

Golang

Сборка, builtin, пакеты



Информация

Данная публикация стала возможной благодаря помощи американского народа, оказанной через Агентство США по международному развитию (USAID). Алиф Академия несёт ответственность за содержание публикации, которое не обязательно отражает позицию USAID или Правительства США.



ПРЕДИСЛОВИЕ



Предисловие

Сегодня мы поговорим о том, как "собирать" наши программы, а также про организацию кода в Go: а именно, разделение на пакеты.



КОМПИЛЯЦИЯ И СБОРКА



Компиляция и сборка

Давайте разбираться, как всё устроено. До этого у нас был только `main.go`, сейчас же мы посмотрим на общую схему:

1. У нас есть файлы с расширением `.go` - это файлы, в которых расположен код на языке Go
2. Мы отдаём эти файлы специальной программе (компилятору), которая из них делает исполняемую программу



Примечание*: мы пока только условно будем называть эту программу компилятором, поскольку там всё немного сложнее.



Исполняемый файл

Что значит исполняемый файл? Это файл со специальной внутренней структурой, который понимает операционная система и может запустить как программу.

Например, в Windows - EXE, в Linux - ELF.

Компиляция и сборка как раз занимаются тем, что ваши текстовые файлы собирает в одну программу и один исполняемый файл, который вы можете запускать через командную строку.

Поскольку у наших приложений нет графического интерфейса, попытка запустить их двойным кликом мыши приведёт лишь к тому, что на короткое мгновение откроется консоль (вы не успеете ничего увидеть).



Компиляция и сборка

```
go main.go > ...
```

```
1 package main
2
3 func main() {
4     println("Hello, Go!")
5 }
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
C:\projects\packages>go mod init packages
go: creating new go.mod: module packages
```

```
C:\projects\packages>go build ← компиляция и сборка
```

```
C:\projects\packages>dir
Том в устройстве C не имеет метки.
Серийный номер тома: EA50-EE70
```

Содержимое папки C:\projects\packages

```
25.08.2020 15:56 <DIR> .
25.08.2020 15:56 <DIR> ..
25.08.2020 15:56      25 go.mod
25.08.2020 14:57      60 main.go
25.08.2020 15:56    1 225 216 packages.exe
          3 файлов    1 225 301 байт
          2 папок   54 960 005 120 байт свободно
```

```
C:\projects\packages>packages.exe ← запуск файла
Hello, Go!
```



Компиляция и сборка

В Linux/Mac всё работает аналогично, единственное изменение - файлы там не имеют расширения `.exe` и запускаются как `./имя_файла`:

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
//> packages go mod init packages ← компиляция и сборка
go: creating new go.mod: module packages
//> packages go build
//> packages dir
go.mod main.go packages
//> packages ./packages ← запуск файла
Hello, Go!
//> packages
```



package main

Команды с предыдущий слайдов будут работать только тогда, когда в вашем файле `main.go` есть:

1. `package main`
2. `func main() { ... }`

В других случаях получить исполняемый файл не получится. В Go, как и во многих других языках, используется принцип соглашений: "вместо того, чтобы что-то настраивать, делай так, как принято". Так вот авторы языка приняли решение, что должны соблюдаться эти два условия.



Командная строка

Командная строка, терминал или шелл (shell) - это интерфейс, в котором вы можете выполнять команды и запускать приложения. В противоположность ему есть графический интерфейс - где вы кликаете мышкой на иконки и т.д.

При разработке и эксплуатации приложений (но не при написании кода) чаще всего используется именно интерфейс командной строки: некоторые приложения предоставляют только его (например, уже знакомая вам команда `go`).



go tools

С командой go мы будем работать очень плотно, поэтому начнём с того, что научимся читать справку (её обязательно нужно читать):

```
C:\projects\packages>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
env	print Go environment information
fix	update packages to use new APIs
fmt	gofmt (reformat) package sources
generate	generate Go files by processing source
get	add dependencies to current module and install them
install	compile and install packages and dependencies
list	list packages or modules
mod	module maintenance
run	compile and run Go program
test	test packages
tool	run specified go tool
version	print Go version
vet	report likely mistakes in packages

Use "go help <command>" for more information about a command.



go tools

Посмотрим справку по **go build**:

```
C:\projects\packages>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.

If the arguments to build are a list of .go files from a single directory, build treats them as a list of source files specifying a single package.

When compiling packages, build ignores files that end in '_test.go'.

When compiling a single main package, build writes the resulting executable to an output file named after the first source file ('go build ed.go rx.go' writes 'ed' or 'ed.exe') or the source code directory ('go build unix/sam' writes 'sam' or 'sam.exe'). The '.exe' suffix is added when writing a Windows executable.

When compiling multiple packages or a single non-main package, build compiles the packages but discards the resulting object, serving only as a check that the packages can be built.

→ The -o flag forces build to write the resulting executable or object to the named output file or directory, instead of the default behavior described in the last two paragraphs. If the named output is a directory that exists, then any resulting executables will be written to that directory.



go build

Т.е. используя флаг `-o`, мы можем сразу указать, файл с каким именем нужно создавать на выходе (по умолчанию название файла берётся из названия модуля - того, что вы прописали в `go mod init`):

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
C:\projects\packages>go build -o main.exe
```

```
C:\projects\packages>main.exe  
Hello, Go!
```

```
C:\projects\packages>
```



ПАКЕТЫ



Пакеты

Пока у нас был только один файл - [main.go](#). А теперь представьте, что вы пишете проект уже 2 или 3 года и кода там накопилось на пару миллионов строк.

Держать всё в одном файле в таком случае не очень разумно, поскольку если вы работаете в команде хотя бы из двух человек, вам каждый раз придётся "сливать" изменения.




Пакеты

Давайте посмотрим, как с этим справляются самые популярные проекты, написанные на Go: [Docker](#) и [Kubernetes](#).



Docker
















AkihiroSuda

Merge pull request [moby#40982](#) from AkihiroSuda/containerd14b...

...


















✖ 7ae5222 4 days ago

 38,899 commits

	.github	Remove refs to jhowardmsft from .go code	11 months ago
	api	Merge pull request moby#41284 from akerouanton/service-ulimits	26 days ago
	builder	Builder: print relative path if COPY/ADD source path was not found	8 days ago
	cli	remove uses of deprecated pkg/term	4 months ago
	client	Check for context error that is wrapped in url.Error	22 days ago
	cmd/dockerd	API: add "prune" events	28 days ago
	container	Merge pull request moby#40856 from cpuguy83/reduce_allocs_on_env...	4 months ago
	contrib	Add multi-user.target to After=	27 days ago
	daemon	Merge pull request moby#41335 from thaJeztah/remove_unneeded_ev...	7 days ago
	distribution	Disable manifest v2 schema 1 push	25 days ago
	dockerversion	Use -X ldflags to set dockerversion package vars	9 months ago
	docs	Merge pull request moby#41284 from akerouanton/service-ulimits	26 days ago



Kubernetes

 k8s-ci-robot Merge pull request #94171 from robscott/endpointslicemirroring...  4db3a09 12 hours ago  93,349 commits		
 .github	Add sigs for root folders	12 days ago
 CHANGELOG	Merge pull request #93972 from puerco/changelog-fix-117	11 days ago
 LICENSES	Update json-patch to v4.9.0 tagged release	5 days ago
 api	Mark componentstatus as deprecated	26 days ago
 build	build: Update to go-runner:buster-v2.0.0	4 days ago
 cluster	Add rbac patch permissions for system:controller:glbc ingresses/status	4 days ago
 cmd	Sign up dims for additional review roles	17 days ago
 docs	Add sigs for root folders	12 days ago
 hack	Fix building with GOFLAGS=-v	10 days ago
 logo	Add sigs for root folders	12 days ago
 pkg	Merge pull request #94171 from robscott/endpointslicemirroring-fix	12 hours ago
 plugin	Sign up dims for additional review roles	17 days ago
 staging	Merge pull request #94146 from MikeSpreitzer/limit-lag	4 days ago
 test	Merge pull request #94171 from robscott/endpointslicemirroring-fix	12 hours ago



Пакеты

Мы видим, что весь проект
разделён на множество каталогов,
внутри которых будут ещё файлы и
каталоги:



```
└─ KUBERNETES
  ├── .github
  ├── api
  ├── build
  ├── CHANGELOG
  ├── cluster
  └─ cmd
    ├── clicheck
    ├── cloud-controller-manager
    └─ app
      ├── apis
      ├── config
      ├── scheme
      │   ├── BUILD
      │   └── scheme.go
      ├── v1alpha1
      │   ├── BUILD
      │   ├── doc.go
      │   ├── register.go
      │   ├── types.go
      │   └── zz_generated.deepcopy.go
      ├── config
      └── options
```



Пакеты

Зачем так сделано? Затем, чтобы было проще работать в команде и проще ориентироваться в файлах. Поэтому мы будем делать с вами точно так же.

Самому Go практически всё равно (исключения мы посмотрим), как и в каких каталогах расположены ваши файлы. Поэтому сообщество выработало свои рекомендации по поводу организации структуры проекта: [Standard Go Project Layout](#).

Поскольку многие проекты следуют этим рекомендациям, мы их тоже будем придерживаться.



cmd

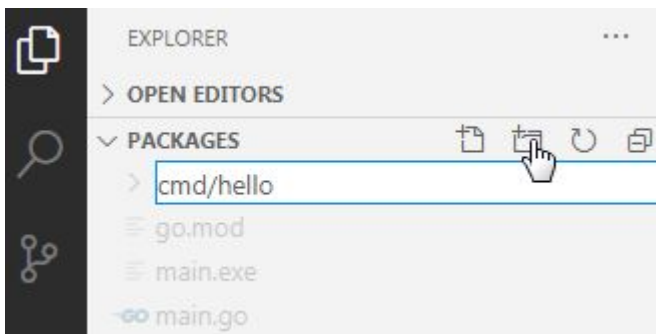
Пока рекомендация будет сводиться к следующему:

1. В нашем проекте мы должны создать каталог `cmd`
2. В этом каталоге будет каталог с названием файла программы, который мы хотим получить (например, `hello`)
3. Внутри каталога из п.2 будет файл `main.go`
4. Компиляцию и сборку мы будем производить командой `go build cmd/hello`



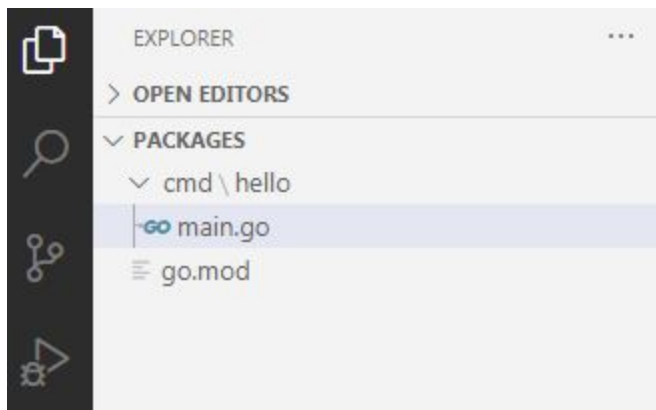
cmd

Что за cmd? В cmd хранятся файлы, на основе которых генерируются исполняемые файлы (exe'шники). К одному проекту может генерироваться много таких файлов (см. слайд с Kubernetes):



← сразу будет создан каталог и подкаталог

После чего просто "перетащить" файл **main.go** в этот каталог:



←



Компиляция и сборка

Теперь команда будет следующая:

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
C:\projects\packages>go build ./...
```

```
C:\projects\packages>hello.exe  
Hello, Go!
```

```
C:\projects\packages>█
```

Что означает `go build ./...`? Это такой специальный синтаксис, который позволяет вам пройтись по всем вложенным каталогам и скомпилировать и собрать все `main` в исполняемые файлы. При этом название будет соответствовать названию каталога, в котором лежит файл.



cmd

Начиная с сегодняшнего дня бот будет требовать наличие каталога `cmd` и файлов `main.go` в нём.



СТАНДАРТНАЯ БИБЛИОТЕКА

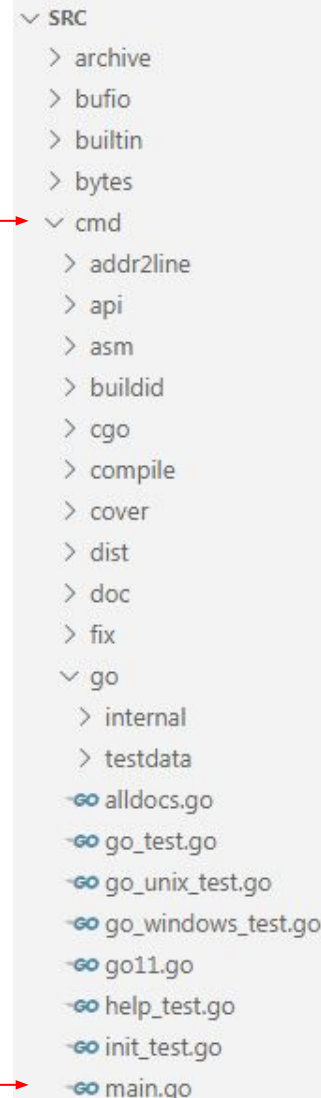


Стандартная библиотека

Пока у нас только один файл `main.go`, зачем же нам вся эта история с пакетами?

Всё дело в том, что стандартная библиотека Go так же разбита на пакеты (откройте `C:\Go\src` в VS Code).

Важно: будьте аккуратны и не редактируйте там файлы (иначе ваш Go будет отличаться от тех, что есть у бота и других пользователей - результат работы после компиляции вашего приложения будет разный)!



```
▼ SRC
  > archive
  > bufio
  > builtin
  > bytes
  > cmd
  > addr2line
  > api
  > asm
  > buildid
  > cgo
  > compile
  > cover
  > dist
  > doc
  > fix
  ▼ go
    > internal
    > testdata
    -go alldocs.go
    -go go_test.go
    -go go_unix_test.go
    -go go_windows_test.go
    -go go11.go
    -go help_test.go
    -go init_test.go
    -go main.go
```



Стандартная библиотека

Стандартная библиотека - это уже готовый код, написанный авторами языка для того, чтобы вы могли быстрее разрабатывать. Например, в нём уже есть готовые функции для создания серверных приложений, работы с криптографией и т.д.

Но давайте по-порядку. У нас с вами есть типы данных и функция `println*`, которую мы с вами использовали. Откуда они берутся?

Примечание*: функция - это кусочек кода, у которого есть имя. И эту функцию можно "запускать", так же как вы на своём смартфоне можете запустить воспроизведение аудио или видео.



builtin

Всё, что мы до этого использовали, описано в пакете `builtin` (файл `builtin/builtin.go`).



```
/*  
    Package builtin provides documentation for Go's predeclared identifiers.  
    The items documented here are not actually in package builtin  
    but their descriptions here allow godoc to present documentation  
    for the language's special identifiers.  
*/
```

```
→ package builtin
```

```
// bool is the set of boolean values, true and false.
```

```
type bool bool
```

```
// true and false are the two untyped boolean values.
```

```
const (
```

```
    true  = 0 == 0 // Untyped bool.
```

```
    false = 0 != 0 // Untyped bool.
```

```
)
```

```
// uint8 is the set of all unsigned 8-bit integers.
```

```
// Range: 0 through 255.
```

```
type uint8 uint8
```

```
...
```

```
// The print built-in function formats its arguments in an
```

```
// implementation-specific way and writes the result to standard error.
```

```
// Print is useful for bootstrapping and debugging; it is not guaranteed
```

```
// to stay in the language.
```

```
func print(args ...Type)
```

```
// The println built-in function formats its arguments in an
```

```
// implementation-specific way and writes the result to standard error.
```

```
// Spaces are always added between arguments and a newline is appended.
```

```
// Println is useful for bootstrapping and debugging; it is not guaranteed
```

```
// to stay in the language.
```

```
→ func println(args ...Type)
```

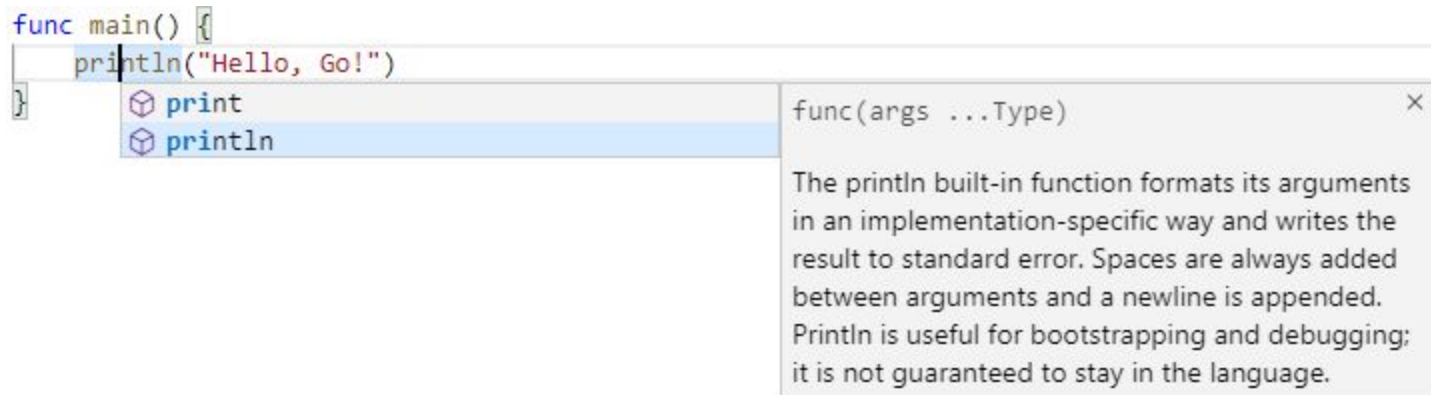


builtin

builtin - это особенный пакет. По факту, в нём просто описана документация того, что **автоматически доступно в любом другом пакете**. Например, та же функция **println**, **int**, **string** и т.д.

Обратите внимание: документация пишется в комментариях и вы через VS Code можете её читать (курсор внутри **println** и **Ctrl + пробел**):

```
func main() {  
    println("Hello, Go!")  
}
```



The screenshot shows a code editor with a Go function `main` containing a `println` call. A dropdown menu is visible below the `println` call, listing `print` and `println`. A tooltip is open next to the `println` entry, displaying its signature `func(args ...Type)` and a detailed description of its behavior.

`func(args ...Type)`

The `println` built-in function formats its arguments in an implementation-specific way and writes the result to standard error. Spaces are always added between arguments and a newline is appended. `Println` is useful for bootstrapping and debugging; it is not guaranteed to stay in the language.



println

Если внимательно прочитать то, что написано в документации на функцию `println`, то написано, что она используется только в целях отладки. Т.е. использовать так, как использовали мы - не стоит.

Что же тогда использовать?

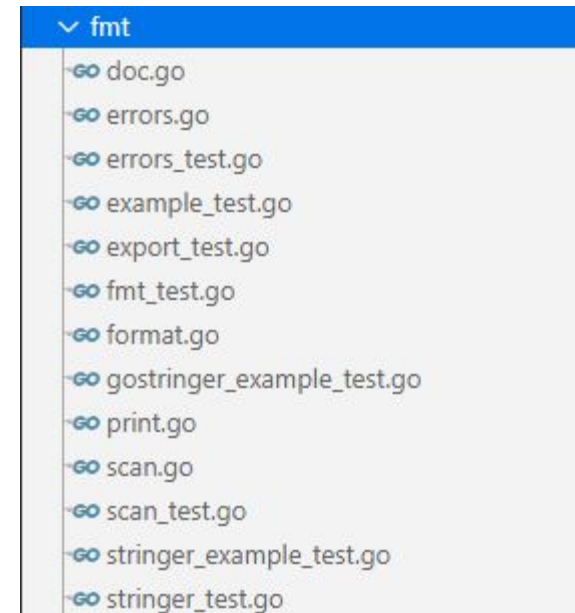


fmt

За форматированный вывод (по умолчанию в консоль) отвечает пакет `fmt`:

Как вы видите, он состоит из нескольких файлов.

Пока наших знаний не хватает, чтобы читать его содержимое, поэтому мы научимся пользоваться ключевой функцией в нём: `Println` (обратите внимание: с большой буквы).

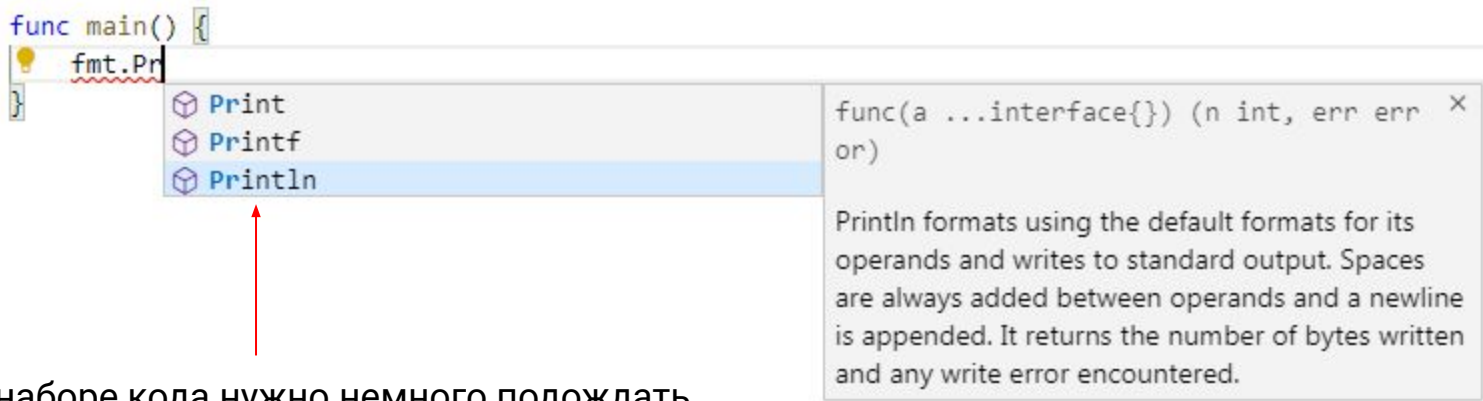


IMPORT



import

Для того, чтобы использовать функции (и другие сущности) из другого пакета, нам необходимо записывать их в формате: **имя_пакета.имя_функции**



при наборе кода нужно немного подождать,
чтобы появилась автоподсказка
(или нажать **Ctrl + пробел**)



import

Если при наборе кода с показанной автоподсказкой вы нажмёте Tab или Enter, то получите следующий код:

```
cmd > hello > -go main.go > ...
```

```
1  package main
2
3  import (
4      |   "fmt"
5  )
6
7  func main() {
8      |   fmt.Println("Hello, Go")
9  }
```

← дописалось автоматически



import

Ключевое слово `import` позволяет вам использовать функции (и другие сущности), объявленные в другом пакете. Т.е. мы работаем в пакете `main`, а используем функции из `fmt`.

Без `import` мы будем получать ошибку:

```
cmd > hello > go main.go > ...  
1 pack  
2  
3 func  
4 | fmt.Println("Hello, Go")  
5 }
```



Поэтому пользуйтесь автодополнением.



builtin

Вопрос к слушателям: а как же мы до этого использовали `println` без всяких `import`?

Ответ: всё, что задокументировано в `builtin` автоматически доступно в любом вашем пакете, это импортировать не нужно.



ИТОГИ



Итоги

В этой лекции мы обсудили

1. Компиляцию Go приложений
2. Систему пакетов и рекомендации по структуре каталогов
3. `builtin`

В домашнем задании вы потренируетесь в использовании функций из пакетов стандартной библиотеки.



ДОМАШНЕЕ ЗАДАНИЕ



Орг.моменты

Курс состоит из 33 обязательных занятий. Каждый **вторник 19:00 по Душанбе дедлайн** сдачи домашнего задания.

Если не успеете сдать в срок домашнее задания, тогда этот курс будет для вас закончен и вы сможете зарегистрироваться на запуск следующего через несколько месяцев.



ДЗ №1: Welcome

Обычно при регистрации пользователя ему высылается либо email, либо SMS с приветствием и кодом.

Например: "Добро пожаловать, Василий! Ваш код доступа: 92348.". Мы хотим сделать небольшую программу, которая в зависимости от тех аргументов, что мы ей передаём в командной строке генерировала нам подобные сообщения.

О чём тут речь? Мы хотим, чтобы:

- `welcome.exe Василий 92348` => "Добро пожаловать, Василий! Ваш код доступа: 92348."
- `welcome.exe Мария 10234` => "Добро пожаловать, Мария! Ваш код доступа: 10234."
- и т.д. (имена и коды могут быть любые)



ДЗ №1: Welcome

Обратите внимание: имя от кода отделяется пробелом:

- `welcome.exe` Василий 92348
- `welcome.exe` Мария 10234

Но что, если у человека будет имя, содержащее пробел, например, Василий Петрович? Тогда нужно Василий Петрович заключать в кавычки: `welcome.exe "Василий Петрович" 92348`. Если вы этого не сделаете, то в имя у вас придёт Василий, а в код - Петрович, что точно не правильно.



ДЗ №1: Welcome

Для этого нам понадобится три пакета:

1. `strings` - умеет работать со строками
2. `flag` - умеет работать с аргументами командной строки
3. `fmt` - умеет форматировать (в том числе печатать на экран)

Важно: `println` (после того, как мы прошли пакеты) теперь использовать нельзя.



ДЗ №1: Welcome

В пакете `strings` есть функция `ReplaceAll`, которая позволяет взять строку и на базе неё создать новую строку, заменив определённую часть:

```
cmd > welcome > go main.go > ...
```

```
1  package main
2
3  import (
4      "fmt"
5      "strings"
6  )
7
8  func main() {
9      template := `Добро пожаловать, {username}!`
10     message := strings.ReplaceAll(template, "{username}", "Василий")
11     fmt.Println(message)
12 }
```



ДЗ №1: Welcome

Что вообще такое функция? Это просто кусочек кода с именем, который что-то делает. Делает он это только тогда, когда вы после имени ставите круглые скобки (это означает вызов функции):

В общем случае:



ДЗ №1: Welcome

Так же как и с арифметическими выражениями (например, $1024/100$) на место вызова функции подставляется результат, поэтому мы можем спокойно использовать её при инициализации переменных:

```
message := strings.ReplaceAll(template, "{username}", "Василий")
```



вместо этого подставится строка, в которой
{username} заменено на Василий

Откуда мы взяли {username} и {code}? Придумали сами (на самом деле - подглядели у других). Это удобные схема отметки тех частей, которые нам нужно заменить.



ДЗ №1: Welcome

Кроме того, чтобы не придумывать новые имена, мы можем с помощью оператора `=` записать значение в то же самое имя:

```
cmd > welcome > go main.go > ...
```

```
1  package main
2
3  import (
4      "fmt"
5      "strings"
6  )
7
8  func main() {
9      message := `Добро пожаловать, {username}!`
10     message = strings.ReplaceAll(message, "{username}", "Василий")
11     fmt.Println(message)
12 }
```

Но с этим нужно быть аккуратным, т.к. тогда вы "затрёте" оригинальные данные и доступа к ним у вас уже не будет.



ДЗ №1: Welcome

Когда мы с вами вызывали команду `go`, то вызывали её вот так: `go mod init <имя>` или `go help build`. На самом деле, в этот момент запускался файл `go` (`go.exe` в Windows) и то, что шло дальше - каким-то образом передавалось в программу.

На основании параметров команда `go` принимала решения, что же нужно сделать. Например, для `go mod init <имя>` - нужно создать модуль с именем `<имя>`. У нас ситуация аналогичная - мы хотим тоже получать эти данные, чтобы затем их подставить в наше сообщение-приветствие.

То, что передаётся после имени программы, например `welcome.exe "Василий" 14234` называется аргументами командной строки ("`Василий`" и `14234` - это аргументы командной строки, по-английски `command line args`)



ДЗ №1: Welcome

Для того, чтобы работать с аргументами командной строки у нас есть две функции*:

- `flag.Parse` - "разбирает" аргументы командной строки
- `flag.Arg` - возвращает аргумент в определённой позиции (позиции нумеруются с 0, т.е. у "Василий" будет позиция 0)

Примечание*: конечно же, их гораздо больше, но нам пока хватит и двух.



ДЗ №1: Welcome

Сначала нужно вызвать `flag.Parse`, а уже только затем вызывать `flag.Arg`:

cmd > welcome >  main.go > ...

```
1  package main
2
3  import (
4      "fmt"
5      "flag"
6  )
7
8  func main() {
9      flag.Parse()
10     name := flag.Arg(0)
11     fmt.Println(name)
12 }
```

← неиспользуемый импортов (например, `strings`) быть не должно

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

Microsoft Windows [Version 6.1.7601]

(c) Корпорация Майкрософт (Microsoft Corp.), 2009. Все права защищены.

C:\projects\welcome>welcome.exe Василий
Василий

← запускали руками, не через `Ctrl + F5`



ДЗ №1: Welcome

Таким образом, на основании примеров использования `strings.ReplaceAll` и `flag.Parse`, `flag.Arg` вам нужно написать программу, которая удовлетворяет требуемым условиям.

Модуль должен называться `welcome`. Также не забудьте про требования к расположению файла `main.go` и тому, что после компиляции и сборки у вас должен получаться файл `welcome` (для Linux/Mac) или `welcome.exe` (для Windows).



ДЗ №2: Cleaner

Мы пишем большую систему, в которую приходят телефонные номера пользователей из разных других систем:

- веб-сайтов
- систем оплаты
- мобильных приложений
- и т.д.



ДЗ №2: Cleaner

Со всех систем телефоны приходят в разных форматах:

- +992 48 888 1111
- +992 (48) 88-81-111
- +992-48-888-11-11
- +992 (48) 8881111

Единственное, что соблюдается, это + в начале и правильное количество цифр. Конечно, в таком виде это хранить нельзя, потому что если пользователь будет пытаться залогиниться на ваш сайт, введя +992488881111 - это не совпадёт ни с одним из указанных выше номеров (это **разные** строки).



ДЗ №2: Cleaner

Что нужно сделать? Используя знания, полученные в предыдущем ДЗ, замените все символы (,), -, пробел на пустую строку "" (пустые кавычки). Это будет равносильно "вырезанию" их из строки - т.е. вы получите новую строку, в которой ЭТИХ СИМВОЛОВ НЕТ.

Работать ваша программа должна следующим образом:

`cleaner.exe "+992 (48) 888 1111"` ← специально взяли в кавычки, т.к. иначе
это будет несколько аргументов, а не один
`+992488881111`

Важно: телефон не обязательно будет +992488881111.



ДЗ №2: Cleaner

Модуль должен называться `phone`. Также не забудьте про требования к расположению файла `main.go` и тому, что после компиляции и сборки у вас должен получаться файл `cleaner` (для Linux/Mac) или `cleaner.exe` (для Windows).



Спасибо за внимание

alif academy

2020

