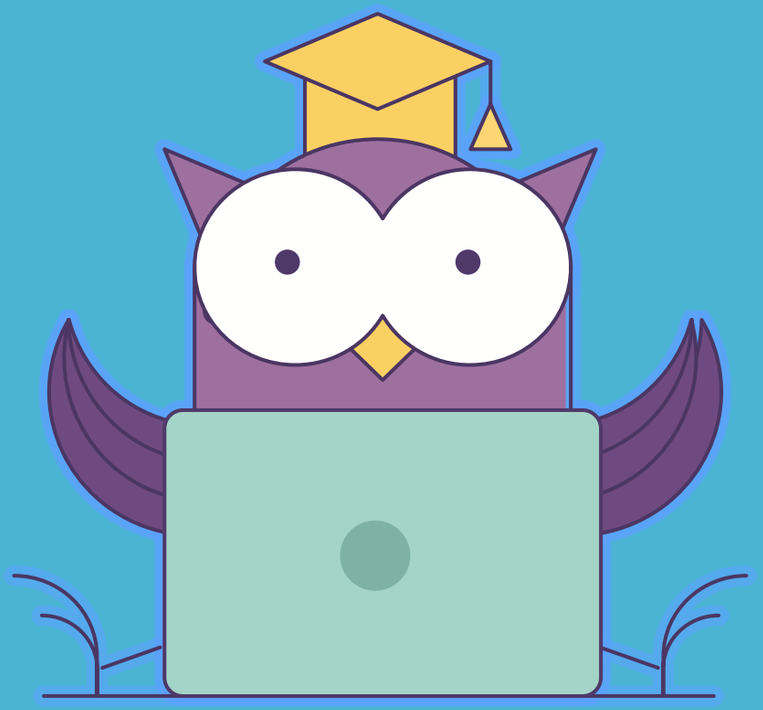
ОНЛАЙН-ОБРАЗОВАНИЕ

OTUS

Взаимодействие с OS

Дмитрий Смаль

Как меня слышно и видно?



> Напишите в чат

+ если все хорошо

- если есть проблемы со звуком или с видео

!проверить запись!

План занятия

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

pflag вместо flag

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 ``)

Использование pflag

```go
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()
}
```

Флаги без значений

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

Флаг	Значение
--flagname=1357	ip=1357
--flagname	ip=4321
[nothing]	ip=1234

POSIX-like:

```
--flag    // boolean flags, or flags with no option default values
--flag x  // only on flags without a default value
--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.

```
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.

Расположение файлов

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

Функция `main`

Располагается в `appName/main.go`

```
package main

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

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

Корневая команда

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 is $HOME/.cobra.yam
  rootCmd.PersistentFlags().StringVarP(&projectBase, "projectbase", "b", "", "base project directory eg
}

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

Подкоманды

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

```
package cmd

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

var source string

var 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)
}
```

Переменные окружения

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

```
# посмотреть текущие переменные окружения
$ env

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

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

Как получить переменные окружения ?

```go
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"
}
```

Использование внешних программ

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")
```

Запуск Cmd

`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).

Более сложный пример

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	`Ctrl+C` в консоли
SIGKILL	Завершить	`kill -9`, 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.

```go
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
```

Временные файлы

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)
    }
}
```

Домашнее задание

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`

Опрос

Заполните пожалуйста опрос

https://otus.ru/polls/3938/

Спасибо за внимание!