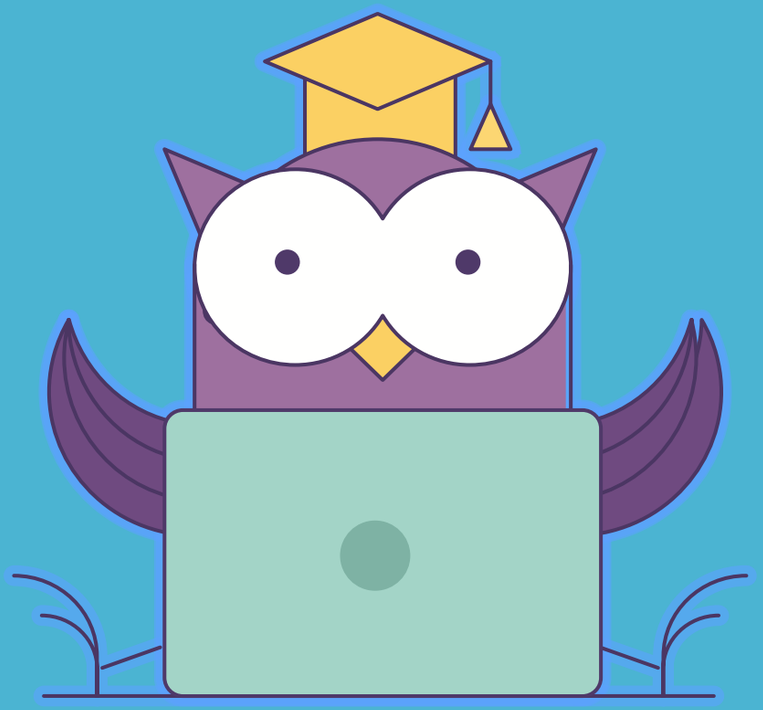
ОНЛАЙН-ОБРАЗОВАНИЕ

Как меня слышно и видно?



> Напишите в чат

+ если все хорошо

- если есть проблемы со звуком или с видео

!проверить запись!

Инструментарий Go, тестирование

Дмитрий Смаль

План занятия

- Подробнее про GOPATH и GOROOT
- Сборка модулей и установка программ: go get, go build, go install
- Кросс-компиляция
- Модули и зависимости: go mod
- Форматирование кода: go fmt, goimports
- Линтеры: go vet, golint, металинтеры
- Тестирование Go программ
- Пакет testing
- Паттерны и анти-паттерны unit-тестирования

Установка Go

Проще всего через `apt-get`

```
sudo apt-get update

sudo apt-get install golang
```

Есть возможность просто скачать с оф. сайта

```
wget https://dl.google.com/go/go1.13.linux-amd64.tar.gz

sudo tar -C /usr/local -xzf go1.13.linux-amd64.tar.gz

sudo ln -s /usr/local/go/bin/go /usr/bin/go
```

Готово!

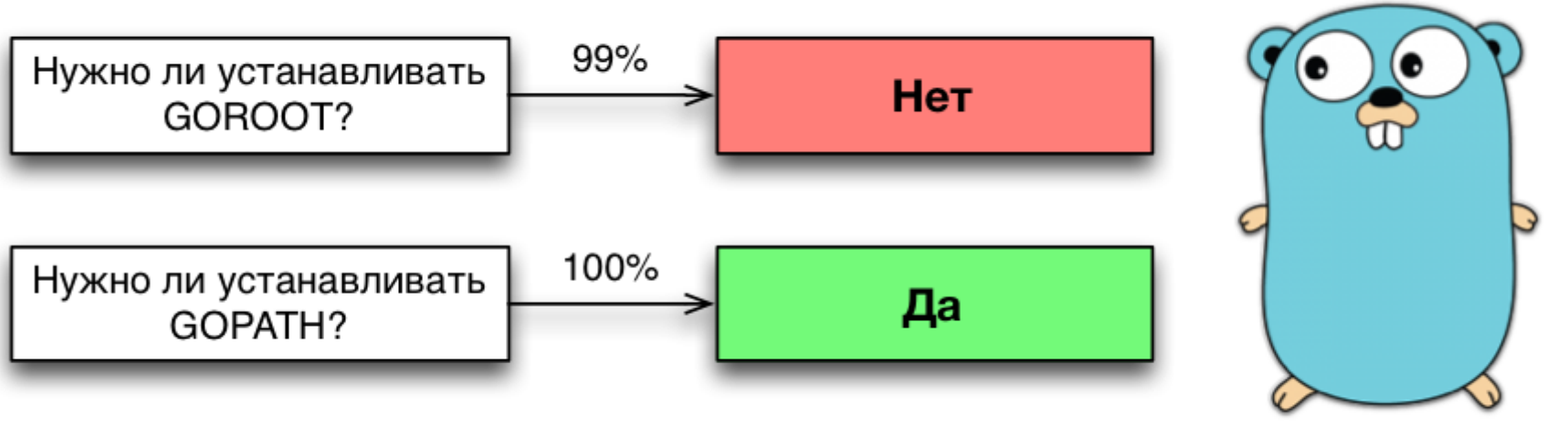
`GOROOT` - переменная, которая указывает где лежит ваш дистрибутив Go, т.е. компилятор, утилиты и стандартная библиотека. В новых версия Go (> 1.0) утилиты сами определяют расположение Go.

Однако, вы можете узнать `GOROOT`

```
go env
...
GOROOT="/usr/local/go"
...
```

И можете посмотреть исходный код Go =)

```
vim /usr/local/go/src/runtime/slice.go
```


Нужно ли устанавливать GOROOT?
99%
Нет
Нужно ли устанавливать GOPATH?
100%
Да

`GOPATH` - переменная окружения, показывает где лежит ваше дерево исходников.

Крайне желательно задать эту переменную явно, добавив в `.bashrc` например

```
export GOPATH=/path/your/go/projects
```

Однако, если не задать, то `GOPATH` будет предполагаться `/home/<username>/go`

Загрузка пакетов

Давайте выполним команду: `go get -d github.com/golang/protobuf/...`

```
$ tree -L 5 ~/go
/home/mialinx/go
└── src
    └── github.com
        └── golang
            └── protobuf
                ├── AUTHORS
                ├── CONTRIBUTORS
                ├── descriptor
                ├── go.mod
                ├── go.sum
                ├── jsonpb
                ├── LICENSE
                ├── Makefile
                ├── proto
                ├── protoc-gen-go
                ├── ptypes
                ├── README.md
                └── regenerate.sh
....
```

Установка пакетов

Теперь выполним команду `go install github.com/golang/protobuf/...`

```
$ tree -L 6 ~/go
/home/mialinx/go
├── bin
│   └── protoc-gen-go
├── pkg
│   └── linux_amd64
│       └── github.com
│           └── golang
│               └── protobuf
│                   ├── descriptor.a
│                   ├── jsonpb
│                   ├── jsonpb.a
│                   ├── proto
│                   ├── proto.a
│                   ├── protoc-gen-go
│                   ├── ptypes
│                   └── ptypes.a
└── src
    └── github.com
        └── golang
            └── protobuf
....
```

`go get -d` - скачивает пакеты из Git репозиториев в `$GOPATH/src`.

`go install` собирает и устанавливает в указанные пакеты в `$GOPATH/pkg` и `$GOPATH/bin`.

`go get` (без флажка `-d`) - так же вызовет `install`.

Многоточия

`github.com/golang/protobuf/...` <= многоточие тут означает "и все дочерние пакеты".

Это необходимо если в пакет сложный, и содержит под-пакеты.
Для простых достаточно `go get github.com/beevik/ntp`

Точечная сборка

`go build` - команда более низкого уровня, заново компилирующая выбранный пакет.

Например:

```
$ go build -o /tmp/thelib.a github.com/beevik/ntp

$ file /tmp/thelib.a
thelib.a: current ar archive
```

Или:

```
$ go build -o /tmp/prog github.com/golang/protobuf/protoc-gen-go

$ file /tmp/prog
prog: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), statically linked, not stripped
```

Результат сборки зависит от пакета (`main` - executable, любой другой - библиотека)

Кросс-компиляция

Go позволяет легко собирать программы для других архитектур и операционных систем.
Для этого при сборке нужно переопределить переменные `GOARCH` и `GOOS`:

```
$ GOOS=windows go build -o /tmp/prog github.com/golang/protobuf/protoc-gen-go

$ file /tmp/prog
prog: PE32+ executable (console) x86-64 (stripped to external PDB), for MS Windows

$ GOARCH=386 GOOS=darwin go build -o /tmp/prog github.com/golang/protobuf/protoc-gen-go

$ file /tmp/prog
prog: Mach-O i386 executable
```

Возможные значения `GOOS` и `GOARCH` https://gist.github.com/asukakenji/f15ba7e588ac42795f421b48b8aede63

Единое дерево кода

Для работы в парадигме `GOPATH` нужно:
- Создать *публичный* проект `github.com/username/projectname`
- Скачать проект в `GOPATH` с помощью `go get github.com/username/projectname/...`
- Изменять, компилировать, и комитить проект из `$GOPATH/src/github.com/username/projectname`

Плюсы:
- Простота. Плоская структура
- Отсутствие версий (?) (master должен быть стабилен).

Минусы:
- Отсутствие версий (!)
- Иногда в проекте не только Go-код.
- Неудобно для корпоративных и других непубличных проектов.

Начиная с Go 1.11 появилась поддержка модулей - системы версионирования и зависимостей, а также разработки вне `GOPATH`.

Стандартные команды (`go get`, `go install`, `go test` и т.д.) работают по-разному внутри модуля и внутри `GOPATH`.

Модуль - любая директория вне `GOPATH`, содержащая файл `go.mod`

Создание Go модуля

- (Опционально) создайте и склонируйте (в любое место) репозиторий с проектом

```
git clone https://github.com/user/otus-go.git /home/user/otus-go
```

- Создайте внутри репозитория нужные вам директории

```
mkdir /home/user/otus-go/hw-1
```

- Зайдите в директорию и инициализируйте Go модуль

```
cd /home/user/otus-go/hw-1
go mod init github.com/user/otus-go/hw-1
```

Теперь `/home/user/otus-go/hw-1` - это Go модуль.

Добавление зависимостей

Внутри модуля, вы так можете добавить пакет точно так же

```
$ go get github.com/beevik/ntp
go: finding golang.org/x/net latest
```

Но при этом пакет попадет не в `$GOPATH/src`, а в `$GOPATH/pkg/mod`

```
$ tree -L 4 ~/go/pkg
/home/mialinx/go/pkg
└── mod
    ├── cache
    │   └── download
    │       ├── github.com
    │       └── golang.org
    ├── github.com
    │   └── beevik
    │       └── ntp@v0.2.0
    └── golang.org
        └── x
            ├── net@v0.0.0-20190827160401-ba9fcec4b297
            └── sys@v0.0.0-20190215142949-d0b11bdaac8a
```

Версии зависимостей

Внутри Go модуля обязательно находится файл `go.mod`, содержащий информацию о версии и зависимостях

```
$ cat go.mod
module github.com/mialinx/foobar

go 1.13

require (
    github.com/beevik/ntp v0.2.0 // indirect
    golang.org/x/net v0.0.0-20190827160401-ba9fcec4b297 // indirect
)
```

А так же `go.sum`, содержащий чек-суммы зависимостей.

Внимание (!) версии зависимостей - фиксируются в момент добавления.

Авто-добавление

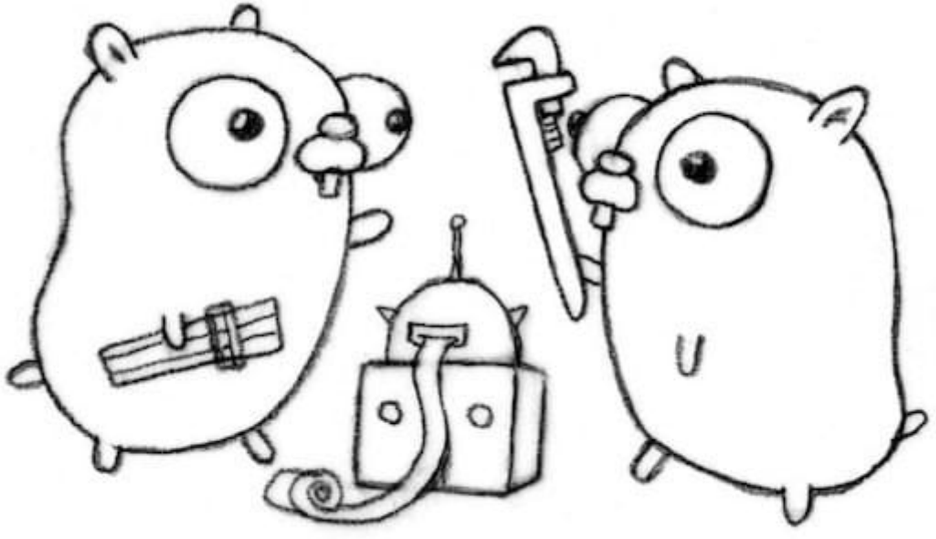
Есть еще более простой способ управлять зависимостями: просто редактируйте код

```
package main
import (
    "fmt"
    "github.com/go-loremipsum/loremipsum"
)
func main() {
    fmt.Println(loremipsum.New().Word())
}
```

А потом запустите

```
$ go mod tidy
```

Это добавит новые и удалит неиспользуемые зависимости



One-shot запуск

Запустить файл "как скрипт".

```
go run ./path/to/your/snippet.go
```

Удобно для проверки кода и синтаксиса.

Так же можно использовать Go PlayGround: https://play.golang.org/p/Fz3j-hbcocv

Получение справки

```
$ go help
Go is a tool for managing Go source code.

Usage:

    go <command> [arguments]

The commands are:
    bug         start a bug report
    build       compile packages and dependencies
    clean       remove object files and cached files
    doc         show documentation for package or symbol
...

$ go help build
usage: go build [-o output] [-i] [build flags] [packages]

Build compiles the packages named by the import paths,
along with their dependencies, but it does not install the results.
...
```

Форматирование кода

В Go нет style guide, зато есть `go fmt path/to/code.go`

Было:

```
package main
import "fmt"



const msg = "%d students in chat\n"
type Student struct{
    Name string
    Age int
}
func main() {
    for i:=99;i>0;i-- {
        fmt.Printf(msg, i, i)
        if i<10{
            break
        }

    }
}
```

Стало:

```
package main

import "fmt"

const msg = "%d students in chat\n"

type Student struct {
    Name string
    Age  int
}

func main() {
    for i := 99; i > 0; i-- {
        fmt.Printf(msg, i, i)
        if i < 10 {
            break
        }

    }
}
```

Обновление импортов с помощью `goimports`

```
$ go get golang.org/x/tools/cmd/goimports
$ ~/go/bin/goimports -w path/to/code.go
```

```
import (
    "strings"
)

func main() {
    fmt.Println(loremipsum.New().Word())
}
```

```
import (
    "fmt"

    "github.com/go-loremipsum/loremipsum"
)

func main() {
    fmt.Println(loremipsum.New().Word())
}
```

Линтер - программа, анализирующая код и сообщающая о потенциальных проблемах.

`go vet` - встроенный линтер

```
$ go vet ./run.go
# command-line-arguments
./run.go:14:3: Printf call needs 1 arg but has 2 args

$ echo $?
2
```

`golint` - популярный сторонний линтер

```
$ go get -u golang.org/x/lint/golint

$ ~/go/bin/golint -set_exit_status ./run.go
run.go:7:6: exported type Student should have comment or be unexported
Found 1 lint suggestions; failing.

$ echo $?
1
```

Металинтеры - обертка, запускающая несколько линтеров за один проход.

```
$ go get github.com/golangci/golangci-lint/cmd/golangci-lint
```

```
$ ~/go/bin/golangci-lint run ./run.go
run.go:14:3: printf: Printf call needs 1 arg but has 2 args (govet)
        fmt.Printf(msg, i, i)
        ^
run.go:7:6: `Student` is unused (deadcode)
type Student struct {
     ^

$ echo $?
1
```

Подробнее https://github.com/golangci/golangci-lint


OK!
NO!

Особые суффиксы файлов

```
$ tree ~/go/src/github.com/boltdb/bolt/
/home/user/go/src/github.com/boltdb/bolt/
├── bolt_386.go
├── bolt_amd64.go
...
├── bolt_linux.go
├── bolt_windows.go
...
├── bucket.go
├── bucket_test.go
...
├── db.go
├── db_test.go
...
```

- `_386`, `_amd64` - только для указанной архитектуры
- `_linux`, `_windows` - только для указанной OS
- `_test` - тесты для пакета

Пример кода

OTUS

В файле `foobar/count.go`

```
package foobar

func Count(s string, r rune) int {
    var cnt int
    for _, l := range s {
        if l == r {
            cnt += 1
        }
    }
    return cnt
}
```

Пример теста

В файле `foobar/count_test.go`

```
package foobar

import "testing"

func TestCount(t *testing.T) {
    s := "qwerasdfe"
    e := 2
    if c := Count(s, 'e'); c != e {
        t.Fatalf("bad count for %s: got %d expected %d", s, c, e)
    }
}
```

Важно:

- `package` тот же что и в основном коде
- Все тесты - *экспортируемые* функции, начинающиеся с букв `Test`
- Тесты принимают `t` - API для установки результата тестирования

Запуск тестов

OTUS

Тест текущей директории

```
$ cd project/foobar
$ go test
PASS
ok      _/home/user/foobar    0.011s
```

Тест произвольного пакета

```
$ go get github.com/gorilla/mux
$ go test github.com/gorilla/mux
ok      github.com/gorilla/mux    0.031s
```

Вместе со вложенными пакетами

```
go test google.golang.org/grpc/...
```

```
t.Fail()   // отметить тест как сломаный, но продолжит выполнение
t.FailNow()  // отметить тест как сломаный и прекратить текущий тест
t.Logf(formar string, ...interface{})  // вывести сообщение с отладкой
t.Errorf(formar string, ...interface{})  // t.Logf + t.Fail
t.Fatalf(formar string, ...interface{})  // t.Logf + t.FailNow
t.SkipNow()  // пропустить тест
```

Пакет `github.com/stretchr/testify`

OTUS

```go
package foobar

import (
    "testing"

    "github.com/stretchr/testify/require"
)

func TestCount(t *testing.T) {
    s := "qwerasdfe"

    require.Equal(t, Count(s, 'e'), 2, "counting 'e' in "+s)

    require.Equal(t, Count(s, 'x'), 0, "counting 'x' in "+s)

    require.Equal(t, Count(s, 'f'), 0, "counting 'f' in "+s)
}
```

Вывод testify

```
$ go test
--- FAIL: TestCount (0.00s)
    require.go:157:
            Error Trace:    count_test.go:16
            Error:          Not equal:
                            expected: 1
                            actual  : 0
            Test:           TestCount
            Messages:       counting 'f' in qwerasdfe
FAIL
exit status 1
FAIL    _/Users/mialinx/foobar    0.016s
```

В данном случае ошибка в тесте (!)

- Тесты должны быть достоверными и исчерпывающим
- Тесты - это тоже код (readability/maintainability)
- Один тест - один тестовый случай
- Тесты должны быть изолированы (нет веерных поломок)
- Тесты должны отрабатывать быстро

Типичные ошибки тестирования

- Вывод чего-либо на экран, для сравнения глазами
- Слишком много проверок в одном тесте
- Зависимости от внешних файлов (вне репозитория)
- Неполное тестирование

Опрос

Заполните пожалуйста опрос

https://otus.ru/polls/4892/

Спасибо за внимание!