

# Введение в golang



Андрей  
Борю



## Андрей Борю

Principal DevOps Engineer, Snapcart



---

# План занятия

1. [Основы go lang](#)
2. [Синтаксис](#)
3. [Компиляция](#)
4. [Тестирование](#)
5. [Итоги](#)
6. [Домашнее задание](#)



# Основы `golang`

---

# Особенности golang

- **Простой и понятный синтаксис.** Это делает написание кода приятным занятием.
- **Статическая типизация.** Позволяет избежать ошибок, допущенных по невнимательности, упрощает чтение и понимание кода, делает код однозначным.
- **Скорость и компиляция.** Скорость у Go в десятки раз быстрее, чем у скриптовых языков, при меньшем потреблении памяти. При этом, компиляция практически мгновенна. Весь проект компилируется в один бинарный файл, без зависимостей.

---

## Особенности `golang`

- **Отход от ООП.** В языке нет классов, но есть структуры данных с методами. Наследование заменяется механизмом встраивания.
- **Параллелизм.** Параллельные вычисления в языке делаются просто, изящно и без головной боли. Горутины (что-то типа потоков) легковесны, потребляют мало памяти.
- **Богатая стандартная библиотека.** В языке есть все необходимое для веб-разработки и не только. Количество сторонних библиотек постоянно растет. Кроме того, есть возможность использовать библиотеки C и C++.

---

# Особенности golang

- **Возможность писать в функциональном стиле.** В языке есть замыкания (closures) и анонимные функции. Функции являются объектами первого порядка, их можно передавать в качестве аргументов и использовать в качестве типов данных.
- **Сильное комьюнити.** Сейчас у языка более 300 контрибьюторов. Язык имеет сильное сообщество и постоянно развивается.
- **Open Source.**

# Установка

- Менеджер пакетов (brew, apt, ...)

или:

- Скачиваем архив <https://golang.org/dl/> .
- Извлекаем его в папку **/usr/local**:

```
tar -C /usr/local -xzf go1.8.3.linux-amd64.tar.gz
```



---

# Переменные окружения

Добавляем папку `/usr/local/go/bin` в переменную окружения **PATH**:

- **`export PATH=$PATH:/usr/local/go/bin`**

---

# Проверяем корректность установки

Создадим файл **test.go**:

```
package main
import "fmt"

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

Запустим его: **go run test.go**



# Синтаксис

# Пакеты

Каждая программа на языке Go состоит из пакетов (**packages**).

Пакет **main** — главный, с него начинается выполнение программы.

В приведённом выше примере импортируется пакет **fmt**.

```
import "fmt"
import "math"
```

```
import (
    "fmt"
    "math"
)
```

# Функции

Общая форма определения функции выглядит следующим образом:

```
func function_name( [список_параметров] ) [возвращаемые_типы] {  
    тело_функции  
}
```

---

# Функции

Количество аргументов может быть разным.

```
package main
import "fmt"

func add(a int, b int) int {
    return a + b
}

func main() {
    fmt.Println("Сумма равна ", add(10, 19))
}
```

# Переменные

Определение переменной в Go означает передачу компилятору информации о типе данных, а так же о месте и объёме хранилища, которое создаётся для этой переменной. Определять переменные одного типа можно по одному и списком.

```
var [перечень переменных] [тип данных];
```

```
package main
import "fmt"
var node, golang, angular bool
func main() {
    var x int
    fmt.Println(x, node, golang, angular)
}
```

# Оператор цикла

В Go один оператор цикла — это **for**.

```
for [условие] {  
    [тело цикла]  
}
```

```
for [ (инициализация; условие; инкремент) ] {  
    [тело цикла]  
}
```

```
for [диапазон] {  
    [тело цикла]  
}
```



# Пример цикла

```
package main

import "fmt"

func main() {
    sum := 0
    for i := 0; i < 8; i++ {
        sum += i
    }
    fmt.Println("Сумма равна ", sum)
}
```

# Условный оператор

Форма определения условного оператора в Go выглядит так:

```
if [условие] {  
    ...  
}
```

Примеры условий:

- **true** — выполняется всегда;
- **a < 10** — выполняется, когда a меньше 10;
- **(a < b) || (a < c)** — выполняется, когда a меньше b или a меньше c;
- **(a < b) && (a < c)** — выполняется, когда a меньше b и a меньше c.

# Условный оператор

```
package main

import (
    "fmt"
)

func main() {
    if true {
        fmt.Println("Это выражение выполнится всегда")
    }

    if false {
        fmt.Println("Это выражение не выполнится никогда")
    }
}
```

# Массивы

Go также поддерживает массивы, которые представляют из себя структуру данных фиксированного размера, состоящую из элементов одного типа.

```
var наименование_переменной [размер] тип_переменной
```

```
var balance [10] float32
```

```
var balance = []float32{1000.0, 2.0, 3.4, 7.0, 50.0}
```

# Массивы

```
package main

import "fmt"

func main() {
    var a [2]string
    a[0] = "Привет"
    a[1] = "Netology"
    fmt.Println(a[0], a[1])
    fmt.Println(a)

    primes := [6]int{2, 3, 5, 7, 11, 13}
    fmt.Println(primes)
}
```

```
$ go run test.go
Привет Netology
[Привет Netology]
[2 3 5 7 11 13]
```

# Срезы

Срезы (Slices) в Go – абстракция над массивами. Хотя встроенных способов увеличить размер массива динамически или сделать вложенный массив в Go нет, срезы убирают это ограничение.

```
var numbers []int /* срез неопределённого размера */  
/* numbers = []int{0,0,0,0,0} */  
numbers = make([]int,5,5) /* срез длиной и ёмкостью равной 5*/
```

- **ёмкость (cap)** – это выделенная память под элементы, при превышении размер автоматически увеличивается в два раза.
- **длина (len)** – это инициализированная память элементов, для превышения (добавления) нужно вручную использовать append.

# Срезы

```
package main

import "fmt"

func main() {
    primes := [6]int{2, 3, 5, 7, 11, 13}
    fmt.Println(primes)

    var s []int = primes[1:4]
    fmt.Println(s)

    var numbers []int
    numbers = make([]int, 5, 5)
    fmt.Print(numbers)
}
```

```
$ go run test.go
[2 3 5 7 11 13]
[3 5 7]
[0 0 0 0 0]
```

# Структуры

Это пользовательский тип данных который комбинирует элементы разных типов. Чтобы объявить структуру, используем выражения **type** и **struct**:

- **Struct** определяет тип данных, которому соответствует два и более элементов.
- **Type** связывает заданное имя с описанием структуры.

```
type struct_name struct {  
    member definition;  
    ...  
    member definition;  
}
```

```
variable_name := struct_name {значение1, ...значениеN}
```



# Структуры

```
package main

import "fmt"

type Vertex struct {
    X int
    Y int
}

func main() {
    v := Vertex{1, 2}
    v.X = 4
    fmt.Println(v.X)
}
```

```
$ go run test.go
4
```



# Компиляция

# go build

Использование команды:

```
$ go build [-o output] [-i] [build flags] [packages]
```

Пример:

```
$ go build [-o output] [-i] [build flags] [packages]

$ cd ~/go/src/github.com/netology/devops
$ go build
$ ./devops
```



# gox

<https://github.com/mitchellh/gox>

Удобный инструмент для кросс-платформенной компиляции кода на golang.

```
$ go get github.com/mitchellh/gox
```

```
...
```

```
$ gox -h
```

```
...
```



# Тестирование

# Создадим функцию

Файл math.go

```
package math
import "fmt"

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

# Напишем для нее тест

Файл math\_test.go

```
package math

import "testing"

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



# Запускаем тесты

go test

```
$ go test
testing: warning: no tests to run
PASS
ok      github.com/netology/devops    0.981s
```



---

# Итоги

Что мы разобрали:

- Особенности и установка Golang
- Синтаксис основных конструкций
- Тестирование кода



# Домашнее задание



# Домашнее задание

Давайте посмотрим ваше [домашнее задание](#).

- Вопросы по домашней работе задавайте **в чате** мессенджера Slack.
- Задачи можно сдавать **по частям**.
- Зачёт по домашней работе проставляется после того, как **приняты все задачи**.

**Задавайте вопросы и  
пишите отзыв о лекции!**

**Андрей Борю**



[andreyborue](https://t.me/andreyborue)



[andreyborue](https://netology.ru/andreyborue)



[andreyborue](https://t.me/andreyborue)