

Go

Начальный курс по Go

СТРОЧНЫЙ ТИП

Строчный тип

string:

- `"""` - в двойных кавычках не могут содержать перенос строки
- ```` - могут быть многострочными

Escape-символы

Есть ряд символов, которые не имеют печатного аналога либо используются в служебных целях, для них используются специальные escape-символы:

`\t` – табуляция

`\n` – перенос строки

`\r` – возврат каретки

`\"` – двойная кавычка

`\'` – одинарная кавычка

`\\` – обратный слэш

ЛОГИЧЕСКИЙ ТИП

Логический тип

bool – true или false

ОПЕРАТОРЫ

Операторы	Ассоциативность
-> ++postfix --postfix [] . () вызов метода	->
! ~ ++prefix --prefix +унарный _унарный	<-
* / %	->
+ -	->
<< >>	->
< <= > >=	->
== !=	->
&	->
^	->
	->
&&	->
	->
= += -= *= /= %= &= = ^= <<= >>=	<-
,	->

ПЗ: Приоритеты операторов

```
count := 3
```

```
fmt.Println(10 % count + count ^ 2 + 22 * count)
```

ПЗ: Операторы

```
var count byte = 255
```

```
count++
```

```
fmt.Println(count) // понять, почему выводится такой результат
```

ОПЕРАТОРЫ

Операторы

Набор операторов фиксирован:

- нельзя создать новый оператор
- нельзя переопределить поведение существующего

АРИФМЕТИЧЕСКИЕ ОПЕРАТОРЫ

Арифметические

$a + b$ – сложение

$a - b$ – вычитание

$a * b$ – умножение

a / b – деление ($5 / 2 = 2$)

$a \% b$ – остаток от деления ($5 \% 2 = 1$)

Операторы

Важно: в зависимости от типа данных,
семантика может быть разная!

ЛИТЕРАЛЫ

Литералы

Литерал – непосредственное значение в коде

var percent float64 = 0.27

var count int = 27



Литералы

Литералы

`int` – по умолчанию для целых

`float64` – по умолчанию для вещественных

ПРИВЕДЕНИЕ ТИПОВ

Приведение типов

запомнить

В Go нет неявного приведения типов, если встречаются в одном выражении два разных типа, то приводить нужно вручную через выражения `int8(x)`, `int16(x)` и т.д.

КОНСТАНТЫ

Константы

const percent = 10

ЯЗЫКОВЫЕ КОНСТРУКЦИИ

Expression, Statement, Block

Expression – выражение, вычисляющее значение

Statement – единица исполнения

Block – блок кода, оформляется {}*

Примечание*: в Go это называется local explicit block

Видимость переменной

Область, в которой переменная доступна по имени

Видимость переменной

```
func main() {  
    outer := 127  
    {  
        fmt.Println(inner)  
        fmt.Println(outer)  
        inner := 10  
    }  
    fmt.Println(inner) // будет ошибка, inner здесь не видно  
}
```

Видимость переменной

```
func main() {
```

```
{
```

```
    variable := 10
```

```
    fmt.Println(variable)
```

```
}
```

```
{
```

```
    variable := 20
```

```
    fmt.Println(variable)
```

```
}
```

```
}
```

это разные переменные



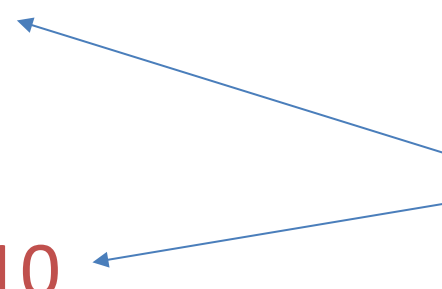
Видимость переменных

Сверху вниз по имени

Снизу вверх по блокам

ПЗ: Identifier shadowing

```
func main() {  
    variable := 127  
    {  
        variable := 10  
    }  
    fmt.Println(variable)  
}
```



это разные переменные

TODO & FIXME

TODO & FIXME

Специальные ключевые слова GoLand: если GoLand встречает их в комментариях, то выносит в отдельную панельку TODO

GoLand: Alt + 6

УСЛОВИЯ

if, else

```
if <expr> { ... }
```

```
if <expr> { ... } else { ... }
```

Важно:

1. <expr> всегда должно быть boolean!
2. Всегда ставьте { }* (кроме одного случая)

Ключевые операторы

`==` - проверка на равенство

`!=` - проверка на неравенство

`>` - больше

`<` - меньше

`>=` - больше или равно

`<=` - меньше или равно

Важно, результат всегда **boolean**!

МАССИВЫ

Массивы

Упорядоченный набор элементов одного типа*

```
var <name>[<length>]<type>
```

```
var data[10]int
```

- Имеет фиксированную длину (определяется `len()`)
- Элементы индексируются с нулевого индекса
- Доступ к элементу по индексу с помощью `[index]`

Важно

Массивы – это отдельный тип данных

FUNCTIONS

Функция

Именованный* блок кода – который может принимать какие-то параметры на вход и возвращать результат

Примечание*: могут быть и без имени

Объявление функции

```
func max(a int, b int) int {  
    // TODO:  
    return ...  
}
```

параметры функции

имя функции

тип результата

возврат результата

Вызов функции

```
func main() {  
    result := max(1, 20)  
}
```

аргументы функции

имя функции

Краткий экскурс в тестирование

UNIT-ТЕСТИРОВАНИЕ

Tools

В Go уже встроены необходимые
инструменты тестирования

Введение в тестирование

Классы эквивалентности:

1. На базе входных значений
2. На базе выходного результата

Unit

Automatic defect prevention + тестирование
изолированное тестирование
функциональности

Плюсы автотестов

- Автоматический запуск
- Можно быть уверенным, что хотя бы что-то работает

Минусы автотестов

- Ложная уверенность
- "У тестов нет глаз"

A-A-A

Общий подход:

1. Подготовка данных (Arrange)
2. Выполнение операций (Act)
3. Проверка результата (Assert)

Unit

Общие принципы:





1. Тестировать изолированно
2. Проверять лучше одно значение в одном тесте*
3. Тесты должны быть простыми для написания
(иначе надо редизайнить сам класс)

Demo в IDE

`ctrl + shift + t` (курсор должен быть на имени функции) -> test for function

```
func nps(scores [3]int) int {
```

Choose Test for **nps** (0 found)

-  Empty test file
-  Test for function
-  Tests for file
-  Tests for package

```
    if value >= promotersLowerBound {
```

Важно

Тесты должны храниться в файле с
постфиксом `_test` (например, `main_test.go`)

Важно

1. Когда ищите какой-то элемент, помните, что он может быть не один
2. Поэтому уточняйте: вам нужны все, первый или последний

SLICES

Слайсы

```
var slice []int
```

```
slice := []int{}
```

nil - слайс

```
slice := make([]int, 5)
```

```
slice := make([]int, 5, 10)
```

```
slice := []int{1, 2, 3}
```

Операции

Добавление в слайс (даже пустой)

```
slice = append(slice, 1)
```

```
slice = append(slice, another...)
```

Операции

Срезы

`copy := slice[:]`

`sub := slice[2:4]`

`head := slice[:1]`

`tail := slice[len(slice) - 2:]`

Операции

Важно: слайс указывает на массив (до тех пор пока Go не решит, что для этого слайса нужно создать новый массив, например, при добавлении в него элементов)

Операции

Желательно не изменять слайс (либо менять копию)

Копию создать можно через функцию **copy**

STRUCT

struct

Тип, который может сгруппировать разные типы элементов:

```
student := struct {  
    name string  
    score int  
} {  
    "Oleg", 5,  
}
```

описание полей структуры

инициализированные значения

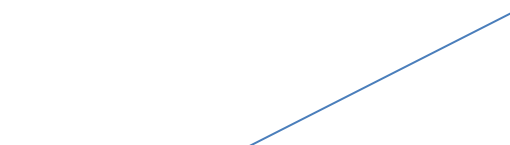
Операции

```
student.name = "Oleg Petrov"
```



запись данных в поле

```
fmt.Println(student.name)
```



чтение данных

struct

Тип, который может сгруппировать разные типы элементов:

```
students := []struct {  
    name string  
    score int  
} {  
    {"Oleg", 5},  
    {"Vasya", 4},  
}
```

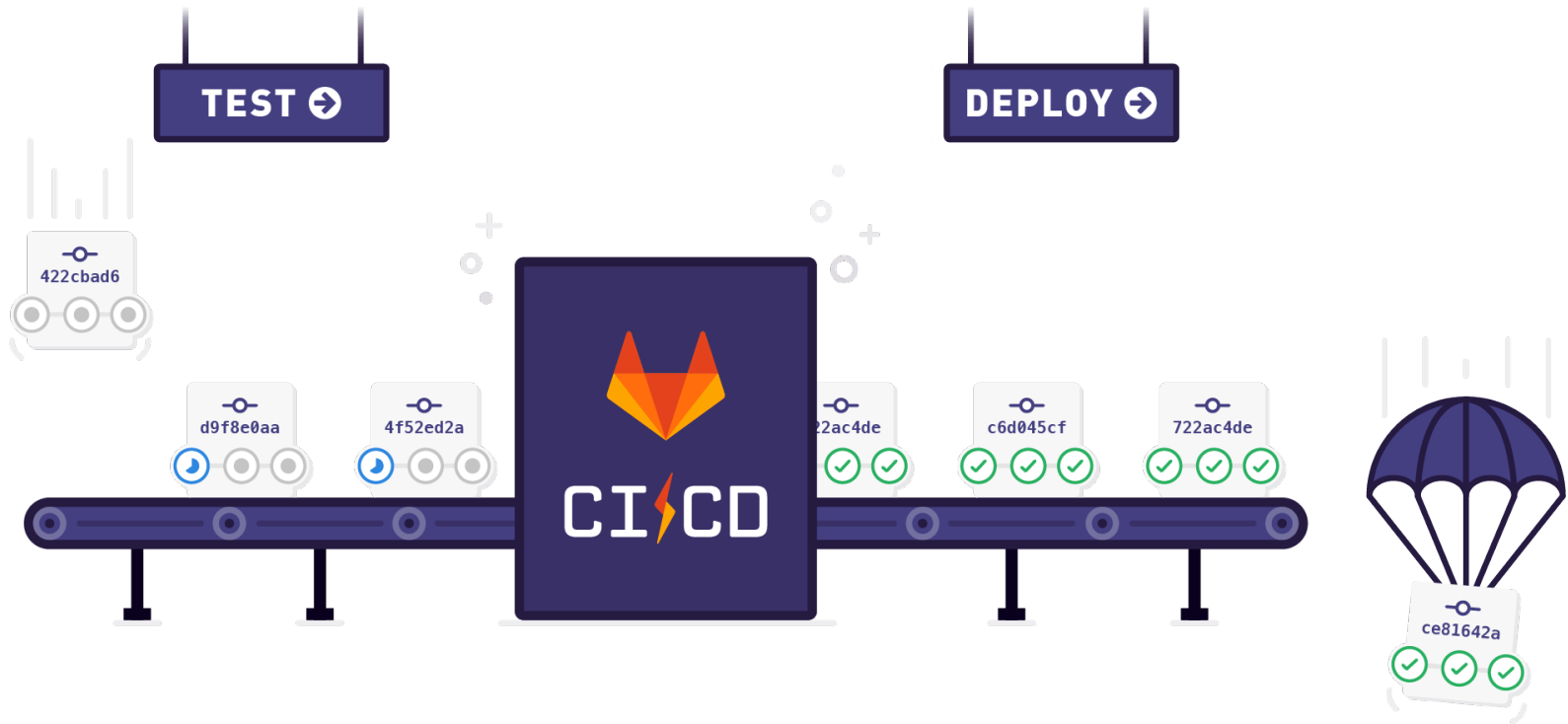
описание полей структуры

инициализированные значения

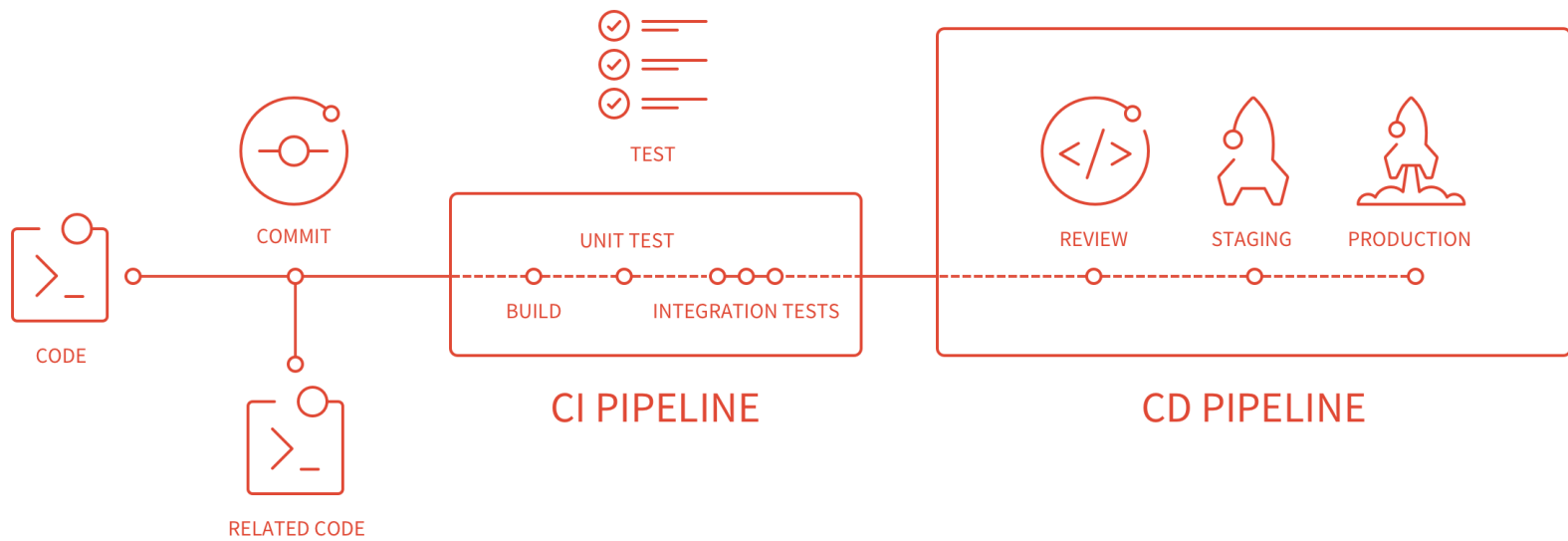
Continuous Integration

CI

CI



CI



Continuous Integration

Непрерывная интеграция – практика, при которой разработчик интегрирует свой код в репозиторий, т.е. запускаются авто-тесты и другие проверки

По шагам

1. Пишете код и функции
2. Пишете авто-тесты на функции (Ctrl + Shift + T)
3. В терминале (Alt + F12) запускаете авто-тесты, удостоверьтесь, что проходят **go test -v**:

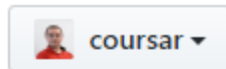
```
=== RUN   Test_nps
--- PASS: Test_nps (0.00s)
PASS
ok      nps      0.012s
```

По шагам

1. Создаёте репозиторий: `git init`
2. Создаёте файл `.gitignore`
3. Добавляете все файлы: `git add .`
4. Делаете коммит: `git commit -m "initial commit"`
5. Создаёте репозиторий на Github

Git remote

Owner



Repository name *

/ go-demo ✓

Задаёте только имя

(должно соответствовать проекту)

Great repository names are short and memorable. Need inspiration? How about **ideal-happiness**?

Description (optional)



Public

Anyone can see this repository. You choose who can commit.



Private

You choose who can see and commit to this repository.

Все настройки оставляете как здесь

Skip this step if you're importing an existing repository.



Initialize this repository with a README

This will let you immediately clone the repository to your computer.

Add .gitignore: None ▾

Add a license: None ▾



Create repository

Git remote

...or push an existing repository from the command line

```
git remote add origin https://github.com/coursar/go-demo.git  
git push -u origin master
```

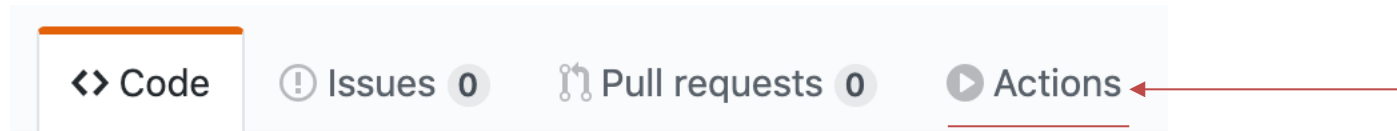


Важно:

1. push можно делать только если у вас есть хотя бы один commit
2. push -u origin master нужно делать только один раз (дальше можно писать просто push)

Добавляете Github Actions

В Github пункт меню "Actions":



Добавляете Github Actions

Build and test your Go repository

Go


Build a Go project.



Set up this workflow

```
go get -v -t -d ./...  
if [ -f Gopkg.toml ]; then  
curl https://raw.githubusercontent.com/golang/dep/master/install.sh  
| sh
```

 actions/starter-workflows

Go 

В самый низ добавляете:

```
26
27     - name: Build
28       run: go build -v .
29
30     - name: Test
31       run: go test -v
```

Это надо добавить
(следите за отступами!)

После чего нажимаете кнопку "Start commit":

Start commit ▼

По шагам

Дальше на своём компьютере делаете: `git pull`

После чего работаете, как обычно

Спасибо за внимание

Ильназ Гильязов

2020г.