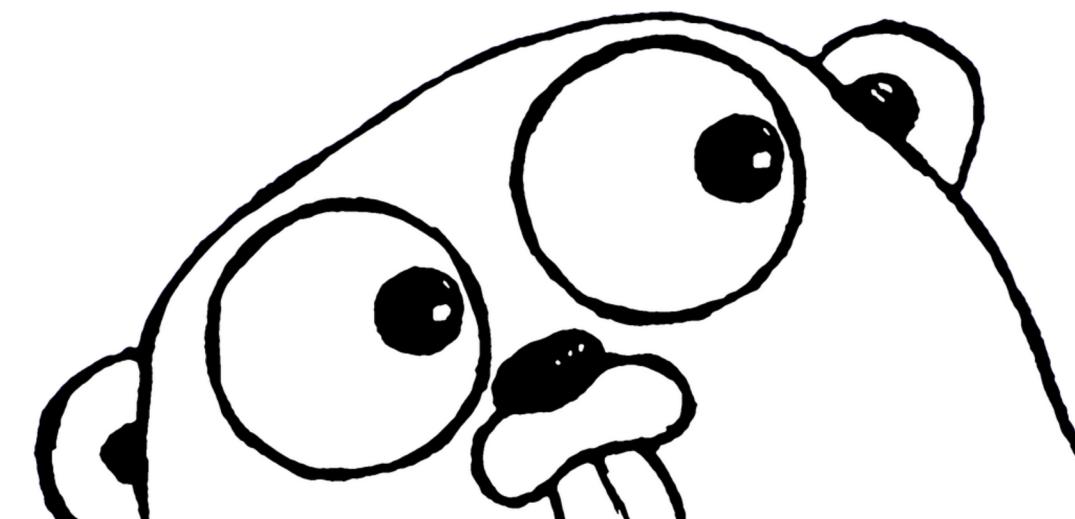


GO

Основы языка-2

Функции. Указатели. Производные типы. Массивы.

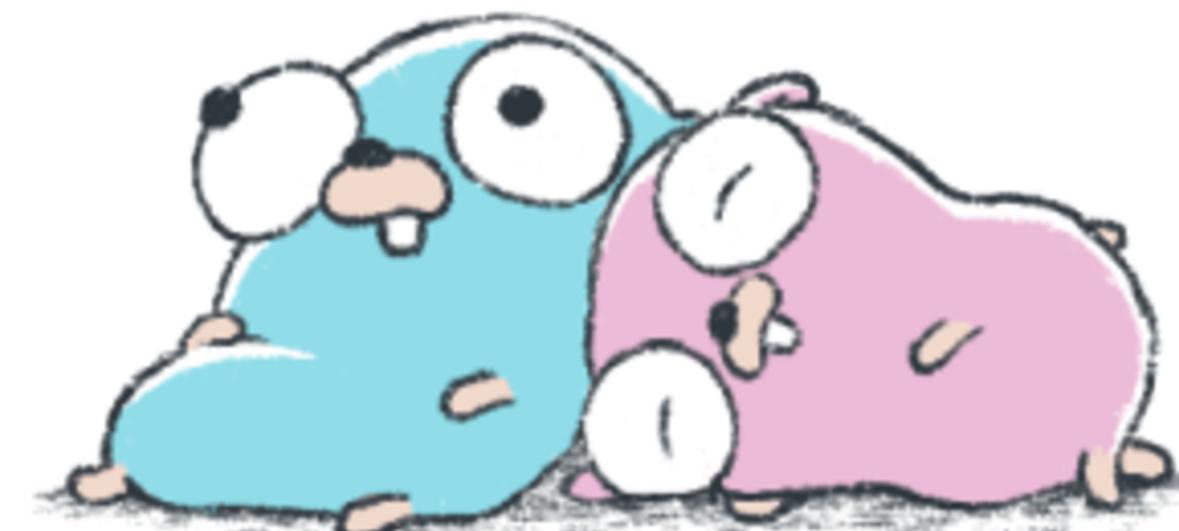


Что такое функция?

Функция – это блок операторов, выполняющий определённую задачу.
Позволяет повторно использовать группу операторов как единое целое.

```
func имя_функции(список_параметров) (типы_возвращаемых_значений) {  
    выполняемые_операторы  
}
```

- func – ключевое слово объявления функции.
- В скобках – список параметров (имя и тип).
- После параметров – тип(ы) возвращаемых значений (если есть).
- Название функции вместе с типами её параметров и возвращаемых значений называют сигнатурой.
- Сигнатура определяет, как функция вызывается и что она возвращает.



Функция main

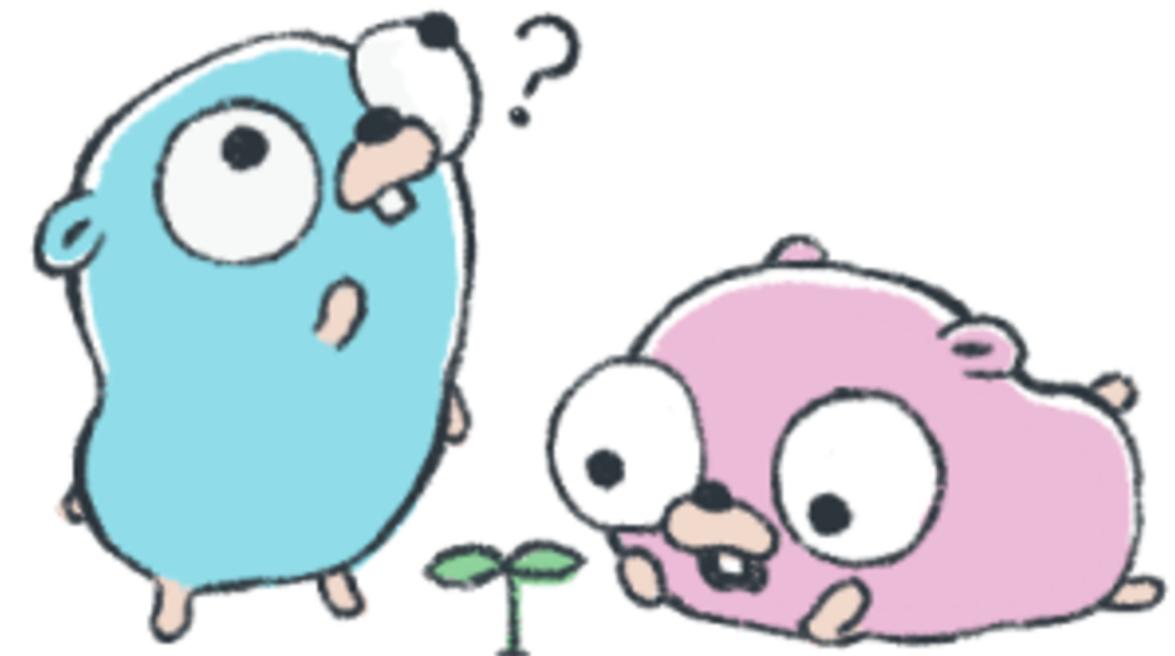
Точка входа

```
package main

import "fmt"

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

- Каждая программа Go должна иметь main в пакете main.
- **main() не принимает параметров и обычно не возвращает значения.**



Определение и вызов функции

Функция выполняется только при вызове (например, из main).

Вызов – имя функции + скобки (параметры, если есть).



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

Пример функции

Комментарии

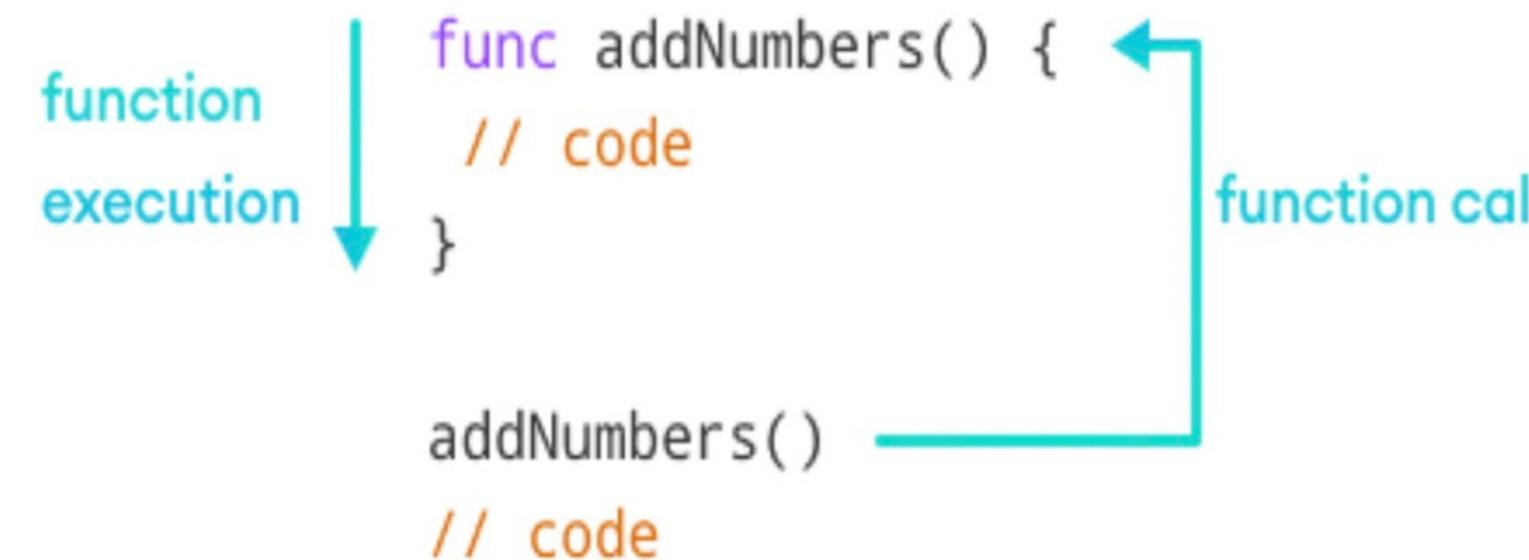
Мы создали функцию с именем addNumbers(). Функция складывает два числа и выводит сумму.

```
package main
import "fmt"

// function to add two numbers
func addNumbers() {
    n1 := 12
    n2 := 8

    sum := n1 + n2
    fmt.Println("Sum:", sum)
}

func main() {
    // function call
    addNumbers()
}
```



```
func addNumbers() {  
    n1 := 12  
    n2 := 8  
    sum := n1 + n2  
    fmt.Println("Sum:", sum)  
}
```

```
package main  
import "fmt"  
  
// define a function with 2 parameters  
func addNumbers(n1 int, n2 int) {  
    sum := n1 + n2  
    fmt.Println("Sum:", sum)  
}  
  
func main() {  
    // pass parameters in function call  
    addNumbers(21, 13)  
}
```

Параметры функции

Мы можем создавать функции, которые принимают внешние значения (параметры) и выполняют с ними операции.

- функция addNumbers() принимает два параметра: n1 и n2.
- int означает, что оба параметра имеют целочисленный тип.

```
func addNumbers(n1 int, n2 int) {  
    // code  
}  
  
addNumbers(21, 13)  
// code
```

function
parameters



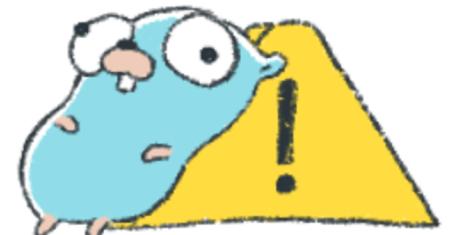
```
package main
import "fmt"

// define a function with 2 parameters
func addNumbers(n1 int, n2 int) {
    sum := n1 + n2
    fmt.Println("Sum:", sum)
}

func main() {
    // pass parameters in function call
    addNumbers(21, 13)
}
```

Параметры функции

Аргументы присваиваются параметрам функции при её вызове.



- Тип данных параметра функции всегда должен соответствовать данным, передаваемым при вызове функции. Здесь тип данных n1 и n2 – int, поэтому при вызове функции мы передаём целочисленные значения 21 и 13.

Возвращаемое значение из функции

В нашем предыдущем примере мы вывели значение суммы внутри самой функции. Однако мы также можем вернуть значение из функции и использовать его в любом месте нашей программы.

- int перед открывающей фигурной скобкой { указывает на тип возвращаемого функцией значения. В данном случае int означает, что функция вернёт целочисленное значение.
- А return sum – это оператор return, который возвращает значение переменной sum .

```
func addNumbers(n1 int, n2 int) int {  
    // code  
    return sum  
}
```

Функция возвращает значение в то место, откуда она была вызвана:

```
// function call  
result := addNumbers(21, 13)
```

```
func addNumbers(n1 int, n2 int) int {  
    // code  
    return sum  
}  
function  
return value  
result = addNumbers(21, 13)  
// code
```



Пример ошибки

Оператор `return` должен быть последним оператором в функции. При обнаружении оператора `return` функция завершается.

```
package main
import "fmt"

// function definition
func addNumbers(n1 int, n2 int) int {
    sum := n1 + n2
    return sum

    // code after return statement
    fmt.Println("After return statement")
}

func main() {
    // function call
    result := addNumbers(21, 13)
    fmt.Println("Sum:", result)
}
```

Возврат нескольких значений из функции

В Go мы также можем возвращать несколько значений из функции.

```
package main
import "fmt"

// function definition
func calculate(n1 int, n2 int) (int, int) {
    sum := n1 + n2
    difference := n1 - n2

    // return two values
    return sum, difference
}

func main() {
    // function call
    sum, difference := calculate(21, 13)

    fmt.Println("Sum:", sum, "Difference:", difference)
}
```



Области видимости переменной

*В Go мы можем объявлять
переменные в двух разных
областях видимости:
локальной и глобальной.*

Область видимости переменной определяет область,
в которой мы можем получить доступ к переменной.

```
package main

import "fmt"

var f float32 = 18
var g float32 = 4.5

func main() {

    var d float64 = 0.23
    var pi float64 = 3.14
    var e float64 = 2.7

    fmt.Println("f: ", f)
    fmt.Println("g: ", g)
    fmt.Println("d: ", d)
    fmt.Println("pi: ", pi)
    fmt.Println("e: ", e)
}
```

Локальные переменные

```
// Program to illustrate local variables

package main
import "fmt"

func addNumbers() {
    // local variables
    var sum int
    sum = 5 + 9
}

func main() {
    addNumbers()
    // cannot access sum out of its local scope
    fmt.Println("Sum is", sum)
}
```

Когда мы объявляем переменные внутри функции, эти переменные имеют локальную область видимости (внутри функции). Мы не можем получить к ним доступ за пределами функции.

Такие переменные называются локальными **переменными**

- Здесь переменная sum является локальной для функции addNumbers(), поэтому доступ к ней возможен только внутри функции.
- Вот почему мы получаем сообщение об ошибке при попытке обратиться к функции из main().
- Чтобы решить эту проблему, мы можем либо вернуть значение локальной переменной и присвоить его другой переменной внутри основной функции. Либо мы можем сделать переменную sum глобальной..

```
// Program to illustrate global variable

package main
import "fmt"

// declare global variable before main
function
var sum int

func addNumbers () {
    // local variable
    sum = 9 + 5
}

func main() {
    addNumbers()

    // can access sum
    fmt.Println("Sum is", sum)
}
```

Глобальные переменные

Когда мы объявляем переменные перед функцией `main()`, эти переменные имеют глобальную область видимости. Мы можем обращаться к ним из любой части программы.

Такие переменные называются глобальными **переменными**

- На этот раз мы можем получить доступ к переменной `sum` внутри функции `main()`. Это возможно, потому что мы создали переменную `sum` как глобальную.
- Теперь `sum` будет доступен из любой области (региона) программы.

Что такое указатель?

Указатель – это переменная, хранящая адрес другой переменной (представленный в виде шестнадцатеричного значения). Через указатель можно читать и менять значение по этому адресу.

```
package main
import "fmt"

func main() {
    var x int = 4          // обычная переменная
    var p *int = &x        // p хранит адрес x
    fmt.Println("Addr:", p) // адрес, например
0xc0000140b0
    fmt.Println("Value:", *p) // 4

    *p = 25                // меняем значение по
адресу
    fmt.Println("x =", x)   // x = 25
}
```

- Объявление: var p *int – p указывает на int.
- Получить адрес: &x
- Разыменование (доступ к значению): *p – чтение/запись.
- Типы указателей типобезопасны: *int ≠ *float64.
- Нулевой указатель: zero value = nil – нельзя разыменовывать (panic) без проверки.



Что такое производные типы?

Функция – это блок операторов, выполняющий определённую задачу.
Позволяет повторно использовать группу операторов как единое целое.

```
func имя_функции(список_параметров) (типы_возвращаемых_значений) {  
    выполняемые_операторы  
}
```

- func – ключевое слово объявления функции.
- В скобках – список параметров (имя и тип).
- После параметров – тип(ы) возвращаемых значений (если есть).
- Название функции вместе с типами её параметров и возвращаемых значений называют сигнатурой.
- Сигнатура определяет, как функция вызывается и что она возвращает.

```
package main
import "fmt"

type MyInt int // новый тип на базе int

func printInt(i int) { fmt.Println(i) }

func main() {
    var a MyInt = 10
    // printInt(a) // ошибка: нельзя неявно
    // преобразовать MyInt в int
    printInt(int(a)) // явное преобразование
}
```

Псевдонимы типов

В Go есть два понятия: *type alias* и *new type*.

type New = Existing – псевдоним (*alias*): *New* – просто другое имя для того же типа.

type New Existing – создание нового типа на основе существующего: *New != Existing*.

- Псевдонимы (=) полезны для переэкспорта типов из пакета или поддержки обратной совместимости.
- Новый тип – когда нужен собственный набор методов или строгая типизация.

```
package main
import "fmt"

type Person struct {
    Name string
    Age  int
}

func (p *Person) HaveBirthday() {
    p.Age++ // метод с указателем меняет саму
    структуру
}

func main() {
    p := Person{Name: "Ivan", Age: 30}
    fmt.Println(p)

    // передача по значению – копия
    q := p
    q.Name = "Petr"
    fmt.Println("p:", p)
    fmt.Println("q:", q)

    // метод с указателем
    p.HaveBirthday()
    fmt.Println("p after birthday:", p)
}
```

Struct (структуры)

struct – составной тип, объединяющий поля разных типов.

- Структуры можно создавать литералами, через new(Person) (возвращает *Person) или &Person{...}.
- Методы могут иметь приёмник по значению или по указателю; выбирайте по семантике изменений и стоимости копирования.

Массивы (arrays)

```
package main
import "fmt"

func main() {
    var a [3]int // массив из 3 элементов,
    инициализирован нулями
    a[0] = 1
    a[1] = 2
    fmt.Println(a, len(a))

    b := [3]int{10, 20, 30}
    // Копирование массива – полное копирование
    // данных
    c := b
    c[0] = 100
    fmt.Println("b:", b)
    fmt.Println("c:", c)
}
```

- Массив в Go – фиксированного размера: $[N]T$.
- Размер является частью типа: $[3]int$ и $[4]int$ – разные типы

Отличие от slice:

- Массив фиксирован по длине и при передаче в функцию копируется целиком.
- Слайс – это динамическое представление поверх массива (ссылка, len, cap).

Когда использовать массивы:

- Редко используются напрямую; полезны для статических, небольших буферов или когда важна семантика размера как части типа.

Slice (срез)

```
package main
import "fmt"

func main() {
    s := []int{1, 2, 3} // литерал
    fmt.Println(s, len(s), cap(s))

    s = append(s, 4, 5) // добавление элементов
    fmt.Println(s, len(s), cap(s))

    s2 := s[1:4] // срез: включает индексы 1..3
    fmt.Println(s2)

    s2[0] = 100 // изменяет базовый массив –
    // отражается в s
    fmt.Println("s:", s)
}
```

slice – динамическая, изменяемая структура над массивом. Хранит ссылку на массив, длину и ёмкость.

- Объявление: var s []int или s := []int{1,2,3}.
- append может выделить новый массив, если не хватает cap – старые ссылки не изменяются автоматически.
- Передача slice в функцию передаёт заголовок (не копируется весь массив), поэтому изменения элементов видны вызывающему.
- Для предварительного резервирования: make([]T, length, capacity).
- Чтобы скопировать: t := make([]T, len(s)); copy(t, s).

Неформальный чат

По материалам, вопросам и предложениям

