Вольный и расширенный перевод оригинального репозитория Go Cheat Sheet на русский язык.

Навигация:

Другие ресурсы

Подборка полезных и бесплатных ресурсов для изучения Go на русском языке:

- Эффективный Go - перевод официальной документации Effective Go (не завершен и устарел).
- Эффективный Go - перевод от сентября 2024 года.
- Go в примерах - исходный код для сборки статического сайта Go в примерах (форк gobyexample).
- Введение в программирование на Go (веб-версия) - перевод книги An Introduction to Programming in Go.
- Маленькая книга о Go - перевод The Little Go Book.
- Паттерны параллельного программирования Go.
- Курс по изучению Golang для начинающих.
- Обучение программированию на языке Go в тренажере (онлайн компилятор).
- Руководство по языку Go от *Metanit*.
- Шпаргалка по Go в переводе с Немецкого языка.
- Гайды Uber по написанию кода на Go - русский перевод оригинального репозитория.
- GUI на Golang на GTK+ 3.

Бесплатные курсы от *Stepik* с получением сертификата:

- Go - первое знакомство - 42 урока, 110 тестов, 45 задач (20к учащихся, рейтинг: 4.9).
- PRO Go. Основы программирования - 38 урока, 121 тестов, 191 задач (13к учащихся, рейтинг: 4.8).
- Программирование на Golang - 35 урока, 64 тестов, 94 задач (65к учащихся, рейтинг: 4.7).

Другие бесплатные курсы:

- Разработка веб-сервисов на Golang - курс по Go от *Mail Ru* на платформе Coursera.
- Основы Go - курс от *Яндекс Практикум* (2 модуля на 30 часов).
- Основы Go - курс от *Хекслет* (34 урока, 97 тестов и 37 упражнений в тренажере).

Участники

Если вы нашли ошибку или хотите расширить список шпаргалок, а также знаете другие источники для изучения, сообщите о них, внеся изменения через Pull Requests.

Источники

Большинство примеров кода взяты из официального тура по Go, который является прекрасным введением для знакомства с языком.

Вы также можете использовать онлайн компилятор на официальном сайте для запуска и проверки блоков кода.

Описание языка

- **Императивный язык**, где описывается последовательность шагов (инструкций), которые необходимо выполнить для достижения результата. В отличии от декларативных языков, где описывается результат, который нужно получить, оставляя процесс выполнения скрытым (например, как в `SQL` или `HTML`).
- **Используется статическая типизация** для проверка типов переменных во время компиляции. Это когда тип переменной не может быть изменен после его присвоения (например, как в `TypeScript` в отличии от `JavaScript`).
- Синтаксис похож на `C` (но меньше скобок и нет точек с запятой в конце каждой строки), а структура — на `Oberon-2`.
- **Компилируется в машинный код без использования промежуточных слоев** (`Runtime`, например, как `JVM` в `Java` или `.NET` в `C#`), который должен быть установлен на машине для работы программы.
- **Нет классов**, но есть структуры с методами.
- **Не предоставляет подклассов, основанного на типах**, но имеет возможность заимствовать части реализации, встраивая типы в структуру или интерфейс (embedding).
- **Функции могут возвращать несколько значений** и их можно присваивать переменным, так как они рассматриваются как объекты.
- Функции можно передавать в другие функции в качестве аргументов, а также функции могут возвращать другие функции как результат.
- **Имеет замыкания** (`closures`), которые позволяют функциям хранит и использовать переменные из внешней области видимости, даже если она выполняется в другом контексте (например, за пределами этой области).
- **Невозможно напрямую изменять значение указателя с помощью арифметических операций** (например, `ptr++`). Это нужно, что бы исключить возможные ошибки, такие как выход за пределы памяти или доступ к неправильным участкам памяти.
- **Встроенные примитивы параллелизма**: горутины и каналы.

- **Поддерживаются динамические и статические срезы** (`slices` , аналог списков или массивов в других языках, где элементы хранятся в порядке их добавления и индексируются числами), а также **карты** (`maps` , аналог словарей или хэш-таблиц, где содержится уникальный ключ и его значение).

Базовый синтаксис

Привет мир

Файл `hello.go` :

```
package main

import "fmt"

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

Запуск:

```
go run hello.go
```

Выведет на экран `Hello Go`

Операторы

Арифметика

Оператор	Описание
+	сложение
-	вычитание
*	умножение
/	деление *

Оператор	Описание
`%`	остаток
`&`	побитовое `и`
`|`	побитовое `или`
`^`	побитовое `исключающее или` * *
`&^`	очистить бит (`и нет`) * * *
`<<`	сдвиг влево * * * *
`>>`	сдвиг вправо

* Если оба операнда имеют целый тип (`int` , `int8` , `int32` , `int64`), результат также будет целым числом, при этом остаток отбрасывается. Если хотя бы один из операндов имеет тип с плавающей точкой (`float32` , `float64`), результат будет дробным числом.

* * Возвращает `0` , если биты двух операндов равны, или `1` , если биты двух операндов различны.

* * * Возвращает `0` , если соответствующий бит второго операнда равен `1` , или бит первого операнда (`0` или `1`), если соответствующий бит второго операнда равен `0` .

* * * * Сдвигает все биты числа влево на указанное количество позиций (аналог умножения числа на `2` в степени количества сдвигов), а новые биты справа заполняются нулями.

Сравнение

Оператор	Описание
`==`	равно
`!=`	не равно
`<`	меньше
`<=`	меньше или равно
`>`	больше
`>=`	больше или равно

Логика

Оператор	Описание
`&&`	логическое `и`
`||`	логическое `или`
`!`	логическое отрецание

Другие

Оператор	Описание
`&`	адрес / создать указатель
`*`	разыменовать указатель
`<-`	оператор отправки / получения

Декларации

Тип указывается после идентификатора (названия переменной):

```
var foo int                 // объявление без инициализации значения
var foo int = 42            // объявление с инициализацией
var foo, bar int = 42, 1302 // объявить и инициализировать несколько переменных одновременно
var foo = 42                // тип пропущен, будет выведен
foo := 42                   // сокращение при объявление переменной (ключевое слово var опущено
const constant = "Это константа, которая используется для хранения неизменяемых данных"

// iota можно использовать для увеличения числа, начиная с 0
const (
    _ = iota
    a
    b
    c = 1 << iota
    d
)
    fmt.Println(a, b) // 1 2 (0 - пропускается)
    fmt.Println(c, d) // 8 16 (2^3, 2^4)
```

Функции

```go
// Простая функция
func functionName() {}

// Функция с параметрами (тип идет после идентификаторов)
func functionName(param1 string, param2 int) {}

// Несколько параметров одного типа
func functionName(param1, param2 int) {}

// Объявление типа для возвращаемого значения (идет после скобок параметров или во вторых скобка
func functionName() int {
    return 42
}

// Может возвращать несколько значений одновременно
func returnMulti() (int, string) {
    return 42, "foobar"
}
var x, str = returnMulti()

// Возвращаем несколько именованных результатов
func returnMulti2() (n int, s string) {
    n = 42
    s = "foobar"
    // Будут возвращены все значения объявленных переменных "n" и "s"
    return
}
var x, str = returnMulti2()

func main() {
    // Присвоить функцию переменной
    add := func(a, b int) int {
        return a + b
    }
    // Используйте имя переменной для вызова функции
    fmt.Println(add(3, 4))
}
```

Замыкания

```
// Дочерние функции могут получить доступ к переменным, объявленным в родительской функции
func scope() func() int{
    outer_var := 2
    foo := func() int { return outer_var}
    return foo
}

func another_scope() func() int{
    // Не скомпилируется, потому что "outer_var" и "foo" не определены в данной области видимост
    outer_var = 444
    return foo
}

func outer() (func() int, int) {
    outer_var := 2
    inner := func() int {
        outer_var += 99 // переменная изменена из внешней области
        return outer_var
    }
    inner()
    return inner, outer_var // вернуть результат внутренней функции и переменной с результатом 1
}
```

Вариативные функции

Вариативная функция работает и вызывается как любая другая функция, за исключением того, что в нее возможно передать произвольное количество аргументов, используя `...` перед типом данных указанного параметра.

```go
func main() {
    fmt.Println(adder(1, 2, 3)) // 6
    fmt.Println(adder(9, 9))    // 18

    nums := []int{10, 20, 30}
    fmt.Println(adder(nums...)) // 60
}

func adder(args ...int) int {
    total := 0
    for _, v := range args { // перебирает все переданные аргументы в цикле
        total += v
    }
    return total
}
```

Типы данных

```go
bool // логический тип (принимает true или false)

string // строка (текст)

int  int8  int16  int32  int64 // целое число
uint uint8 uint16 uint32 uint64 uintptr // беззнаковый целочисленный тип размером

byte // псевдоним для uint8

rune // псевдоним для int32 ~= символ (кодовая точка Unicode)

float32 float64 // число с плавающей точкой одинарной и двойной точности

complex64 complex128 // комплексное число (1 + 2i или 3.14 + 4.2i), имеющие реальную и мнимую ча

interfae{} // универсальный тип, который может позволяет работать с переменными неизвестного или
```

Все предварительно объявленные идентификаторы `Go` определены в пакете builtin.

Преобразование типов

```
var i int = 42
var f float64 = float64(i)  // преобразуем тип данных int в float64
var u uint = uint(f)        // преобразуем тип данных float64 в unit

// Альтернативный синтаксис
i := 42
f := float64(i)
u := uint(f)
```

Структуры управления

Условия if

```
func main() {
    // Базовый
    if x > 10 {
        return x
    } else if x == 10 {
        return 10
    } else {
        return -x
    }

    // Возможно поставить одно утверждение перед условием
    if a := b + c; a < 42 {
        return a
    } else {
        return a - 42
    }

    // Утверждение (проверка) типа внутри условия
    var val interface{} = "foo"
    // Проверяется, содержит ли переменная val значение типа string
    if str, ok := val.(string); ok {
        // Если тип не совпадает, значение не вернется.
        // При этом panic не вызывается, т.к. используется безопасное утверждение типа (ok)
        fmt.Println(str)
    }
}
```

Условия switch

После выполнения условия при использование переключателей, прерывания обрабатываются автоматически.

```go
package main

import (
    "fmt"
    "os"
)

func main() {
    var operatingSystem string = runtime.GOOS
    // Используем оператор (ключевое слово) switch
    switch operatingSystem {
    case "darwin":
        fmt.Println("Используется macOS")
    case "linux":
        fmt.Println("Используется Linux")
    // Условие по умолчанию (аналог else в if)
    default:
        fmt.Println("Используется Windows, OpenBSD, FreeBSD или другая")
    }
}

// Как в случае с "for" и "if", возможно иметь оператор присваивания перед значением switch
switch os := runtime.GOOS; os {
    case "darwin": ...
}

// Возможно использовать сравнения
number := 42
switch {
    case number < 42:
        fmt.Println("Переданное значение:", number, "меньше 42 в условие")
    case number == 42:
        fmt.Println("Переданное значение:", number, "равно 42 в условие")
    case number > 42:
        fmt.Println("Переданное значение:", number, "больше 42 в условие")
}

// Все случаи могут быть представлены в виде списков, разделенных запятыми
var char byte = '?'
switch char {
    case ' ', '?', '&', '=', '#', '+', '%':
```

```
        fmt.Println("Переданное значение присутствует в списке")
}
```

Циклы

В `Go` используются только универсальные циклы `for`, другие операторы (например, `while` или `until`) отсутствуют.

```
// Используется 9 интераций с 1 по 9 (до 10)
for i := 1; i < 10; i++ {
}
// Цикл (loop) - while
for ; i < 10;  {
}
// Если есть только условие, точки с запятой опускаются
for i < 10  {
}
// Если опустить условие, равноценно использованию while (true)
for {
}

// Использование пропуска и прерывания в цикле
// Метка here (произвольное имя) позволяет указать целевой цикл, на который будут ссылаться опер
here:
    // Используем 2 интерации в внешнем цикле (от 0 до 1)
    for i := 0; i < 2; i++ {
        // Внутренний цикл: переменная j начинается с i+1 и проходит до 3
        for j := i + 1; j < 3; j++ {
            if i == 0 {
                // Пропустить интерацию внешнего цикла по названию его метки
                continue here
            }
            fmt.Println(j)
            if j == 2 {
                // Завершить внутренний цикл
                break
            }
        }
    }

// 1-я интерация: внешний цикл с значением i=0 в внутреннем цикле пропускает интерацию внешнего
// 2-я интерация: внешний цикл с значением i=1 в внутреннем цикле пропускает условие i==0
// Переменная j получает значение 2, которое печатается и завершает внутренний (текущий) цикл во
// Программа завершается, т.к. интерации внешнего цикла закончились

there:
    for i := 0; i < 2; i++ {
        for j := i + 1; j < 3; j++ {
            if j == 1 {
                // Пропускаем интерацию внутреннего цикла
                continue
```

```
        }
        fmt.Println(j)
        if j == 2 {
            // Завершаем выполнение внешнего цикла
            break there
        }
    }
}
```

Примеры циклов

```go
package main

import "fmt"

// Функция, возвращающая название месяца через условную конструкцию switch
func getMonthName(month int) string {
    switch month {
    case 1:
        return "January"
    case 2:
        return "February"
    case 3:
        return "March"
    case 4:
        return "April"
    case 5:
        return "May"
    case 6:
        return "June"
    case 7:
        return "July"
    case 8:
        return "August"
    case 9:
        return "September"
    case 10:
        return "October"
    case 11:
        return "November"
    case 12:
        return "December"
    default:
        return "Invalid month (range: 1-12)"
    }
}

// Второй вариант функции через классическое условие по индеку массива
func getMonthName2(month int) string {
    months := []string{"", "January", "February", "March", "April", "May", "June", "July", "Augu
    if month >= 1 && month <= 12 {
        return months[month]
```

```
    }
    return "Invalid month"
}

func main() {
    // Классический цикл из 13-ти итераций
    for i := 1; i <= 13; i++ {
        fmt.Printf("Month %d: %s\n", i, getMonthName(i))
    }

    // Увеличение индекса итерации в теле цикла
    j := 1
    for j <= 13 {
        fmt.Printf("Month %d: %s\n", j, getMonthName(j))
        j++
    }

    // Бесконечный цикл
    months := []int{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12}
    k := 0
    for {
        // Пропускаем итерацию, если 6-й месяц (5-й индекс)
        if k == 5 {
            k++ // переход к следующей итерации
            continue
        }
        // Выходим из цикла, если индекс больше или равен длине массива
        if k >= len(months) {
            break
        }
        fmt.Printf("Month %d: %s\n", months[k], getMonthName(months[k]))
        k++
    }

    // Конструкция range используется для перебора всех элементов в коллекциях (массивы, слайсы,
    for _, month := range months {
        fmt.Printf("Month %d: %s\n", month, getMonthName(month))
    }

    // Индекс может использоваться для карты (map) как ключ
    m := map[string]int{"a": 1, "b": 2, "c": 3}
    for index, value := range m {
        fmt.Println("Key:", index, "Value:", value)
```

```
    }

    // Перебор строки по символам
    s := "string"
    for index, char := range s {
        fmt.Println("Index:", index, "Char:", string(char))
    }

    // Перебор канала
    ch := make(chan int, 3)
    ch <- 1
    ch <- 2
    ch <- 3
    close(ch)
    for val := range ch {
        fmt.Println(val)
    }
}
```

Типы последовательностей

Массивы, срезы и диапазоны представляют собой структуры данных, хранящие упорядоченные наборы значений.

Статические срезы (массивы)

Статические срезы подразумеваются как массивы.

```
var a [10]int // объявить массив int длиной 10 (длина массива является частью типа)
a[3] = 42     // присвоить значение элементу, по его порядковому номеру
i := a[3]     // прочитать элементы

// Возможные варианты объявление с инициализацией значений
var a = [2]int{1, 2}
// Массив из двух элементов: [1 2]
a := [2]int{1, 2}
// Многоточие используется компилятором для вычисления длины массива
a := [...]int{1, 2}
```

Динамические срезы

Динамический срез объявляется аналогично статическому, но длина не указывается.

```
var a []int                                 // объявить срез
var a = []int {1, 2, 3, 4}                  // объявить и инициализировать срез
a := []int{1, 2, 3, 4}                      // [1 2 3 4]
chars := []string{0:"a", 2:"c", 1: "b"}     // [a b c]

var b = a[lo:hi]                            // создать срез от индекса lo до hi-1
var b = a[1:4]                              // срез с индекса 1 по 3 (до 4)
var b = a[:3]                               // отсутствие первого индекса подразумевает 0
var b = a[3:]                               // отсутствие последнего индекса подразумевает len(a
a =  append(a,17,3)                         // добавление элементов к срезу a с помощью функции
c := append(a,b...)                         // объединение срезов a и b

a = make([]byte, 5, 5)                      // первый аргумент длина, второй емкость
a = make([]byte, 5)                         // емкость необязательна

x := [3]string{"Лайка", "Белка", "Стрелка"} // создаем массив
s := x[:]                                   // создать срез из массива
```

Операции с срезами

`len(a)` возвращает длину среза/массива. Это встроенная функция, а не метод массива.

```
// Цикл по массиву/срезу
for i, e := range a {
    // "i" — индекс, "e" — элемент
}

// Если нужен только элемент "e"
for _, e := range a {
    // Используется только элемент "e"
}

// Если нужен только индекс
for i := range a {
}
```

Карты

```
package main

import "fmt"

type Vertex struct {
        Lat, Long float64
}

func main() {
    m := make(map[string]int) // объявить карту
    m["key"] = 42             // инициализировать карту
    fmt.Println(m["key"])     // вывести содержимое значения (value) по его уникальному названию
    delete(m, "key")          // удалить элемент из карты
    elem, ok := m["key"]      // проверка, если ключ присутствует, то получить его значение
    fmt.Println(ok, elem)

    var m2 = map[string]Vertex{
        "Bell Labs": {40.68433, -74.39967},
        "Google":    {37.42202, -122.08408},
    }

    // Перебрать содержимое карты в цикле
    for key, value := range m2 {
        fmt.Println(key)
        fmt.Println(value)
    }
}
```

Примеры срезов и карт

```go
package main

import "fmt"

func main() {
    // Массив фиксированного размера с 5-ю элементами, все элементы инициализируются значением 0
    var arr [5]int
    // Присвоить значения указанным элементам массива по индексу
    arr[0] = 1
    arr[1] = 2
    fmt.Println("Array:", arr)

    // Слайсы (массивы переменной длины)
    slice := []int{1, 2, 3, 4, 5}
    // Добавить новое значение в массив с помощью функции append
    slice = append(slice, 6)
    // Удалить элемент с индексом 5
    index := 4
    // Создается 2 слайса с начала до указанного индекса и срез с следующего элемента после инде
    slice = append(slice[:index], slice[index+1:]...)
    fmt.Println("Slice:", slice) // [1, 2, 3, 4, 6]
    // Вывести срез слайсов по индексу с 1 и до 4 (по 3, не включая 4) элемент
    fmt.Println(slice[1:4]) // [2, 3, 4]
    // Очистить слайс (удалить все элементы)
    slice = slice[:0] // []
    fmt.Println(len(slice) == 0) // true
    // Объединение двух слайсов
    slice1 := []int{1, 2}
    slice2 := []int{3, 4}
    combined := append(slice1, slice2...)
    fmt.Println(combined) // [1, 2, 3, 4]

    // Слайс с заданной вместимостью:
    makeSlice := make([]int, 5, 10)
    fmt.Println(makeSlice) // [0 0 0 0 0]
    fmt.Println(cap(makeSlice)) // 10

    // Создаем пустую карту (map) с ключами типа string и значениями типа int
    m := make(map[string]int)
    m["Day"] = 30       // добавляем элемент с ключом "Day" и значением 30
```

```
    m["Day"] = 31       // обновляем  значение для ключа
    m["Month"] = 12
    fmt.Println(m) // map[Day:31 Month:12]
    // Создать карту с заданными значениями
    m2 := map[string]int{
        "Day": 31,
        "Month": 12,
    }
    // Читаем значение
    value := m2["Day"]
    fmt.Println(value) // 31
    // Если ключа нет в карте, то получаем нулевое значение для типа значения
    value, exists := m["Year"]
    fmt.Println(value, exists) // 0 false
    // Удалить элемент
    delete(m, "Day")
    fmt.Println(m) // map[Month:12]
}
```

Структуры

Вместо классов (`class`) в `Go` используются структуры (тип данных `struct`), которые могут иметь методы. Поля структуры всегда инициализируются нулевыми значениями при её объявлении.

```
// Объявление структуры с названием Vertex с помощью ключевого слова "type"
type Vertex struct {
    X, Y float64
}

// Создание структуры
var v = Vertex{1, 2}                // инициализация данных в структуре
var v = []Vertex{{1,2},{5,2},{5,5}} // инициализация среза в структуре
var v = Vertex{X: 1, Y: 2}          // создание структуры с опредилением значений с помощью ключ
v.X = 4                             // доступ к значениям

// Объявление метода (принимающий тип), находится между ключевым словом func и именем метода
// Структура копируется при каждом вызове метода
func (v Vertex) Abs() float64 {
    return math.Sqrt(v.X*v.X + v.Y*v.Y)
}

// Вызов метода
v.Abs()

// Для мутирующих методов необходимо использовать указатель (см. ниже) на Struct в качестве типа
// При этом значение структуры не копируется для вызова метода
func (v *Vertex) add(n float64) {
    v.X += n
    v.Y += n
}
```

Анонимные структуры

Безопаснее и дешевле, чем использование `map[string]interface{}` .

```
point := struct {
        X, Y int
} {1, 2}
```

Указатели

```
p := Vertex{1, 2}  // "p" это структура Vertex
q := &p            // "q" указывает на структуру Vertex
r := &Vertex{1, 2} // "r" также указывает на структуру Vertex

// Объявление переменной с указателем на структуру *Vertex
var s *Vertex = new(Vertex) // функция "new" создает указатель на новый экземпляр структуры
```

Интерфейсы

Интерфейс - это набор методов (требований), которые должен иметь тип, чтобы соответствовать этому интерфейсу.

```
// Объявление интерфейса с одинм методом Awesomize(), который возвращает строку
type Awesomizer interface {
    Awesomize() string
}

// Обычная структура, которая может реализовывать методы
type Foo struct {}

// Добавление (реализация) метода Awesomize() в структуре Foo
// Тип автоматически соответствует интерфейсу, если он реализует все его методы
func (foo Foo) Awesomize() string {
    return "Awesome!"
}
```

Встраивание

В `Go` нет подклассов, вместо этого используется встраивание интерфейса и структуры, которое добавляет методы встроенной структуры к внешней.

```
// В структуру Server встраиваются все методы, которые есть у метода Logger из структуры log
type Server struct {
    Host string
    Port int
    *log.Logger
}

// Структура Server инициализируется с помощью указателя на log.Logger
server := &Server{"localhost", 80, log.New(...)}

// Когда вызывается server.Log(...), Go автоматически перенаправляет вызов к server.Logger.Log(
server.Log(...)

// Поле встроенного типа доступно через его имя, по этому переменной можно присвоить ссылку на 
var logger *log.Logger = server.Logger
```

Обработка ошибок

Обработка исключений отсутствует. Вместо этого функции, которые могут выдать ошибку, просто объявляют дополнительное возвращаемое значение типа error (чаще всего вторым возвращаемым параметром).

Встроенный тип интерфейса `error` — это общепринятый интерфейс для представления состояния ошибки, при этом нулевое значение не представляет ошибки.

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

Пример:

```go
package main

import (
	"errors"
	"fmt"
	"math"
)

// Определение функции sqrt должно быть вне main
func sqrt(x float64) (float64, error) {
	if x < 0 {
	// Создаем объект типа error с текстовым описанием ошибки
		return 0, errors.New("ошибка: отрицательное значение")
	}
	return math.Sqrt(x), nil
}

func main() {
	val, err := sqrt(-1)
	if err != nil {
		// Обработка ошибки
		fmt.Println(err) // отрицательное значение
		return
	}
	// Если все хорошо (переданное значение не отрицательное), вывести содержимое "val"
	fmt.Println(val)
}
```

Параллелизм

Горутины

Горутины — это легковесные потоки (управляемые `Go` , а не потоками ОС).

`go f(a, b)` запускает новую горутину, которая запускает `f` (при условии, что `f` — это функция).

```go
// Просто функция (которая позже может быть запущена в горутине)
func doStuff(s string) {
    fmt.Println(s)
}

func main() {
    // Запуск существующий функции в горутине по ее имени
    go doStuff("foobar")

    // Использование анонимной внутренней функции в горутине
    go func (x int) {
        fmt.Println(x)
    } (42) // Параметр анонимной функции
}
```

Синхронизация

Пакет `sync` используется для ожидания завершения всех запущенных горутин.

```go
package main

import (
        "fmt"
        "sync"
)

func doStuff(s string, wg *sync.WaitGroup) {
        fmt.Println(s)
    defer wg.Done()
}

func main() {
    // Объект для отслеживания завершение групп горутин
        var wg sync.WaitGroup

        // Задаем счетчик для запуска 2-х горутин
        wg.Add(2)

        // Запуск функции в горутине
        go doStuff("foobar", &wg)

        // Запуск анонимной функции в горутине
        go func(x int, wg *sync.WaitGroup) {
                fmt.Println(x)
        // Уменьшить счётчик WaitGroup, когда горутина завершится
        defer wg.Done()
        }(42, &wg)

        // Ожидание завершения всех горутин
        wg.Wait()
}
```

Таймер

Таймеры из пакета `time` используются для задержки (паузы) на указанное время:

```go
package main

import (
    "fmt"
    "time"
)

func goRun() {
    // Симуляция работы
    time.Sleep(2 * time.Second)
    fmt.Println("Выполнение горутины завершено")
}

func main() {
    // Запуск горутины
    go goRun()
     // Основная функция продолжает работать параллельно
    fmt.Println("Запуск выполнения основной функции и ожидание завершения горутины")
    // Ждем завершения выполнения горутины
    time.Sleep(3 * time.Second)
}
```

Небуферизованный канал

Небуферизованный канал блокирует операцию записи, пока не будет выполнено чтение, и наоборот.

```go
// Создаем небуферизованный канал типа "int"
ch := make(chan int)
// Отправляем значение 42 в канал "ch"
// Операция блокирует текущую горутину, пока другая горутина не прочитает его значение
ch <- 42
// Получаем значение из канала "ch"
// Это также блокирует выполнение, пока не будет доступно значение для чтения в канале
v := <-ch
```

Буферизованный канал

Буферизованный канал позволяет отправлять и получать данные без блокировки, пока размер буфера не будет превышен, как только буфер заполняется, запись блокируется, пока другие горутины не начнут извлекать значения из канала.

Закрытие канала — это сигнал получателю, что больше значений не будет отправляться в канал, при этом отправленные в него данные не удаляются. Это необходимо для того, чтобы получатели знали, что можно завершить чтение. Закрытие канала происходило только в той горутине, которая отправляет данные.

```
package main

import "fmt"

func main() {
	// Создаем буферизованный канал с размером буфера 100
	ch := make(chan int, 100)

	// Отправляем некоторое количество значений в канал
	for i := 0; i < 10; i++ {
		ch <- i
	}

	// Закрываем канал, чтобы цикл мог завершиться
	close(ch)

	// Читать из канала, пока он не будет закрыт
	for i := range ch {
		fmt.Println(i)
	}

	// Прочитать данные из канала и проверить, закрыт ли он
	v, ok := <-ch
	if !ok {
		fmt.Println("Канал закрыт, данные не доступны")
	} else {
		fmt.Println("Прочитано из канала:", v)
	}
}
```

Вывод: `0 1 2 3 4 5 6 7 8 9 Канал закрыт, данные не доступны`

Селекторы

Оператор `select` работает как многоканальный оператор `switch` . Выбор блоков в операциях с несколькими каналами, если один из них разблокируется, выполняется соответствующие условие. Он блокируется до тех пор, пока одно из выражений `case` не будет готов к выполнению, при этом остальные игнорируются.

```go
package main

import (
        "fmt"
        "time"
)

func doStuff(channelOut, channelIn chan int) {
    select {
    case channelOut <- 42:
        fmt.Println("Отправить значение 42 в channelOut")
    case x := <-channelIn:
        fmt.Println("Прочитать из channelIn:", x)
    case <-time.After(time.Second * 1):
        fmt.Println("Задержка в одну секунду")
    }
}

func main() {
    // Создание двух каналов (один для записи, другой для чтения)
    channelOut := make(chan int)
    channelIn := make(chan int)

    // Запуск горутины для записи в канал "channelOut"
    go func() {
        time.Sleep(500 * time.Millisecond) // Пауза перед отправкой
        channelOut <- 42                   // Отправить значение 42
        fmt.Println("Значение 42 отправлено в channelOut")
    }()

    // Запуск горутины для чтения из канала "channelIn"
    go func() {
        time.Sleep(200 * time.Millisecond) // Пауза перед отправкой
        channelIn <- 99                    // Отправить значение 99
        fmt.Println("Значение 99 отправлено в channelIn")
    }()

    // Запуск функции doStuff с двумя каналами
    doStuff(channelOut, channelIn)
}
```

Аксиомы канала

- Отправка в пустой канал блокирует навсегда и вызывает фатальную ошибку:

```
var c chan string
c <- "Hello, World!"
```

- Чтение из нулевого канала блокируется навсегда:

```
var c chan string
fmt.Println(<-c)
```

- Отправка в закрытый канал вызывает панику:

```
var c = make(chan string, 1)
c <- "Hello, World!"
close(c)
c <- "Hello, Panic!"
```

- Прием из закрытого канала немедленно возвращает нулевое значение:

```
var c = make(chan int, 2)
c <- 1
c <- 2
close(c)
for i := 0; i < 3; i++ {
    fmt.Printf("%d ", <-c)
}
// 1 2 0
```

Примеры каналов и горутин

```go
package main

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

func goRun(ch chan string) {
    time.Sleep(2 * time.Second)
    // Возвращяем сообщене о выполнение в канал
    ch <- "Первая горутина завершена за 2 секунды"
}

func goRunThree(ch chan string) {
    time.Sleep(3 * time.Second)
    ch <- "Вторая горутина завершена за 3 секунды"
}

func printMessage(msg string, wg *sync.WaitGroup) {
    // Уменьшает счётчик в WaitGroup, когда горутина завершена
    defer wg.Done()
    fmt.Println(msg)
}

func main() {
    // Создаем канал
    ch := make(chan string)
    // Запускаем горутину
    go goRun(ch)
    fmt.Println("Ожидаем завершения горутины в канале")
    // Блокируем main, пока не получим сообщение от горутины
    result := <-ch
    // После получения вывода, программа продолжает выполнение
    fmt.Println(result)

    // Создаем два канала и запускаем две горутины
    ch1 := make(chan string)
    ch2 := make(chan string)
    go goRun(ch1)
```

```
    go goRunThree(ch2)
    fmt.Println("Ожидаем завершения первой выполненной горутины")
    // Используем select для ожидания данных с двух каналов и выбора первого завершенного канала
    select {
    case msg1 := <-ch1:
        fmt.Println("Ответ:", msg1)
    case msg2 := <-ch2:
        fmt.Println("Ответ:", msg2)
    }

    // Создаем группу ожидания для синхронизации выполнения нескольких горутин
    var wg sync.WaitGroup
    fmt.Println("Ожидаем выполнения всех запущенных горутин")
    // Указать количество горутин, за которыми нужно следить
    wg.Add(2)
    go printMessage("Результат первой горутины", &wg)
    go printMessage("Результат второй горутины", &wg)
    // Ожидаем завершения всех горутин
    wg.Wait()
    fmt.Println("Все горутины завершили свою работу")
}
```

Вывод

```
package main

import "fmt"

func main() {
    fmt.Println("Hello, 你好, नमस्ते, Привет, ಹಲೊ") // базовый вывод
    p := struct{ X, Y int }{17, 2}
    fmt.Println("My point:", p, "x coord=", p.X)       // вывод структуры, цифр
    s := fmt.Sprintln("My point:", p, "x coord=", p.X) // вывод в переменную с типом данных str
    fmt.Println(s)

    fmt.Printf("%d hex:%x bin:%b fp:%f sci:%e", 17, 17, 17, 17.0, 17.0) // C-образный формат
    s2 := fmt.Sprintf("%d %f", 17, 17.0)                               // форматировать вывод в
    fmt.Println(s2)

        // Многострочный строковый литерал
    hellomsg := `
        "Hello" in Chinese is 你好 ('Ni Hao')
        "Hello" in Hindi is नमस्ते ('Namaste')
    `
    fmt.Println(hellomsg)
}
```

Переключение типа

Переключение типа похоже на обычный оператор `switch` , но в условиях указывается типы (а не на значения), которые сравниваются с типом значения, содержащегося в данном значении интерфейса.

```go
package main

import "fmt"

func do(i interface{}) {
	switch v := i.(type) {
	case int:
		fmt.Printf("Число %v равно %v по типу данных\n", v, v*2)
	case string:
		fmt.Printf("Значение %q равно %v bytes\n", v, len(v))
	default:
		fmt.Printf("Тип %T неизвестен\n", v)
	}
}

func main() {
	do(21)
	do("hello")
	do(true)
}

// Число 21 равно 42 по типу данных
// Значение "hello" равно 5 bytes
// Тип bool неизвестен
```

Математические вычисления

```go
package main

import (
        "fmt"
        "math"
)

func customCeil(numerator int, denominator int) int {
    result := numerator / denominator
    if numerator%denominator != 0 {
        result++
    }
    return result
}

func main() {
    fmt.Println("Возвращает наименьшее значение из двух чисел 9 и 10:", math.Min(9, 10)) // 9
    fmt.Println("Возвращает наибольшее значение из двух чисел 9 и 10:", math.Max(9, 10)) // 10
    fmt.Println("Округляет число в меньшую сторону 10 / 3:", math.Floor(10/3)) // 3
    fmt.Println("Округляет число в большую сторону 10 / 3:", math.Ceil(10.0/3))
    fmt.Println("Округляет число в большую сторону 10 / 3:", customCeil(10, 3)) // 4
    fmt.Println("Отбрасывает дробную часть числа (не округляет) 4,9:", math.Trunc(4.9)) // 4
    fmt.Println("Округляет число до ближайшего целого в большую сторону от 4,5:", math.Round(4.5
    fmt.Println("Округляет число до ближайшего целого в меньшую сторону от 4,5:", math.Round(4.4
    fmt.Println("Возвращает абсолютное значение числа -7:", math.Abs(-7)) // 7
    fmt.Println("Возводит число 2 в степень 3:", math.Pow(2, 3)) // 8
    fmt.Println("Вычисляет квадратный корень числа 16:", math.Sqrt(16)) // 4
}
```

Встроенные пакеты

- Декларация пакета (объявление через `import`) производится в начале каждого исходного файла.
- Исполняемые файлы находятся в пакете `main`.
- Имя пакета соответствует последнему имени в пути импорта (путь импорта `math/rand` => пакет `rand`).

- Идентификатор функции в верхнем регистре является экспортируемый (доступны из других пакетов).
- Идентификатор функции в нижнем регистре является частный (недоступны из других пакетов).

Встраивание файлов

Программы `Go` могут встраивать статические файлы с помощью пакета `embed` и директиву `go:embed path/filename`:

```go
package main

import (
    "embed"
    "fmt"
    "io"
    "log"
    "net/http"
)

//go:embed static/*
var content embed.FS

func main() {
    http.Handle("/", http.FileServer(http.FS(content)))
    go func() {
        log.Fatal(http.ListenAndServe(":8080", nil))
    }()

    // Чтение содержимого файлов из файловой системы
    entries, err := content.ReadDir("static")
    if err != nil {
        log.Fatal(err)
    }

    for _, e := range entries {
        resp, err := http.Get("http://localhost:8080/static/" + e.Name())
        if err != nil {
            log.Fatal(err)
        }
        body, err := io.ReadAll(resp.Body)
        if err != nil {
            log.Fatal(err)
        }
        if err := resp.Body.Close(); err != nil {
            log.Fatal(err)
        }
        fmt.Printf("%q: %s", e.Name(), body)
    }

    // Блокировка программы, чтобы сервер продолжал работать для доступа к статическим файлам че
    select {}
}
```

```
// Имитация реальных файлов с их содержимым для запуска Playground
-- static/a.txt --
hello a
-- static/b.txt --
hello b
```

HTTP сервер

Реализация простого `API` сервера на базе встроенной библиотеки `net/http` :

```go
package main

import (
	"encoding/json"
	"fmt"
	"net/http"
)

// Обработчик API
func apiHandler(w http.ResponseWriter, r *http.Request) {
	// Устанавливаем заголовок для ответа (Content-Type: application/json)
	w.Header().Set("Content-Type", "application/json")

	switch r.Method {
	case "GET":
		// Получение параметра "name" из URL
		name := r.URL.Query().Get("name")
		if name == "" {
			name = "Guest" // Значение по умолчанию, если параметр отсутствует
		}
		// Формируем JSON-ответ
		json.NewEncoder(w).Encode(map[string]string{
			"message": fmt.Sprintf("Hi %s", name),
		})

	case "POST":
		// Парсим JSON из тела запроса
		var data map[string]interface{}
		if err := json.NewDecoder(r.Body).Decode(&data); err != nil {
			http.Error(w, "invalid JSON", http.StatusBadRequest)
			return
		}

		// Формируем JSON-ответ
		json.NewEncoder(w).Encode(map[string]interface{}{
			"received": data,
			"status":   "OK",
		})

	default:
		// Обработка неподдерживаемых методов
		http.Error(w, "Метод не поддерживается", http.StatusMethodNotAllowed)
	}
```

```
}

func main() {
        // Регистрируем обработчик для пути /api
        http.HandleFunc("/api", apiHandler)

        // Запуск сервера
        fmt.Println("Сервер запущен на http://localhost:8080")
        http.ListenAndServe(":8080", nil)
}
```

Делаем запрос к `API` через `curl`:

```
curl -s "http://localhost:8080/api" | jq .message # "Hi Guest"
curl -s "http://localhost:8080/api?name=Alex" | jq .message # "Hi Alex"
curl -s -X POST -d '{"key":"value"}' -H "Content-Type: application/json" http://localhost:8080/a
curl -s -X POST "http://localhost:8080/api" # invalid JSON
```

HTTP клиент

Делаем запрос к `API` в `Go`:

```go
package main

import (
	"bytes"
	"encoding/json"
	"fmt"
	"io"
	"net/http"
)

func main() {
	// URL для отправки POST-запроса
	url := "http://localhost:8080/api"

	// Тело запроса в формате JSON
	requestBody := map[string]string{"key": "value"}
	jsonData, err := json.Marshal(requestBody)
	if err != nil {
		fmt.Println("Ошибка при создании тела запроса в формате JSON:", err)
		return
	}

	// Создаем запрос
	resp, err := http.Post(url, "application/json", bytes.NewBuffer(jsonData))
	if err != nil {
		fmt.Println("Ошибка при отправке запроса:", err)
		return
	}
	defer resp.Body.Close()

	// Читаем тело ответа
	body, err := io.ReadAll(resp.Body)
	if err != nil {
		fmt.Println("Ошибка при чтении ответа:", err)
		return
	}

	// Разбираем ответ в формате JSON
	var response map[string]interface{}
	if err := json.Unmarshal(body, &response); err != nil {
		fmt.Println("Ошибка при парсинге JSON:", err)
		return
	}
```

```
        // Выводим значение "key" из ответа
        if received, ok := response["received"].(map[string]interface{}); ok {
                if value, exists := received["key"]; exists {
                        fmt.Println(value) // "value"
                } else {
                        fmt.Println("Ключ 'key' не найден в ответе")
                }
        } else {
                fmt.Println("Ответ не содержит ожидаемую структуру received")
        }
}
```

`HTTP` запрос к `API` для получения последней версии релиза указаного репозитория в GitHub:

```go
package main

import (
	"encoding/json"
	"fmt"
	"log"
	"net/http"
)

// Формируем структуру ответа от API
type GitHubRelease struct {
	TagName string `json:"tag_name"`
}

func main() {
	// Формируем URL для получения информации
    repos := "Lifailon/lazyjournal"
	url := fmt.Sprintf("https://api.github.com/repos/%s/releases/latest", repos)
	// Выполнение GET-запроса
	resp, err := http.Get(url)
	if err != nil {
		log.Fatal("Ошибка при выполнении запроса:", err)
	}
	defer resp.Body.Close()
	// Проверка на успешный ответ
	if resp.StatusCode != http.StatusOK {
		log.Fatalf("Ошибка HTTP: %s", resp.Status)
	}
	// Декодирование JSON-ответа в заданную структуру
	var release GitHubRelease
	err = json.NewDecoder(resp.Body).Decode(&release)
	if err != nil {
		log.Fatal("Ошибка при декодировании JSON:", err)
	}
	// Вывод последней версии
	fmt.Println("Latest version:", release.TagName)
}
```

```
go run main.go
```

Вызов системных команд

Проверка доступности всех хостов в указанной подсети (асинхронный `ICMP` опрос):

```go
package main

import (
	"fmt"
	"os"
	"os/exec"
	"strings"
	"sync"
)

func pingHost(ip string, wg *sync.WaitGroup) {
	defer wg.Done()
	// Запускаем команду ping
	cmd := exec.Command("ping", "-n", "1", ip)
	output, err := cmd.CombinedOutput()
	if err != nil {
		return
	}
	// Обрабатываем вывод команды
	if strings.Contains(string(output), "TTL=") {
		fmt.Printf("%s - доступен\n", ip)
	}
}

func main() {
	if len(os.Args) < 2 {
		fmt.Println("Использование: go run main.go <подсеть>")
		return
	}
	// Извлекаем аргумент
	subnet := os.Args[1]
	// Убираем последний октет
	ipBase := subnet[:len(subnet)-1]
	var wg sync.WaitGroup
	for i := 1; i <= 254; i++ {
		ip := fmt.Sprintf("%s%d", ipBase, i)
		wg.Add(1)
		// Запускаем асинхронный пинг
		go pingHost(ip, &wg)
	}
	// Ждем завершения всех горутин
```

```
        wg.Wait()
}
```

```
go run main.go 192.168.3.0
```

Регулярные выражения

Элементы синтаксиса

Основные элементы синтаксиса регулярных выражений:

Символ	Описание
`.`	любой символ, кроме символа новой строки
`*`	`0` или более повторений
`+`	`1` или более повторений
`{n}`	точно `n` повторений (например, `a{3}`, соответствует: `"aaa"`)
`{n,}`	минимум `n` повторений (например, `a{2,}`, соответствует: `"aa"`, `"aaa"` и т.д.)
`{n,m}`	от `n` до `m` повторений (например, `a{2,4}`, соответствует: `"aa"`, `"aaa"`, `"aaaa"`)
`?`	`0` или `1` повторений
`^`	начало строки
`$`	конец строки
`[]`	группа символов (например, `[a-z]`)
`\s`	любой пробельный символ (пробел, табуляция, новая строка и другие пробельные символы)
`\d`	цифра (эквивалентно `[0-9]`)
`\D`	любой символ, не являющийся цифрой (эквивалентно `[^0-9]`)
`\w`	буквенно-цифровой символ (буквы, цифры и символ подчеркивания, эквивалентно `[a-zA-Z0-9_]`)

Символ	Описание
`\W`	не буквенно-цифровой символ (эквивалентно `[^a-zA-Z0-9_]`)
`\b`	граница слова (например, `\bword\b` соответствует `"word"`, и не подхоит `"wordy"`)
`(?i)`	делает выражение нечувствительным к регистру
`\`	экранирование специальных символов
`()`	группа захвата
`|`	логическое `или` (например, `a

Основные функции пакета ``:

Функции regexp

- `regexp.MatchString` — проверяет, соответствует ли строка регулярному выражению.

```
package main

import (
        "fmt"
        "regexp"
)

func main() {
    pattern := `^[a-z]+$`
        str := "string"
    matched, err := regexp.MatchString(pattern, str)
        if err != nil {
                fmt.Println("Ошибка в регулярном выражении:", err)
                return
        }
        fmt.Printf("Строка '%s' соответствует регулярному выражению '%s' (результат: %v)", str,
}
```

- `regexp.Compile` — компилирует регулярное выражение и возвращает объект типа `*regexp.Regexp`, если выражение корректное, или возвращается ошибка.

```go
package main

import (
    "fmt"
    "regexp"
)

func main() {
    // Компилируем регулярное выражение
    r, err := regexp.Compile(`\d+`)
    if err != nil {
        fmt.Println("Ошибка компиляции регулярного выражения:", err)
        return
    }
    // Применяем регулярное выражение к строке
    fmt.Println(r.FindString("123 abc 456")) // 123
}
```

- `regexp.FindAllString` — находит все подстроки в строке, которые соответствуют регулярному выражению, и возвращает их в виде среза строк.

```go
package main

import (
    "fmt"
    "regexp"
)

func main() {
    r, err := regexp.Compile(`\d+`)
    if err != nil {
        fmt.Println("Ошибка компиляции регулярного выражения:", err)
        return
    }
    matches := r.FindAllString("123abc456", -1)
    fmt.Println(matches) // [123 456]
}
```

- `regexp.ReplaceAllString` — заменяет все соответствующие части строки.

```go
package main

import (
    "fmt"
    "regexp"
)

func main() {
    pattern := `\d+`
    str := "Диапазон от 1 до 10"
    // Заменяем все цифры на "X"
    r, err := regexp.Compile(pattern)
    if err != nil {
        fmt.Println("Ошибка компиляции регулярного выражения:", err)
        return
    }
    result := r.ReplaceAllString(str, "X")
    fmt.Println(result)
}
```

- Группы захвата

```go
package main

import (
    "fmt"
    "regexp"
)

func main() {
    // Регулярное выражение с группой захвата для даты в формате "dd.mm.yyyy"
    pattern := `(\d{2}).(\d{2}).(\d{4})`
    r, err := regexp.Compile(pattern)
    if err != nil {
        fmt.Println("Ошибка компиляции регулярного выражения:", err)
        return
    }
    // Поиск и извлечение данных
    result := r.FindStringSubmatch("01.12.2024")
    if len(result) > 0 {
        fmt.Println("День:", result[1])
        fmt.Println("Месяц:", result[2])
        fmt.Println("Год:", result[3])
    }
}
```

- Извлечение логина и домена из почтовых адресов

```go
package main

import (
    "fmt"
    "regexp"
)

func main() {
    pattern := `([a-zA-Z0-9._%+-]+)@([a-zA-Z0-9.-]+\.[a-zA-Z]{2,})`
    str := "contact@example.com, support@example.net"
    r, err := regexp.Compile(pattern)
    if err != nil {
        fmt.Println("Ошибка компиляции регулярного выражения:", err)
        return
    }
    matches := r.FindAllStringSubmatch(str, -1)
    for _, match := range matches {
        fmt.Printf("Логин: %s, Домен: %s\n", match[1], match[2])
    }
}
```