



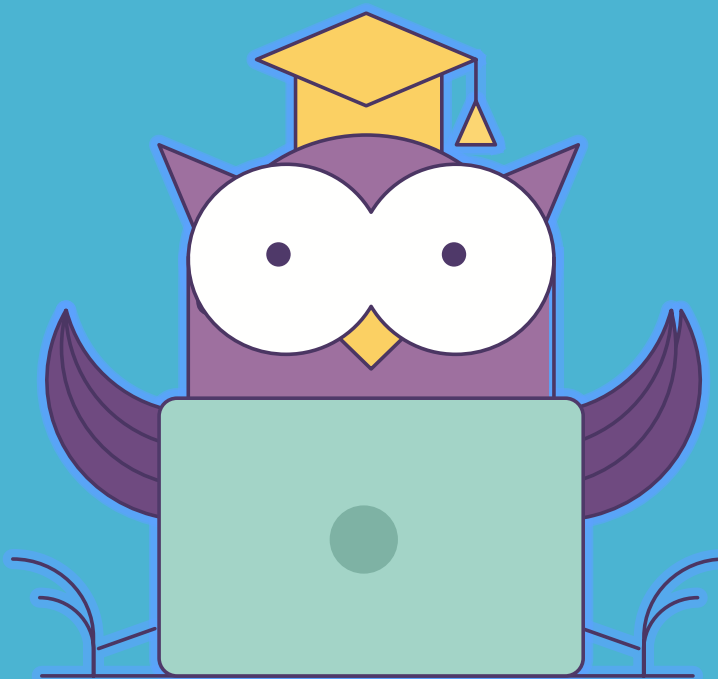
ОНЛАЙН-ОБРАЗОВАНИЕ

Взаимодействие с OS

Антон Телышев



Как меня слышно и видно?



> Напишите в чат

+ если все хорошо

- если есть проблемы со звуком или с видео

!проверить запись!



- Ведущий Golang-разработчик в “Центр недвижимости от Сбербанка” (ДомКлик)
- Окончил магистратуру МГТУ им Н.Э. Баумана (кафедра “Компьютерные системы и сети”)
- Разрабатывал и поддерживал системы мониторинга и внутренние сервисы в Mail.Ru Group (Python/Golang)
- Разрабатывал и поддерживал Kaspersky Fraud Prevention Cloud (Python/C++)
- Окончил Технопарк Mail.Ru, где впоследствии преподавал Подготовительную программу по C++
- Telegram: **@antonboom**

- Обработка аргументов командной строки: flags, pflags, cobra
- Работа с переменными окружения
- Запуск внешних программ
- Работа с файловой системой
- Временные файлы
- Обработка сигналов

<https://golang.org/pkg/flag/>

```
import "flag"

// *int
var ip = flag.Int("flagname", 1234, "help message for flagname")
```

```
var flagvar int
func init() {
    flag.IntVar(&flagvar, "flagname", 1234, "help message for flagname")
}
```

```
// После определения всех флагов, вызовите Parse()
flag.Parse()

fmt.Println("ip has value ", *ip)
fmt.Println("flagvar has value ", flagvar)
```

flag: синтаксис в командной строке

Разрешенные формы

```
-flag  
-flag=x  
-flag x // Не для boolean флагов (fix: -flag=false)
```

Целочисленные флаги

```
1234, 0664, 0x1234, -100
```

Логические флаги

```
1, 0, t, f, T, F, true, false, TRUE, FALSE, True, False
```

Можно два минуса

```
-flag  
--flag
```

<https://github.com/spf13/pflag>

Установка

```
$ go get github.com/spf13/pflag  
$ go test github.com/spf13/pflag
```


В чем преимущества `pflag` ?

- POSIX стиль флагов (`--flag`)
- Однобуквенные сокращения (`--verbose` и `-v`)
- Можно отличать флаг без значения от незаданного (`--buf` от `--buf=1` от)



Просто заменяем flag на библиотеку и все работает

```
import flag "github.com/spf13/pflag"

// Указатель!
var ip *int = flag.Int("flagname", 1234, "help message for flagname")

// Просто значение
var flagvar int
var verbose bool

func init() {
    flag.IntVar(&flagvar, "flagname", 1234, "help message for flagname")\

    // Заметьте суффикс P в имени функции
    flag.BoolVarP(&verbose, "verbose", "v", true, "help message")
}

func main() {
    flag.Parse()
}
```

```
var ip = flag.IntP("flagname", "f", 1234, "help message")
func init() {
    flag.Lookup("flagname").NoOptDefVal = "4321"
}
```

Флаг

--flagname=1357

--flagname

[nothing]

Значение

ip=1357

ip=4321

ip=1234

POSIX-like:

```
--flag      // Логические флаги или флаги без значений по умолчанию  
--flag x     // Только для флагов без значений по умолчанию  
--flag=x
```

Использование одного дефиса отличается от пакет `flag`.

В пакете `flag` текст `-abc` это флаг `abc`.

В пакете `pflag` текст `-abc` это набор из трех флагов `a`, `b`, `c`.

Многие CLI приложения имеют команды и подкоманды.

```
docker help
docker run --help

git add file1 file2
git commit -m 123

aws s3 ls s3://bucket-name
```

CLI имеют как общие флаги (`--verbose`), так и специфичные для подкоманд.

Фреймворк **Cobra** (github.com/spf13/cobra) позволяет сильно упростить написание таких CLI.



<https://github.com/golang-standards/project-layout>

Расположение файлов

```
▼ appName/  
  ▼ pkg/  
  ▼ internal/  
  ▼ cmd/  
    add.go  
    your.go  
    commands.go  
    here.go  
  main.go
```

Располагается в `appName/main.go`

```
package main

import (
    "github.com/username/appName/cmd"
)

func main() {
    cmd.Execute()
}
```

Корневая команда обычно располагается в `appName/cmd/root.go`

```
var cfgFile, projectBase string

var rootCmd = &cobra.Command{
    Use:     "hugo",
    Short:   "Hugo is a very fast static site generator",
    Run:     func(cmd *cobra.Command, args []string) {
        // Основной код команды, имеет смысл не раздувать...
    },
}

func init() {
    // Флаги для всех команд и подкоманд
    rootCmd.PersistentFlags().StringVar(&cfgFile, "config", "", "config file (default")
    rootCmd.PersistentFlags().StringVarP(&projectBase, "projectbase", "b", "", "base p")
}

func Execute() {
    if err := rootCmd.Execute(); err != nil {
        fmt.Println(err)
        os.Exit(1)
    }
}
```


Подкоманды располагаются в соответствующих файлах, например в `appName/cmd/add.go`

```
package cmd

import (
    "fmt"
    "github.com/spf13/cobra"

    "source string"

    addCmd = &cobra.Command{
        Use:     "add",
        Short:   "Adds some files to storage",
        Run: func(cmd *cobra.Command, args []string) {
            runAdd(cmd, args)
        },

        func init() {
            addCmd.Flags().StringVarP(&source, "source", "s", "", "Source directory to read from")
            rootCmd.AddCommand(addCmd)
        }
    }
```

Получаем бинарь кобры

```
go get github.com/spf13/cobra/cobra
```

Генерируем команды

```
cobra init --pkg-name github.com/Antonboom/example  
cobra add serve  
cobra add config
```

Получившееся приложение

```
▼ app/  
  ▼ cmd/  
    serve.go  
    config.go  
    root.go  
    main.go
```

Подробнее: <https://github.com/spf13/cobra/blob/master/cobra/README.md>

Переменные окружения - набор строк, которые передаются в программу при запуске.

```
# посмотреть текущие переменные окружения
$ env

# запустить программу prog с дополнительной переменной
$ NEWVAR=val prog

# запустить программу prog с чистым окружением и переменной NEWVAR
$ env -i NEWVAR=val prog
```

Реальное применение (Travis CI)

```
script:
  - env GO111MODULE=on go build ./...
```

Как получить переменные окружения ?

```
import (  
    "os"  
    "fmt"  
)  
  
func main() {  
    var env []string  
    env = os.Environ() // слайс (!) строк  
    fmt.Println(env[0]) // NEWVAR=val  
  
    var newvar string  
    newvar, ok := os.LookupEnv("NEWVAR")  
    fmt.Printf(newvar) // val  
  
    os.Setenv("NEWVAR", "val2") // установить  
    os.Unsetenv("NEWVAR")       // удалить  
    fmt.Printf(os.ExpandEnv("$USER have a ${NEWVAR}")) // "шаблонизация"  
}
```

Команды:

- set
- env
- declare
- export

Для запуска внешних команд используется пакет `os/exec`. Основной тип - `Cmd`.

```
type Cmd struct {  
    // Путь к запускаемой программе  
    Path string  
  
    // Аргументы командной строки  
    Args []string  
  
    // Переменные окружения (слайс!)  
    Env []string  
  
    // Рабочая директория  
    Dir string  
  
    // Поток ввода, вывода и ошибок для программы (/dev/null если nil!)  
    Stdin io.Reader  
    Stdout io.Writer  
    Stderr io.Writer  
    ...  
}  
  
cmd := exec.Command("prog", "--arg=1", "arg2")
```

`cmd.Run()` запускает команду и дожидается ее завершения.

```
cmd := exec.Command("sleep", "1")
err := cmd.Run()
// ошибка запуска или выполнения программы
log.Printf("Command finished with error: %v", err)
```

`cmd.Start()` запускает программу, но не дожидается завершения.

`cmd.Wait()` дожидается завершения.

```
err := cmd.Start()
if err != nil {
    log.Fatal(err) // ошибка запуска
}
log.Printf("Waiting for command to finish...")
err = cmd.Wait() // ошибка выполнения
log.Printf("Command finished with error: %v", err)
```

С помощью `cmd.Output()` можно получить STDOUT выполненной команды.

```
out, err := exec.Command("date").Output()
if err != nil {
    log.Fatal(err)
}
fmt.Printf("The date is %s\n", out)
```

С помощью `cmd.CombinedOutput()` можно получить STDOUT и STDERR (перемешанные).

Как сделать аналог bash команды `ls | wc -l` ?

```
import (  
    "os"  
    "os/exec"  
)  
  
func main() {  
    c1 := exec.Command("ls")  
    c2 := exec.Command("wc", "-l")  
    pipe, _ := c1.StdoutPipe()  
    c2.Stdin = pipe  
    c2.Stdout = os.Stdout  
    _ = c1.Start()  
    _ = c2.Start()  
    _ = c1.Wait()  
    _ = c2.Wait()  
}
```

Сигналы - механизм OS, позволяющий посылать уведомления программе в особых ситуациях.

Сигнал	Поведение	Применение
SIGTERM	Завершить	<code>ctrl+c</code> в консоли
SIGKILL	Завершить	<code>kill -9</code> , остановка зависших программ
SIGHUP	Завершить	Сигнал для переоткрытия логов и перечитывания конфига
SIGUSR1		На усмотрение пользователя
SIGUSR2		На усмотрение пользователя
SIGPIPE	Завершить	Отправляется при записи в закрытый файловый дескриптор
SIGSTOP	Остановить	При использовании отладчика
SIGCONT	Продолжить	При использовании отладчика

Некоторые сигналы, например `SIGTERM` , `SIGUSR1` , `SIGHUP` , можно игнорировать или установить обработчик.

Некоторые, например `SIGKILL` , обработать нельзя.

```
import (  
    "fmt"  
    "os"  
    "os/signal"  
    "syscall"  
)  
  
func signalHandler(c <-chan os.Signal) {  
    s := <- c  
    // TODO: handle  
    fmt.Println("Got signal:", s)  
}  
  
func main() {  
    c := make(chan os.Signal, 1)  
    signal.Notify(c, syscall.SIGUSR1)  
    signal.Ignore(syscall.SIGINT)  
    go signalHandler(c)  
    businessLogic()  
}
```

В пакете `os` содержится большое количество функций для работы с файловой системой.

```
// изменить права доступа к файлу
func Chmod(name string, mode FileMode) error

// изменить владельца
func Chown(name string, uid, gid int) error

// создать директорию
func Mkdir(name string, perm FileMode) error

// создать директорию (вместе с родительскими)
func MkdirAll(path string, perm FileMode) error

// переименовать файл/директорию
func Rename(oldpath, newpath string) error

// удалить файл (пустую директорию)
func Remove(name string) error

// удалить рекурсивно rm -rf
func RemoveAll(path string) error
```

Иногда бывает необходимо создать временный файл, для сохранения в нем данных.

```
import (  
    "io/ioutil"  
    "log"  
    "os"  
)  
  
func main() {  
    content := []byte("temporary file's content")  
    // файл будет создан в os.TempDir, например /tmp/example-Jsm22jkn  
    tmpfile, err := ioutil.TempFile("", "example-")  
    if err != nil {  
        log.Fatal(err)  
    }  
    defer os.Remove(tmpfile.Name()) // не забываем удалить  
    if _, err := tmpfile.Write(content); err != nil {  
        log.Fatal(err)  
    }  
    if err := tmpfile.Close(); err != nil {  
        log.Fatal(err)  
    }  
}
```

Реализовать утилиту envdir на Go.

Эта утилита позволяет запускать программы получая переменные окружения из определенной директории. Пример использования:

```
go-envdir /path/to/env/dir some_prog
```

Если в директории /path/to/env/dir содержатся файлы

- `A_ENV` с содержимым `123`
- `B_VAR` с содержимым `another_val` То программа `some_prog` должать быть запущена с переменными окружения `A_ENV=123 B_VAR=another_val`

Заполните пожалуйста опрос

<https://otus.ru/polls/4902/>



Спасибо за внимание!

