  
    fmt.Println(val)  
}  
  
// [map[age:25 name:Alice] map[age:23 name:Emma] map[age:27 name:Grace]]
```

Проблема в логике работы `Decode()`. Он всегда начитывает следующее JSON-значение с *текущей позиции*. В начале работы цикла текущая позиция — это открывающая скобка массива `[`. Относительно нее следующее JSON-значение — это весь массив целиком. Его и начитывает `Decode()`.

Способ читать JSON-документы поточно — существует. Но прежде чем мы к нему перейдем, посмотрим на декодер более пристально.

[песочница](#)

10

Декодер и токены

Декодер умеет считывать не только полные JSON-значения, но и отдельные *токены* (token). Проще всего показать на примере.

Вот наш JSON-документ:

```
[
  {
    "name": "Alice",
    "age": 25
  },
  {
    "name": "Emma",
    "age": 23
  },
  {
    "name": "Grace",
    "age": 27
  }
]
```

Прочитаем и напечатаем все токены в нем:

```
f, err := os.Open("people.json")
// ...

dec := json.NewDecoder(bufio.NewReader(f))
for {
    tok, err := dec.Token()
    // ...
    fmt.Printf("%T: %v", tok, tok)
    if dec.More() {
```

```

        fmt.Print("...")
    }
    fmt.Print("\n")
}

```

Метод `Token()` начитывает очередной токен из источника. Токенами считаются названия полей, атомарные значения (строки, числа, true/false и null), а также разделители `[] { }`. Запятые и двоеточия игнорируются.

Результат:

```
json.Delim: [...
```

```
json.Delim: {...
```

```
string: name...
```

```
string: Alice...
```

```
string: age...
```

```
float64: 25
```

```
json.Delim: }...
```

```
json.Delim: {...
```

```
string: name...
```

```
string: Emma...
```

```
string: age...
```

```
float64: 23
```

```
json.Delim: }...
```

```
json.Delim: {...
```

```
string: name...
```

```
string: Grace...
```

```
string: age...
```

```
float64: 27
```

```
json.Delim: }
```

```
json.Delim: ]
```

(пустые строки я добавил для наглядности)

Метод `More()` сообщает, есть ли еще значения внутри объекта или массива, в котором находится текущая позиция декодера. Несколько примеров (текущая позиция отмечена знаками `^`):

```

[
  {

```



```

    "name": "Alice",
        ^^^^^^
    "age": 25
},
...
]

```

Декодер находится внутри объекта Алисы. После "Alice" есть еще "age", так что `More() == true`.

```

[
  {
    "name": "Alice",
    "age": 25
    ^^^^^
  },
  ...
]

```

Декодер находится внутри объекта Алисы. После "age" есть еще 25, так что `More() == true`.

```

[
  {
    "name": "Alice",
    "age": 25
        ^^
  },
  ...
]

```

Декодер находится внутри объекта Алисы. После 25 значений нет, так что `More() == false`.

```

[
  {
    "name": "Alice",
    "age": 25
  },
  ^
  {
    "name": "Emma",
    "age": 23
  },
  ...
]

```

Декодер вышел из объекта Алисы и теперь находится внутри массива верхнего уровня. После объекта Алисы есть еще объект Эммы, так что `More() == true`.

И так далее.

Сочетание методов `Token()` и `More()` поможет нам прочитать JSON-документ поточно. Давайте посмотрим, как.

[песочница](#)

11

Поточное декодирование (снова)

Давайте прочитаем объекты из нашего JSON-документа поточно, не загружая все содержимое в память:

```
[
  {
    "name": "Alice",
    "age": 25
  },
  {
    "name": "Emma",
    "age": 23
  },
  {
    "name": "Grace",
    "age": 27
  }
]
```

Идея следующая:

- В самом начале сдвинем позицию декодера с `[` на ближайшую `{`, чтобы следующим значением для декодирования стал объект Алисы, а не весь массив.
- Декодируем Алису, тем самым передвинув текущую позицию на Эмму.
- Декодируем Эмму, тем самым передвинув текущую позицию на Грейс.
- И так далее, пока в массиве верхнего уровня остались еще объекты (в нашем случае на Грейс все и закончится).

```
f, err := os.Open("people.json")
// ...
```

```

dec := json.NewDecoder(bufio.NewReader(f))

// сдвигаем декодер на Алису
if _, err := dec.Token(); err != nil {
    panic(err)
}

var person Person

// пока в массиве остались еще объекты
for dec.More() {
    // декодируем очередной объект
    err := dec.Decode(&person)
    if err == io.EOF {
        break
    }
    if err != nil {
        panic(err)
    }
    fmt.Println(person)
}

```

```

{Alice 25}
{Emma 23}
{Grace 27}

```

Уфффф. Насколько все же было проще с JSON Lines.

[песочница](#)

Поточное кодирование

Раз есть поточное декодирование, должно быть и кодирование. За него отвечает тип `json.Encoder`. Он принимает выходной поток типа `io.Writer` и пишет в него через метод `Encode()`:

```

type Person struct {
    Name string `json:"name"`
    Age  int    `json:"age"`
}

func main() {
    people := []Person{
        {"Alice", 25},
        {"Emma", 23},
    }
}

```

```

        {"Grace", 27},
    }

    f, err := os.Create("people.jl")
    // ...

    w := bufio.NewWriter(f)
    enc := json.NewEncoder(w)
    for _, person := range people {
        err := enc.Encode(person)
        if err != nil {
            panic(err)
        }
    }

    // ...
}

```

Метод `Encode()` кодирует переданное ему значение в JSON и записывает в выходной поток. Еще он автоматически дописывает после значения символ перевода строки `\n`. В результате файл `people.jl` получится таким:

```

{"name": "Alice", "age": 25}
{"name": "Emma", "age": 23}
{"name": "Grace", "age": 27}

```

[песочница](#)

12

Почтовый фильтр

Давайте напишем фильтр, который отделяет «хорошие» письма от «плохих» (например, спама).

Есть источник с метаданными о письмах. Каждая строчка — информация об отдельном письме:

```

{ "from": "alice@go.dev",      "to": "zet@php.net",      "subject":
  "How are you?" }
{ "from": "bob@temp-mail.org", "to": "yolanda@java.com",      "subject":
  "Re: Indonesia" }
{ "from": "cindy@go.dev",      "to": "xavier@rust-lang.org",      "subject":
  "Go vs Rust" }
{ "from": "diane@dart.dev",     "to": "wanda@typescriptlang.org", "subject":
  "Our crypto startup" }

```

Есть приемник, в который должны попадать только хорошие письма.

И есть функция `FilterEmails()`, которая считывает все письма из источника и записывает в приемник тех из них, кто проходит проверку:

```
// Email описывает письмо
type Email struct {
    // ...
}

// FilterEmails читает все письма из src и записывает в dst тех,
// кто проходит проверку predicate
func FilterEmails(dst io.Writer, src io.Reader, predicate func(e Email)
bool) (int, error) {
    // ...
}
```

Пример: отсеиваем письма, в теме которых упоминается крипто:

```
func main() {
    src := strings.NewReader(`...`)
    dst := os.Stdout

    predicate := func(email Email) bool {
        return !strings.Contains(email.Subject, "crypto")
    }

    n, err := FilterEmails(dst, src, predicate)
    if err != nil {
        panic(err)
    }
    fmt.Println(n, "good emails")

    // {"from":"alice@go.dev","to":"zet@php.net","subject":"How are you?"}
    // {"from":"bob@temp-mail.org","to":"yolanda@java.com","subject":"Re:
Indonesia"}
    // {"from":"cindy@go.dev","to":"xavier@rust-lang.org","subject":"Go vs
Rust"}
    // 3 good emails
}
```

Дополните тип `Email` и реализуйте логику `Filter` через поточное декодирование и кодирование.

Полный код — в песочнице. Отправляйте в качестве решения только фрагмент, отмеченный комментариями «начало решения» и «конец решения».

[песочница](#)

Sample Input:

Sample Output:

PASS

```
package main

import (
    "encoding/json"
    "fmt"
    "io"
    "os"
    "strings"
)

// начало решения

// Email описывает письмо
type Email struct {
    From    string    `json:"from"`
    To      string    `json:"to"`
    Subject string    `json:"subject"`
}

// FilterEmails читает все письма из src и записывает в dst тех,
// кто проходит проверку predicate
func FilterEmails(dst io.Writer, src io.Reader, predicate func(e Email)
bool) (int, error) {
    dec := json.NewDecoder(src) // str -> obj
    enc := json.NewEncoder(dst) // obj -> str
    counter := 0

    for {
        var mail Email
        err := dec.Decode(&mail)
        if err == io.EOF {
            break
        }
    }
}
```

```

        if err != nil {
            return counter, err
        }

        if !predicate(mail) {
            continue
        }

        err = enc.Encode(mail)
        if err != nil {
            return counter, err
        }
        counter++
    }
    return counter, nil
}

```

// конец решения

```

func main() {
    src := strings.NewReader(`
        { "from": "alice@go.dev",      "to": "zet@php.net",
"subject": "How are you?" }
        { "from": "bob@temp-mail.org", "to": "yolanda@java.com",
"subject": "Re: Indonesia" }
        { "from": "cindy@go.dev",      "to": "xavier@rust-lang.org",
"subject": "Go vs Rust" }
        { "from": "diane@dart.dev",    "to": "wanda@typescriptlang.org",
"subject": "Our crypto startup" }
    `)
    dst := os.Stdout

    predicate := func(email Email) bool {
        return !strings.Contains(email.Subject, "crypto")
    }

    n, err := FilterEmails(dst, src, predicate)
    if err != nil {
        panic(err)
    }
    fmt.Println(n, "good emails")

    // {"from":"alice@go.dev","to":"zet@php.net","subject":"How are you?"}

```

```
// {"from":"bob@temp-mail.org","to":"yolanda@java.com","subject":"Re:
Indonesia"}
// {"from":"cindy@go.dev","to":"xavier@rust-lang.org","subject":"Go vs
Rust"}
// 3 good emails
}
```

13

XML

За работу с XML в Go отвечает пакет `encoding/xml`. Кодирование и декодирование XML работают аналогично JSON, так что ограничусь парой примеров.

Кодирование в XML

Человек с адресом и друзьями:

```
type Address struct {
    Country string `xml:"country,attr"` // (1)
    City    string `xml:"city,attr"`      // (2)
}

type Person struct {
    XMLName    xml.Name `xml:"person"` // (3)
    Name       string   `xml:"name"`
    IsAwesome  bool     `xml:"awesome,attr,omitempty"` // (4)
    Residence  *Address `xml:"residence"`
    Friends    []*Person `xml:"friends>person"` // (5)
}
```

Теги задают правила преобразования значений в XML. Поскольку это более сложный формат, чем JSON, то и возможностей больше:

- `Address.Country` и `Address.City` будут кодироваться как атрибуты, а не как элементы ❶ ❷
- Значения типа `Person` будут записаны внутри элемента `person` ❸
- `Person.IsAwesome` будет кодироваться как атрибут. Причем значение `false` вовсе не будет записано, только `true` ❹
- Массив друзей будет записан внутри вложенного элемента `friends` ❺

Кодированием занимаются уже знакомые нам функции `Marshal()` / `MarshalIndent()`:

```
paris := Address{"France", "Paris"}
emma  := Person{Name: "Emma"}
grace := Person{Name: "Grace"}
```



```

alice := Person{
    Name:      "Alice",
    IsAwesome: true,
    Residence: &paris,
    Friends:   []*Person{&emma, &grace},
}

b, err := xml.MarshalIndent(alice, "", "    ")
if err != nil {
    panic(err)
}
fmt.Println(string(b))

```

Результат:

```

<person awesome="true">
  <name>Alice</name>
  <residence country="France" city="Paris"></residence>
  <friends>
    <person>
      <name>Emma</name>
      <friends></friends>
    </person>
    <person>
      <name>Grace</name>
      <friends></friends>
    </person>
  </friends>
</person>

```

[песочница](#)

Декодирование из XML

Декодированием, как и с JSON, занимается функция `Unmarshal()`:

```

var alice Person
err := xml.Unmarshal([]byte(src), &alice)
if err != nil {
    panic(err)
}

fmt.Printf(alice.Name)
if alice.IsAwesome {
    fmt.Printf(" ✓ awesome")
}

```

```

}
fmt.Println()
fmt.Printf("from %s, %s\n", alice.Residence.City, alice.Residence.Country)
fmt.Println("friends:")
for _, person := range alice.Friends {
    fmt.Printf("- %s\n", person.Name)
}

```

Результат:

```

Alice ✓ awesome
from Paris, France
friends:
- Emma
- Grace

```

[песочница](#)

14

CSV

CSV — более примитивный формат, чем JSON или тем более XML. Единственные значения, которые Go умеет кодировать и декодировать в CSV — это срезы строк. Преобразованием более сложных типов в строки приходится заниматься самостоятельно.

Разберемся на примере.

Кодирование в CSV

Человек, который умеет превратить себя в срез строк:

```

type Person struct {
    Name      string
    Age       int
    IsAwesome bool
}

func (p Person) Slice() []string {
    return []string{p.Name, strconv.Itoa(p.Age),
        strconv.FormatBool(p.IsAwesome)}
}

```

Создадим список людей и закодируем их в CSV с помощью `csv.Writer`:

```

people := []Person{
    {"Alice", 25, true},
    {"Emma", 23, false},
    {"Grace", 27, false},
}

w := csv.NewWriter(os.Stdout)
for _, person := range people {
    err := w.Write(person.Slice())
    if err != nil {
        panic(err)
    }
}
w.Flush() // (1)

if err := w.Error(); err != nil { // (2)
    panic(err)
}

```

`csv.Writer` похож на знакомый нам по предыдущему уроку `bufio.Writer`. Разве что по какой-то загадочной причине метод `Flush()` не возвращает ошибку ❶. Ее приходится проверять через отдельный метод `Error()` ❷

Результат:

```

Alice,25,true
Emma,23,false
Grace,27,false

```

[песочница](#)

Декодирование из CSV

Тут придется написать конструктор, который создает человека из среза строк:

```

func MakePerson(record []string) (Person, error) {
    if len(record) != 3 {
        return Person{}, fmt.Errorf("invalid person slice: %v", record)
    }
    name := record[0]
    age, err := strconv.Atoi(record[1])
    if err != nil {
        return Person{}, fmt.Errorf("invalid Age: %v", record[1])
    }
    isAwesome, err := strconv.ParseBool(record[2])

```

```

    if err != nil {
        return Person{}, fmt.Errorf("invalid IsAwesome: %v", record[2])
    }
    return Person{name, age, isAwesome}, nil
}

```

Затем прочитать людей из CSV с помощью `csv.Reader`:

```

src := `...`
r := csv.NewReader(strings.NewReader(src))

var people []Person
for {
    record, err := r.Read()
    if err == io.EOF {
        break
    }
    if err != nil {
        panic(err)
    }
    person, err := MakePerson(record)
    if err != nil {
        panic(err)
    }
    people = append(people, person)
}

fmt.Println(people)

```

Результат:

```
[{Alice 25 true} {Emma 23 false} {Grace 27 false}]
```

CSV-разделитель настраивается с помощью свойства `Comma` у `csv.Reader` и `csv.Writer`.

[песочница](#)

15

XML → CSV

Есть XML-документ с информацией о сотрудниках организации:

```

<organization>
  <department>

```

```

<code>hr</code>
<employees>
  <employee id="11">
    <name>Дарья</name>
    <city>Самара</city>
    <salary>70</salary>
  </employee>
  <employee id="12">
    <name>Борис</name>
    <city>Самара</city>
    <salary>78</salary>
  </employee>
</employees>
</department>
<department>
  <code>it</code>
  <employees>
    <employee id="21">
      <name>Елена</name>
      <city>Самара</city>
      <salary>84</salary>
    </employee>
  </employees>
</department>
</organization>

```

Документ иерархический: организация состоит из департаментов, департаменты — из сотрудников.

Требуется преобразовать этот документ в плоский CSV со списком сотрудников:

```

id,name,city,department,salary
11,Дарья,Самара,hr,70
12,Борис,Самара,hr,78
21,Елена,Самара,it,84

```

Здесь иерархии нет, а код департамента стал атрибутом сотрудника (четвертое поле в каждой записи).

Напишите функцию `ConvertEmployees()`, которая выполнит такое преобразование:

```

// ConvertEmployees преобразует XML-документ с информацией об организации
// в плоский CSV-документ с информацией о сотрудниках
func ConvertEmployees(outCSV io.Writer, inXML io.Reader) error {

```

```
// ...  
}
```

Обратите внимание на нюансы:

- Если в исходном документе у сотрудника не хватает каких-то атрибутов, в CSV-документе они должны принимать нулевое значение соответствующего типа (пустая строка для строк, 0 для чисел).
- Если у организации вовсе нет сотрудников, это не ошибка. CSV-документ в таком случае должен содержать только заголовки.
- Если при чтении или записи произошла ошибка, `ConvertEmployees()` должна возвращать ошибку.

Полный код — в песочнице. Отправляйте в качестве решения только фрагмент, отмеченный комментариями «начало решения» и «конец решения».

[песочница](#)

Sample Input:

Sample Output:

PASS

```
package main  
  
import (  
    "encoding/csv"  
    "encoding/xml"  
    "os"  
  
    // "fmt"  
    "io"  
    "strconv"  
    "strings"  
)  
  
// начало решения  
type Org struct {  
    XMLName      xml.Name    `xml:"organization"`  
    Organization []Department `xml:"department"`  
}  
  
type Department struct {
```

```

XMLName    xml.Name    `xml:"department"`
Code       string      `xml:"code"`
Employees []Employee   `xml:"employees>employee"`
}

```

```

type Employee struct {
    XMLName xml.Name `xml:"employee"`
    Id      int      `xml:"id,attr"`
    Name    string   `xml:"name"`
    City    string   `xml:"city"`
    Salary  int      `xml:"salary"`
}

```

// csv умеет писать только срезы строк,

// сделаем из организации срез срезов строк по сотрудникам

```

func (org Org) ToList() [][]string {
    var result [][]string
    header := []string{"id", "name", "city", "department", "salary"}
    result = append(result, header)

    for _, dept := range org.Organization {
        for _, e := range dept.Employees {
            emp_str := []string{
                strconv.Itoa(e.Id),
                e.Name,
                e.City,
                dept.Code,
                strconv.Itoa(e.Salary),
            }
            result = append(result, emp_str)
        }
    }
    return result
}

```

*// ConvertEmployees преобразует XML-документ с информацией об организации
 // в плоский CSV-документ с информацией о сотрудниках*

```

func ConvertEmployees(outCSV io.Writer, inXML io.Reader) error {
    var b []byte
    buf := make([]byte, 32)
    for {
        n, err := inXML.Read(buf)

```

```

    for _, ch := range buf[:n] {
        b = append(b, ch)
    }
    if err == io.EOF {
        break
    }
}

var org Org
err := xml.Unmarshal(b, &org)

if err != nil {
    return err
}

w := csv.NewWriter(outCSV)
for _, emp_str := range org.ToList() {
    err = w.Write(emp_str)
    if err != nil {
        return err
    }
}
w.Flush()

if err := w.Error(); err != nil {
    return err
}

return nil
}

// конец решения

/* Правильный вариант (декодер без размаршалинга)
// Определим структуры для декодирования организации, департамента и
сотрудника из XML:

// Organization описывает организацию
type Organization struct {
    Departments []Department `xml:"department"`
}

```



```

// Department описывает департамент организации
type Department struct {
    Code      string    `xml:"code"`
    Employees []Employee `xml:"employees>employee"`
}

// Employee описывает сотрудника департамента
type Employee struct {
    Id      int    `xml:"id,attr"`
    Name    string `xml:"name"`
    City    string `xml:"city"`
    Salary  float64 `xml:"salary"`
}

// Логика трансформации XML → CSV можно полностью уложить в
ConvertEmployees(), но тогда она получится довольно громоздкой. Поэтому я
добавил вспомогательную функцию decodeOrganization(), которая декодирует
организацию из XML:

func decodeOrganization(in io.Reader) (Organization, error) {
    var org Organization
    decoder := xml.NewDecoder(in)
    err := decoder.Decode(&org)
    return org, err
}

// И вспомогательный тип employeeWriter, который пишет сотрудников в CSV:

// employeeWriter записывает сотрудников в CSV
type employeeWriter struct {
    w    *csv.Writer
    err  error
}

// writeHeader записывает заголовок
func (ew *employeeWriter) writeHeader() {
    if ew.err != nil {
        return
    }
    header := []string{"id", "name", "city", "department", "salary"}
    ew.err = ew.w.Write(header)
}

// writeEmployee записывает сотрудника в строку

```

```

func (ew *employeeWriter) writeEmployee(depCode string, emp Employee) {
    if ew.err != nil {
        return
    }
    fields := []string{
        strconv.Itoa(emp.Id),
        emp.Name,
        emp.City,
        depCode,
        strconv.FormatFloat(emp.Salary, 'f', -1, 64),
    }
    ew.err = ew.w.Write(fields)
}

```

// flush финализирует данные

```

func (ew *employeeWriter) flush() error {
    ew.w.Flush()
    if ew.err == nil {
        ew.err = ew.w.Error()
    }
    return ew.err
}

```

// newEmployeeWriter создает нового писателя сотрудников в CSV

```

func newEmployeeWriter(w io.Writer) *employeeWriter {
    return &employeeWriter{w: csv.NewWriter(w)}
}

```

// Теперь ConvertEmployees() будет легче читаться:

// декодируем организацию;

// записываем заголовок CSV;

// проходим по департаментам и сотрудникам внутри них;

// записываем каждого сотрудника в строку CSV;

// финализируем CSV.

```

func ConvertEmployees(outCSV io.Writer, inXML io.Reader) error {
    org, err := decodeOrganization(inXML)
    if err != nil {
        return fmt.Errorf("failed to parse xml: %w", err)
    }
}

```

```

w := newEmployeeWriter(outCSV)

```

```

w.writeHeader()

```

```

for _, dep := range org.Departments {
    for _, emp := range dep.Employees {
        w.writeEmployee(dep.Code, emp)
    }
}

if err := w.flush(); err != nil {
    return fmt.Errorf("failed writing csv: %w", err)
}

return nil
}
*/

```

```

func main() {
    src := `
<department>
    <code>hr</code>
    <employees>
        <employee id="11">
            <name>Дарья</name>
            <city>Самара</city>
            <salary>70</salary>
        </employee>
        <employee id="12">
            <name>Борис</name>
            <city>Самара</city>
            <salary>78</salary>
        </employee>
    </employees>
</department>
<department>
    <code>it</code>
    <employees>
        <employee id="21">
            <name>Елена</name>
            <city>Самара</city>
            <salary>84</salary>
        </employee>
    </employees>
</department>
</organization>`

```

```
in := strings.NewReader(src)
out := os.Stdout
ConvertEmployees(out, in)
```

```
/*
    id,name,city,department,salary
    11,Дарья,Самара,hr,70
    12,Борис,Самара,hr,78
    21,Елена,Самара,it,84
*/
```

```
}
```