

Введение в язык Go

Что такое Go

Последнее обновление: 17.02.2024

Go представляет компилируемый статически типизированный язык программирования от компании Google. Язык Go предназначен для создания различного рода приложений, но прежде всего это веб-сервисы и клиент-серверные приложения. Хотя также язык обладает возможностями по работе с графикой, низкоуровневыми возможностями и т.д.

Работа над языком Go началась в 2007 в недрах компании Google. Одним из авторов является Кен Томпсон, который, к слову, является и одним из авторов языка Си (наряду с Денисом Ритчи). 10 ноября 2009 года язык был анонсирован, а в марте 2012 года вышла версия 1.0. При этом язык продолжает развиваться. Текущей версией на момент написания данной статьи является версия 1.22, которая вышла в феврале 2024 года.

Язык Go развивается как open source, то есть представляет проект с открытым исходным кодом, и все его коды и компилятор можно найти и использовать бесплатно. Официальный сайт проекта - <https://go.dev/>, где можно много полезной информации о языке.

Go является кроссплатформенным, он позволяет создавать программы под различные операционные системы - Windows, Mac OS, Linux, FreeBSD. Код обладает переносимостью: программы, написанные для одной из этих операционных систем, могут быть легко с перекомпиляцией перенесены на другую ОС.

Основные особенности языка Go:

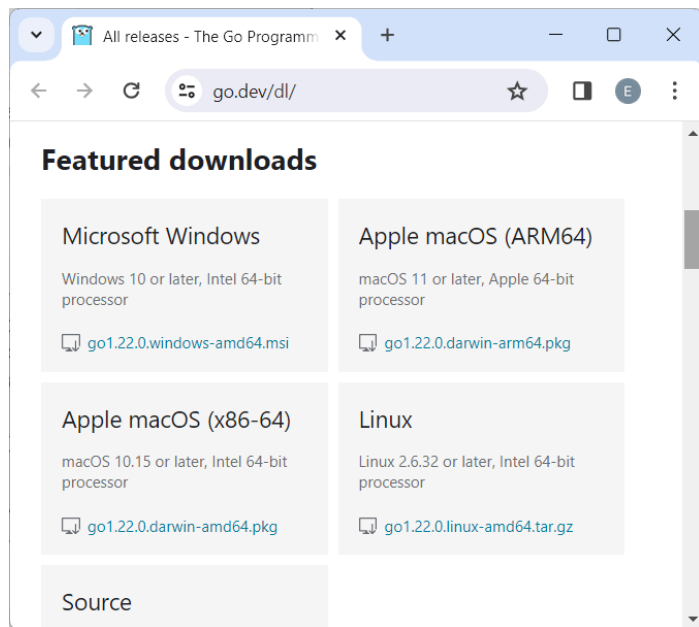
- компилируемый - компилятор транслирует программу на Go в машинный код, понятный для определенной платформы
- статически типизированный
- присутствует сборщик мусора, который автоматически очищает память
- поддержка работы с сетевыми протоколами
- поддержка многопоточности и параллельного программирования

В настоящее время Go находит широкое применение в различных сферах. В частности, среди известных проектов, которые применяют Go, можно найти следующие: Google, Dropbox, Netflix, Kubernetes, Docker, Twitch, Uber, CloudFlare и ряд других.

Что нужно для работы с Go? Прежде всего необходим текстовый редактор для набора кода и компилятор для преобразования кода в исполняемый файл. Также можно использовать специальные интегрированные среды разработки (IDE), которые поддерживают Go, например, GoLand от компании JetBrains. Существуют плагины для Go для других IDE, в частности, IntelliJ IDEA и Netbeans.

Установка Go

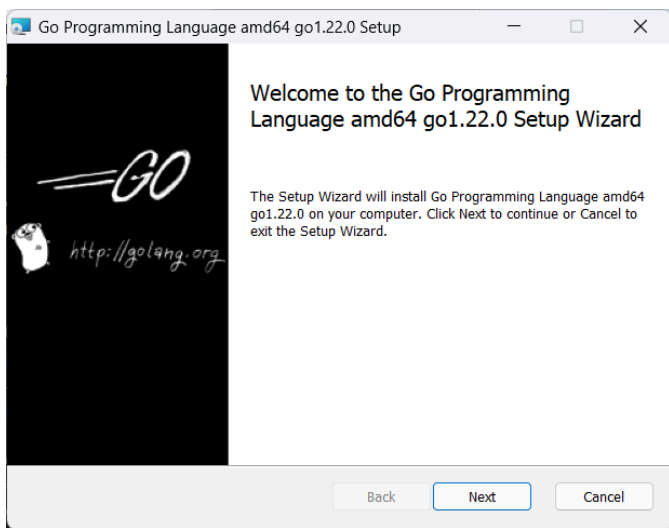
Пакет для установки компилятора можно загрузить с официального сайта <https://go.dev/dl/>.



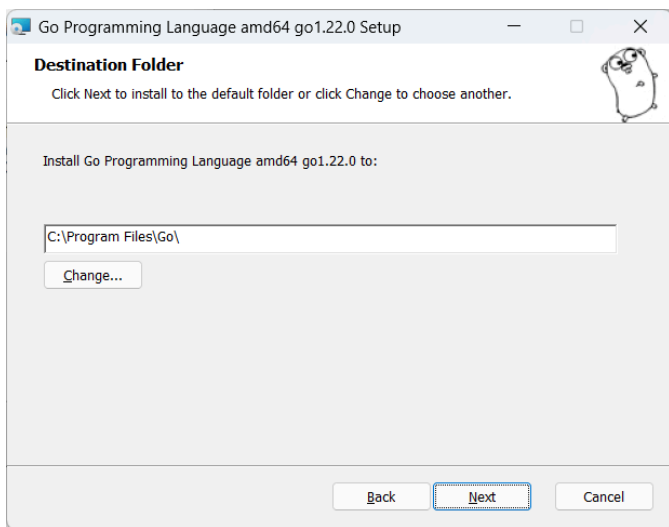
По этому адресу пакеты установщиков для различных операционных систем. Обратите внимание на версии поддерживаемых систем. Так, на момент написания текущей статьи поддерживались только версии Windows 10 и выше, MacOS 10.15 и выше и Linux 2.6.32 и выше, но все версии должны быть 64-разрядными. Загрузим подходящий для нашей ОС пакет установщика и запустим его. Процесс установки относительно прост - надо лишь прощелкать на кнопки в окнах установщика.

Установка на Windows

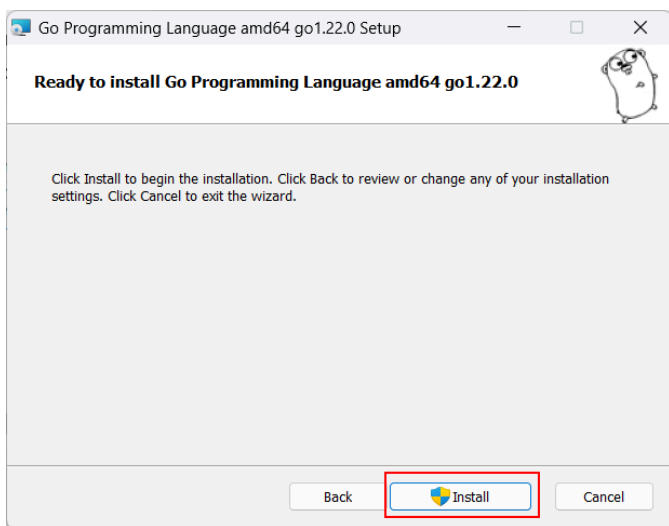
Например, стартовое окно установщика для Windows:



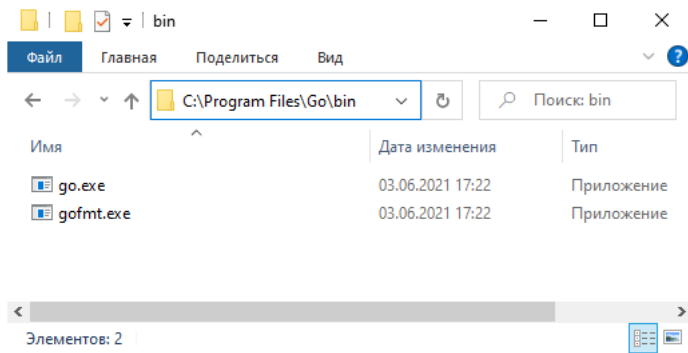
После принятия лицензионного соглашения отобразится окно для выбора места установки:



На Windows, например, по умолчанию используется путь "C:\Program Files\Go". Оставим путь по умолчанию и перейдем к следующему окну, на котором нажмем на кнопку Install:

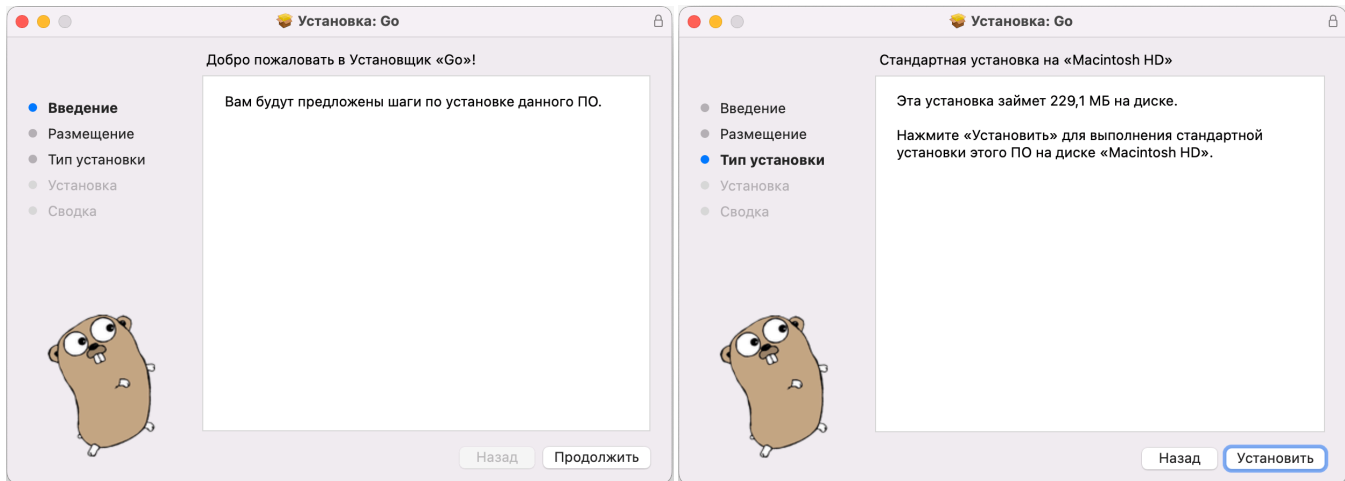


После успешной установки в папке установки будут установлены все файлы, необходимые для работы с Go. В частности, в подкаталоге bin можно найти файл go (go.exe на Windows), который выполняет роль компилятора:



Установка на MacOS

Установка на MacOS аналогична - с помощью мастера установки также надо последовательно нажать на кнопки:



Установка на Linux

Для Linux оф. сайт предоставляет архив. Например, в моем случае это файл `go1.22.0.linux-amd64.tar.gz`. Для установки этого архива выполним команду

```
sudo rm -rf /usr/local/go && tar -C /usr/local -xzf go1.22.0.linux-amd64.tar.gz
```

Эта команда удаляет ранее установленную версию (если таковая имеется) и устанавливает новую в папку `/usr/local/go`, соответственно компилятор будет располагаться в папке `/usr/local/go/bin`.

После этого внесем путь к папке `"/usr/local/go/bin"` в переменные среды. Для этого добавим в конец файла `$HOME/.profile` или `/etc/profile` следующую строку

```
export PATH=$PATH:/usr/local/go/bin
```

И для немедленного применения ее выполним команду

```
source $HOME/.profile
```

Проверка установки Go

После установки мы можем проверить версию языка, запустив в консоли команду `go version`:

```
C:\Users\eugen>go version
go version go1.22.0 windows/amd64
```

```
C:\Users\eugen>
```

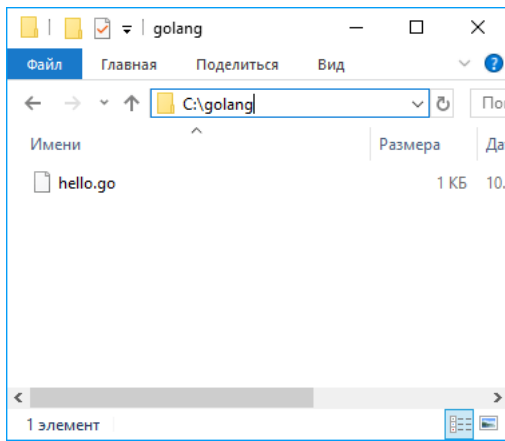
Первая программа

Последнее обновление: 03.12.2021

Создадим первую программу на языке Go. Для написания кода нам потребуется какой-нибудь текстовый редактор. Можно взять любой редактор, например, встроенный блокнот или популярный Notepad++ или любой другой. Для трансляции исходного кода в приложение необходим компилятор.

Создание программы

Определим на жестком диске папку для хранения файлов с исходным кодом. Допустим, в моем случае это будет папка `C:\golang`. В этой папке создадим новый текстовый файл, который переименуем в `hello.go`.



Откроем этот файл в любом текстовом редакторе и определим в нем следующий код:

```
package main
import "fmt"

func main() {
    fmt.Println("Hello Go!")
}
```

Что в этой программе делается? Программа на языке Go определяется в виде пакетов. Программный код должен быть определен в каком-то определенном пакете. Поэтому в самом начале файла с помощью оператора `package` указывается, к какому пакету будет принадлежать файл. В данном случае это пакет `main`:

```
package main
```

Причем пакет должен называться именно `main`, так как именно данный пакет определяет исполняемый файл.

При составлении программного кода нам может потребоваться функционал из других пакетов. В Go есть множество встроенных пакетов, которые содержат код, выполняющий определенные действия. Например, в нашей программе мы будем выводить сообщение на консоль. И для этого нам нужна функция `Println`, которая определена в пакете `fmt`. Поэтому второй строкой с помощью директивы `import` мы подключаем этот пакет:

```
import "fmt"
```

Далее идет функция `main`. Это главная функция любой программы на Go. По сути все, что выполняется в программе, выполняется именно функции `main`.

Определение функции начинается со слова `func`, после которого следует название функции, то есть `main`. После названия функции в скобках идет перечисление параметров. Так как функция `main` не принимает никаких параметров, то в данном случае указываются пустые скобки.

Затем в фигурных скобках определяется тело функции `main` - те действия, которые собственно и выполняет функция.

```
func main() {
```

В нашем случае функция выводит на консоль строку "Hello Go!". Для этого применяется функция `Println()`, которая определена в пакете `fmt`. Поэтому при вызове функции вначале указывается имя пакета и через точку имя функции. А в скобках функции передается то сообщение, которое она должна выводить на консоль:

```
    fmt.Println("Hello Go!")
```

Компиляция и выполнение программы

Теперь скомпилируем и выполним данную программу. Для этого необходимо передать файл с исходным кодом компилятору `go.exe` и указать нужную команду. Для этого откроем командную строку(терминал) и перейдем в ней с помощью команды `cd` к папке, где храниться файл с исходным кодом `hello.go` (в моем случае это папка `C:\golang`):

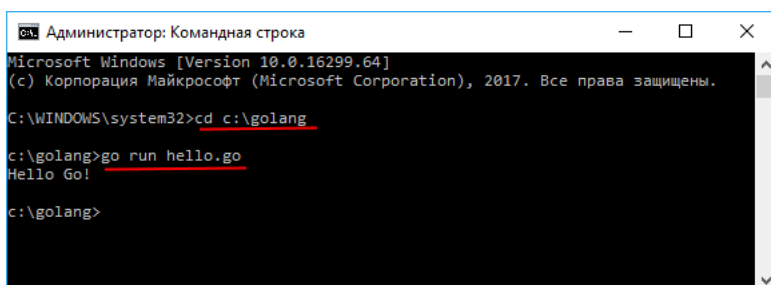
```
cd C:\golang
```

Затем выполним программу с помощью следующей команды:

```
go run hello.go
```

`go` - это компилятор. Поскольку при установке путь к компилятору автоматически прописывается в переменную `PATH` в переменных окружения, то нам не надо указывать полный путь `C:\Go\bin\go.exe`, а достаточно написать просто имя приложения `go`. Далее идет параметр `run`, который говорит, что мы просто хотим выполнить программу. И в конце указывается собственно файл программы `hello.go`.

В итоге после выполнения на консоль будет выведено сообщение "Hello Go!".



Данная команды выполняет, но не компилирует программу в отдельный исполняемый файл. Для компиляции выполним другую команду:

```
go build hello.go
```

После выполнения этой команды в папке с исходным файлом появится еще один файл, который будет называться `hello.exe` и который мы можем запускать. После этого опять же мы можем выполнить программу, запустив в консоли этот файл:

```

Microsoft Windows [Version 10.0.16299.64]
(c) Корпорация Майкрософт (Microsoft Corporation), 2017. Все права защищены.

C:\WINDOWS\system32>cd c:\golang

c:\golang>go run hello.go
Hello Go!

c:\golang>go build hello.go

c:\golang>hello
Hello Go!

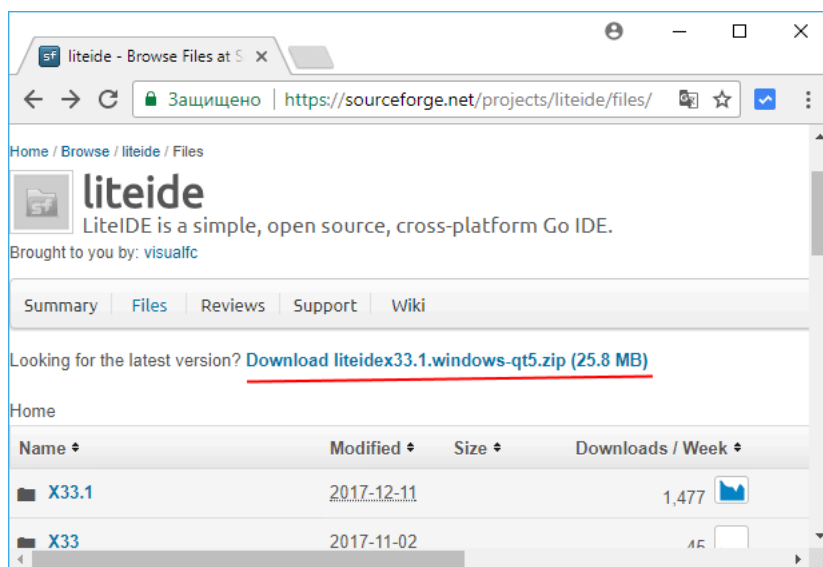
c:\golang>
  
```

Go в LiteIDE

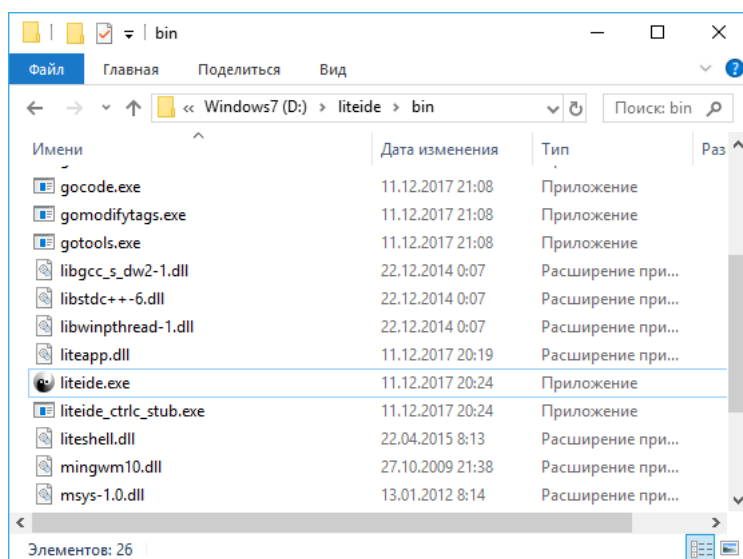
Последнее обновление: 20.12.2017

Использование интегрированных сред разработки (IDE) в ряде случаев упрощает управление проектом и создание приложения. Для языка Go одной из популярных сред разработки является LiteIDE. Это бесплатная кроссплатформенная среда, которую можно свободно загрузить себе на рабочий компьютер. Официальный сайт IDE - <http://liteide.org/en/>.

Непосредственно загрузить все файлы данной IDE можно по ссылке: <https://sourceforge.net/projects/liteide/files/>.

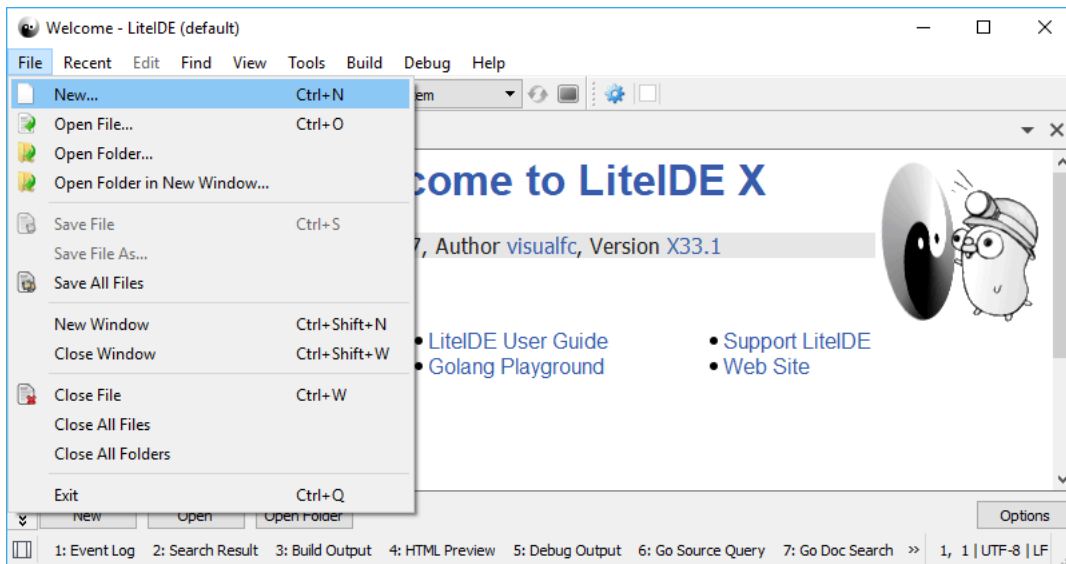


На этой странице можно найти ссылку для загрузки. Причем ничего не надо устанавливать. По ссылке загружается zip-архив, который после загрузки достаточно распаковать. После распаковки архива в корневой папке в каталоге `bin` можно найти файл исполняемой программы `liteide.exe` (на Windows), через который и можно запустить среду разработки:

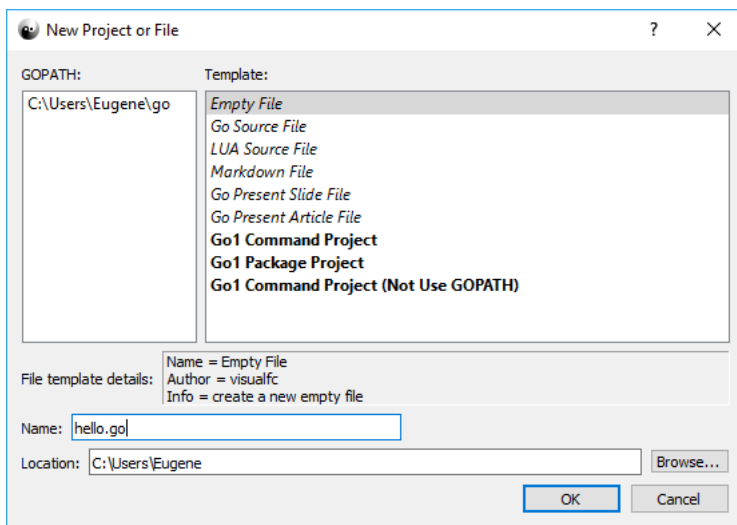


Запустим этот файл, и по умолчанию нам откроется стартовая страница приветствия. LiteIDE имеет локализованные версии (в том числе русифицированную версию), и через параметры можно переключить язык среды на любой из предложенных.

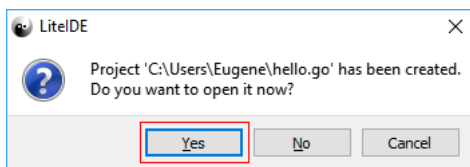
Для создания нового проекта перейдем к меню `File(Файл) -> New (Создать)` :



После этого отобразится окно выбора шаблона проекта:



В качестве шаблона выберем Empty File . А внизу в поле Name укажем для файла имя hello.go . В другом поле можно увидеть путь к каталогу, где будет располагаться файл. Этот путь мы также можем изменить. И затем проект будет создан, и откроется окно, где надо будет подтвердить открытие проекта:

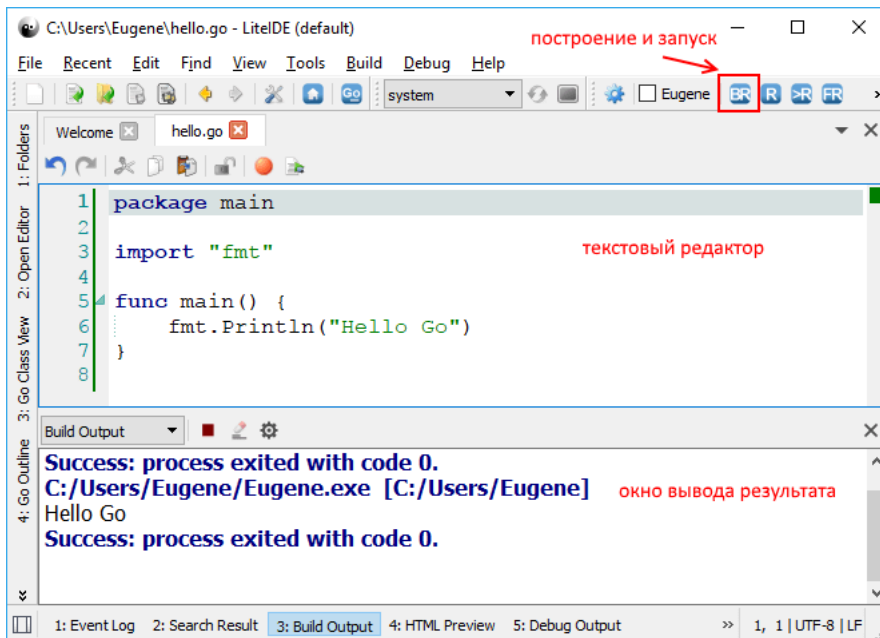


После этого в центральной части окна программы откроется текстовый редактор. Введем в него следующий код:

```
package main

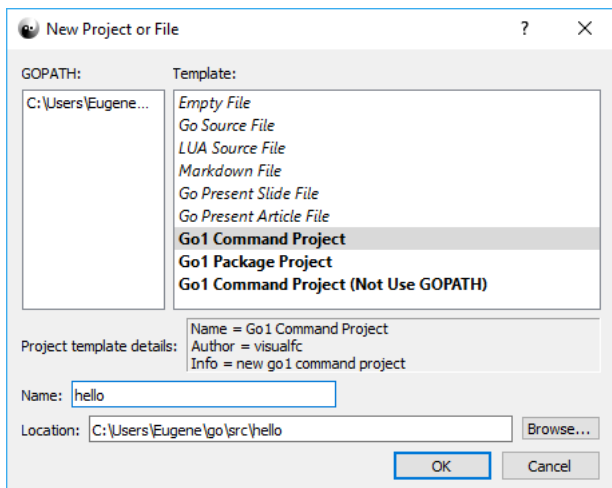
import "fmt"

func main() {
    fmt.Println("Hello Go")
}
```

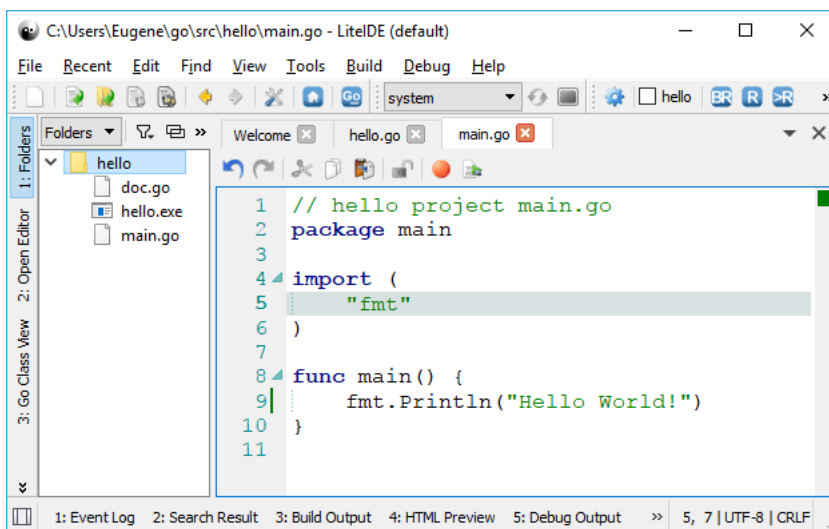


Для сборки и запуска проекта нажмем на панели инструментов кнопку BR. И внизу окна IDE отобразится поле вывода, где мы можем увидеть результат выполнения программы.

Также в LiteIDE можно создавать и использовать другие типы проектов. Например, создадим проект Go1 Command Project, который пусть называется hello:



В результате будет создан проект, который располагается в отдельной папке и который по умолчанию состоит из двух файлов. И данный проект подобным образом мы можем запустить:



Go в Visual Studio Code

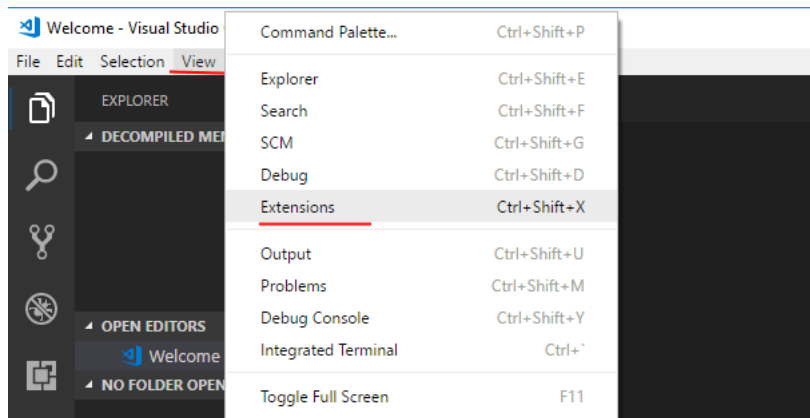
Последнее обновление: 21.12.2017

Visual Studio Code представляет кроссплатформенный подвинутый легковесный текстовый редактор от компании Microsoft, который поддерживает подсветку синтаксиса, интеллектуальную подсказку для разных языков программирования и многое другое. Рассмотрим, как мы можем использовать

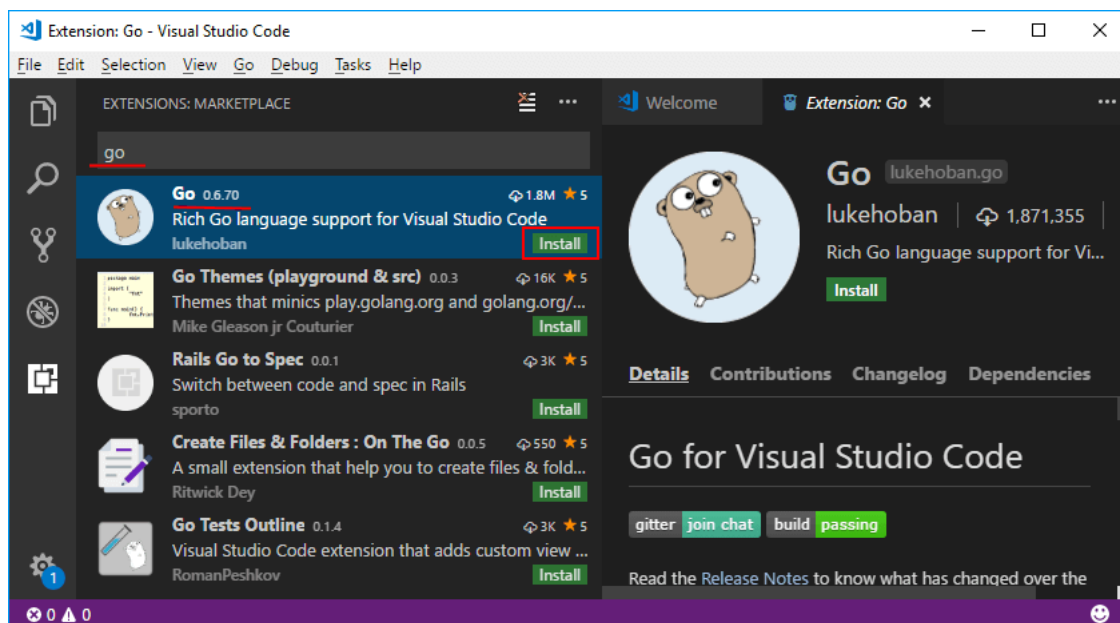
данный текстовый редактор для разработки на языке Go.

Прежде всего установим данный текстовый редактор. Инсталлятор для нужной операционной системы (есть поддержка для Windows, Mac OS, Linux) можно найти по адресу <https://code.visualstudio.com/>.

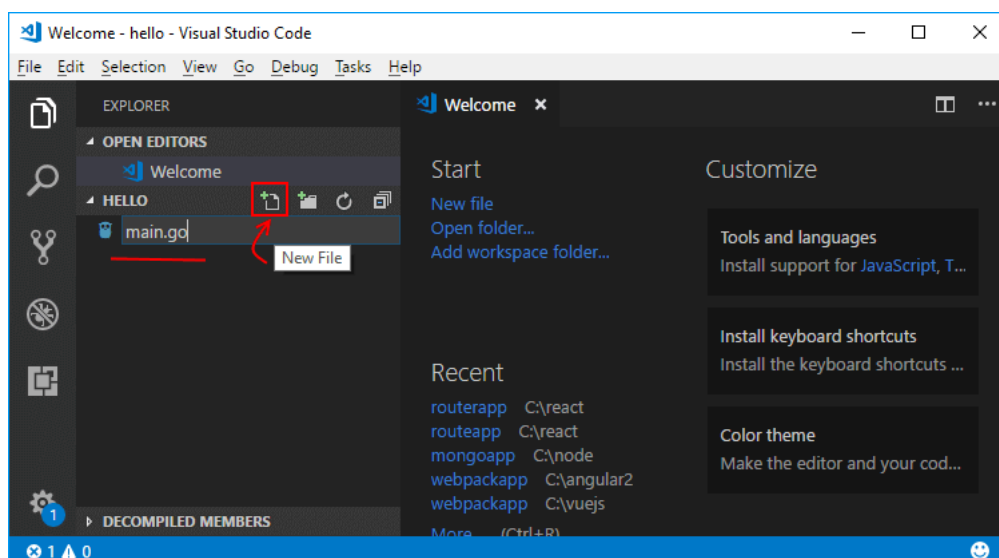
После установки Visual Studio Code по умолчанию не имеет никакой поддержки языка Go. Поэтому необходимо установить соответствующее расширение. Для этого перейдем в Visual Studio Code в меню View -> Extensions



В строку поиска расширений введем "go", и нам отобразится список найденных расширений. Нам нужно установить первое в этом списке, которое имеет больше всего установок:



После установки расширения определим на жестком диске папку для хранения файлов проекта и откроем эту папку в Visual Studio Code. Открыть папку можно через пункт меню File -> Open Folder. Затем создадим в VS Code новый файл, который назовем main.go :



Откроем файл main.go и введем в него следующий код:

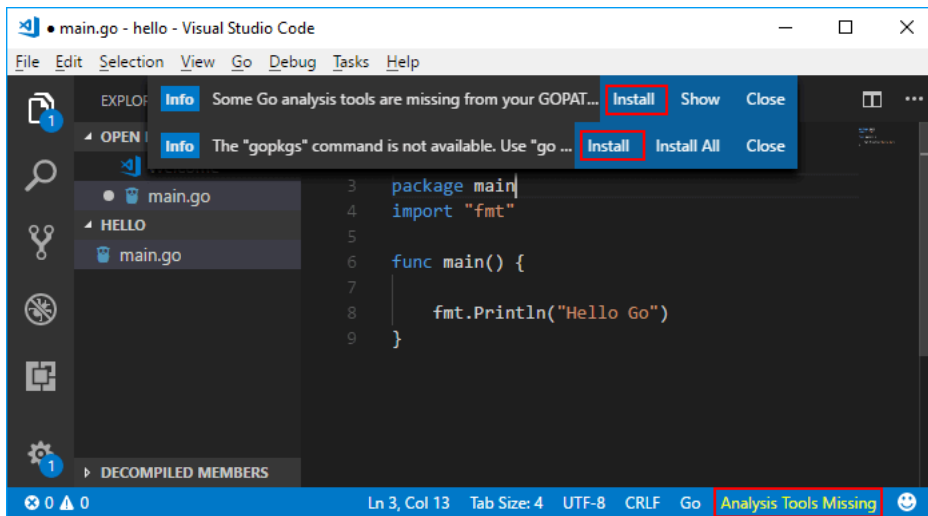
```
package main

import "fmt"
```



```
func main() {
    fmt.Println("Hello Go")
}
```

Сохраним введенный код, нажав на комбинацию Ctrl+S.



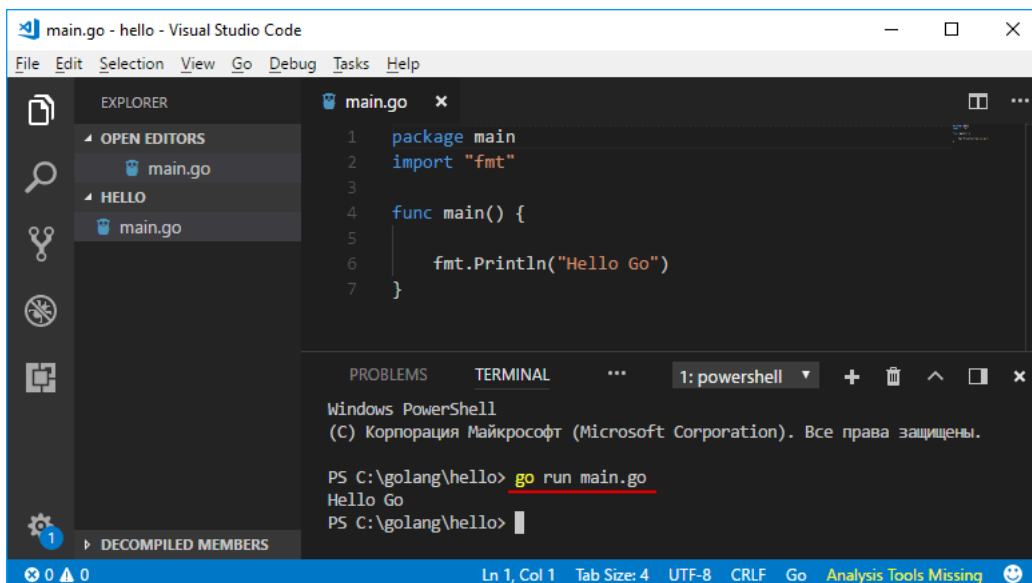
При работе с файлами go в VS Code могут появляться различные сообщения о необходимости установки дополнительных плагинов. Например, внизу окна в статусной строке может отображаться сообщение "Analysis Tools Missing". Можно нажать на это сообщение, и вверху VS Code отобразится список пакетов, которые желательно доустановить. Для их установки надо нажать на кнопку Install.

Преимуществом VS Code является то, что этот редактор имеет встроенный терминал. Откроем терминал через пункт меню View -> Integrated Terminal . После этого внизу VS Code откроется встроенный терминал. По умолчанию в нем открывается текущая папка проекта.

Введем в терминал следующую команду и нажмем Enter:

```
go run main.go
```

После этого во встроенном терминале мы увидим вывод программы:



Основы языка Go

Структура программы

Последнее обновление: 12.12.2017

Программа на языке Go хранится в одном или нескольких файлах. Каждый файл с программным кодом должен принадлежать какому-нибудь пакету. И в начале каждого файла должно идти объявление пакета, к которому этот файл принадлежит. Пакет объявляется с помощью ключевого слова `package` .

В файле может использоваться функционал из других пакетов. В этом случае используемые пакеты надо импортировать с помощью ключевого слова `import` . Импортируемые пакеты должны идти после объявления пакета для текущего файла:

```
package main
import "fmt"
```

Например, в данном случае текущий файл будет находиться в пакете `main`. И далее он подключает пакет `fmt`.

Причем главный пакет программы должен называться "main". Так как именно данный пакет определяет, что будет создаваться исполняемый файл приложения, который после компиляции можно будет запускать на выполнение.

После подключения других пакетов располагаются объявления типов, переменных, функций, констант.

При этом входной точкой в приложения является функция с именем `main` . Она обязательно должна быть определена в программе. Все, что выполняется в программе, выполняется именно в функции `main`.

```
package main
import "fmt"

func main() {
    fmt.Println("Hello Go!")
}
```

Базовым элементом программы являются инструкции. Например, вызов функции `fmt.Println("Hello Go!")` представляет отдельную инструкцию. Каждая инструкция выполняет определенное действие и размещается на новой строке:

```
package main
import "fmt"

func main() {
    fmt.Println("Hello Go!")
    fmt.Println("Hello Golang!")
    fmt.Println("Hello Go!")
}
```

Здесь функция `main` содержит три инструкции, которые выводит на консоль строку, и каждая из инструкций размещается на новой строке.

Можно размещать несколько инструкций и на одной строке, но тогда их надо отделять точкой запятой:

```
package main
import "fmt"

func main() {
    fmt.Println("Hello Go!");fmt.Println("Hello Golang!");fmt.Println("Hello Go!")
}
```

В то же время размещение инструкций на новой строке представляет более читабельный формат, поэтому более предпочтительно для использования.

Комментарии

Программа может иметь комментарии. Комментарии служат для описания действий, которые производит программа или какие-то ее части. При компиляции комментарии не учитываются и не оказывают никакого влияния на работу приложения. Комментарии бывают однострочными и многострочными.

Однострочный комментарий располагается в одну строку после двойного слеша `///` . Все, что идет после этих символов, воспринимается компилятором как комментарий. Многострочный комментарий заключается между символами `/*` и `*/` и может занимать несколько строк:

```
/*
    Первая программа
    на языке Go
*/
package main    // определение пакета для текущего файла
import "fmt"     // подключение пакета fmt

// определение функции main
func main() {
    fmt.Println("Hello Go!")           // вывод строки на консоль
}
```

Переменные

Последнее обновление: 20.12.2017

Для хранения данных в программе применяются переменные. Переменная представляет именованный участок в памяти, который может хранить некоторое значение. Для определения переменной применяется ключевое слово `var` , после которого идет имя переменной, а затем указывается ее тип:

```
var имя_переменной тип_данных
```

Имя переменной представляет произвольный идентификатор, который состоит из алфавитных и цифровых символов и символа подчеркивания. При этом первым символом должен быть либо алфавитный символ, либо символ подчеркивания. При этом имена не должны представлять одно из ключевых слов: `break`, `case`, `chan`, `const`, `continue`, `default`, `defer`, `else`, `fallthrough`, `for`, `func`, `go`, `goto`, `if`, `import`, `interface`, `map`, `package`, `range`, `return`, `select`, `struct`, `switch`, `type`, `var`.

Например, простейшее определение переменной:

```
var hello string
```

Данная переменная называется `hello` и она представляет тип `string` , то есть некоторую строку.

Можно одновременно объявить сразу несколько переменных через запятую:

```
var a, b, c string
```

В данном случае определены переменные `a`, `b`, `c`, которые имеют тип `string` . В этом случае опять же в конце указывается тип данных, и все переменные принадлежат этому типу.

После определения переменной ей можно присвоить какое-либо значение, которое соответствует ее типу:

```
package main
import "fmt"

func main() {
    var hello string
    hello = "Hello world"
    fmt.Println(hello)
}
```

Поскольку переменная `hello` представляет тип `string`, ей можно присвоить строку. В данном случае переменная `hello` хранит строку `"Hello world"`. С помощью функции `Println` значение этой переменной выводится на консоль.

Также важно учитывать, что Go - регистрозависимый язык, то есть переменные с именами `"hello"` и `"Hello"` будут представлять разные переменные:

```
var hello string
hello = "Hello world"
fmt.Println(Hello)      // ! Ошибка переменной Hello нет, есть переменная hello
```

Также можно сразу при объявлении переменной присвоить ей начальное значение. Такой прием называется инициализацией:

```
package main
import "fmt"

func main() {
    var hello string = "Hello world"
    fmt.Println(hello)
}
```

Если мы хотим сразу определить несколько переменных и присвоить им начальные значения, то можно обернуть их в скобки:

```
package main
import "fmt"

func main() {
    var (
        name string = "Tom"
        age int = 27
    )

    fmt.Println(name)      // Tom
    fmt.Println(age)       // 27
}
```

Отличительной особенностью переменных является то, что их значение можно многократно изменять:

```
package main
import "fmt"

func main() {
    var hello string = "Hello world"
    fmt.Println(hello)      // Hello world

    hello = "Hello Go"
    fmt.Println(hello)      // Hello Go

    hello = "Go Go Go Ole Ole Ole"
    fmt.Println(hello)      // Go Go Go Ole Ole Ole
}
```

Краткое определение переменной

Также допустимо краткое определение переменной в формате:

`имя_переменной := значение`

После имени переменной ставится двоеточие и равно и затем указывается ее значение.

```
package main
import "fmt"

func main() {
    name := "Tom"
    fmt.Println(name)
}
```

В этом случае тип данных явным образом не указывается, он выводится автоматически из присваиваемого значения.

Типы данных

Последнее обновление: 20.12.2017

Все данные, которые хранятся в памяти, по сути представляют просто набор битов. И именно тип данных определяет, как будут интерпретироваться эти данные и какие операции с ними можно производить. Язык Go является статически типизированным языком, то есть все используемые в программе данные имеют определенный тип.

Go имеет ряд встроенных типов данных, а также позволяет определять свои типы. Рассмотрим базовые встроенные типы данных, которые мы можем использовать.

Целочисленные типы

Ряд типов представляют целые числа:

- `int8` : представляет целое число от -128 до 127 и занимает в памяти 1 байт (8 бит)
- `int16` : представляет целое число от -32768 до 32767 и занимает в памяти 2 байта (16 бит)
- `int32` : представляет целое число от -2147483648 до 2147483647 и занимает 4 байта (32 бита)
- `int64` : представляет целое число от -9 223 372 036 854 775 808 до 9 223 372 036 854 775 807 и занимает 8 байт (64 бита)
- `uint8` : представляет целое число от 0 до 255 и занимает 1 байт
- `uint16` : представляет целое число от 0 до 65535 и занимает 2 байта

- `uint32` : представляет целое число от 0 до 4294967295 и занимает 4 байта
- `uint64` : представляет целое число от 0 до 18 446 744 073 709 551 615 и занимает 8 байт
- `byte` : синоним типа `uint8` , представляет целое число от 0 до 255 и занимает 1 байт
- `rune` : синоним типа `int32` , представляет целое число от -2147483648 до 2147483647 и занимает 4 байта
- `int` : представляет целое число со знаком, которое в зависимости о платформы может занимать либо 4 байта, либо 8 байт. То есть соответствовать либо `int32`, либо `int64`.
- `uint` : представляет целое беззнаковое число только без знака, которое, аналогично типу `int`, в зависимости о платформы может занимать либо 4 байта, либо 8 байт. То есть соответствовать либо `uint32`, либо `uint64`.

Здесь несложно запомнить, что есть типы со знаком (то есть которые могут быть отрицательными) и есть беззнаковые положительные типы, которые начинаются на префикс `u` (`uint32`). Ну и также есть `byte` - синоним для `uint8` и `rune` - синоним для `int32`.

Стоит отметить типы `int` и `uint`. Они имеют наиболее эффективный размер для определенной платформы (32 или 64 бита). Это наиболее используемый тип для представления целых чисел в программе. Причем различные компиляторы могут предоставлять различный размер для этих типов даже для одной и той же платформы.

Примеры определения переменных, которые представляют целочисленные типы:

```
var a int8 = -1
var b uint8 = 2
var c byte = 3 // byte - синоним типа uint8
var d int16 = -4
var f uint16 = 5
var g int32 = -6
var h rune = -7 // rune - синоним типа int32
var j uint32 = 8
var k int64 = -9
var l uint64 = 10
var m int = 102
var n uint = 105
```

Числа с плавающей точкой

Для представления дробных чисел есть два типа:

- `float32` : представляет число с плавающей точкой от $1.4 \cdot 10^{-45}$ до $3.4 \cdot 10^{38}$ (для положительных). Занимает в памяти 4 байта (32 бита)
- `float64` : представляет число с плавающей точкой от $4.9 \cdot 10^{-324}$ до $1.8 \cdot 10^{308}$ (для положительных) и занимает 8 байт.

Тип `float32` обеспечивает шесть десятичных цифр точности, в то время как точность, обеспечиваемая типом `float64`, составляет около 15 цифр

Примеры использования типов `float32` и `float64`:

```
var f float32 = 18
var g float32 = 4.5
var d float64 = 0.23
var pi float64 = 3.14
var e float64 = 2.7
```

В качестве разделителя между целой и дробной частью применяется точка.

Комплексные числа

Существуют отдельные типы для представления комплексных чисел:

- `complex64` : комплексное число, где вещественная и мнимая части представляют числа `float32`
- `complex128` : комплексное число, где вещественная и мнимая части представляют числа `float64`

Пример использования:

```
var f complex64 = 1+2i
var g complex128 = 4+3i
```

Тип bool

Логический тип или тип `bool` может иметь одно из двух значений: `true` (истина) или `false` (ложь).

```
var isAlive bool = true
var isEnabled bool = false
```

Строки

Строки представлены типом `string` . В Go строке соответствует строковый литерал - последовательность символов, заключенная в двойные кавычки:

```
var name string = "Том Сойер"
```

Кроме обычных символов строка может содержать специальные последовательности (управляющие последовательности), которые начинаются с обратного слеша `\`. Наиболее распространенные последовательности:

- `\n` : переход на новую строку
- `\r` : возврат каретки
- `\t` : табуляция
- `\"` : двойная кавычка внутри строк

- \: обратный слеш

Значение по умолчанию

Если переменной не присвоено значение, то она имеет значение по умолчанию, которое определено для ее типа. Для числовых типов это число 0, для логического типа - false, для строк - ""(пустая строка).

Неявная типизация

При определении переменной мы можем опускать тип в том случае, если мы явно инициализируем переменную каким-нибудь значением:

```
var name = "Tom"
```

В этом случае компилятор на основании значения неявно выводит тип переменной. Если присваивается строка, то соответственно переменная будет представлять тип string, если присваивается целое число, то переменная представляет тип int и т.д.

То же самое по сути происходит при кратком определении переменной, когда также явным образом не указывается тип данных:

```
name := "Tom"
```

При этом стоит учитывать, что если мы не указываем у переменной тип, то ей обязательно надо присвоить некоторое начальное значение. Объявление переменной одновременно без указания типа данных и начального значения будет ошибкой:

```
var name // ! Ошибка
```

Надо либо указать тип данных (в этом случае переменная будет иметь значение по умолчанию):

```
var name string
```

Либо указать начальное значение, на основании которого выводится тип данных:

```
var name = "Tom"
```

Либо и то, и другое одновременно:

```
var name string = "Tom"
```

Неявная типизация нескольких переменных:

```
var (
    name = "Tom"
    age = 27
)
```

или так:

```
var name, age = "Tom", 27
```

Константы

Последнее обновление: 18.12.2017

Константы, как и переменные, хранят некоторые данные, но в отличие от переменных значения констант нельзя изменить, они устанавливаются один раз. Вычисление констант производится во время компиляции. Благодаря этому уменьшается количество работы, которую необходимо произвести во время выполнения, упрощается поиск ошибок, связанных с константами (так как некоторые из них можно обнаружить на момент компиляции).

Для определения констант применяется ключевое слово const :

```
const pi float64 = 3.1415
```

И в отличие от переменной мы не можем изменить значение константы. А если и попробуем это сделать, то при компиляции мы получим ошибку:

```
const pi float64 = 3.1415
pi = 2.7182 // ! Ошибка
```

В одном определении можно объявить сразу несколько констант:

```
const (
    pi float64 = 3.1415
    e float64 = 2.7182
)
```

или так:

```
const pi, e = 3.1415, 2.7182
```

Если у константы не указан тип, то он выводится неявно на основании того значения, которым инициализируется константа:

```
const n = 5 // тип int
```

В то же время необходимо обязательно инициализировать константу начальным значением при ее объявлении. Например, следующие определения констант являются недопустимыми, так как они не инициализируются:

```
const d
const n int
```

Если определяется последовательность констант, то инициализацию значением можно опустить для всех констант, кроме первой. В этом случае константа без значения получит значение предыдущей константы:

```
const (
    a = 1
    b
    c
)
```

```

    d = 3
    f
)
fmt.Println(a, b, c, d, f)           // 1, 1, 1, 3, 3

```

Константы можно инициализировать только константными значениями, например, литералами типа чисел или строк, или значениями других констант. Но инициализировать константу значением переменной мы не можем:

```

var m int = 7
// const k = m           // ! Ошибка: m - переменная
const s = 5              // Норм: 5 - числовая константа
const n = s              // Норм: s - константа

```

Арифметические операции

Последнее обновление: 18.12.2017

Язык Go поддерживает все основные арифметические операции, которые производятся над числами. Значения, которые участвуют в операции, называются операндами. Результатом операции также является число. Список поддерживаемых арифметических операций:

- +

Операция сложения возвращает сумму двух чисел:

```

package main
import "fmt"

func main() {
    var a = 4
    var b = 6
    var c = a + b
    fmt.Println(c)           // 10
}

```

- -

Операция вычитания возвращает разность двух чисел:

```

package main
import "fmt"

func main() {
    var a = 4
    var b = 6
    var c = a - b
    fmt.Println(c)           // -2
}

```

- *

Операция умножения возвращает произведение двух чисел:

```

var a = 4
var b = 6
var c = a * b    // 24

```

- /

Операция деления двух чисел:

```

var a int = 10
var b int = 4
var c int = a / b
fmt.Println(c)           // 2

var k float32 = 10
var l float32 = 4
var m float32 = k / l
fmt.Println(m)           // 2.5

```

При делении стоит быть внимательным, так как если в операции участвуют два целых числа, то результат деления будет округляться до целого числа, даже если результат присваивается переменной типа float32/float64:

```

var m float32 = 10 / 4           // 2

```

Результат представлял вещественное число, один из операндов также должен представлять вещественное число:

```

var m float32 = 10 / 4.0           // 2.5

```

- %

Возвращает остаток от деления (в этой операции могут принимать участие только целочисленные операнды):

```

var c int = 35 % 3           // 2 (35 - 33 = 2)

```

- Постфиксный инкремент (x++). Увеличивает значение переменной на единицу:

```

var a int = 8
a++
fmt.Println(a)           // 9

```

- Постфиксный декремент (x--). Уменьшает значение переменной на единицу:

```

var a int = 8
a--
fmt.Println(a)           // 7

```

Условные выражения

Последнее обновление: 18.12.2017

Условные выражения представляют логические операции и операции отношения. Они представляют некоторое условие и возвращают значение типа bool: true (если условие истинно) или false (если условие ложно).

Операции отношения

Операции отношения позволяют сравнить два значения. В языке Go есть следующие операции отношения:

- ==

Операция "равно". Возвращает true, если оба операнда равны, и false, если они не равны:

```
package main
import "fmt"

func main() {
    var a int = 8
    var b int = 3
    var c bool = a == b
    fmt.Println(c)    // false
}
```

- >

Операция "больше чем". Возвращает true, если первый операнд больше второго, и false, если первый операнд меньше второго:

```
var a int = 8
var b int = 3
var c bool = a > b    // true
```

- <

Операция "меньше чем". Возвращает true, если первый операнд меньше второго, и false, если первый операнд больше второго:

```
var a int = 8
var b int = 3
var c bool = a < b    // false
```

- <=

Операция "меньше или равно". Возвращает true, если первый операнд меньше или равен второму, и false, если первый операнд больше второго:

```
var a int = 8
var b int = 3
var c bool = a <= b    // false
```

- >=

Операция "больше или равно". Возвращает true, если первый операнд больше или равен второму, и false, если первый операнд меньше второго:

```
var a int = 8
var b int = 3
var c bool = a >= b    // true
```

- !=

Операция "не равно". Возвращает true, если первый операнд не равен второму, и false, если оба операнда равны:

```
var a int = 8
var b int = 3
var c bool = a != b    // true
var d bool = a != 8    // false
```

Как правило, операции отношения применяются в условных конструкциях типа if...else, которые мы далее рассмотрим.

Логические операции

Логические операции сравнивают два условия. Как правило, они применяются к отношениям и объединяют несколько операций отношения. К логическим операциям относят следующие:

- ! (операция отрицания)

Инвертирует значение. Если операнд равен true, то возвращает false, иначе возвращает true.

```
var a bool = true
var b bool = !a    //false
var c bool = !b    // true
```

- && (конъюнкция, логическое умножение)

Возвращает true, если оба операнда не равны false. Возвращает false, если хотя бы один операнд равен false.

```
var b bool = 4 > 5 && 6 > 8    //false
var c bool = 3 <= 5 && 10 > 8    // true
```

- || (дизъюнкция, логическое сложение)

Возвращает true, если хотя бы один операнд не равен false. Возвращает false, если оба операнда равны false.

```
var b bool = 4 > 5 || 6 > 8    //false
var c bool = 3 == 5 || 10 > 8    // true
```

Поразрядные операции

Последнее обновление: 18.12.2017

Поразрядные операции выполняются над отдельными разрядами чисел в бинарном представлении. Например, число пять в двоичной системе имеет три разряда: 101, а число восемь - четыре разряда: 1000.

Операции сдвига

И операции сдвига позволяют сдвинуть двоичное представление числа на несколько разрядов вправо или влево. Операции сдвига применяются только к целочисленным операндам. Есть две операции:

- <<

Сдвигает битовое представление числа, представленного первым операндом, влево на определенное количество разрядов, которое задается вторым операндом.

- >>

Сдвигает битовое представление числа вправо на определенное количество разрядов.

Применение операций:

```
var b int = 2 << 2;           // 10  на два разряда влево = 1000 - 8
var c int = 16 >> 3;          // 10000 на три разряда вправо = 100 - 2
```

Число 2 в двоичном представлении 10. Если сдвинуть число 10 на два разряда влево, то получится 1000, что в десятичной системе равно числу 8.

Число 16 в двоичном представлении 10000. Если сдвинуть число 10000 на три разряда вправо (три последних разряда отбрасываются), то получится 10, что в десятичной системе представляет число 2.

Поразрядные операции

Поразрядные операции также проводятся только над разрядами целочисленных операндов:

- & : поразрядная конъюнкция (операция И или поразрядное умножение). Возвращает 1, если оба из соответствующих разрядов обоих чисел равны 1. Возвращает 0, если разряд хотя бы одного числа равен 0
- | : поразрядная дизъюнкция (операция ИЛИ или поразрядное сложение). Возвращает 1, если хотя бы один из соответствующих разрядов обоих чисел равен 1
- ^ : поразрядное исключающее ИЛИ. Возвращает 1, если только один из соответствующих разрядов обоих чисел равен 1
- &^ : сброс бита (И НЕ). В выражении $z = x \ \&\wedge\ y$ каждый бит z равен 0, если соответствующий бит y равен 1. Если бит y равен 0, то берется значение соответствующего бита из x .

Применение операций:

```
package main
import "fmt"

func main() {
    var a int = 5 | 2;           // 101 | 010 = 111 - 7
    var b int = 6 & 2;           // 110 & 010 = 10 - 2
    var c int = 5 ^ 2;           // 101 ^ 010 = 111 - 7
    var d int = 5 &^ 6;          // 101 &^ 110 = 001 - 1
}
```

Например, выражение $5 \mid 2$ равно 7. Число 5 в двоичной записи равно 101, а число 2 - 10 или 010. Сложим соответствующие разряды обоих чисел. При сложении если хотя бы один разряд равен 1, то сумма обоих разрядов равна 1. Поэтому получаем:

```
1 0 1
0 1 0
1 1 1
```

В итоге получаем число 111, что в десятичной записи представляет число 7.

Возьмем другое выражение $6 \& 2$. Число 6 в двоичной записи равно 110, а число 2 - 10 или 010. Умножим соответствующие разряды обоих чисел. Произведение обоих разрядов равно 1, если оба этих разряда равны 1. Иначе произведение равно 0. Поэтому получаем:

```
1 1 0
0 1 0
0 1 0
```

Получаем число 010, что в десятичной системе равно 2.

Массивы

Последнее обновление: 20.12.2017

Массивы представляют последовательность элементов определенного типа. Массив определяется следующим способом:

```
var numbers [число_элементов]тип_элементов
```

Например, массив из пяти элементов типа int:

```
var numbers [5]int
```


При таком определении все элементы массива инициализируются значениями по умолчанию. Но также можно инициализировать элементы массива другими значениями:

```
var numbers [5]int = [5]int{1,2,3,4,5}
```

Значения передаются в фигурных скобках через запятую. При этом значений не может быть больше длины массива. В данном случае длина массива равна 5, поэтому нельзя в фигурных скобках определить больше пяти элементов. Но можно определить меньше элементов:

```
var numbers [5]int = [5]int{1,2}
fmt.Println(numbers)    // [1 2 0 0 0]
```

В этом случае элементы, для которых не указано значение, будут иметь значение по умолчанию.

Также можно применять сокращенное определение переменной массива:

```
numbers := [5]int{1,2,3,4,5}
```

Если в квадратных скобках вместо длины указано троеточие, то длина массива определяется, исходя из количества переданных ему элементов:

```
var numbers = [...]int{1,2,3,4,5}    // длина массива 5
numbers2 := [...]int{1,2,3}         // длина массива 3
fmt.Println(numbers)                // [1 2 3 4 5]
fmt.Println(numbers2)               // [1 2 3]
```

При этом длина массива является частью его типа. И, к примеру, следующие два массива представляют разные типы данных, хотя они и хранят данные одного типа:

```
var numbers [3]int = [3]int{1, 2, 3}
var numbers2 [4]int = [4]int{1, 2, 3, 4}
numbers = numbers2    // ! Ошибка
```

И в данном случае при присвоении мы получим ошибку, так как данные одного типа пытаемся передать переменной другого типа.

Индексы

Для обращения к элементам массива применяются индексы - номера элементов. При этом нумерация начинается с нуля, то есть первый элемент будет иметь индекс 0. Индекс указывается в квадратных скобках. По индексу можно получить значение элемента, либо изменить его:

```
package main
import "fmt"

func main() {
    var numbers [5]int = [5]int{1,2,3,4,5}
    fmt.Println(numbers[0])    // 1
    fmt.Println(numbers[4])    // 5
    numbers[0] = 87
    fmt.Println(numbers[0])    // 87
}
```

Индексы в массиве фактически выступают в качестве ключей, по которым можно обратиться к соответствующему значению. И в принципе мы можем явным образом указать, какому ключу какое значение будет соответствовать. При этом числовые ключи необязательно располагать в порядке возрастания:

```
colors := [3]string{2: "blue", 0: "red", 1: "green"}
fmt.Println(colors[2])    // blue
```

Условные конструкции

Последнее обновление: 20.12.2017

Условные конструкции проверяют истинность некоторого условия и в зависимости от результатов проверки позволяют направить ход программы по одному из путей.

if...else

Конструкция if принимает условие - выражение, которое возвращает значение типа bool . И если это условие истинно, то выполняется последующий блок инструкций.

```
package main
import "fmt"

func main() {
    a := 6
    b := 7
    if a < b {
        fmt.Println("a is less than b")
    }
}
```

Условие ставится после оператора if. В данном случае проверяется, меньше ли значение переменной a чем значение переменной b. Поскольку в данном случае значение переменной a действительно меньше значения переменной b, то есть условие возвращает true, то будет выполняться последующий блок кода, который выводит на консоль сообщение.

Если необходимо задать альтернативную логику, которая выполняется, в случае если условие неверно, то добавляется выражение else:

```
package main
import "fmt"

func main() {
    a := 6
    b := 7
    if a < b {
        fmt.Println("a меньше b")
    }
}
```

```

    }else{
        fmt.Println("a больше b")
    }
}

```

Таким образом, если условное выражение после оператора if истинно, то выполняется блок после if, если ложно - выполняется блок после else.

Если необходимо проверить несколько альтернативных вариантов, то можно добавить выражения else if :

```

package main
import "fmt"

func main() {
    a := 8
    b := 8
    if a < b {
        fmt.Println("a меньше b")
    }else if a > b{
        fmt.Println("a больше b")
    } else{
        fmt.Println("a равно b")
    }
}

```

Таким образом, если выражение после if истинно, то срабатывает блок if. Иначе проверяется выражение после else if. Если оно истинно, то выполняется блок else if. Если оно ложно, то выполняется блок else.

Выражений else if может быть множество:

```

if a ==9 {
    fmt.Println("a = 9")
}else if a == 8{
    fmt.Println("a = 8")
} else if a == 7{
    fmt.Println("a == 7")
}

```

switch

Конструкция switch проверяет значение некоторого выражения. С помощью операторов case определяются значения для сравнения. Если значение после оператора case совпадает со значением выражения из switch, то выполняется код данного блока case.

```

package main
import "fmt"

func main() {
    a := 8
    switch(a) {
        case 9:
            fmt.Println("a = 9")
        case 8:
            fmt.Println("a = 8")
        case 7:
            fmt.Println("a = 7")
    }
}

```

В качестве выражения конструкция switch использует переменную a. Ее значение последовательно сравнивается со значениями после операторов case. Поскольку переменная a равна 8, то будет выполняться блок case 8: `fmt.Println("a = 8")` . Остальные блоки case не выполняются.

При этом после оператора switch мы можем указывать любое выражение, которое возвращает значение. Например, операцию сложения:

```

a := 7
switch(a + 2) {
    case 9:
        fmt.Println("9")
    case 8:
        fmt.Println("8")
    case 7:
        fmt.Println("7")
}

```

Также конструкция switch может содержать необязательных блок default , который выполняется, если ни один из операторов case не содержит нужного значения:

```

package main
import "fmt"

func main() {
    a := 87
    switch(a) {
        case 9:
            fmt.Println("a = 9")
        case 8:
            fmt.Println("a = 8")
        case 7:
            fmt.Println("a = 7")
        default:
            fmt.Println("значение переменной a не определено")
    }
}

```

Также можно указывать после оператора case сразу несколько значений:

```

a := 5
switch(a) {
    case 9: fmt.Println("a = 9")
}

```

```

case 8: fmt.Println("a = 8")
case 7: fmt.Println("a = 7")
case 6, 5, 4:
    fmt.Println("a = 6 или 5 или 4, но это не точно")
default:
    fmt.Println("значение переменной a не определено")
}

```

Циклы

Последнее обновление: 20.12.2017

Циклы позволяют в зависимости от определенного условия выполнять некоторые действия множество раз. Фактически в Go есть только один цикл - цикл `for`, который может принимать разные формы. Этот цикл имеет следующее формальное определение:

```

for [инициализация счетчика]; [условие]; [изменение счетчика]{
    // действия
}

```

Например, выведем с помощью цикла квадраты чисел:

```

package main
import "fmt"

func main() {
    for i := 1; i < 10; i++){
        fmt.Println(i * i)
    }
}

```

Объявление цикла `for` разбивается на три части. Вначале идет инициализация счетчика: `i := 1`. Фактически она представляет объявление переменной, которая будет использоваться внутри цикла. В данном случае это счетчик `i`, начальное значение которого равно 1.

Вторая часть представляет условие: `i < 10`. Пока это условие истинно, то есть возвращает `true`, будет продолжаться цикл.

Третья часть представляет изменение (увеличение) счетчика на единицу.

В теле цикла на консоль выводится квадрат числа `i`.

Таким образом, цикл сработает 9 раз, пока значение `i` не станет равным 10. И каждый раз это значение будет увеличиваться на 1. Каждый отдельный проход цикла называется итерацией. То есть в данном случае будет 9 итераций. В итоге мы получим следующий консольный вывод:

```

1
4
9
16
25
36
49
64
81

```

Нам необязательно указывать все условия при объявлении цикла. Например, можно вынести объявление переменной вовне:

```

var i = 1
for ; i < 10; i++){
    fmt.Println(i * i)
}

```

Можно убрать изменение счетчика в само тело цикла и оставить только условие:

```

var i = 1
for ; i < 10;{
    fmt.Println(i * i)
    i++
}

```

Если цикл использует только условие, то его можно сократить следующим образом:

```

var i = 1
for i < 10{
    fmt.Println(i * i)
    i++
}

```

Вложенные циклы

Циклы могут быть вложенными, то есть располагаться внутри других циклов. Например, выведем на консоль таблицу умножения:

```

package main
import "fmt"

func main() {
    for i := 1; i < 10; i++){
        for j := 1; j < 10; j++){
            fmt.Print(i * j, "\t")
        }
        fmt.Println()
    }
}

```

```

1      2      3      4      5      6      7      8      9
2      4      6      8      10     12     14     16     18
3      6      9      12     15     18     21     24     27
4      8      12     16     20     24     28     32     36
5      10     15     20     25     30     35     40     45

```

6	12	18	24	30	36	42	48	54
7	14	21	28	35	42	49	56	63
8	16	24	32	40	48	56	64	72
9	18	27	36	45	54	63	72	81

Перебор массивов

Для перебора массивов можно использовать следующую форму цикла for:

```
for индекс, значение := range массив{
    // действия
}
```

При переборе мы можем по отдельности получить индекс элемента в массиве и значение этого элемента. Например, перебираем массив строк:

```
var users = [3]string{"Tom", "Alice", "Kate"}
for index, value := range users{
    fmt.Println(index, value)
}
```

Консольный вывод:

```
0 Tom
1 Alice
2 Kate
```

Если мы не планируем использовать значения или индексы элементов, то мы можем вместо них указать прочерк. Например, нам не нужны индексы:

```
for _, value := range users{
    fmt.Println(value)
}
```

Но также для перебора массива можно использовать и стандартную версию цикла for:

```
var users = [3]string{"Tom", "Alice", "Kate"}
for i:= 0; i < len(users); i++){
    fmt.Println(users[i])
}
```

В данном случае счетчик `i` играет роль индекса. Цикл выполняется, пока счетчик `i` не станет равным длине массива, которую можно получить с помощью функции `len()`

Операторы break и continue

Может возникнуть ситуация, когда нам надо при определенных условиях завершить текущую итерацию цикла, не выполняя все инструкции цикла, а сразу перейти к следующей итерации. В этом случае можно использовать оператор `continue`. Например, в массиве могут быть, как положительные, так и отрицательные числа. Допустим, нам нужна сумма только положительных чисел, поэтому, если нам встретится отрицательное число, мы можем просто перейти к следующей итерации с помощью `continue`:

```
var numbers = [10]int{1, -2, 3, -4, 5, -6, -7, 8, -9, 10}
var sum = 0

for _, value := range numbers{
    if value < 0{
        continue           // переходим к следующей итерации
    }
    sum += value
}
fmt.Println("Sum:", sum)    // Sum: 27
```

Оператор `break` полностью осуществляет выход из цикла:

```
var numbers = [10]int{1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
var sum = 0

for _, value := range numbers{
    if value > 4{
        break           // если число больше 4 выходим из цикла
    }
    sum += value
}
fmt.Println("Sum:", sum)    // Sum: 10
```

Функции и их параметры

Последнее обновление: 20.12.2017

Функция представляет блок операторов, которые все вместе выполняют какую-то определенную задачу. С помощью функций можно многократно вызывать ее блок операторов как единое целое в других частях программы.

Функция объявляется следующим образом:

```
func имя_функции (список_параметров) (типы_возвращаемых_значений){
    выполняемые_операторы
}
```

Функция определяется с помощью ключевого слова `func`, после которого идет имя функции. Затем в скобках идет список параметров. После списка параметров определяются типы возвращаемых из функции значений (если функция возвращает значения). И далее в фигурных скобках идут собственно те операторы, из которых состоит функция.

Название функции вместе с типами ее параметров и типами возвращаемых значений еще называют сигнатурой.

По умолчанию каждая программа на языке Go должна содержать как минимум одну функцию - функцию `main`, которая является входной точкой в приложение:

```
package main
import "fmt"

func main() {
    fmt.Println("Hello Go")
}
```

Функция `main` начинается с ключевого слова `func`, затем идет название - `main`. Функция не принимает никаких параметров, поэтому после названия идут пустые скобки. Функция `main` не возвращает никакого результата, поэтому после пустых скобок не указывается тип возвращаемого значения. И тело функции в фигурных скобках фактически состоит из вызова другой функции - `fmt.Println()`, которая выводит строку на консоль.

Теперь определим еще одну функцию:

```
package main
import "fmt"

func main() {

}

func hello(){
    fmt.Println("Hello World")
}
```

В данном случае мы определили функцию `hello`, которая не принимает никаких параметров, ничего не возвращает и просто выводит на консоль строку. Определить функцию можно в том же файле, где расположена функция `main`. Но если мы запустим эту программу, но на консоли мы ничего не увидим. Потому что программа выполняет только те действия, которые определены внутри функции `main`. Имя она является входной точкой в приложение. И если мы хотим выполнить в программе нашу функцию `hello`, то нам надо вызвать ее в функции `main`:

```
package main
import "fmt"

func main() {
    hello()
    hello()
    hello()
}

func hello(){
    fmt.Println("Hello World")
}
```

Для вызова функции пишется ее имя, после которого в скобках указываются значения для параметров функции. Но поскольку в данном случае функция `hello` не имеет параметров, то после ее названия пишем просто пустые скобки. И таким образом, при выполнении программы на консоль три раза будет выведена строка "Hello World":

```
Hello World
Hello World
Hello World
```

Таким образом, оформление группы операторов в виде функции позволяет не писать каждый раз всю группу операторов, а обращаться к ним по имени функции.

Параметры функции

Через параметры функция получает входные данные. Параметры указываются в скобках после имени функции. Для каждого параметра указывается имя и тип (как для переменной). Друг от друга параметров разделяются запятыми. При вызове функции необходимо передать значения для всех ее параметров. Например, мы хотим использовать функцию, которая складывает два любых числа:

```
package main
import "fmt"

func main() {
    add(4, 5)        // x + y = 9
    add(20, 6)       // x + y = 26
}

func add(x int, y int){
    var z = x + y
    fmt.Println("x + y = ", z)
}
```

Функция `add` имеет два параметра: `x` и `y`. Оба параметра представляют тип `int`, то есть целые числа. В самой функции определяется переменная, которая хранит сумму этих чисел. И затем сумма чисел выводится на консоль.

В функции `main` вызывается функция `add`. Так как функция принимает два параметра, то при вызове ей необходимо передать значения для этих параметров или два аргумента. Причем эти значения должны соответствовать параметрам по типу. То есть если параметр представляет тип `int`, то ему необходимо передать число.

Значения передаются по позиции. То есть первое значение получит первый параметр, второе значение - входной параметр и так далее. В итоге мы получим следующий консольный вывод:

```
x + y = 9
x + y = 26
```

Если несколько параметров подряд имеют один и тот же тип, то мы можем указать тип только для последнего параметра, а предыдущие параметры также будут представлять этот тип:

```
package main
import "fmt"

func main() {
    add(1, 2, 3.4, 5.6, 1.2)
}

func add(x, y int, a, b, c float32){
    var z = x + y
    var d = a + b + c
}
```

```

    fmt.Println("x + y = ", z)
    fmt.Println("a + b + c = ", d)
}

```

В качестве аргументов при вызове функции можно передавать и значения переменных, результаты операций или других функций, но при этом следует учитывать, что аргументы в функцию всегда передаются по значению:

```

package main
import "fmt"

func main() {
    var a = 8
    fmt.Println("a before: ", a)
    increment(a)
    fmt.Println("a after: ", a)
}
func increment(x int){

    fmt.Println("x before: ", x)
    x = x + 20
    fmt.Println("x after: ", x)
}

```

Консольный вывод:

```

a before: 8
x before: 8
x after: 28
a after: 8

```

В данном случае в качестве аргумента в функцию `increment` передается значение переменной `a`. Параметр `x` получает это значение, и оно увеличивается на 20. Однако несмотря на то, что значение параметра `x` увеличилось, значение переменной `a` никак не изменилось. Потому что при вызове функции передается копия значения переменной.

Неопределенное количество параметров

В Go функция может принимать неопределенное количество параметров одного типа. Например, нам надо получить сумму чисел, но мы точно не знаем, сколько чисел будут переданы в функцию:

```

package main
import "fmt"

func main() {
    add(1, 2, 3)           // sum = 6
    add(1, 2, 3, 4)        // sum = 10
    add(5, 6, 7, 2, 3)     // sum = 23
}

func add(numbers ...int){
    var sum = 0
    for _, number := range numbers{
        sum += number
    }
    fmt.Println("sum = ", sum)
}

```

Для определения параметра, который представляет неопределенное количество значений, перед типом этих значений ставится многоточие: `numbers ...int`. То есть через подобный параметр мы получаем несколько значений типа `int`.

При вызове мы можем передать в функцию `add` разное количество чисел:

```

add(1, 2, 3)           // sum = 6
add(1, 2, 3, 4)        // sum = 10
add(5, 6, 7, 2, 3)     // sum = 23

```

От этого случае следует отличать передачу среза в качестве параметра:

```

add([]int{1, 2, 3})
add([]int{1, 2, 3, 4})
add([]int{5, 6, 7, 2, 3})

```

В данном случае мы получим ошибку, так как передача среза не эквивалентна передаче неопределенного количества параметров того же типа. Если мы хотим передать срез, то надо указать после аргумента-массива многоточие:

```

add([]int{1, 2, 3}...)
add([]int{1, 2, 3, 4}...)

var nums = []int{5, 6, 7, 2, 3}
add(nums...)

```

Возвращение результата из функции

Последнее обновление: 20.12.2017

Функции могут возвращать результат. Для этого нужно после списка параметров функции указать тип возвращаемого результата. А в теле функции использовать оператор `return`, после которого указывается возвращаемое значение:

```

func имя_функции (список_параметров) тип_возвращаемого_значения {
    выполняемые_операторы
    return возвращаемое_значение
}

```

Например, мы хотим вернуть из функции сумму двух чисел:

```

package main
import "fmt"

```

```
func main() {
    var a = add(4, 5)      // 9
    var b = add(20, 6)     // 26
    fmt.Println(a)
    fmt.Println(b)
}

func add(x, y int) int {
    return x + y
}
```

Функция `add` возвращает значение типа `int`, поэтому данный тип указан после списка параметров. В самой функции после оператора `return` указывается возвращаемое значение. При этом данное значение может быть значением переменной, литералом, либо же, как в данном случае, результатом операции или вызова функции. То есть выражение `x + y` определяет возвращаемое значение.

Поскольку функция возвращает значение, то при вызове функции мы можем получить это значение и передать его переменной:

```
var a = add(4, 5)      // 9
var b = add(20, 6)     // 26
```

Именованные возвращаемые результаты

Возвращаемый результат может быть именован:

```
func add(x, y int) (z int) {
    z = x + y
    return
}
```

В скобках после списка параметров фактически определяется переменная, значение которой будет возвращаться. В конце функции ставится оператор `return`, но теперь необязательно после этого оператора ставить возвращаемое значение. Фактически мы также могли бы написать:

```
func add(x, y int) int {
    var z int = x + y
    return z
}
```

Возвращение нескольких значений

В Go функция может возвращать сразу несколько значений. В этом случае после списка параметров указывается в скобках список типов возвращаемых значений. А после оператора `return` располагаются через запятую все возвращаемые значения:

```
package main
import "fmt"

func main() {
    var age, name = add(4, 5, "Tom", "Simpson")
    fmt.Println(age)      // 9
    fmt.Println(name)     // Tom Simpson
}

func add(x, y int, firstName, lastName string) (int, string) {
    var z int = x + y
    var fullName = firstName + " " + lastName
    return z, fullName
}
```

Функция `add` принимает четыре параметра: два числа и две строки. Возвращает число (значение типа `int`) и строку. Возвращаемые значения указаны после оператора `return`.

Поскольку функция теперь возвращает два значения, то при вызове этой функции мы можем присвоить ее результат двум переменным:

```
var age, name = add(4, 5, "Tom", "Simpson")
```

Первое возвращаемое значение передается первой переменной `age`, а второе значение передается второй переменной `name`.

Альтернативный способ передачи переменным результатов функции:

```
age, name := add(4, 5, "Tom", "Simpson")
```

И также в данном случае можно было бы использовать именованные результаты:

```
func add(x, y int, firstName, lastName string) (z int, fullName string) {
    z = x + y
    fullName = firstName + " " + lastName
    return
}
```

Тип функции

Последнее обновление: 26.12.2017

Каждая функция имеет определенный тип, который складывается из списка типов параметров и списка типов возвращаемых результатов. Например, возьмем следующую функцию:

```
func add(x int, y int) int{
    return x + y
}
```

Эта функция представляет тип `func(int, int) int`. Этому же типу будет соответствовать следующая функция:

```
func multiply(x int, y int) int{
    return x * y
}
```

Хотя эта функция называется иначе, выполняет другие действия, но по типу параметров и по типу возвращаемого результата она соответствует выше указанному типу функции.

Возьмем еще одну функцию:

```
func display(message string){
    fmt.Println(message)
}
```

Эта функция имеет тип `func(string)`. То есть опять же вначале идет слово `func`, а затем в скобках типы параметров. Так как функция не возвращает никакого результата, то соответственно возвращаемый тип не указывается.

Но какое значение имеет тип функции? Это значит, что мы можем определять переменные или параметры функций, которые будут представлять определенный тип функции. То есть фактически переменная может быть функцией. Например:

```
package main

import "fmt"

func add(x int, y int) int{
    return x + y
}

func main() {

    var f func(int, int) int = add
    fmt.Println(f(3, 4))

    var x = f(4, 5)          // 9
    fmt.Println(x)
}
```

Здесь переменная `f` имеет тип `func(int, int) int`, то есть представляет любую функцию, которая принимает два параметра типа `int` и возвращает значение типа `int`. Поэтому мы можем присвоить этой переменной функцию `add`, которая соответствует данному типу:

```
var f func(int, int) int = add
```

После этого мы можем вызывать присвоенную функцию через переменную, передавая нужные значения для ее параметров:

```
var x = f(4, 5) // 9
```

При этом переменная может изменять функцию, на которую она указывает, но при этом функция обязательно должна соответствовать ее типу:

```
package main

import "fmt"

func add(x int, y int) int{ return x + y}
func multiply(x int, y int) int{ return x * y}

func display(message string){
    fmt.Println(message)
}

func main() {

    f := add                //или так var f func(int, int) int = add
    fmt.Println(f(3, 4))    // 7

    f = multiply            // теперь переменная f указывает на функцию multiply
    fmt.Println(f(3, 4))    // 12

    // f = display          // ошибка, так как функция display имеет тип func(string)

    var f1 func(string) = display // норм
    f1("hello")
}
```

Функции как параметры других функций

Также функция может передаваться в качестве параметра в другую функцию. Например:

```
package main

import "fmt"

func add(x int, y int) int {

    return x + y
}
func multiply(x int, y int) int {

    return x * y
}
func action(n1 int, n2 int, operation func(int, int) int){

    result := operation(n1, n2)
    fmt.Println(result)
}
func main() {

    action(10, 25, add)          // 35
    action(5, 6, multiply)      // 30
}
```

Здесь функция `action` принимает три параметра. Первые два параметра - числа, а третий параметр - функция, которая соответствует типу: `func(int, int) int`. То есть третий параметр представляет некоторое действие и может быть представлен любой функцией, которая принимает два значения типа `int` и возвращает также значение типа `int`. Для примера здесь как раз определены две подобных функции, которые соответствуют данному типу: `add` и `multiply`. Через имя параметра `operation` мы сможем вызывать данную функцию.

Или еще один пример:

```
package main

import "fmt"

func isEven(n int) bool{
    return n % 2 == 0
}
func isPositive(n int) bool{
    return n > 0
}

func sum(numbers []int, criteria func(int) bool) int{
    result := 0
    for _, val := range numbers{
        if(criteria(val)){
            result += val
        }
    }
    return result
}

func main() {
    slice := []int{-2, 4, 3, -1, 7, -4, 23}

    sumOfEvens := sum(slice, isEven)           // сумма четных чисел
    fmt.Println(sumOfEvens)                   // -2

    sumOfPositives := sum(slice, isPositive)    // сумма положительных чисел
    fmt.Println(sumOfPositives)                // 37
}
```

Здесь функция `sum` вычисляет сумму элементов среза. Но не всех элементов, а только тех, которые соответствуют условию. Условие передается в виде функции в качестве второго параметра. Условие должно соответствовать функции типа `func(int) bool`. То есть функция должна принимать в качестве параметра значение типа `int` и возвращать значение типа `bool`, которое указывает, соответствует ли переданное число условию.

Для примера здесь также определены две вспомогательные функции: `isEven` (возвращает `true`, если число четное) и `isPositive` (возвращает `true`, если число положительное). Эти функции соответствуют типу `func(int) bool`, поэтому их можно использовать в качестве условия.

Функция как результат другой функции

Функция также может возвращаться из другой функции в качестве результата:

```
package main

import "fmt"

func add(x int, y int) int{ return x + y}
func subtract(x int, y int) int{ return x - y}
func multiply(x int, y int) int{ return x * y}

func selectFn(n int) (func(int, int) int){
    if n==1 {
        return add
    }else if n==2{
        return subtract
    }else{
        return multiply
    }
}

func main() {
    f := selectFn(1)
    fmt.Println(f(3, 4))           // 7

    f = selectFn(3)
    fmt.Println(f(3, 4))           // 12
}
```

Здесь в зависимости от значения параметра функция `selectFn` возвращает одну из трех функций: `add`, `subtract` или `multiply`.

Анонимные функции

Последнее обновление: 26.12.2017

Анонимные функции - это функции, которым не назначено имя. Они отличаются от обычных функций также тем, что они могут определяться внутри других функций и также могут иметь доступ к контексту выполнения.

Анонимные функции позволяют нам определить некоторое действие непосредственно там, где оно применяется. Например, нам надо выполнить сложение двух чисел, но больше нигде это действие в программе не нужно:

```
package main

import "fmt"

func main() {
    f := func(x, y int) int{ return x + y}
    fmt.Println(f(3, 4))           // 7
    fmt.Println(f(6, 7))           // 13
}
```

Фактически переменная `f` определена как:

```
var f func(int, int) int = func(x, y int) int{ return x + y}
```

То есть переменной `f` можно присвоить любую функцию, которая соответствует типу `func(int, int) int`.

Анонимная функция как аргумент функции

Очень удобно использовать анонимные функции в качестве аргументов других функций:

```
package main

import "fmt"

func action(n1 int, n2 int, operation func(int, int) int){
    result := operation(n1, n2)
    fmt.Println(result)
}

func main() {
    action(10, 25, func (x int, y int) int { return x + y }) // 35
    action(5, 6, func (x int, y int) int { return x * y }) // 30
}
```

Анонимная функция как результат функции

Анонимная функция может быть результатом другой функции:

```
package main

import "fmt"

func selectFn(n int) (func(int, int) int){
    if n==1 {
        return func(x int, y int) int{ return x + y}
    }else if n==2{
        return func(x int, y int) int{ return x - y}
    }else{
        return func(x int, y int) int{ return x * y}
    }
}

func main() {
    f := selectFn(1)
    fmt.Println(f(2, 3)) // 5
    fmt.Println(f(4, 5)) // 9
    fmt.Println(f(7, 6)) // 13
}
```

Доступ к окружению

Преимуществом анонимных функций является то, что они имеют доступ к окружению, в котором они определяются. Например:

```
package main

import "fmt"

func square() func() int {
    var x int = 2
    return func() int {
        x++
        return x * x
    }
}

func main() {
    f := square()
    fmt.Println(f()) // 9
    fmt.Println(f()) // 16
    fmt.Println(f()) // 25
}
```

Здесь функция `square` определяет локальную переменную `x` и возвращает анонимную функцию. Анонимная функция увеличивает значение переменной `x` и возвращает ее квадрат.

Таким образом, мы можем зафиксировать у внешней функции `square` состояние в виде переменной `x`, которое будет изменяться в анонимной функции.

Рекурсивные функции

Последнее обновление: 26.12.2017

Рекурсивная функция представляет такую функцию, которая вызывает саму себя. Рекурсивные функции представляют мощный инструмент для обработки рекурсивных структур данных, например, различных деревьев.

Например, определим функцию вычисления факториала числа, которая получает результат рекурсивным способом:

```
package main

import "fmt"

func factorial(n uint) uint{
    if n == 0{
        return 1
    }
    return n * factorial(n - 1)
}

func main() {
```

```

    fmt.Println(factorial(4))    // 24
    fmt.Println(factorial(5))    // 120
    fmt.Println(factorial(6))    // 720
}

```

Здесь функция `factorial` получает некоторое положительное число, для которого надо вычислить факториал. Полученный результат возвращается из функции. Вначале идет условие, что если число равно 0, то функция возвращает 1. Иначе функция возвращает произведение числа `n` на результат этой же функции для числа `n-1`.

При создании рекурсивной функции в ней обязательно должен быть некоторый базовый вариант, который использует оператор `return` и помещается в начале функции. В случае с факториалом это `if x == 0 {return 1}`.

И, кроме того, все рекурсивные вызовы должны обращаться к подфункциям, которые в конце концов сходятся к базовому варианту. Так, при передаче в функцию положительного числа при дальнейших рекурсивных вызовах подфункций в них будет передаваться каждый раз число, меньшее на единицу. И в конце концов мы дойдем до ситуации, когда число будет равно 0, и будет использован базовый вариант.

Например, вызов `factorial(4)` фактически можно расписать следующим образом:

- `factorial(4)`
- `4 * factorial(3)`
- `4 * 3 * factorial(2)`
- `4 * 3 * 2 * factorial(1)`
- `4 * 3 * 2 * 1 * factorial(0)`
- `4 * 3 * 2 * 1 * 1`

Другим распространенным показательным примером рекурсивной функции служит функция, вычисляющая числа Фибоначчи. `n`-й член последовательности Фибоначчи определяется по формуле: $f(n)=f(n-1) + f(n-2)$, причем $f(0)=0$, а $f(1)=1$.

```

package main

import "fmt"

func fibonacci(n uint) uint{
    if n == 0 {
        return 0
    }
    if n == 1 {
        return 1
    }
    return fibonacci(n - 1) + fibonacci(n - 2)
}

func main() {

    fmt.Println(fibonacci(4))    // 3
    fmt.Println(fibonacci(5))    // 5
    fmt.Println(fibonacci(6))    // 8
}

```

defer и panic

Последнее обновление: 21.01.2017

defer

Оператор `defer` позволяет выполнить определенную функцию в конце программы, при этом не важно, где в реальности вызывается эта функция. Например:

```

package main
import "fmt"

func main() {
    defer finish()
    fmt.Println("Program has been started")
    fmt.Println("Program is working")
}

func finish(){
    fmt.Println("Program has been finished")
}

```

Здесь функция `finish` вызывается с оператором `defer`, поэтому данная функция в реальности будет вызываться в самом конце выполнения программы, несмотря на то, что ее вызов определен в начале функции `main`. В частности, мы получим следующий консольный вывод:

```

Program has been started
Program is working
Program has been finished

```

Если несколько функций вызываются с оператором `defer`, то те функции, которые вызываются раньше, будут выполняться позже всех. Например:

```

package main
import "fmt"

func main() {

    defer finish()
    defer fmt.Println("Program has been started")
    fmt.Println("Program is working")
}

```

```
func finish(){
    fmt.Println("Program has been finished")
}
```

Консольный вывод:

```
Program is working
Program has been started
Program has been finished
```

panic

Оператор panic позволяет сгенерировать ошибку и выйти из программы:

```
package main
import "fmt"

func main() {
    fmt.Println(divide(15, 5))
    fmt.Println(divide(4, 0))
    fmt.Println("Program has been finished")
}
func divide(x, y float64) float64{
    if y == 0{
        panic("Division by zero!")
    }
    return x / y
}
```

Оператору panic мы можем передать любое сообщение, которое будет выводиться на консоль. Например, в данном случае в функции divide, если второй параметр равен 0, то осуществляется вызов panic("Division by zero!") .

В функции main в вызове fmt.Println(divide(4, 0)) будет выполняться оператор panic, поскольку второй параметр функции divide равен 0. И в этом случае все последующие операции, которые идут после этого вызова, например, в данном случае это вызов fmt.Println("Program has been finished") , не будут выполняться. В этом случае мы получим следующий консольный вывод:

```
3
panic: Division by zero!
```

И в конце вывода будет идти диагностическая информация о том, где возникла ошибка.

Срезы

Последнее обновление: 21.12.2017

Срезы (slice) представляют последовательность элементов одного типа переменной длины. В отличие от массивов длина в срезах не фиксирована и динамически может меняться, то есть можно добавлять новые элементы или удалять уже существующие.

Срез определяется также, как и массив, за тем исключением, что у него не указывается длина:

```
var users []string
```

Также срез можно инициализировать значениями:

```
var users = []string{"Tom", "Alice", "Kate"}
// или так
users2 := []string{"Tom", "Alice", "Kate"}
```

К элементам среза обращение происходит также, как и к элементам массива - по индексу и также мы можем перебирать все элементы с помощью цикла for:

```
var users []string = []string{"Tom", "Alice", "Kate"}
fmt.Println(users[2]) // Kate
users[2] = "Katherine"

for _, value := range users{
    fmt.Println(value)
```

С помощью функции make() можно создать срез из нескольких элементов, которые будут иметь значения по умолчанию:

```
var users []string = make([]string, 3)
users[0] = "Tom"
users[1] = "Alice"
users[2] = "Bob"
```

Добавление в срез

Для добавления в срез применяется встроенная функция append(slice, value) . Первый параметр функции - срез, в который надо добавить, а второй параметр - значение, которое нужно добавить. Результатом функции является увеличенный срез.

```
users := []string{"Tom", "Alice", "Kate"}
users = append(users, "Bob")

for _, value := range users{
    fmt.Println(value)
}
```

Оператор среза

Оператор среза s[i:j] создает из последовательности s новый срез, который содержит элементы последовательности s с i по j-1. При этом должно соблюдаться условие 0 ≤ i ≤ j ≤ cap(s) . В качестве исходной последовательности, из которой берутся элементы, может использоваться массив, указатель на массив или другой срез. В итоге в полученном срезе будет j-i элементов.

Если значение `i` не указано, то применяется по умолчанию значение 0. Если значение `j` не указано, то вместо него используется длина исходной последовательности `s`.

```
// исходный массив
initialUsers := [8]string{"Bob", "Alice", "Kate", "Sam", "Tom", "Paul", "Mike", "Robert"}
users1 := initialUsers[2:6]           // с 3-го по 6-й
users2 := initialUsers[:4]            // с 1-го по 4-й
users3 := initialUsers[3:]            // с 4-го до конца

fmt.Println(users1)                   // ["Kate", "Sam", "Tom", "Paul"]
fmt.Println(users2)                   // ["Bob", "Alice", "Kate", "Sam"]
fmt.Println(users3)                   // ["Sam", "Tom", "Paul", "Mike", "Robert"]
```

Удаление элемента

Что делать, если необходимо удалить какой-то определенный элемент? В этом случае мы можем комбинировать функцию `append` и оператор среза:

```
users := [8]string{"Bob", "Alice", "Kate", "Sam", "Tom", "Paul", "Mike", "Robert"}
//удаляем 4-й элемент
var n = 3
users = append(users[:n], users[n+1:]...)
fmt.Println(users)                  //[ "Bob", "Alice", "Kate", "Tom", "Paul", "Mike", "Robert"]
```

Отображения

Последнее обновление: 21.01.2017

Отображение или `map` представляет ссылку на хеш-таблицу - структуру данных, где каждый элемент представляет пару "ключ-значение". При этом каждый элемент имеет уникальный ключ, по которому можно получить значение элемента. Отображение определяется как объект типа `map[K]V`, где `K` представляет тип ключа, а `V` - тип значения. Причем тип ключа `K` должен поддерживать операцию сравнения `==`, чтобы отображение могло сопоставить значение с одним из ключей и хеш-таблицы.

Например, определение отображения, которое в качестве ключей имеет тип `string`, а в качестве значений - тип `int`:

```
var people map[string]int           // Ключи представляют тип string, значения - тип int
```

Установка значений отображения:

```
var people = map[string]int{
    "Tom": 1,
    "Bob": 2,
    "Sam": 4,
    "Alice": 8,
}
fmt.Println(people)                 // map[Tom:1 Bob:2 Sam:4 Alice:8]
```

Как и в массиве или в срезе элементы помещаются в фигурные скобки. Каждый элемент представляет пару ключ -значение. Вначале идет ключ и через двоеточие значение. Определение элемента завершается запятой.

Для обращения к элементам нужно применять ключи:

```
var people = map[string]int{
    "Tom": 1,
    "Bob": 2,
    "Sam": 4,
    "Alice": 8,
}
fmt.Println(people["Alice"])         // 8
fmt.Println(people["Bob"])           // 2
people["Bob"] = 32
fmt.Println(people["Bob"])           // 32
```

Для проверки наличия элемента по определенному ключу можно применять выражение `if`:

```
var people = map[string]int{
    "Tom": 1,
    "Bob": 2,
    "Sam": 4,
    "Alice": 8,
}
if val, ok := people["Tom"]; ok{
    fmt.Println(val)
}
```

Если значение по заданному ключу имеется в отображении, то переменная `ok` будет равна `true`, а переменная `val` будет хранить полученное значение. Если переменная `ok` равна `false`, то значения по ключу в отображении нет.

Для перебора элементов применяется цикл `for`:

```
var people = map[string]int{
    "Tom": 1,
    "Bob": 2,
    "Sam": 4,
    "Alice": 8,
}
for key, value := range people{
    fmt.Println(key, value)
}
```

Консольный вывод программы:

```
Tom 1
Bob 2
Sam 4
Alice 8
```

Функция `make` представляет альтернативный вариант создания отображения. Она создает пустую хеш-таблицу:

```
people := make(map[string]int)
```

Добавление и удаление элементов

Для добавления элементов достаточно просто выполнить установку значения по новому ключу и элемент с этим ключом будет добавлен в коллекцию:

```
var people = map[string]int{ "Tom": 1, "Bob": 2}
people["Kate"] = 128
fmt.Println(people)           // map[Tom:1 Bob:2 Kate:128]
```

Для удаления применяется встроенная функция `delete(map, key)`, первым параметром которой является отображение, а вторым - ключ, по которому надо удалить элемент.

```
var people = map[string]int{ "Tom": 1, "Bob": 2, "Sam": 8}
delete(people, "Bob")
fmt.Println(people)           // map[Tom:1 Sam:8]
```

Указатели

Что такое указатели

Последнее обновление: 24.12.2017

Указатели представляют собой объекты, значением которых служат адреса других объектов (например, переменных).

Указатель определяется как обычная переменная, только перед типом данных ставится символ звездочки *. Например, определение указателя на объект типа `int`:

```
var p *int
```

Этому указателю можно присвоить адрес переменной типа `int`. Для получения адреса применяется операция `&`, после которой указывается имя переменной (`&x`).

```
package main

import "fmt"

func main() {

    var x int = 4           // определяем переменную
    var p *int              // определяем указатель
    p = &x                 // указатель получает адрес переменной
    fmt.Println(p)         // значение самого указателя - адрес переменной x
}
```

Здесь указатель `p` хранит адрес переменной `x`. Что важно, переменная `x` имеет тип `int`, и указатель `p` указывает именно на объект типа `int`. То есть должно быть соответствие по типу. И если мы попробуем вывести адрес переменной на консоль, то увидим, что он представляет шестнадцатеричное значение:

```
0xc0420120a0
```

В каждом отдельном случае адрес может отличаться, но к примеру, в моем случае машинный адрес переменной `x` - `0xc0420120a0`. То есть в памяти компьютера есть адрес `0xc0420120a0`, по которому располагается переменная `x`.

По адресу, который хранит указатель, мы получить значение переменной `x`. Для этого применяется операция `*` или операция разыменования. Результатом этой операции является значение переменной, на которую указывает указатель. Применим данную операцию и получим значение переменной `x`:

```
package main

import "fmt"

func main() {

    var x int = 4
    var p *int = &x           // указатель получает адрес переменной
    fmt.Println("Address:", p) // значение указателя - адрес переменной x
    fmt.Println("Value:", *p)  // значение переменной x
}
```

Консольный вывод:

```
Address: 0xc0420c058
Value: 4
```

И также используя указатель, мы можем менять значение по адресу, который хранится в указателе:

```
var x int = 4
var p *int = &x
*p = 25
fmt.Println(x)           // 25
```

Для определения указателей можно использовать также сокращенную форму:

```
f := 2.3
pf := &f

fmt.Println("Address:", pf)
fmt.Println("Value:", *pf)
```

Пустой указатель

Если указателю не присвоен адрес какого-либо объекта, то такой указатель по умолчанию имеет значение `nil` (по сути отсутствие значения). Если мы попробуем получить значение по такому пустому указателю, то мы столкнемся с ошибкой:

```
var pf *float64
fmt.Println("Value:", *pf)      // ! ошибка, указатель не указывает на какой-либо объект
```

Поэтому при работе с указателями иногда бывает целесообразно проверять на значение nil:

```
var pf *float64
if pf != nil{
    fmt.Println("Value:", *pf)
}
```

Функция new

Переменная представляет именованный объект в памяти. Язык Go также позволяет создавать безымянные объекты - они также размещаются в памяти, но не имеют имени как переменные. Для этого применяется функция new(type) . В эту функцию передается тип, объект которого надо создать. Функция возвращает указатель на созданный объект:

```
package main

import "fmt"

func main() {

    p := new(int)
    fmt.Println("Value:", *p)          // Value: 0 - значение по умолчанию
    *p = 8                             // изменяем значение
    fmt.Println("Value:", *p)          // Value: 8
}
```

В данном случае указатель p будет иметь тип *int , поскольку он указывает на объект типа int. Создаваемый объект имеет значение по умолчанию (для типа int это число 0).

Объект, созданный с помощью функции new, ничем не отличается от обычной переменной. Единственное что, чтобы обратиться к этому объекту - получить или изменить его адрес, необходимо использовать указатель.

Указатели и функции

Последнее обновление: 24.12.2017

Указатели как параметры функции

По умолчанию все параметры передаются в функцию по значению. Например:

```
package main
import "fmt"

func changeValue(x int){
    x = x * x
}

func main() {

    d := 5
    fmt.Println("d before:", d)          // 5
    changeValue(d)                      // изменяем значение
    fmt.Println("d after:", d)           // 5 - значение не изменилось
}
```

Функция changeValue изменяет значение параметра, возводя его в квадрат. Но после вызова этой функции мы видим, что значение переменной d, которая передается в changeValue, не изменилось. Ведь функция получает копию данной переменной и работает с ней независимо от оригинальной переменной d. Поэтому d никак не изменяется.

Однако что, если нам все таки надо менять значение передаваемой переменной? И в этом случае мы можем использовать указатели:

```
package main

import "fmt"

func changeValue(x *int){
    *x = (*x) * (*x)
}

func main() {

    d := 5
    fmt.Println("d before:", d)          // 5
    changeValue(&d)                      // изменяем значение
    fmt.Println("d after:", d)           // 25 - значение изменилось!
}
```

Теперь функция changeValue принимает в качестве параметра указатель на объект типа int. При вызове функции changeValue в нее передается адрес переменной d (changeValue(&d)). И после ее выполнения мы видим, что значение переменной d изменилось.

Указатель как результат функции

Функция может возвращать указатель:

```
package main
import "fmt"

func createPointer(x int) *int{
    p := new(int)
    *p = x
    return p
}

func main() {

    p1 := createPointer(7)
```

```

    fmt.Println("p1:", *p1)      // p1: 7
    p2 := createPointer(10)
    fmt.Println("p2:", *p2)      // p2: 10
    p3 := createPointer(28)
    fmt.Println("p3:", *p3)      // p3: 28
}

```

В данном случае функция `createPointer` возвращает указатель на объект `int`.

Производные типы

Именованные типы и псевдонимы

Последнее обновление: 18.02.2024

Оператор `type` позволяет определять именованный тип на основе другого. Например:

```
type mile uint
```

В данном случае определяется именованный тип `mile`, который основывается на типе `uint`. По сути `mile` представляет тип `uint` и работа с ним будет производиться также, как и с типом `uint`. Однако в то же время фактически это новый тип.

Мы можем определять переменные данного типа, работать с ними как с объектами базового типа `uint`:

```

package main
import "fmt"

type mile uint

func main() {
    var distance mile = 5
    fmt.Println(distance)
    distance += 5
    fmt.Println(distance)
}

```

Но может возникнуть вопрос, а зачем это нужно, зачем определять именованный тип, если он все равно ведет себя как тип `uint`? Рассмотрим следующую ситуацию:

```

package main
import "fmt"

type mile uint
type kilometer uint

func distanceToEnemy (distance mile){
    fmt.Println("расстояние для противника:")
    fmt.Println(distance, "миль")
}

func main() {
    var distance mile = 5
    distanceToEnemy(distance)

    // var distance1 uint = 5
    // distanceToEnemy(distance1)    // !Ошибка

    // var distance2 kilometer = 5
    // distanceToEnemy(distance2)    // ! ошибка
}

```

Здесь определены два именованных типа: `mile` и `kilometer`, которые по сути представляют тип `uint` и которые предназначены для выражения расстояния в милях и километрах соответственно. И также определена функция `distanceToEnemy()`, которая отображает расстояние в милях до условного противника. В качестве параметра принимает значение `mile` - именно значение типа `mile`, а не типа `uint`. Это позволит нам уменьшить вероятность передачи некорректных данных. То есть передаваемые данные должны быть явным образом определены в программе как значение типа `mile`, а не типа `uint` или типа `kilometer`. Два именованных типа считаются разными, даже если они основаны на некотором общем типе (как `uint` в данном случае).

Также именованные типы позволяют придать типу некоторый дополнительный смысл. Так, использование в коде типа `"kilometer"` или `"mile"` позволит указать на предназначение переменной или параметра и будет более описательным, чем просто тип `uint`.

Еще одна ситуация, где можно применять именованные типы - это сокращение названия типов в том случае, если они слишком длинные или громоздкие. Например, рассмотрим следующий пример:

```

package main

import "fmt"

func action(n1 int, n2 int, op func(int, int) int){
    result := op(n1, n2)
    fmt.Println(result)
}

func add(x int, y int) int {
    return x + y
}

func main() {
    var myOperation func(int, int) int = add
    action(10, 25, myOperation)    // 35
}

```


Здесь определена функция action, которая принимает два числа и некоторую другую функцию с типом `func(int, int) int` - то есть функцию, которая принимает два числа и также возвращает число. В функции main определяется переменная myOperation, которая как раз представляет функцию типа `func(int, int) int`, получает ссылку на функцию add и передается в вызов `action(10, 25, myOperation)`

Теперь определим именованный тип для типа `func(int, int) int`:

```
package main

import "fmt"

type BinaryOp func(int, int) int

func action(n1 int, n2 int, op BinaryOp){
    result := op(n1, n2)
    fmt.Println(result)
}

func add(x int, y int) int {
    return x + y
}

func main() {
    var myOperation BinaryOp = add
    action(10, 35, myOperation)    // 45
}
```

Теперь тип функции `func(int, int) int` проецируется на именованный тип BinaryOp, который представляет бинарную операцию над двумя операндами:

```
type BinaryOp func(int, int) int
```

Такое название короче оригинального определения типа и в то же время является более описательным (по крайней мере для меня). Соответственно далее его можно использовать для указания типа параметра:

```
func action(n1 int, n2 int, op BinaryOp){ ... }
```

или переменной:

```
var myOperation BinaryOp = add
```

Псевдонимы

На именованные типы похожи псевдонимы. Они также определяются с помощью оператора `type`, только при присвоении типа применяется операция присваивания:

```
type псевдоним = имеющийся_тип
```

Однако псевдоним НЕ определяет нового типа и все псевдонимы одного и того же типа считаются идентичными. Например:

```
package main
import "fmt"

type mile = uint
type kilometer = uint

func distanceToEnemy (distance mile){
    fmt.Println("расстояние для противника:")
    fmt.Println(distance, "миль")
}

func main() {
    var distance mile = 5
    distanceToEnemy(distance)

    var distance1 uint = 5
    distanceToEnemy(distance1)    // норм

    var distance2 kilometer = 5
    distanceToEnemy(distance2)    // норм
}
```

Здесь для типа `uint` определяются два псевдонима - `mile` и `kilometer`. И несмотря на то, что параметр функции `distanceToEnemy` определен как параметр типа `mile`, ему можно передать и значение собственно типа `uint`, и значение его псевдонима - `kilometer`.

Обычно псевдонимы применяются для сокращения названий других типов или для определения более описательного имени.

Структуры

Последнее обновление: 24.12.2017

Структуры представляют тип данных, определяемый разработчиком и служащий для представления каких-либо объектов. Структуры содержат набор полей, которые представляют различные атрибуты объекта. Для определения структуры применяются ключевые слова `type` и `struct`:

```
type имя_структуры struct{
    поля_структуры
}
```

Каждое поле имеет название и тип данных, как переменная. Например, определим структуру, которая представляет человека:

```
type person struct{
    name string
}
```

```
    age int
}
```

Структура называется person. Она имеет два поля: name (имя человека, представляет тип string) и age (возраст человека, представляет тип int).

Создание и инициализация структуры

Структура представляет новый тип данных, и мы можем определить переменную данного типа:

```
var tom person
```

С помощью инициализатора можно передать структуре начальные значения:

```
var tom person = person{"Tom", 23}
```

Инициализатор представляет набор значений в фигурных скобках. Причем эти значения передаются полям структуры в том порядке, в котором поля определены в структуре. Например, в данном случае строка "Tom" передается первому полю - name, а второе значение - 23 передается второму полю - age.

Также мы можем явным образом указать какие значения передаются свойствам:

```
var alice person = person{age: 23, name: "Alice"}
```

Также можно использовать сокращенные способы инициализации переменной структуры:

```
var tom = person {name: "Tom", age: 24}
bob := person {name: "Bob", age: 31}
```

Можно даже не указывать никаких значений, в этом случае свойства структуры получают значения по умолчанию:

```
undefined := person {} // name - пустая строка, age - 0
```

Обращение к полям структуры

Для обращения к полям структуры после переменной ставится точка и указывается поле структуры:

```
package main
import "fmt"

type person struct{
    name string
    age int
}

func main() {

    var tom = person {name: "Tom", age: 24}
    fmt.Println(tom.name)      // Tom
    fmt.Println(tom.age)       // 24

    tom.age = 38 // изменяем значение
    fmt.Println(tom.name, tom.age) // Tom 38
}
```

Указатели на структуры

Как и в случае с обычными переменными, можно создавать указатели на структуры.

```
package main
import "fmt"

type person struct{
    name string
    age int
}

func main() {

    tom := person {name: "Tom", age: 22}
    var tomPointer *person = &tom
    tomPointer.age = 29
    fmt.Println(tom.age) // 29
    (*tomPointer).age = 32
    fmt.Println(tom.age) // 32
}
```

Для инициализации указателя на структуру необязательно присваивать ему адрес переменной. Можно присвоить адрес безымянного объекта следующим образом:

```
var tom *person = &person{name:"Tom", age:23}
var bob *person = new(person)
```

Для обращения к полям структуры через указатель можно использовать операцию разыменования ((*tomPointer).age), либо напрямую обращаться по указателю (tomPointer.age).

Также можно определять указатели на отдельные поля структуры:

```
tom := person {name: "Tom", age: 22}
var agePointer *int = &tom.age // указатель на поле tom.age
*agePointer = 35 // изменяем значение поля
fmt.Println(tom.age) // 35
```

Вложенные структуры

Последнее обновление: 24.12.2017

Поля одних структур могут представлять другие структуры. Например:

```
package main
import "fmt"

type contact struct{
    email string
    phone string
}

type person struct{
    name string
    age int
    contactInfo contact
}

func main() {
    var tom = person {
        name: "Tom",
        age: 24,
        contactInfo: contact{
            email: "tom@gmail.com",
            phone: "+1234567899",
        },
    }
    tom.contactInfo.email = "supertom@gmail.com"

    fmt.Println(tom.contactInfo.email)    // supertom@gmail.com
    fmt.Println(tom.contactInfo.phone)    // +1234567899
}
```

В данном случае структура person имеет поле contactInfo, которое представляет другую структуру contact .

Можно сократить определение поля следующим образом:

```
package main
import "fmt"

type contact struct{
    email string
    phone string
}

type person struct{
    name string
    age int
    contact
}

func main() {
    var tom = person {
        name: "Tom",
        age: 24,
        contact: contact{
            email: "tom@gmail.com",
            phone: "+1234567899",
        },
    }
    tom.email = "supertom@gmail.com"

    fmt.Println(tom.email)    // supertom@gmail.com
    fmt.Println(tom.phone)    // +1234567899
}
```

Поле contact в структуре person фактически эквивалентно свойству contact contact , то есть свойство называется contact и представляет тип contact. Это позволяет нам сократить путь к полям вложенной структуры. Например, мы можем написать tom.email , а не tom.contact.email . Хотя можно использовать и второй вариант.

Хранения ссылки на структуру того же типа

При этом надо учитывать, что структура не может иметь поле, которое представляет тип этой же структуры. Например:

```
type node struct{
    value int
    next node
}
```

Подобное определение будет неправильным. Вместо этого поле должно представлять указатель на структуру:

```
package main
import "fmt"

type node struct{
    value int
    next *node
}

// рекурсивный вывод списка
func printNodeValue(n *node){
    fmt.Println(n.value)
    if n.next != nil{
        printNodeValue(n.next)
    }
}

func main() {
    first := node{value: 4}
    second := node{value: 5}
```

```

    third := node{value: 6}

    first.next = &second
    second.next = &third

    var current *node = &first
    for current != nil{
        fmt.Println(current.value)
        current = current.next
    }
}

```

Здесь определена структура `node`, которая представляет типичный узел односвязного списка. Она хранит значение в поле `value` и ссылку на следующий узел через указатель `next`.

В функции `main` создаются три связанных структуры, и с помощью цикла `for` и вспомогательного указателя `current` выводятся их значения.

```

4
5
6

```

Методы

Последнее обновление: 26.12.2017

Метод представляет функцию, связанную с определенным типом. Методы определяются также как и обычные функции за тем исключением, что в определении метода также необходимо указать получателя или `receiver`. Получатель - это параметр того типа, к которому прикрепляется метод:

```

func (имя_параметра тип_получателя) имя_метода (параметры) (типы_возвращаемых_результатов){
    тело_метода
}

```

Допустим, у нас будет определен именованный тип, представляющий срез из строк:

```
type library []string
```

Для вывода всех элементов из среза мы можем определить следующий метод:

```

func (l library) print(){
    for _, val := range l{
        fmt.Println(val)
    }
}

```

Та часть, которая расположена между ключевым словом `func` и именем метода и представляет определение получателя, для которого будет определен этот метод: `(l library)`. Используя параметр получателя (в данном случае `l`), мы можем обращаться к получателю. Например, в нашем случае получатель представляет срез - набор объектов. С помощью цикла `for` можно пройтись по этому срезу и вывести все его элементы на консоль.

Поскольку `print` представляет именно метод, который определен для типа `library`, а не обычную функцию, то мы можем вызвать этот метод у любого объекта типа `library`:

```

var lib library = library{ "Book1", "Book2", "Book3" }
lib.print()

```

`lib` является объектом типа `library`, поэтому для него мы можем вызвать метод `print`. В данном случае объект `lib` - это и будет то значение, которое будет передаваться в функцию `print` через параметр `(l library)`.

Методы структур

Подобным образом мы можем определять методы и для структур:

```

package main

import "fmt"

type person struct{
    name string
    age int
}

func (p person) print(){
    fmt.Println("Имя:", p.name)
    fmt.Println("Возраст:", p.age)
}

func (p person) eat(meal string){
    fmt.Println(p.name, "ест", meal)
}

func main() {

    var tom = person { name: "Tom", age: 24 }
    tom.print()
    tom.eat("борщ с капустой, но не красный")
}

```

Консольный вывод данной программы:

```

Имя: Tom
Возраст: 24
Tom ест борщ с капустой, но не красный

```

В данном случае для структуры `person` определены две функции: `print` и `eat`. Функция `print` выводит информацию о текущем объекте `person`. А функция `eat` имитирует употребление пищи. Каждая из этих функций определяет объект и тип структуры, к которой функция относится:

```
func (p person) имя_функции
```

С помощью объекта `p` мы можем обращаться к свойствам структуры `person`. В остальном это обычные функции, которые могут принимать параметры и возвращать результат.

Для обращения к функциям структуры указывается переменная структуры и через точку идет вызов функции.

```
tom.print()
tom.eat("борщ")
```

В данном случае `tom` - это и будет объект `p person` в определении функции.

Методы указателей

Последнее обновление: 26.12.2017

При вызове метода, объект структуры, для которого определен метод, передается в него по значению. Что это значит? Рассмотрим следующий пример:

```
package main
import "fmt"

type person struct{
    name string
    age int
}
func (p person) updateAge(newAge int){
    p.age = newAge
}

func main() {
    var tom = person { name: "Tom", age: 24 }
    fmt.Println("before", tom.age)
    tom.updateAge(33)
    fmt.Println("after", tom.age)
}
```

Для структуры `person` определен метод `updateAge`, который принимает параметр `newAge` и изменяет значение поля `age` у структуры. То есть при вызове этого метода мы ожидаем, что возраст человека изменится. Однако консольный вывод нам показывает, что значение поля `age` не изменяется:

```
before 24
after 24
```

Потому что при вызове `tom.updateAge(33)` метод `updateAge` получает копию структуры `tom`. То есть структура `tom` копируется в другой участок памяти, и далее метод `updateAge` работает с копией, никак не затрагивая оригинальную структуру `tom`.

Однако такое поведение может быть нежелательным. Что если мы все-таки хотим таким образом изменять состояние структуры? В этом случае необходимо использовать указатели на структуры:

```
package main
import "fmt"

type person struct{
    name string
    age int
}
func (p *person) updateAge(newAge int){
    (*p).age = newAge
}

func main() {
    var tom = person { name: "Tom", age: 24 }
    var tomPointer *person = &tom
    fmt.Println("before", tom.age)
    tomPointer.updateAge(33)
    fmt.Println("after", tom.age)
}
```

Теперь метод `updateAge` принимает указатель на структуру `person`: `p *person`, то есть фактически адрес структуры в памяти. С помощью операции разыменования получаем значение по этому адресу в памяти и меняем поле `age`:

```
(*p).age = newAge
```

В функции `main` определяем указатель на структуру `person` и передаем ему адрес структуры `tom`:

```
var tomPointer *person = &tom
```

Затем вызываем метод `updateAge`:

```
tomPointer.updateAge(33)
```

Таким образом, метод `updateAge` получит адрес, который хранится в `tomPointer` и по этому адресу обратиться к структуре `tom`, изменив значение ее свойства `age`.

```
before 24
after 33
```

Стоит отметить, что несмотря на то, что метод `updateAge` определен для указателя на структуру `person`, мы по-прежнему можем вызывать этот метод и для объекта `person`:

```
var tom = person { name: "Tom", age: 24 }
fmt.Println("before", tom.age)      // before 24
tom.updateAge(33)
fmt.Println("after", tom.age)      // after 33
```

Пакеты и модули

Пакеты и их импорт

Последнее обновление: 24.12.2017

Весь код в языке Go организуется в пакеты. Пакеты представляют удобную организацию разделения кода на отдельные части или модули. Модульность позволяет определять один раз пакет с нужной функциональностью и потом использовать его многократно в различных программах.

Код пакета располагается в одном или нескольких файлах с расширением `go`. Для определения пакета применяется ключевое слово `package`. Например:

```
package main
import "fmt"

func main() {

    fmt.Println("Hello Go")
}
```

В данном случае пакет называется `main`. Определение пакета должно идти в начале файла.

Есть два типа пакетов: исполняемые (`executable`) и библиотеки (`reusable`). Для создания исполняемых файлов пакет должен иметь имя `main`. Все остальные пакеты не являются исполняемыми. При этом пакет `main` должен содержать функцию `main`, которая является входной точкой в приложение.

Импорт пакетов

Если уже есть готовые пакеты с нужной нам функциональностью, которую мы хотим использовать, то для их использования мы можем их импортировать в программу с помощью оператора `import`. Например, в примере выше задействуется функциональность вывода сообщения на консоль с помощью функции `Println`, которая определена в пакете `fmt`. Соответственно чтобы использовать эту функцию, необходимо импортировать пакет `fmt`:

```
import "fmt"
```

Нередко программы подключают сразу несколько внешних пакетов. В этом случае можно последовательно импортировать каждый пакет:

```
package main
import "fmt"
import "math"

func main() {

    fmt.Println(math.Sqrt(16))    // 4
}
```

В данном случае подключается встроенный пакет `math`, который содержит функцию `Sqrt()`, возвращающую квадратный корень числа.

Либо чтобы сократить определение импорта пакетов можно заключить все пакеты в скобки:

```
package main
import (
    "fmt"
    "math"
)

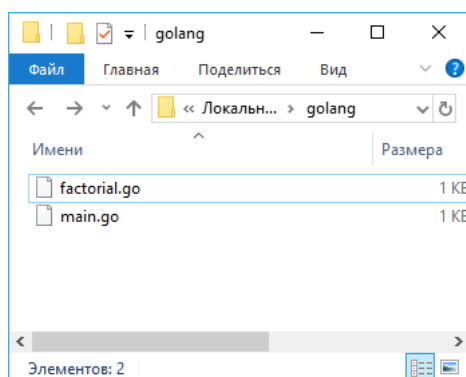
func main() {

    fmt.Println(math.Sqrt(16))
}
```

Подобным образом мы можем импортировать как встроенные пакеты, так и свои собственные. Полный список встроенных пакетов в Go можно найти по адресу <https://golang.org/pkg/>.

Пакет из нескольких файлов

Один пакет может состоять из нескольких файлов. Например, определим в папке два файла:



В файле `factorial.go` определим функцию для подсчета факториала:

```
package main

func factorial(n int) int {

    var result = 1
    for i:=1; i <= n; i++){
        result *= i
    }
    return result
}
```

Данный файл принадлежит пакету `main`.

В файле main.go используем функцию для вычисления факториала:

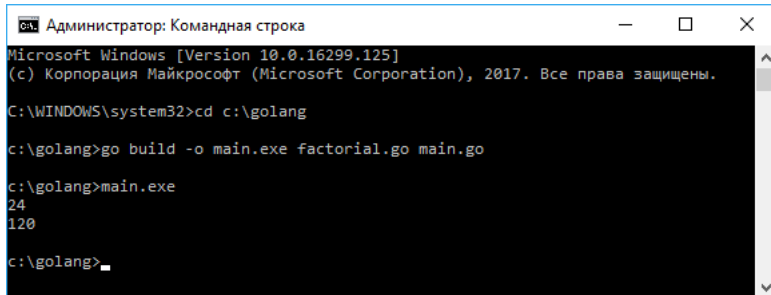
```
package main
import "fmt"

func main() {
    fmt.Println(factorial(4))
    fmt.Println(factorial(5))
}
```

Данный файл также принадлежит пакету main. Файлов может и быть и больше. Теперь скомпилируем из этих файлов программу. Для этого перейдем в консоли к папке проекта и выполним команду:

```
go build -o main.exe factorial.go main.go
```

Флаг -o указывает, как будет называться выходной файл - в данном случае main.exe. Затем идут все компилируемые файлы. После выполнения этой команды будет создан файл main.exe, который мы сможем запустить в консоли:



```
Администратор: Командная строка
Microsoft Windows [Version 10.0.16299.125]
(c) Корпорация Майкрософт (Microsoft Corporation), 2017. Все права защищены.

C:\WINDOWS\system32>cd c:\golang

c:\golang>go build -o main.exe factorial.go main.go

c:\golang>main.exe
24
120

c:\golang>
```

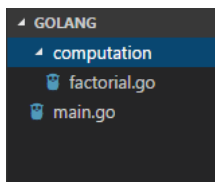
Файлы в разных пакетах

Теперь рассмотрим другую ситуацию, когда файлы нашей программы разделены по разным пакетам. Определим в папке проекта каталог computation. Затем в каталог computation добавим следующий файл factorial.go:

```
package computation

func Factorial(n int) int {
    var result = 1
    for i:=1; i <= n; i++){
        result *= i
    }
    return result
}
```

Код файла factorial.go принадлежит пакету computation. Важно отметить, что название функции начинается с заглавной буквы. Благодаря этому данная функция будет видна в других пакетах.



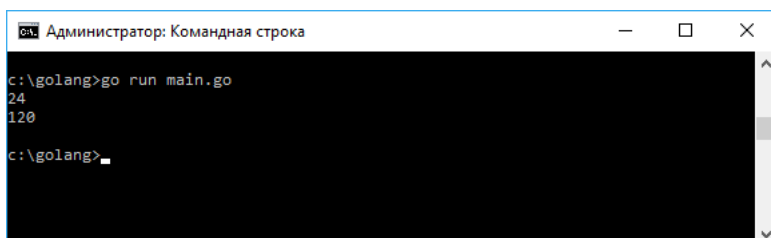
И чтобы использовать функцию factorial, надо импортировать этот пакет в файле main.go:

```
package main
import (
    "fmt"
    "./computation"
)

func main() {
    fmt.Println(computation.Factorial(4))
    fmt.Println(computation.Factorial(5))
}
```

Путь "./computation" указывает, что пакет находится в папке computation.

Компиляция и выполнение программы осуществляется также как и ранее без необходимости указания всех файлов из других пакетов:



```
Администратор: Командная строка

c:\golang>go run main.go
24
120

c:\golang>
```

Введение в модули

Последнее обновление: 03.12.2021

Модули представляют набор пакетов Go, которые имеют встроенное версионирование и которые можем опубликовать для использования в других проектах.

Язык Go обладает богатой функциональностью, одной этой встроенной функциональности может быть недостаточно для построения приложения. Однако кроме того, Go имеет большую экосистему разработчиков, которые разрабатывают и публикуют различные модули. Эти модули могут предоставлять функциональность, которая отсутствует во встроенной библиотеке пакетов Go, но которая может быть нам необходима. И Go также позволяет использовать эти модули. Рассмотрим как создавать свои модули и подключать и использовать внешние модули в своем приложении. Но, начиная с версии Go 1.16, для использования внешних модулей необходимо определить свой модуль.

Создание модуля

Для создания модуля применяется команда `go mod init`, которой передается имя модуля.

Сначала определим папку, где будет располагаться наш модуль. Пусть, к примеру это будет папка `C:\golang\`. Далее перейдем в терминале/командной строке к этой папке. Допустим, наш модуль будет называться `helloapp`. Для его создания выполним команду:

```
go mod init helloapp
```

```

Администратор: Командная строка
Microsoft Windows [Version 10.0.19043.1348]
(c) Корпорация Майкрософт (Microsoft Corporation). Все права защищены.

C:\WINDOWS\system32>cd c:\golang

c:\golang>go mod init helloapp
go: creating new go.mod: module helloapp

c:\golang>
  
```

После выполнения этой команды в папке появится файл `go.mod`. По умолчанию он будет иметь следующее содержимое:

```
module helloapp
```

```
go 1.17
```

Первая строка с директивой `module` определяет путь модуля - `"helloapp"`. Вторая строка определяет используемую для модуля версию `go` - в данном случае `1.17`.

Кроме пути модуля и версии `go` этот файл позволяет управлять зависимостями - внешними модулями, которые подключаются в приложение.

Загрузка внешнего модуля

Теперь подключим в наш модуль `helloapp` внешний модуль. Для примера возьмем модуль `"rsc.io/quote"`. Это модуль, определенный специально для целей тестирования. Он содержит набор функций, который возвращают некоторый текст.

Для его загрузки модуля `"rsc.io/quote"` выполним команду:

```
go get rsc.io/quote
```

При выполнении этой команды Go загрузит необходимые зависимости:

```

Администратор: Командная строка
Microsoft Windows [Version 10.0.19043.1348]
(c) Корпорация Майкрософт (Microsoft Corporation). Все права защищены.

C:\WINDOWS\system32>cd c:\golang

c:\golang>go mod init helloapp
go: creating new go.mod: module helloapp

c:\golang>go get rsc.io/quote
go: downloading rsc.io/quote v1.5.2
go: downloading rsc.io/sampler v1.3.0
go: downloading golang.org/x/text v0.0.0-20170915032832-14c0d48ead0c
go get: added golang.org/x/text v0.0.0-20170915032832-14c0d48ead0c
go get: added rsc.io/quote v1.5.2
go get: added rsc.io/sampler v1.3.0

c:\golang>
  
```

Все загружаемые пакеты хранятся по пути `$GOPATH\pkg\mod`. Обычно переменная среды `GOPATH` указывает на папку `[Папка текущего пользователя]\go`. Например, в моем случае пакеты будут загружаться по адресу `C:\Users\Eugene\go\pkg\mod`

И если мы заново откроем файл `go.mod`, то мы увидим, что его содержимое изменилось:

```
module helloapp
```

```
go 1.17
```

```

require (
    golang.org/x/text v0.0.0-20170915032832-14c0d48ead0c // indirect
    rsc.io/quote v1.5.2 // indirect
    rsc.io/sampler v1.3.0 // indirect
)
  
```

В него добавилась директива `require()`, которая содержит определения подключаемых зависимостей. В данном случае это все те зависимости, которые необходимы для работы с пакетом `"rsc.io/quote"`.

Кроме того, мы также можем заметить, что в папке проекта `c:\golang` появился еще один файл - `go.sum` с содержимым тиа следующего:

```

golang.org/x/text v0.0.0-20170915032832-14c0d48ead0c h1:qg0Y6WgZ0aTkIIMiVjBQcw93ERBE4m30iBm00nkL0i8=
golang.org/x/text v0.0.0-20170915032832-14c0d48ead0c/go.mod h1:NqM8EU0U14njKJ3fqMW+pc6Ldnwhi/IjpwHt7yyuwOQ=
  
```

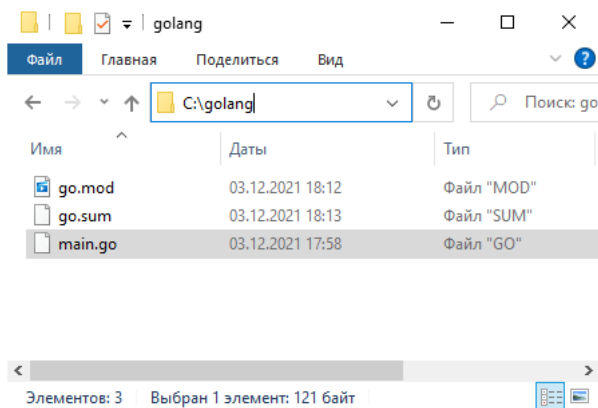


```
rsc.io/quote v1.5.2 h1:w5fcysjrx7yqtD/a0+QwRjYZ0KnaM9Uh2b40tELTs3Y=
rsc.io/quote v1.5.2/go.mod h1:LzX7hefJvL54yjeFDEDHNDjII0t9xZLPXsUe+TKr0=
rsc.io/sampler v1.3.0 h1:7uVklFmeBqHfdjD+gZwtXXI+RODJ2Wc407MPEh/QiW4=
rsc.io/sampler v1.3.0/go.mod h1:T1hPZKmBbMNahiBKfy5HrXp6adAjACjK9JXDNKaTXpA=
```

Этот файл содержит контрольную сумму для подключаемых пакетов.

Подключение внешнего модуля

Пакеты из внешнего модуля подключаются также, как и другие пакеты с помощью директивы `import`. Например, определим в папке `c:\golang` собственно файл кода `main.go`



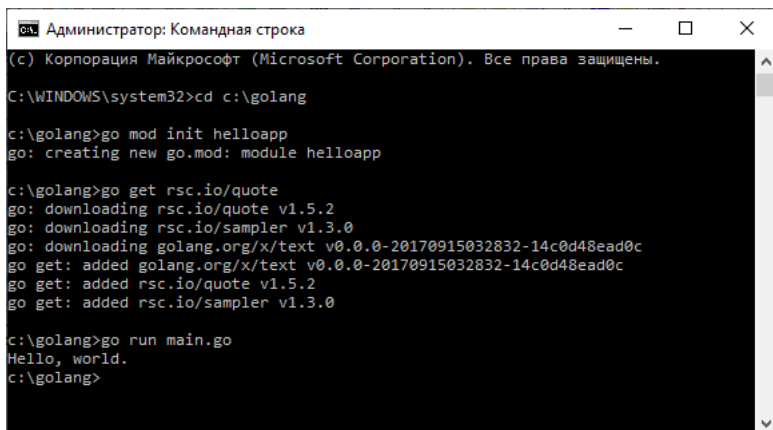
Определим в `main.go` следующее содержимое:

```
package main

import (
    "fmt"
    "rsc.io/quote"
)

func main() {
    message := quote.Hello()
    fmt.Printf(message)
}
```

В данном случае вызывается функция `quote.Hello()` из пакета `"rsc.io/quote"`, которая возвращает некоторое сообщение (точнее строку `"Hello World"`). И в конце запустим программу на выполнение:



Интерфейсы

Введение в интерфейсы

Последнее обновление: 28.12.2017

Интерфейсы представляют абстракцию поведения других типов. Интерфейсы позволяют определять функции, которые не привязаны к конкретной реализации. То есть интерфейсы определяют некоторый функционал, но не реализуют его.

Для определения интерфейса применяется ключевое слово `interface`:

```
type имя_интерфейса interface{
    определения_функций
}
```

Например, простейшее определение интерфейса:

```
type vehicle interface{
    move()
}
```

Данный интерфейс называется `vehicle`. Допустим, данный интерфейс представляет некоторое транспортное средство. Он определяет функцию `move()`, которая не принимает никаких параметров и ничего не возвращает.

При этом важно понимать, что интерфейс - это именно абстракция, а не конкретный тип, как `int`, `string` или структуры. К примеру, мы не можем напрямую создать объект интерфейса:

```
var v vehicle = vehicle{}
```

Интерфейс представляет своего рода контракт, которому должен соответствовать тип данных. Чтобы тип данных соответствовал некоторому интерфейсу, данный тип должен реализовать в виде методов все функции этого интерфейса. Например, определим две структуры:

```
package main

import "fmt"

type Vehicle interface{
    move()
}

// структура "Автомобиль"
type Car struct{ }

// структура "Самолет"
type Aircraft struct{}

func (c Car) move(){
    fmt.Println("Автомобиль едет")
}
func (a Aircraft) move(){
    fmt.Println("Самолет летит")
}

func main() {

    var tesla Vehicle = Car{}
    var boing Vehicle = Aircraft{}
    tesla.move()
    boing.move()
}
```

Здесь определены две структуры: `Car` и `Aircraft`, которые, предположим, представляют автомобиль и самолет соответственно. Для каждой из структур определен метод `move()`, который имитирует перемещение транспортного средства. Этот метод `move` соответствует функции `move` интерфейса `vehicle` по типу параметров и типу возвращаемых значений. Поскольку между методом структур и функций в интерфейсе есть соответствие, то подобные структуры неявно реализуют данный интерфейс.

В Go интерфейс реализуется неявно. Нам не надо специально указывать, что структуры применяют определенный интерфейс, как в некоторых других языках программирования. Для реализации типу данных достаточно реализовать методы, которые определяет интерфейс.

Поскольку структуры `Car` и `Aircraft` реализуют интерфейс `Vehicle`, то мы можем определить переменные данного интерфейса, передав им объекты структур:

```
var tesla Vehicle = Car{}
var boing Vehicle = Aircraft{}
```

Где нам могут помочь интерфейсы? Интерфейсы позволяют определить какую-то обобщенную реализацию без привязки к конкретному типу. Например, рассмотрим следующую ситуацию:

```
package main

import "fmt"

type Car struct{ }
type Aircraft struct{}

func (c Car) move(){
    fmt.Println("Автомобиль едет")
}
func (a Aircraft) move(){
    fmt.Println("Самолет летит")
}

func driveCar(c Car){
    c.move()
}
func driveAircraft(a Aircraft){
    a.move()
}

func main() {

    var tesla Car = Car{}
    var boing Aircraft = Aircraft{}
    driveCar(tesla)
    driveAircraft(boing)
}
```

Допустим, в данном случае определены две структуры `Car` и `Aircraft`, которые представляют автомобиль и самолет. Для каждой из структур определен метод перемещения `move()`, который условно перемещает транспортное средство. И также определены две функции `driveCar()` и `driveAircraft()`, которые принимают соответственно структуры `Car` и `Aircraft` и предназначены для вождения этих транспортных средств.

И отчетливо видно, что обе функции `driveCar` и `driveAircraft` фактически идентичны, они выполняют один и те же действия, только для разных типов. И было бы неплохо, если можно было бы определить одну обобщенную функцию для разных типов. Особенно учитывая, что у нас может быть и больше транспортных средств - велосипед, корабль и т.д. И для вождения каждого транспортного средства придется определять свой метод, что не очень удобно. И как раз в этом случае можно воспользоваться интерфейсами:

```
package main
import "fmt"

type Vehicle interface{
```

```

        move()
    }

    func drive(vehicle Vehicle){
        vehicle.move()
    }

    type Car struct{ }
    type Aircraft struct{}

    func (c Car) move(){
        fmt.Println("Автомобиль едет")
    }
    func (a Aircraft) move(){
        fmt.Println("Самолет летит")
    }

    func main() {

        tesla := Car{}
        boing := Aircraft{}
        drive(tesla)
        drive(boing)
    }

```

Теперь вместо двух функций определена одна общая функция - `drive()` , которая в качестве параметра принимает значение типа `Vehicle`. Поскольку этому интерфейсу соответствуют обе структуры `Car` и `Aircraft`, то мы можем передавать эти структуры в функцию `drive` в качестве аргументов.

Соответствие интерфейсу

Последнее обновление: 28.12.2017

Чтобы тип данных соответствовал интерфейсу, он должен реализовать все методы этого интерфейса. Например:

```

package main

import "fmt"

type Stream interface{
    read() string
    write(string)
    close()
}

func writeToStream(stream Stream, text string){
    stream.write(text)
}
func closeStream(stream Stream){
    stream.close()
}

// структура файл
type File struct{
    text string
}
// структура папка
type Folder struct{}

// реализация методов для типа *File
func (f *File) read() string{
    return f.text
}
func (f *File) write(message string){
    f.text = message
    fmt.Println("Запись в файл строки", message)
}
func (f *File) close(){
    fmt.Println("Файл закрыт")
}
// релизация методов для типа *Folder
func (f *Folder) close(){
    fmt.Println("Папка закрыта")
}

func main() {

    myFile := &File{}
    myFolder := &Folder{}

    writeToStream(myFile, "hello world")
    closeStream(myFile)
    //closeStream(myFolder)          // Ошибка: тип *Folder не реализует интерфейс Stream
    myFolder.close()                 // Так можно
}

```

Здесь определен интерфейс `Stream`, который условно представляет некоторый поток данных и определяет три метода: `close`, `read` и `write`. И также есть две структуры `File` и `Folder`, которые представляют соответственно файл и папку. Для типа `*File` реализованы все три метода интерфейса `Stream`. А для типа `*Folder` реализован только один метод интерфейса `Stream`. Поэтому тип `*File` реализует интерфейс `Stream` и соответствует этому интерфейсу, а тип `*Folder` не соответствует интерфейсу `Stream`. Поэтому везде, где требуется объект `Stream` мы можем использовать объект типа `*File` , но никак не `*Folder`.

Например, в функцию `closeStream()` , которая условно закрывает поток, в качестве параметра передается объект `Stream`. При вызове в эту функцию можно передать объект типа `*File`, который соответствует интерфейсу `Stream`:

```
closeStream(myFile)
```

А вот объект `*Folder` передать нельзя:

```
closeStream(myFolder)          // Ошибка: тип *Folder не реализует интерфейс Stream
```

Но мы по-прежнему можем вызывать у объекта *Folder метод close, но он будет рассматриваться как собственный метод, который не имеет никакого отношения к интерфейсу Stream.

Еще важно отметить, что в данном случае методы реализованы именно для типа *File, то есть указателя на объект File, а не для типа File. Это два разных типа. Поэтому тип File по-прежнему НЕ соответствует интерфейсу Stream. И мы, к примеру, не можем написать следующее:

```
myFile2 := File{}
closeStream(myFile2)           // ! Ошибка: тип File не соответствует интерфейсу Stream
```

Реализация нескольких интерфейсов

При этом тип данных необязательно должен реализовать только методы интерфейса, для типа данных можно определить его собственные методы или также реализовать методы других интерфейсов. Например:

```
package main

import "fmt"

type Reader interface{
    read()
}

type Writer interface{
    write(string)
}

func writeToStream(writer Writer, text string){
    writer.write(text)
}
func readFromStream(reader Reader){
    reader.read()
}

type File struct{
    text string
}

func (f *File) read(){
    fmt.Println(f.text)
}
func (f *File) write(message string){
    f.text = message
    fmt.Println("Запись в файл строки", message)
}

func main() {

    myFile := &File{}
    writeToStream(myFile, "hello world")
    readFromStream(myFile)
}
```

В данном случае для типа *File реализованы методы обоих интерфейсов - Reader и Writer. Соответственно мы можем использовать объекты типа *File в качестве объектов Reader и Writer.

Вложенные интерфейсы

Одни интерфейсы могут содержать другие интерфейсы.

```
type Reader interface{
    read()
}

type Writer interface{
    write(string)
}

type ReaderWriter interface{
    Reader
    Writer
}
```

В этом случае для соответствия подобному интерфейсу типы данных должны реализовать все его вложенные интерфейсы. Например:

```
package main

import "fmt"

type Reader interface{
    read()
}

type Writer interface{
    write(string)
}

type ReaderWriter interface{
    Reader
    Writer
}

func writeToStream(writer ReaderWriter, text string){
    writer.write(text)
}
func readFromStream(reader ReaderWriter){
    reader.read()
}

type File struct{
    text string
}
```

```

}

func (f *File) read(){
    fmt.Println(f.text)
}
func (f *File) write(message string){
    f.text = message
    fmt.Println("Запись в файл строки", message)
}

func main() {

    myFile := &File{}
    writeToStream(myFile, "hello world")
    readFromStream(myFile)
    writeToStream(myFile, "lolly bomb")
    readFromStream(myFile)
}

```

В данном случае определено три интерфейса. Причем интерфейс ReaderWriter содержит интерфейсы Reader и Writer. Чтобы структура File соответствовала интерфейсу ReaderWriter, она должна реализовать методы read и write, то есть методы обоих вложенных интерфейсов, что в принципе здесь и сделано.

Полиморфизм

Последнее обновление: 06.01.2018

Полиморфизм представляет способность принимать многообразные формы. В частности, в предыдущих статьях было рассмотрено использование интерфейсов, которым могут соответствовать различные структуры. Например:

```

package main
import "fmt"

type Vehicle interface{
    move()
}

type Car struct{ model string}
type Aircraft struct{ model string}

func (c Car) move(){
    fmt.Println(c.model, "едет")
}
func (a Aircraft) move(){
    fmt.Println(a.model, "летит")
}

func main() {

    tesla := Car{"Tesla"}
    volvo := Car{"Volvo"}
    boeing := Aircraft{"Boeing"}

    vehicles := [...]Vehicle{tesla, volvo, boeing}
    for _, vehicle := range vehicles{
        vehicle.move()
    }
}

```

В данном случае определен массив vehicles, который содержит набор структур, которые соответствуют интерфейсу Vehicle, то есть объекты Car и Aircraft. То есть объект Vehicle может принимать различные формы: или структуру Car, или структуру Aircraft. При переборе массива для каждого объекта вызывается метод move. И в зависимости от реального типа структуры динамически определяется, какая именно реализация метода move для какой структуры должна вызываться.

Консольный вывод программы:

```

Tesla едет
Volvo едет
Boeing летит

```

Параллельное программирование. Горутины

Горутины

Последнее обновление: 12.01.2018

Горутины (goroutines) представляют параллельные операции, которые могут выполняться независимо от функции, в которой они запущены. Главная особенность горутинов состоит в том, что они могут выполняться параллельно. То есть на многоядерных архитектурах есть возможность выполнять отдельные горутины на отдельных ядрах процессора, тем самым горутины будут выполняться параллельно, и программа завершится быстрее.

Каждая горутина, как правило, представляет вызов функции, и последовательно выполняет все свои инструкции. Когда мы запускаем программу на Go, мы уже работаем как минимум с одной горутиной, которая представлена функцией main. Эта функция последовательно выполняет все инструкции, которые определены внутри нее.

Для определения горутинов применяется оператор go, который ставится перед вызовом функции:

```
go вызов_функции
```

Например, определим несколько горутинов, вычисляющих факториал числа:

```

package main
import "fmt"

```

```
func main() {
```

```

        for i := 1; i < 7; i++){
            go factorial(i)
        }
        fmt.Println("The End")
    }
}

func factorial(n int){
    if(n < 1){
        fmt.Println("Invalid input number")
        return
    }
    result := 1
    for i := 1; i <= n; i++){
        result *= i
    }
    fmt.Println(n, "-", result)
}

```

В цикле последовательно запускаются шесть горутин с помощью вызова `go factorial(i)`. То есть фактически это обычный вызов функции с оператором `go`.

Однако вместо шести факториалов на консоли при вызове программы мы можем увидеть только строку "The End":

The End

"Можем увидеть" означает, что поведение программы в данном случае недетерминировано. Например, вывод может быть и таким:

```

2 - 2
1 - 1
4 - 24
The End
5 - 120

```

Почему так происходит? После вызова `go factorial(i)` функция `main` запускает горутины, которая начинает выполняться в своем контексте, независимо от функции `main`. То есть фактически `main` и `factorial` выполняются параллельно. Однако главной горутинной является вызов функции `main`. И если завершается выполнение этой функции, то завершается и выполнение всей программы. Поскольку вызовы функции `factorial` представляют горутин, то функция `main` не ждет их завершения и после их запуска продолжает свое выполнение. Какие-то горутин могут завершиться раньше функции `main`, и соответственно мы сможем увидеть на консоли их результат. Но может сложиться ситуация, что функция `main` выполнится раньше вызовов функции `factorial`. Поэтому мы можем не увидеть на консоли факториалы чисел.

Чтобы все таки увидеть результат выполнения горутин, поставим в конце функции `main` вызов функции `fmt.Scanln()`, которая ожидает ввода пользователя с консоли:

```

package main
import "fmt"

func main() {
    for i := 1; i < 7; i++){
        go factorial(i)
    }
    fmt.Scanln()           // ждем ввода пользователя
    fmt.Println("The End")
}

func factorial(n int){
    if(n < 1){
        fmt.Println("Unvalid input number")
        return
    }
    result := 1
    for i := 1; i <= n; i++){
        result *= i
    }
    fmt.Println(n, "-", result)
}

```

Теперь мы сможем увидеть результаты всех вызовов функции `factorial`:

```

1 - 1
3 - 6
5 - 120
4 - 24
2 - 2
6 - 720

The End

```

Стоит отметить, что так как каждая горутина запускается в своем собственном контексте и выполняется независимо и параллельно по сравнению с другими горутин, то в данном случае нельзя четко детерминировать, какая из горутин завершится раньше. Например, горутина `go factorial(2)` запускается до `go factorial(5)`, однако может завершиться после.

Горутин также могут представлять вызовы анонимных функций:

```

package main
import "fmt"

func main() {
    for i := 1; i < 7; i++){
        go func(n int){
            result := 1
            for j := 1; j <= n; j++){
                result *= j
            }
            fmt.Println(n, "-", result)
        }(i)
    }
}

```

```

    fmt.Scanln()
    fmt.Println("The End")
}

```

Каналы

Последнее обновление: 23.02.2024

Каналы (channels) представляют инструменты коммуникации между горутинами. Для определения канала применяется ключевое слово `chan` :

```
chan тип_элемента
```

После слова `chan` указывается тип данных, которые будут передаваться с помощью канала. Например:

```
var intCh chan int
```

Здесь переменная `intCh` представляет канал, который передает данные типа `int`.

Для передачи данных в канал или, наоборот, из канала применяется операция `<-` (направленная влево стрелка). Например, передача данных в канал:

```
intCh <- 5
```

В данном случае в канал посылается число 5. Получение данных из канала в переменную:

```
val := <- intCh
```

Если ранее в канал было отправлено число 5, то при выполнении операции `<- intCh` мы можем получить это число в переменную `val`.

Стоит учитывать, что мы можем отправить в канал и получить из канала данные только того типа, который представляет канал. Так, в примере с каналом `intCh` это данные типа `int`.

Как правило, отправителем данных является одна горутина, а получателем - другая горутина.

При простом определении переменной канала она имеет значение `nil`, то есть по сути канал неинициализирован. Для инициализации применяется функция `make()`. В зависимости от определения емкости канала он может быть буферизованным или небуферизованным.

Небуферизированные каналы

Для создания небуферизованного канала вызывается функция `make()` без указания емкости канала:

```
var intCh chan int = make(chan int) // канал для данных типа int
strCh := make(chan string) // канал для данных типа string

```

Если канал пустой, то горутина-получатель блокируется, пока в канале не окажутся данные. Когда горутина-отправитель посылает данные, горутина-получатель получает эти данные и возобновляет работу.

Горутина-отправитель может отправлять данные только в пустой канал. Горутина-отправитель блокируется до тех пор, пока данные из канала не будут получены. Например:

```

package main
import "fmt"

func main() {
    intCh := make(chan int)

    go func(){
        fmt.Println("Go routine starts")
        intCh <- 5 // блокировка, пока данные не будут получены функцией main
    }()
    fmt.Println(<-intCh) // получение данных из канала
    fmt.Println("The End")
}

```

Через небуферизированный канал `intCh` горутина, представленная анонимной функцией, передает число 5:

```
intCh <- 5
```

А функция `main` получает это число:

```
fmt.Println(<-intCh)
```

Общий ход выполнения программы выглядит следующим образом:

1. Запускается функция `main`. Она создает канал `intCh` и запускает горутина в виде анонимной функции.
2. Функция `main` продолжает выполняться и блокируется на строке `fmt.Println(<-intCh)`, пока не будут получены данные.
3. Параллельно выполняется запущенная горутина в виде анонимной функции. В конце своего выполнения она отправляет данные через канал: `intCh <- 5`. Горутина блокируется, пока функция `main` не получит данные.
4. Функция `main` получает отправленные данные, деблокируется и продолжает свою работу.

В данном случае горутина определена в виде анонимной функции и поэтому она имеет доступ к окружению, в том числе к переменной `intCh`. Если же мы работаем с обычными функциями, то объекты каналов надо передавать через параметры:

```

package main
import "fmt"

func main() {
    intCh := make(chan int)

    go factorial(5, intCh) // вызов горутин
    fmt.Println(<-intCh) // получение данных из канала
}

```

```

        fmt.Println("The End")
    }

    func factorial(n int, ch chan int){

        result := 1
        for i := 1; i <= n; i++){
            result *= i
        }
        fmt.Println(n, "-", result)

        ch <- result                // отправка данных в канал
    }

```

Обратите внимание, как определяется параметр, который представляет канал данных типа `int`: `ch chan int`. Консольный вывод данной программы:

```

5 - 120
120
The End

```

Таким образом, при использовании канала вызывающий поток - функция `main` ожидает завершения выполнения горутин.

Стоит отметить, что одновременно одна горутин должна отправлять данные, а другая - получать. Например, если мы определим отправку и получение данных через канал в самой функции `main`, то мы столкнемся с взаимоблокировкой:

```

package main
import "fmt"

func main() {

    intCh := make(chan int)
    intCh <- 10                // функция main блокируется
    fmt.Println(<-intCh)
}

```

От этой ситуации следует отличать ситуацию, когда две горутин попеременно обмениваются данными, при этом одновременно опять же одна горутин выступает отправителем, а другая - получателем. Например:

```

package main
import "fmt"

func main() {

    intCh := make(chan int)

    go square(intCh)           // square ожидает получения через канал
    intCh <- 4                  // отправляем в канал число
    fmt.Println("result := ", <-intCh) // получаем из канала результат
    fmt.Println("The End")
}
// функция возведения в квадрат
func square(ch chan int){

    num := <-ch                // получаем из канала число
    fmt.Println("num := ", num)
    ch <- num * num             // обратно отправляем квадрат числа
}

```

Здесь определена функция `square`, которая получает через канал число, возводит его в квадрат и возвращает обратно в канал. Функция `main` запускает функцию `square` в виде горутин, отправляет в канал некоторое число и ожидает получить в ответ из канала квадрат этого числа.

В итоге сначала запускается горутин `square`:

```
go square(intCh)
```

Горутин `square` блокируется на строке

```
num := <-ch
```

В этот момент получателем является горутин `square`, а отправителем функция `main`. И функция `main` отправляет данные в поток:

```
intCh <- 4
```

После этого горутин `square` получает из канала число, и роли меняются: теперь отправителем становится горутин `square`, а получателем - функция `main`, которая в ожидании данных блокируется на строке

```
fmt.Println("result := ", <-intCh)
```

Функция `square` обрабатывает полученное число и отправляет квадрат числа в поток:

```
ch <- num * num
```

Функция `main` получает из канала квадрат числа и завершает свою работу. Консольный вывод программы:

```

num := 4
result := 16
The End

```

Буферизированные каналы

Буферизированные каналы также создаются с помощью функции `make()`, только в качестве второго аргумента в функцию передается емкость канала. Если канал пуст, то получатель ждет, пока в канале появится хотя бы один элемент.

При отправке данных горутин-отправитель ожидает, пока в канале не освободится место для еще одного элемента и отправляет элемент, только тогда, когда в канале освобождается для него место.


```
package main
import "fmt"

func main() {

    intCh := make(chan int, 3)
    intCh <- 10
    intCh <- 3
    intCh <- 24
    fmt.Println(<-intCh)           // 10
    fmt.Println(<-intCh)           // 3
    fmt.Println(<-intCh)           //24
}
```

В данном случае отправителем и получателем данных является функция main. В ней создается канал из трех элементов, и последовательно отправляются три значения типа int.

В то же время в данном случае должно быть соответствие между количеством отправляемых и получаемых данных. Если в функции main будет одновременно отправлено значений больше, чем вмещает канал, то функция заблокируется:

```
package main
import "fmt"

func main() {

    intCh := make(chan int, 3)
    intCh <- 10
    intCh <- 3
    intCh <- 24
    intCh <- 15    // блокировка - функция main ждет, когда освободится место в канале

    fmt.Println(<-intCh)
    fmt.Println("The End")
}
```

С помощью встроенных функций cap() и len() можно получить соответственно емкость и количество элементов в канале:

```
package main
import "fmt"

func main() {

    intCh := make(chan int, 3)
    intCh <- 10

    fmt.Println(cap(intCh))        // 3
    fmt.Println(len(intCh))        // 1

    fmt.Println(<-intCh)
}
```

Однонаправленные каналы

В Go можно определить канал, как доступный только для отправки данных или только для получения данных.

Определение канала только для отправки данных:

```
var inCh chan<- int
```

Определение канала только для получения данных:

```
var outCh <-chan int
```

Например:

```
package main
import "fmt"

func main() {

    intCh := make(chan int, 2)
    go factorial(5, intCh)
    fmt.Println(<-intCh)
    fmt.Println("The End")
}

func factorial(n int, ch chan<- int){

    result := 1
    for i := 1; i <= n; i++){
        result *= i
    }
    ch <- result
}
```

Здесь второй параметр функции factorial определен как канал, доступный только для отправки данных: ch chan<- int . Соответственно внутри функции factorial мы можем только отправлять данные в канал, но не получать их.

Возвращение канала

Канал может быть возвращаемым значением функции. Однако следует внимательно подходить к операциям записи и чтения в возвращаемом канале. Например:

```
package main
import "fmt"

func main() {
    fmt.Println("Start")
    // создание канала и получение из него данных
}
```

```

    fmt.Println(<-createChan(5))    // блокировка
    fmt.Println("End")
}
func createChan(n int) chan int{
    ch := make(chan int)    // создаем канал
    ch <- n    // отправляем данные в канал
    return ch    // возвращаем канал
}

```

Функция createChan возвращает канал. Однако при выполнении операции `ch <- n` мы столкнемся с блокировкой, поскольку происходит ожидание получения данных из канала. Поэтому следующее выражение `return ch` не будет выполняться.

И если все таки необходимо определить функцию, которая возвращает канал, то все операции чтения-записи в канал следует вынести в отдельную горутину:

```

package main
import "fmt"

func main() {
    fmt.Println("Start")
    // создание канала и получение из него данных
    fmt.Println(<-createChan(5))    // 5
    fmt.Println("End")
}
func createChan(n int) chan int{
    ch := make(chan int)    // создаем канал
    go func(){
        ch <- n    // отправляем данные в канал
    }()    // запускаем горутину
    return ch    // возвращаем канал
}

```

Заккрытие канала

Последнее обновление: 12.01.2018

После инициализации канал готов передавать данные. Он находится в открытом состоянии, и мы можем с ним взаимодействовать, пока не будет закрыт с помощью встроенной функции `close()`:

```

package main
import "fmt"

func main() {

    intCh := make(chan int, 3)
    intCh <- 10
    intCh <- 3
    close(intCh)
    // intCh <- 24    // ошибка - канал уже закрыт
    fmt.Println(<-intCh)    // 10
    fmt.Println(<-intCh)    // 3
    fmt.Println(<-intCh)    // 0
}

```

После закрытия канала мы не сможем послать в него новые данные. Если мы попробуем это сделать, то получим ошибку. Однако мы можем получить ранее добавленные данные. Но при попытке получить данные из канала, которых нет, мы получим значение по умолчанию. Например, в примере выше в канал добавляются два значения. При попытке получить третье значение, которого нет в канале, мы получим значение по умолчанию - число 0.

Чтобы не столкнуться с проблемой, когда канал уже закрыт, мы можем проверять состояние канала. В частности, из канала мы можем получить два значения:

```
val, opened := <-intCh
```

Первое значение - `val` - это собственно данные из канала, а `opened` представляет логическое значение, которое равно `true`, если канал открыт и мы можем успешно считать из него данные. Например, мы можем проверять с помощью условной конструкции состояние канала:

```

package main
import "fmt"

func main() {

    intCh := make(chan int, 3)
    intCh <- 10
    intCh <- 3
    close(intCh)    // канал закрыт

    for i := 0; i < cap(intCh); i++ {
        if val, opened := <-intCh; opened {
            fmt.Println(val)
        } else {
            fmt.Println("Channel closed!")
        }
    }
}

```

Консольный вывод программы:

```

10
3
Channel closed!

```

Синхронизация

Последнее обновление: 12.01.2018

Использование каналов открывает нам возможности по синхронизации между различными горутинами. Например, одна горутина производит некоторое действие, результат которой используется в другой горутине. В этом плане мы можем использовать каналы для синхронизации. Например, одна горутина

вычисляет факториал числа, а результат выводится в другой горутине:

```
package main
import "fmt"

func main() {
    intCh := make(chan int)

    go factorial(5, intCh)

    fmt.Println(<-intCh)
}

func factorial(n int, ch chan int){
    result := 1
    for i:=1; i <= n; i++){
        result *= i
    }
    ch <- result
}
```

Канал не обязательно должен нести данные, которые представляют некоторый результат, от которого зависит дальнейшее выполнение горутин. Иногда это может быть холостой объект, например, пустая структура, которая необходима только для синхронизации горутин:

```
package main
import "fmt"

func main() {
    results := make(map[int]int)
    structCh := make(chan struct{})

    go factorial(5, structCh, results)

    <-structCh           // ожидаем закрытия канала structCh

    for i, v := range results{
        fmt.Println(i, " - ", v)
    }
}

func factorial(n int, ch chan struct{}, results map[int]int){
    defer close(ch)       // закрываем канал после завершения горутин
    result := 1
    for i:=1; i <= n; i++){
        result *= i
        results[i] = result
    }
}
```

В данном случае функция factorial по-прежнему вычисляет факториал, но помещает все факториалы чисел от 1 до n в отображение results, где ключи представляют числа, а значения - факториалы чисел.

Канал, через который горутин взаимодействуют, представляет тип `chan struct{}`. Причем функция factorial не отправляет конкретные данные в канал, а просто закрывает его после выполнения всех своих инструкций с помощью вызова `defer close(ch)`. После закрытия канала в функции main получает соответствующий сигнал в строке `<-structCh`, и после этого функция main продолжает свою работу.

Передача потоков данных

Последнее обновление: 12.01.2018

Нередко одна горутин транслирует другой горутине через канал не одиночные значения, а некоторый поток данных. В этом случае общий алгоритм состоит в том, что горутин-отправитель в течение некоторого периода отправляет данные. Когда данные для отправки закончились, работа сделала, отправитель закрывает канал.

Горутин-получатель в бесконечном цикле получает данные из канала. Если будет получен маркер закрытия канала, то осуществляется выход из бесконечного цикла.

```
package main
import "fmt"

func main(){
    intCh := make(chan int)

    go factorial(7, intCh)

    for {
        num, opened := <- intCh           // получаем данные из потока
        if !opened {
            break           // если поток закрыт, выход из цикла
        }
        fmt.Println(num)
    }
}

func factorial(n int, ch chan int){
    defer close(ch)
    result := 1
    for i := 1; i <= n; i++){
        result *= i
        ch <- result           // посылаем по числу
    }
}
```

В данном случае функция main и горутин factorial взаимодействуют через канал intCh. Функция factorial последовательно вычисляем факториалы чисел от 1 до n. И все вычисленные значения передаются в канал. По завершении функции factorial канал закрывается вызовом `defer close(ch)`.

В функции main в бесконечном цикле отправленные данные извлекаются из канала. При этом также проверяется, открыт ли канал. И если вдруг канал закрыт и соответственно нет смысла получать из него данные, происходит выход из бесконечного цикла:

```
for {
    num, opened := <- intCh
    if !opened {
        break
    }
    fmt.Println(num)
}
```

Консольный вывод программы:

```
1
2
6
24
120
720
5040
```

При извлечении значений из канала мы можем использовать ту же форму цикла for, которая применяется для перебора коллекций:

```
for переменная := канал{
    //.....
}
```

Например, перепишем предыдущий пример:

```
package main
import "fmt"

func main(){
    intCh := make(chan int)

    go factorial(7, intCh)

    for num := range intCh{
        fmt.Println(num)
    }
}

func factorial(n int, ch chan int){
    defer close(ch)
    result := 1
    for i := 1; i <= n; i++){
        result *= i
        ch <- result
    }
}
```

Когда канал будет закрыт, то автоматически произойдет выход из цикла for.

Мьютексы

Последнее обновление: 12.01.2018

Для упрощения синхронизации между горутинами в Go имеется пакет sync, который предоставляет ряд возможностей, в частности мьютексы. Мьютексы позволяют разграничить доступ к некоторым общим ресурсам, гарантируя, что только одна горутина имеет к ним доступ в определенный момент времени. И пока одна горутина не освободит общий ресурс, другая горутина не может с ним работать.

На уровне кода мьютекс представляет тип sync.Mutex. Для блокирования доступа к общему разделяемому ресурсу у мьютекса вызывается метод Lock(), а для разблокировки доступа - метод Unlock().

В какой ситуации нам могут помочь мьютексы? Рассмотрим следующую ситуацию:

```
package main
import "fmt"

var counter int = 0 // общий ресурс
func main() {

    ch := make(chan bool) // канал
    for i := 1; i < 5; i++){
        go work(i, ch)
    }
    // ожидаем завершения всех горутин
    for i := 1; i < 5; i++){
        <-ch
    }
    fmt.Println("The End")
}

func work (number int, ch chan bool){
    counter = 0
    for k := 1; k <= 5; k++){
        counter++
        fmt.Println("Goroutine", number, "-", counter)
    }
    ch <- true
}
```

Функция work сбрасывает значение переменной counter к нулю и в цикле последовательно увеличивает ее значение до 5. В функции main запускается четыре горутин work. Но какой в данном случае будет консольный вывод? Он может быть, например, таким:

```
Goroutine 3 - 1
Goroutine 3 - 2
Goroutine 3 - 3
Goroutine 3 - 4
Goroutine 3 - 5
```

```
Goroutine 2 - 1
Goroutine 2 - 6
Goroutine 2 - 7
Goroutine 2 - 8
Goroutine 2 - 9
Goroutine 1 - 1
Goroutine 1 - 10
Goroutine 1 - 11
Goroutine 1 - 12
Goroutine 1 - 13
Goroutine 4 - 1
Goroutine 4 - 14
Goroutine 4 - 15
Goroutine 4 - 16
Goroutine 4 - 17
The End
```

Несмотря на то, что в каждой горутине значение counter сбрасывается к 0, а затем увеличивается до 5, мы видим, что несколько горутин после сброса переменной работают совсем с другим значением. То есть при запуске горутин каждая из них получает значение переменной counter и начинает с ней работать. Пока одна горутина еще не закончила работу с counter в цикле, с этой же переменной начинает работать и другая горутина. То есть к одному и тому же разделяемому общему ресурсу - переменной counter одновременно работают сразу несколько горутин. Это может привести к некорректным результатам, как в данном случае.

С помощью мьютексов можно ограничить доступ к переменной таким образом, чтобы только одна горутина имела к ней монопольный доступ в один момент времени:

```
package main
import (
    "fmt"
    "sync"
)

var counter int = 0 // общий ресурс
func main() {

    ch := make(chan bool) // канал
    var mutex sync.Mutex // определяем мьютекс
    for i := 1; i < 5; i++{
        go work(i, ch, &mutex)
    }

    for i := 1; i < 5; i++{
        <-ch
    }

    fmt.Println("The End")
}
func work (number int, ch chan bool, mutex *sync.Mutex){
    mutex.Lock() // блокируем доступ к переменной counter
    counter = 0
    for k := 1; k <= 5; k++{
        counter++
        fmt.Println("Goroutine", number, "-", counter)
    }
    mutex.Unlock() // деблокируем доступ
    ch <- true
}
```

Теперь функция work принимает указатель на мьютекс. С помощью вызова mutex.Lock() мьютекс блокируется данной горутинной. Это значит, что к последующему коду имеет доступ только та горутина, которая первая заблокировала мьютекс. Остальные горутинны ждут пока, мьютекс освободится.

Далее горутина сбрасывает значение переменной counter к нулю и затем в цикле последовательно увеличивает его. В конце, когда все действия с общим ресурсом уже выполнены, горутина освобождает мьютекс с помощью вызова mutex.Unlock(). Ожидающие горутинны получают сигнал, что мьютекс освободился, и одна из горутин блокирует мьютекс и начинает выполнять действия с переменной counter. И так далее горутинны последовательно захватывают и освобождают мьютекс. В итоге к следующей секции:

```
mutex.Lock() // блокируем доступ к переменной counter
counter = 0
for k := 1; k <= 5; k++{
    counter++
    fmt.Println("Goroutine", number, "-", counter)
}
mutex.Unlock() // деблокируем доступ
```

будет иметь доступ только та горутина, которая первая заблокировала мьютекс. В итоге мы получим следующий результат:

```
Goroutine 1 - 1
Goroutine 1 - 2
Goroutine 1 - 3
Goroutine 1 - 4
Goroutine 1 - 5
Goroutine 4 - 1
Goroutine 4 - 2
Goroutine 4 - 3
Goroutine 4 - 4
Goroutine 4 - 5
Goroutine 3 - 1
Goroutine 3 - 2
Goroutine 3 - 3
Goroutine 3 - 4
Goroutine 3 - 5
Goroutine 2 - 1
Goroutine 2 - 2
Goroutine 2 - 3
Goroutine 2 - 4
Goroutine 2 - 5
The End
```

Последнее обновление: 09.02.2020

Еще одну возможность по синхронизации горутин представляет использование типа `sync.WaitGroup`. Этот тип позволяет определить группу горутин, которые должны выполняться вместе как одна группа. И можно установить блокировку, которая приостановит выполнение функции, пока не завершит выполнение вся группа горутин. Например:

```
package main
import (
    "fmt"
    "sync"
    "time"
)

func main() {
    var wg sync.WaitGroup
    wg.Add(2) // в группе две горутин
    work := func(id int) {
        defer wg.Done()
        fmt.Printf("Горутина %d начала выполнение \n", id)
        time.Sleep(2 * time.Second)
        fmt.Printf("Горутина %d завершила выполнение \n", id)
    }

    // вызываем горутин
    go work(1)
    go work(2)

    wg.Wait() // ожидаем завершения обеих горутин
    fmt.Println("Горутин завершили выполнение")
}
```

Вначале определяем группу в виде переменной `wg sync.WaitGroup`. С помощью метода `Add` определяем, что группа будет состоять из двух элементов:

```
wg.Add(2)
```

Число, которое передается в метод `Add` определяет значение внутреннего счетчика активных элементов.

Все элементы группы `wg` будут представлять анонимную функцию в виде переменной `work`, которая в качестве параметра принимает условный числовой идентификатор горутин. Эта функция будет вызываться в виде горутин. Чтобы сигнализировать, что элемент группы завершил свое выполнение, в горутине необходимо вызвать метод `Done()`:

```
defer wg.Done()
```

Вызов метода `wg.Done()` уменьшает внутренний счетчик активных элементов на единицу.

В самой функции `work()` с помощью задержки времени на две секунды (`time.Sleep(2 * time.Second)`) имитируется работа горутин

Далее вызываем две горутин:

```
go work(1)
go work(2)
```

Причем количество горутин, которые вызывают метод `wg.Done()` должно соответствовать количеству элементов группы `wg`, то есть в данном случае 2 элемента.

Затем вызывается метод `Wait()`, который ожидает завершения всех горутин из группы `wg`:

```
wg.Wait()
```

Метод деблокирует функцию `main`, когда внутренний счетчик активных элементов в группе `wg` стает равен 0. Поэтому когда все горутин из группы `wg` завершат выполнение, функция `main` продолжит свою работу.

Результат работы программы:

```
Горутина 1 начала выполнение
Горутина 2 начала выполнение
Горутина 1 завершила выполнение
Горутина 2 завершила выполнение
Горутин завершили выполнение
```

Потоки и файлы

Операции ввода-вывода. Reader и Writer

Последнее обновление: 15.01.2018

Язык Go имеет свою модель работы с потоками ввода-вывода, которая позволяет получать данные из различных источников - файлов, сетевых интерфейсов, объектов в памяти и т.д.

Поток данных в Go представлен байтовым срезом (`[]byte`), из которого можно считывать байты или в который можно заносить данные. Ключевыми типами для работы с потоками являются интерфейсы `Reader` и `Writer` из пакета [io](#).

io.Reader

Интерфейс `io.Reader` предназначен для считывания данных. Он имеет следующее определение:

```
type Reader interface {
    Read(p []byte) (n int, err error)
}
```

Метод `Read` возвращает общее количество считанных байт из среза байт и информацию об ошибке, если она возникнет. Если в потоке больше нет данных, то метод должен возвращать ошибку типа `io.EOF`.

Рассмотрим простейший пример. Например, нам необходимо считывать номера телефонов, которые могут иметь различные форматы:

```
package main
import (
    "fmt"
    "io"
)

type phoneReader string

func (ph phoneReader) Read(p []byte) (int, error){
    count := 0
    for i := 0; i < len(ph); i++){
        if(ph[i] >= '0' && ph[i] <= '9'){
            p[count] = ph[i]
            count++
        }
    }
    return count, io.EOF
}

func main() {
    phone1 := phoneReader("+1(234)567 9010")
    phone2 := phoneReader("+2-345-678-12-35")

    buffer := make([]byte, len(phone1))
    phone1.Read(buffer)
    fmt.Println(string(buffer))           // 12345679010

    buffer = make([]byte, len(phone2))
    phone2.Read(buffer)
    fmt.Println(string(buffer))           // 23456781235
}
```

Для считывания номеров телефонов определен тип `phoneReader`, который по сути представляет тип `string`. Однако `phoneReader` при этом реализует интерфейс `Reader`, то есть определяет его метод `Read`. В методе `Read` считываем данные из строки, которую представляет объект `phoneReader` и, если символы строки представляют числовые данные, передаем их в срез байтов. На выходе возвращаем количество считанных данных и маркер окончания чтения `io.EOF`. В результате при считывании из строки метод `Read` возвратит номер телефона, который состоит только из цифр.

При вызове метода `Read` создается срез байтов достаточной длины, который передается в метод `Read`:

```
buffer := make([]byte, len(phone1))
phone1.Read(buffer)
```

Затем с помощью инициализатора `string` мы можем преобразовать срез байтов в строку:

```
fmt.Println(string(buffer))           // 12345679010
```

io.Writer

Интерфейс `io.Writer` предназначен для записи в поток. Он определяет метод `Write()`:

```
type Writer interface {
    Write(p []byte) (n int, err error)
}
```

Метод `Write` предназначен для копирования данных их среза байт `p` в определенный ресурс - файл, сетевой интерфейс и т.д. Метод возвращает количество записанных байтов и объект ошибки.

Рассмотрим примитивный пример:

```
package main
import "fmt"

type phoneWriter struct{ }

func (p phoneWriter) Write(bs []byte) (int, error){
    if len(bs) == 0 {
        return 0, nil
    }
    for i := 0; i < len(bs); i++){
        if(bs[i] >= '0' && bs[i] <= '9'){
            fmt.Print(string(bs[i]))
        }
    }
    fmt.Println()
    return len(bs), nil
}

func main() {
    bytes1 := []byte("+1(234)567 9010")
    bytes2 := []byte("+2-345-678-12-35")

    writer := phoneWriter{}
    writer.Write(bytes1)
    writer.Write(bytes2)
}
```

Здесь структура `phoneWriter` реализует интерфейс `Writer`. В методе `Write` она принимает срез байтов. Предполагается, что срез байтов хранит номер телефона. Эта информация должным образом обрабатывается: из нее выделяются цифры, которые выводятся на консоль. То есть тип `phoneWriter` осуществляет запись потока байт на консоль.

В качестве результата метод возвращает длину среза и значение `nil`.

Для имитации потока байт определяются два среза байт на основе строк, которые передаются в метод `Write`.

На основе выше рассмотренных интерфейсов `Writer` и `Reader` основана вся система ввода-вывода в Go, и впоследствии мы более детально рассмотрим их применение при работе с файлами и сетевыми потоками.

Создание и открытие файлов

Последнее обновление: 15.01.2018

Для работы с файлами мы можем использовать функциональность пакета [os](#). Все файлы в Go представлены типом `os.File`. Этот тип реализует ряд интерфейсов, например, `io.Reader` и `io.Writer`, которые позволяют читать содержимое файла и сохранять данные в файл.

С помощью функции `os.Create()` можно создать файл по определенному пути. Путь к файлу передается в качестве параметра. Если подобный файл уже существует, то он перезаписывается:

```
file, err := os.Create("hello.txt")
```

Функция возвращает объект `os.File` для работы с файлом и информацию об ошибке, которая может возникнуть при создании файла.

Ранее созданный файл можно открыть с помощью функции `os.Open()`:

```
file, err := os.Open("hello.txt")
```

Эта функция также возвращает объект `os.File` для работы с файлом и информацию об ошибке, которая может возникнуть при открытии файла.

Также в нашем распоряжении есть функция `os.OpenFile()`, которая открывает файл, а если файла нет, то создает его. Она принимает три параметра:

- путь к файлу
- режим открытия файла (для чтения, для записи и т.д.)
- разрешения для доступа к файлу

Например:

```
// открытие файла для чтения
f1, err := os.OpenFile("sometext.txt", os.O_RDONLY, 0666)
// открытие файла для записи
f2, err := os.OpenFile("common.txt", os.O_WRONLY, 0666)
```

После окончания работы с файлом его следует закрыть с помощью метода `Close()`.

```
package main
import (
    "fmt"
    "os"
)

func main() {
    file, err := os.Create("hello.txt")           // создаем файл
    if err != nil {                               // если возникла ошибка
        fmt.Println("Unable to create file:", err)
        os.Exit(1)                               // выходим из программы
    }
    defer file.Close()                           // закрываем файл
    fmt.Println(file.Name())                     // hello.txt
}
```

С помощью функции `os.Exit()` можно выйти из программы. А метод `Name()`, определенный для типа `os.File`, позволяет получить имя файла.

Чтение и запись файлов

Последнее обновление: 15.01.2018

Запись в файл

Для записи текстовой информации в файл можно применять метод `WriteString()` объекта `os.File`, который заносит в файл строку:

```
package main
import (
    "fmt"
    "os"
)

func main() {
    text := "Hello Gold!"
    file, err := os.Create("hello.txt")

    if err != nil {
        fmt.Println("Unable to create file:", err)
        os.Exit(1)
    }
    defer file.Close()
    file.WriteString(text)

    fmt.Println("Done.")
}
```

В данном случае создается файл `hello.txt`, в который записывается строка `"Hello Gold!"`.

Для записи нетекстовой бинарной информации в виде набора байт применяется метод `Write()` (реализация интерфейса `io.Writer`):

```
package main
import (
    "fmt"
    "os"
)

func main() {
    data := []byte("Hello Bold!")
    file, err := os.Create("hello.bin")
```



```

    if err != nil{
        fmt.Println("Unable to create file:", err)
    }
    os.Exit(1)
}
defer file.Close()
file.Write(data)

    fmt.Println("Done.")
}

```

Чтение из файла

Поскольку тип `io.File` реализует интерфейс `io.Reader` , то для чтения из файла мы можем использовать метод `Read()` . Этот метод позволяет получить содержимое файла в виде набора байт:

```

package main
import (
    "fmt"
    "os"
    "io"
)

func main() {
    file, err := os.Open("hello.txt")
    if err != nil{
        fmt.Println(err)
    }
    os.Exit(1)
}
defer file.Close()

    data := make([]byte, 64)

    for{
        n, err := file.Read(data)
        if err == io.EOF{           // если конец файла
            break                  // выходим из цикла
        }
        fmt.Print(string(data[:n]))
    }
}

```

Для считывания данных определяется срез из 64 байтов. В бесконечном цикле содержимое файла считывается в срез, а когда будет достигнут конец файла, то есть мы получим ошибку `io.EOF`, то произойдет выход из цикла. Ну и поскольку данные представляют срез байтов, хотя файл `hello.txt` хранит текстовую информацию, то для вывода текста на консоль преобразуем срез байтов в строку: `string(data[:n])` .

Стандартные потоки ввода-вывода и `io.Copу`

Последнее обновление: 15.01.2018

Пакет `os` определяет три переменных: `os.Stdin` , `os.Stdout` и `os.Stderr` , которые представляют стандартные потоки ввода, вывода и вывода ошибок соответственно. Так, стандартный поток вывода `os.Stdout` представляет вывод информации на консоль.

Например, мы можем использовать функцию `io.Copу()` для копирования данных из одного потока в другой:

```
n, err = io.Copy(io.Writer, io.Reader)
```

Эта функция упрощает копирование данных из объекта `io.Reader` в объект `io.Writer` . В качестве результата функция возвращает количество скопированных файлов и информацию об ошибке.

И при выводе из файла текстовой информации на консоль гораздо проще передать данные из файлового потока в `os.Stdout`, через вывести данные отдельными порциями:

```

package main
import (
    "fmt"
    "os"
    "io"
)

func main() {
    file, err := os.Open("hello.txt")
    if err != nil{
        fmt.Println(err)
    }
    os.Exit(1)
}
defer file.Close()

    io.Copy(os.Stdout, file)
}

```

В качестве `io.Reader` можно использовать свои кастомные объекты, которые реализуют данный интерфейс. Например:

```

package main
import (
    "fmt"
    "io"
    "os"
)

type phoneReader string

func (p phoneReader) Read(bs []byte) (int, error){
    count := 0
    for i := 0; i < len(p); i++){
        if(p[i] >= '0' && p[i] <= '9'){
            bs[count] = p[i]
            count++
        }
    }
}

```

```

    }
    return count, io.EOF
}

func main() {
    phone1 := phoneReader("+1(234)567 90-10")
    io.Copy(os.Stdout, phone1)
    fmt.Println()
}

```

В данном случае в качестве интерфейса `io.Reader` передается объект `phoneReader`, который считывает цифровые символы из номера телефона.

Форматированный вывод

Последнее обновление: 15.01.2018

Ряд возможностей по чтению и записи файлов предоставляет пакет [fmt](#). Этот пакет предоставляет ряд функций для записи данных в произвольный объект, который реализует интерфейс `io.Writer` : `fmt.Fprint()` , `fmt.Fprintln()` и `fmt.Fprintf()` .

Функции `Fprint` и `Fprintln`

Функции `Fprint` и `Fprintln` имеют примерно одинаковое определение:

```

func Fprint(w io.Writer, a ...interface{}) (n int, err error)
func Fprintln(w io.Writer, a ...interface{}) (n int, err error)

```

Первым параметром передается объект, который реализует интерфейс `io.Writer`. А второй параметр представляет набор объектов, которые записываются в поток. Обе функции возвращают количество записанных байтов и информацию об ошибке. Отличием функции `Fprintln` является то, что она добавляет при выводе перевод строки, то есть фактически записывает строку. Например:

```

package main
import (
    "fmt"
    "os"
)

func main() {
    file, err := os.Create("confeve.txt")
    if err != nil {
        fmt.Println(err)
    }
    os.Exit(1)
}
defer file.Close()
fmt.Fprint(file, "Сегодня ")
fmt.Fprintln(file, "хорошая погода")
}

```

В данном случае обе функции записывают некоторый текст в файл `confeve.txt`, который будет создан в той же папке, где расположен выполняемый скрипт.

Форматирование и `Fprintf`

Функция `Fprintf` упрощает запись сложных по структуре данных:

```

func Fprintf(w io.Writer, format string, a ...interface{}) (n int, err error)

```

Первым параметром также идет объект `io.Writer`. Второй параметр представляет строку форматирования, которая указывает, как данные будут форматироваться при записи. И третий параметр - набор значений, которые передаются в строку форматирования и записываются в поток вывода.

Строка форматирования представляет набор спецификаторов. Каждый спецификатор представляет набор символов, которые интерпретируются определенным образом и предваряются знаком процента `%`. Каждый спецификатор представляет определенный тип данных:

- `%t` : для вывода значений типа `boolean` (`true` или `false`)
- `%b` : для вывода целых чисел в двоичной системе
- `%c` : для вывода символов, представленных числовым кодом
- `%d` : для вывода целых чисел в десятичной системе
- `%o` : для вывода целых чисел в восьмеричной системе
- `%q` : для вывода символов в одинарных кавычках
- `%x` : для вывода целых чисел в шестнадцатеричной системе, буквенные символы числа имеют нижний регистр `a-f`
- `%X` : для вывода целых чисел в шестнадцатеричной системе, буквенные символы числа имеют верхний регистр `A-F`
- `%U` : для вывода символов в формате кодов `Unicode`, например, `U+1234`
- `%e` : для вывода чисел с плавающей точкой в экспоненциальном представлении, например, `-1.234456e+78`
- `%E` : для вывода чисел с плавающей точкой в экспоненциальном представлении, например, `-1.234456E+78`
- `%f` : для вывода чисел с плавающей точкой, например, `123.456`
- `%F` : то же самое, что и `%f`
- `%g` : для длинных чисел с плавающей точкой используется `%e`, для других - `%f`
- `%G` : для длинных чисел с плавающей точкой используется `%E`, для других - `%F`
- `%s` : для вывода строки

- `%p` : для вывода значения указателя - адреса в шестнадцатеричном представлении

Также можно применять универсальный спецификатор `%v` , который для типа `boolean` аналогичен `%t` , для целочисленных типов - `%d` , для чисел с плавающей точкой - `%g` , для строк - `%s` .

К спецификаторам можно добавлять различные флаги, которые влияют на форматирование значений. Например, число перед спецификатором указывает, какую минимальную длину в символах будет занимать выводимое значение. Например, `%9f` - число с плавающей точкой будет занимать как минимум 9 позиций. Если ширина больше, чем требуется значению, то заполняется пробелами.

Для чисел с плавающей точкой можно указать точность или количество символов в дробной части. Для этого количество символов указывается после точки: `%.2f` - две цифры в дробной части после точки. Например, варианты форматирования чисел с плавающей точкой:

`%f` : точность и ширина значения по умолчанию

`%9f` : ширина - 9 символов и точность по умолчанию

`%.2f` : ширина по умолчанию и точность - 2 символа

`%9.2f` : ширина - 9 и точность - 2

`%9.f` : ширина - 9 и точность - 0

Также из флагов следует отметить дефис -, который дополняет значение пробелами не справа, как по умолчанию, а слева.

Применим функцию `Fprintf` для вывода в файл:

```
package main
import (
    "fmt"
    "os"
)
type person struct {
    name string
    age  int32
    weight float64
}
func main() {
    tom := person {
        name: "Tom",
        age: 24,
        weight: 68.5,
    }
    file, err := os.Create("person.dat")
    if err != nil {
        fmt.Println(err)
        os.Exit(1)
    }
    defer file.Close()
    fmt.Fprintf(file,
        "%-10s %-10d %-10.3f\n",
        tom.name, tom.age, tom.weight)
}
```

Функция `Fprintf()` в качестве первого параметра также принимает файл, а в качестве второго параметра - строку форматирования, которая определяет, как данные будут форматироваться. После строки форматирования перечисляются значения, которые вставляются вместо спецификаторов. При этом значения передаются на место спецификаторов по позиции. Например, первое значение передается вместо первого спецификатора, второе значение - вместо второго спецификатора и так далее. При этом значения должны соответствовать спецификаторам по типу: на место спецификатора `%s` должна передаваться строка, на место `%d` - целое число и т.д.

Таким образом, в примере выше будет создан в одной папке со скриптом файл `person.dat`, в который будет записаны данные объекта `person`.

Вывод на консоль

Последнее обновление: 15.01.2018

Стандартным потоком вывода в Go является объект `os.Stdout` , который фактически представляет консоль. Например, мы могли бы вывести в этот поток данные следующим образом:

```
package main
import (
    "fmt"
    "os"
)
func main() {
    fmt.Fprintln(os.Stdout, "hello cold")
}
```

Здесь используется рассмотренная в прошлой теме функция `Fprintln()` , которая выводит в поток вывода набор значений. То есть фактически в данном случае идет запись или вывод на консоль.

Однако поскольку запись в стандартный поток `os.Stdout` - довольно распространенная задача, то вместо функций `Fprint/Fprintln/Fprintf` применяются их двойники: `Println()` , `Print()` и `Printf()` соответственно, которые по умолчанию выводят данные в `os.Stdout`:

```
package main
import "fmt"

type person struct {
    name string
    age  int32
    weight float64
}
func main() {
    tom := person {
        name: "Tom",
        age: 24,
```

```

        weight: 68.5,
    }
    fmt.Printf("%-10s %-10d %-10.3f\n",
        tom.name, tom.age, tom.weight)
    fmt.Print("Hello ")
    fmt.Println("cold!")
}

```

Форматируемый ввод

Последнее обновление: 15.01.2018

Пакет `fmt` чтение из объекта, который реализует интерфейс `io.Reader`. Для этого применяются следующие функции: `Fscan()`, `Fscanln()` и `Fscanf()`.

Функции `Fscan` и `Fscanln`

Через параметры функций `Fscan` и `Fscanln` можно получить вводимые значения:

```

func Fscan(r io.Reader, a ...interface{}) (n int, err error)
func Fscanln(r io.Reader, a ...interface{}) (n int, err error)

```

В качестве первого параметра передается объект `io.Reader`, из которого надо считывать данные, а второй параметр представляет объекты, в которые считываются данные. В качестве результата обе функции возвращают количество считанных байт и информацию об ошибке. Например:

```

package main
import (
    "fmt"
    "os"
)

type person struct {
    name string
    age int32
    weight float64
}
func main() {
    filename := "hello2.txt"
    writeData(filename)
    readData(filename)
}

func writeData(filename string){
    // начальные данные
    var name string = "Tom"
    var age int    = 24

    file, err := os.Create(filename)
    if err != nil {
        fmt.Println(err)
        os.Exit(1)
    }
    defer file.Close()

    fmt.Fprintln(file, name)
    fmt.Fprintln(file, age)
}
func readData(filename string){
    var name string
    var age int

    file, err := os.Open(filename)
    if err != nil{
        fmt.Println(err)
        os.Exit(1)
    }
    defer file.Close()

    fmt.Fscanln(file, &name)
    fmt.Fscanln(file, &age)
    fmt.Println(name, age)
}

```

В данном случае вначале записываем две переменных в файл с помощью `fmt.Fprintln`, а затем считываем записанные значения с помощью `fmt.Fscanln`.

`Fscanf`

Функция `fmt.Fscanf()` считывает данные с применением форматирования:

```

func Fscanf(r io.Reader, format string, a ...interface{}) (n int, err error)

```

Первый параметр функции представляет объект `io.Reader`. Второй параметр - строка форматирования, которая содержит спецификаторы и определяет последовательность считывания данных. Третий параметр - набор объектов, в которые надо считать данные. Например:

```

package main
import (
    "fmt"
    "os"
)

type person struct {
    name string
    age int32
    weight float64
}
func main() {
    filename := "person.dat"
    writeData(filename)
    readData(filename)
}

```

```

}

func writeData(filename string){
    // начальные данные
    tom := person { name:"Tom", age: 24, weight: 68.5 }

    file, err := os.Create(filename)
    if err != nil {
        fmt.Println(err)
        os.Exit(1)
    }
    defer file.Close()

    // сохраняем данные в файл
    fmt.Fprintf(file, "%s %d %.2f\n", tom.name, tom.age, tom.weight)
}

func readData(filename string){

    // переменные для считывания данных
    var name string
    var age int
    var weight float64

    file, err := os.Open(filename)
    if err != nil{
        fmt.Println(err)
        os.Exit(1)
    }
    defer file.Close()

    // считывание данных из файла
    _, err = fmt.Fscanf(file, "%s %d %f\n", &name, &age, &weight)
    if err != nil{
        fmt.Println(err)
        os.Exit(1)
    }
    fmt.Printf("%-8s %-8d %-8.2f\n", name, age, weight)
}

```

Здесь вначале данные структуры person записываются в файл, а затем считываются из него в три переменных. При записи данных файл мы знаем его структуру. Поэтому мы можем взять строку форматирования с той же последовательностью спецификаторов и выполнить обратное действие - считать данные. При считывание в объекты в функцию передаются их адреса:

```
fmt.Fscanf(file, "%s %d %f\n", &name, &age, &weight)
```

При определении строки форматирования и передаче объектов для считывания действуют те же правила, что и при записи с помощью fmt.Fprintf. Так, первый спецификатор связан с первым объектом, второй спецификатор - со вторым объектом и так далее. И также спецификаторы должны соответствовать объектам по типу.

В итоге при выполнении этой программы на консоль будет выведено:

```
Tom      24      68.50
```

При этом объекты, в которые производится считывание, необязательно должны представлять переменные примитивных типов. Например, это может быть и структура:

```

func readData(filename string){

    // переменная для считывания данных
    tom := person{}

    file, err := os.Open(filename)
    if err != nil{
        fmt.Println(err)
        os.Exit(1)
    }
    defer file.Close()

    // считывание данных из файла
    _, err = fmt.Fscanf(file, "%s %d %f\n", &tom.name, &tom.age, &tom.weight)

    if err != nil{
        fmt.Println(err)
        os.Exit(1)
    }
    fmt.Printf("%-8s %-8d %-8.2f\n", tom.name, tom.age, tom.weight)
}

```

Рассмотрим более сложный пример, когда файл содержит набор данных структур person:

```

package main
import (
    "fmt"
    "os"
    "io"
)

type person struct {
    name string
    age int32
    weight float64
}

func main() {
    filename := "people.dat"
    writeData(filename)
    readData(filename)
}

func writeData(filename string){
    // начальные данные
    var people = []person{
        { "Tom", 24, 68.5 },

```

```

        { "Bob", 25, 64.2 },
        { "Sam", 27, 73.6 },
    }

    file, err := os.Create(filename)
    if err != nil {
        fmt.Println(err)
        os.Exit(1)
    }
    defer file.Close()

    for _, p := range people{
        fmt.Fprintf(file, "%s %d %.2f\n", p.name, p.age, p.weight)
    }
}

func readData(filename string){

    var name string
    var age int
    var weight float64

    file, err := os.Open(filename)
    if err != nil{
        fmt.Println(err)
        os.Exit(1)
    }
    defer file.Close()

    for{
        _, err = fmt.Fscanf(file, "%s %d %f\n", &name, &age, &weight)
        if err != nil{
            if err == io.EOF{
                break
            } else{
                fmt.Println(err)
                os.Exit(1)
            }
        }
        fmt.Printf("%-8s %-8d %-8.2f\n", name, age, weight)
    }
}

```

Сначала функция writeData записывает в файл набор объектов person. А затем в функции readData из файла считываются данные в бесконечном цикле. Когда файл закончится, функция Fscanf возвратит ошибку io.EOF.

Чтение с консоли

Последнее обновление: 15.01.2018

В Go имеется объект os.Stdin , который реализует интерфейс io.Reader и позволяет считывать данные с консоли. Например, мы можем использовать функцию fmt.Fscan() для считывания с консоли с помощью os.Stdin:

```

package main
import (
    "fmt"
    "os"
)

func main() {
    var name string
    var age int
    fmt.Print("Введите имя: ")
    fmt.Fscan(os.Stdin, &name)

    fmt.Print("Введите возраст: ")
    fmt.Fscan(os.Stdin, &age)

    fmt.Println(name, age)
}

```

При запуске программы мы сможем вводить данные с консоли, и они перейдут в переменные name и age:

```

Введите имя: Том
Введите возраст: 24
Том 24

```

Однако также для получения ввода с консоли мы можем использовать встроенные функции fmt.Scan() , fmt.Scanln() и fmt.Scanf() , которые аналогичны соответственно функциям fmt.Fscan() , fmt.Scanln() и fmt.Fscanf() :

```

func Scan(a ...interface{}) (n int, err error)
func Scanf(format string, a ...interface{}) (n int, err error)
func Scanln(a ...interface{}) (n int, err error)

```

Все эти функции уже по умолчанию считывают данные с потока os.Stdin:

```

package main
import (
    "fmt"
    "os"
)

func main() {
    var name string
    var age int
    fmt.Print("Введите имя: ")
    fmt.Scan(&name)
    fmt.Print("Введите возраст: ")
    fmt.Scan(&age)
}

```

```

    fmt.Println(name, age)
}

или так

package main
import (
    "fmt"
    "os"
)

func main() {
    var name string
    var age int
    fmt.Print("Введите имя и возраст: ")
    fmt.Scan(&name, &age)
    fmt.Println(name, age)

    // альтернативный вариант
    //fmt.Println("Введите имя и возраст:")
    //fmt.Sprintf("%s %d", &name, &age)
    //fmt.Println(name, age)
}

```

В случае если вводятся сразу несколько значений, то разделителем между ними является пробел. Хотя теоретически строка может включать внутренние пробелы, тем не менее данные функции считывают значение строки и других типов данных до пробела:

```

Введите имя и возраст: Tom 34
Tom 34

```

Буферизированный ввод-вывод

Последнее обновление: 16.01.2018

Большинство встроенных операций ввода-вывода не используют буфер. Это может иметь отрицательный эффект для производительности приложения. Для буферизации потоков чтения и записи в Go определены ряд возможностей, которые сосредоточены в пакете `bufio`.

Запись через буфер

Для записи в источник данных через буфер в пакете `bufio` определен тип `Writer`. Чтобы записать данные, можно воспользоваться одним из его методов:

```

func (b *Writer) Write(p []byte) (nn int, err error)
func (b *Writer) WriteByte(c byte) error
func (b *Writer) WriteRune(r rune) (size int, err error)
func (b *Writer) WriteString(s string) (int, error)

```

- `Write()` : записывает срез байтов
- `WriteByte()` : записывает один байт
- `WriteRune()` : записывает один объект типа `rune`
- `WriteString()` : записывает строку

При выполнении этих методов данные вначале накапливаются в буфере, а чтобы сбросить их в источник данных, необходимо вызвать метод `Flush()`.

Для создания потока вывода через буфер применяется функция `bufio.NewWriter()`:

```

func NewWriter(w io.Writer) *Writer

```

Она принимает объект `io.Writer` - это может быть любой объект, в который идет запись: `os.Stdout`, файл и т.д. В качестве результата возвращается объект `bufio.Writer`:

```

package main
import (
    "fmt"
    "os"
    "bufio"
)
func main() {
    rows := []string{
        "Hello Go!",
        "Welcome to Golang",
    }

    file, err := os.Create("some.dat")
    writer := bufio.NewWriter(file)
    if err != nil {
        fmt.Println(err)
        os.Exit(1)
    }
    defer file.Close()

    for _, row := range rows {
        writer.WriteString(row)           // запись строки
        writer.WriteString("\n")         // перевод строки
    }
    writer.Flush()                       // сбрасываем данные из буфера в файл
}

```

В данном случае в файл через буферизированный поток вывода записываются две строки.

Чтение через буфер

Для чтения из источника данных через буфер в пакете `bufio` определен тип `Reader`. Для чтения данных можно воспользоваться одним из его методов:

```
func (b *Reader) Read(p []byte) (n int, err error)
func (b *Reader) ReadByte() (byte, error)
func (b *Reader) ReadBytes(delim byte) ([]byte, error)
func (b *Reader) ReadLine() (line []byte, isPrefix bool, err error)
func (b *Reader) ReadRune() (r rune, size int, err error)
func (b *Reader) ReadSlice(delim byte) (line []byte, err error)
func (b *Reader) ReadString(delim byte) (string, error)
```

- `Read(p []byte)` : считывает срез байтов и возвращает количество прочитанных байтов
- `ReadByte()` : считывает один байт
- `ReadBytes(delim byte)` : считывает срез байтов из потока, пока не встретится байт `delim`
- `ReadLine()` : считывает строку в виде среза байт
- `ReadRune()` : считывает один объект типа `rune`
- `ReadSlice(delim byte)` : считывает срез байтов из потока, пока не встретится байт `delim`
- `ReadString(delim byte)` : считывает строку, пока не встретится байт `delim`

Для создания потока ввода через буфер применяется функция `bufio.NewReader()` :

```
func NewReader(rd io.Reader) *Reader
```

Она принимает объект `io.Reader` - это может быть любой объект, с которого производится чтение: `os.Stdin`, файл и т.д. В качестве результата возвращается объект `bufio.Reader`:

```
package main
import (
    "fmt"
    "os"
    "bufio"
    "io"
)
func main(){
    file, err := os.Open("some.data")
    if err != nil {
        fmt.Println("Unable to open file:", err)
        return
    }
    defer file.Close()

    reader := bufio.NewReader(file)
    for {
        line, err := reader.ReadString('\n')
        if err != nil {
            if err == io.EOF {
                break
            } else {
                fmt.Println(err)
                return
            }
        }
        fmt.Print(line)
    }
}
```

В данном случае идет считывания из ранее записанного файла. Для этого объект файла `os.File` передается в функцию `bufio.NewReader`, на основании которого создается объект `bufio.Reader`. Поскольку идет построчное считывание, то каждая строка считывается из потока, пока не будет обнаружен символ перевода строки `\n`.

Сетевое программирование

Последнее обновление: 23.01.2018

Одной из ключевых возможностей языка Go является возможность работы с сетевыми сервисами: отправлять запросы к ресурсам в сети и, наоборот, обрабатывать входящие запросы. Основной функционал по работе с сетью представлен пакетом [net](#). Этот пакет предоставляет различные низкоуровневые сетевые примитивы, через которые идет взаимодействие по сети.

Отправка запросов

Для отправки запросов к ресурсам в сети применяется функция `net.Dial()` :

```
func Dial(network, address string) (Conn, error)
```

Эта функция принимает два параметра: `network` - тип протокола и `address` - адрес ресурса.

Есть следующие типы протоколов:

- `tcp`, `tcp4`, `tcp6` : протокол TCP. `tcp` по умолчанию представляет `tcp4`, цифра в конце указывает, какой тип адресов будет использоваться: IPv4 или IPv6
- `udp`, `udp4`, `udp6` : протокол UDP. `udp` по умолчанию представляет `udp4`
- `ip`, `ip4`, `ip6` : протокол IP. `ip` по умолчанию представляет `ip4`
- `unix`, `unixgram`, `unixpacket` : сокеты Unix

Второй параметр представляет сетевой адрес ресурса (для адресов в сети интернет это домен). Это может быть числовой сетевой адрес, например, `"127.0.0.1"` . Он может включать указание порта, например, `"127.0.0.1:80"` . Это также может быть адрес в формате IPv6, например, `"::1"` или `"[2516:b7f0:3421:b16::71]:80"` .

Функция возвращает объект, который реализует интерфейс `net.Conn`. Этот интерфейс, в свою очередь, применяет интерфейсы `io.Reader` и `io.Writer`, то есть может использоваться как поток для чтения и записи. Пакет `net` предоставляет базовые реализации этого интерфейса в виде типов `IPConn`, `UDPConn`, `TCPConn`. В зависимости от используемого протокола возвращается соответствующий тип.

Таким образом, используя данную функцию, мы можем отправлять запросы по протоколу TCP и UDP. Например:

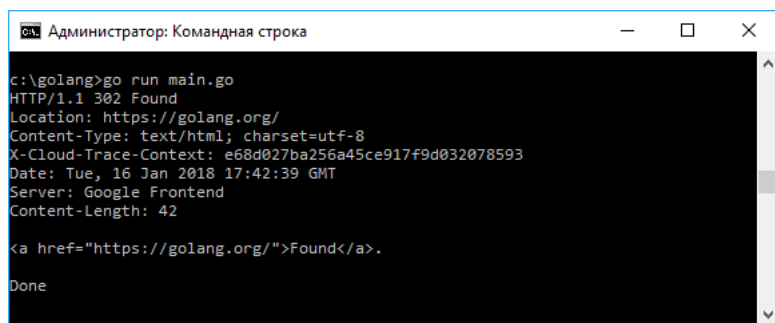
```
package main
import (
    "fmt"
    "os"
    "net"
    "io"
)
func main() {
    httpRequest := "GET / HTTP/1.1\n" +
        "Host: golang.org\n\n"
    conn, err := net.Dial("tcp", "golang.org:80")
    if err != nil {
        fmt.Println(err)
        return
    }
    defer conn.Close()

    if _, err = conn.Write([]byte(httpRequest)); err != nil {
        fmt.Println(err)
        return
    }

    io.Copy(os.Stdout, conn)
    fmt.Println("Done")
}
```

В данном случае мы фактически обращаемся к веб-ресурсу `golang.ru`. Так как `net.Conn` реализует интерфейсы `io.Reader` и `io.Writer`, то в данный объект можно записывать данные - фактически посылать по сети данные и можно считывать с него данные - получать данные из сети. Например, `conn.Write([]byte(httpRequest))` посылает данные, которые здесь представлены переменной `httpRequest`. Так как метод `Write` отправляет срез байтов, то любые данные надо преобразовать в срез байтов.

Как и любой объект `io.Reader`, мы можем передать `net.Conn` в функцию `io.Copy` и считать полученные по сети данные, например, на консоль: `io.Copy(os.Stdout, conn)`.



Стоит отметить, что в примере выше осуществляет запрос к сетевому ресурсу сети интернет по протоколу TCP. Однако для этой же цели куда более удобнее использовать возможности пакета `net/http`, который предназначен специально для протокола HTTP, который работает поверх TCP.

Сервер. Обработка подключений

Последнее обновление: 16.01.2018

Для прослушивания и приемы входящих запросов в пакете `net` определена функция `net.Listen`:

```
func Listen(network, laddr string) (net.Listener, error)
```

Функция принимает два параметра: `network` - протокол, по которому приложение будет получать запросы, и `laddr` представляет локальный адрес, по которому будет запускаться сервер. Протокол должен представлять одно из значений: `"tcp"`, `"tcp4"`, `"tcp6"`, `"unix"`, `"unixpacket"`. Локальный адрес может содержать только номер порта, например, `":8080"`. В этом случае приложение будет обслуживать по всем.

В случае успешного выполнения функция возвращает объект интерфейса `net.Listener`, который представляет функционал для приема входящих подключений. В зависимости от типа используемого протокола возвращаемый объект `Listener` может представлять тип `net.TCPListener` или `net.UnixListener` (оба этих типа реализуют интерфейс `net.Listener`).

Основные методы, которые представляет `net.Listener`: `Accept()` (принимает входящее подключение) и `Close()` (закрывает подключение).

```
package main
import (
    "fmt"
    "net"
)
func main() {
    message := "Hello, I am a server" // отправляемое сообщение
    listener, err := net.Listen("tcp", ":4545")

    if err != nil {
        fmt.Println(err)
        return
    }
    defer listener.Close()
    fmt.Println("Server is listening...")
    for {
        conn, err := listener.Accept()
        if err != nil {
            fmt.Println(err)
            return
        }
    }
}
```

```

    }
    conn.Write([]byte(message))
    conn.Close()
}
}

```

Вначале в функции `net.Listen("tcp", ":4545")` устанавливается 4545 порт для прослушивания подключений по протоколу TCP. После вызова этой функции сервер запущен и готов принимать подключения. Затем в бесконечном цикле `for` получаем входящие подключения с помощью вызова `listener.Accept()`. Этот метод возвращает объект `net.Conn`, который представляет подключенного клиента. Затем мы можем каким-нибудь образом обработать это подключение. Например, с помощью метода `Write` отправить ему сообщение. Поскольку данный метод принимает срез байтов, то любые сообщения надо транслировать в срез байтов: `conn.Write([]byte(message))`

Для тестирования сервера определим еще одну программу - клиент:

```

package main
import (
    "fmt"
    "os"
    "net"
    "io"
)
func main() {
    conn, err := net.Dial("tcp", "127.0.0.1:4545")
    if err != nil {
        fmt.Println(err)
        return
    }
    defer conn.Close()

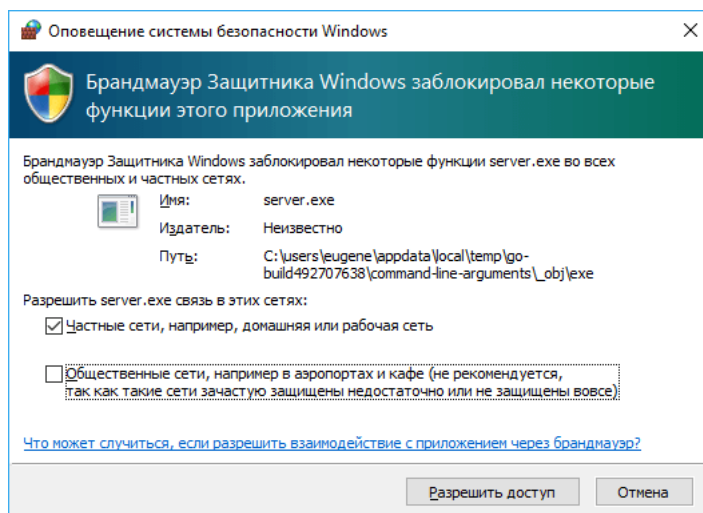
    io.Copy(os.Stdout, conn)
    fmt.Println("\nDone")
}

```

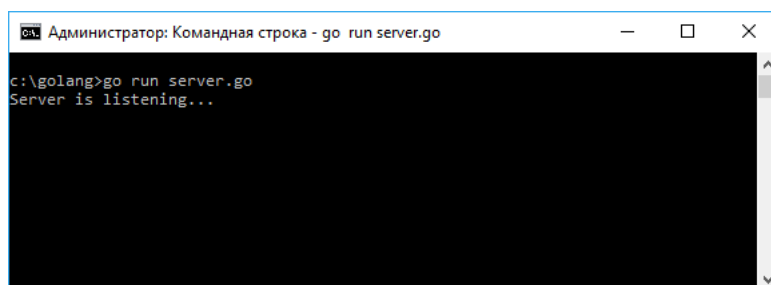
Поскольку сервер будет запущен на локальном компьютере на порте 4545, то клиент подключается к этому адресу: `net.Dial("tcp", "127.0.0.1:4545")`

После этого к серверу будет отправляться запрос, и с помощью вызова `io.Copy(os.Stdout, conn)` выводим полученный ответ на консоль.

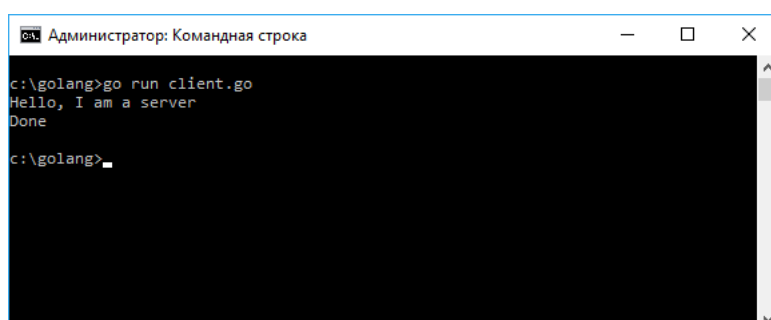
Вначале запустим сервер. На Windows может отобразиться окно с предложением разрешить доступ:



И после этого будет запущен сервер:



Затем запустим клиент:



После запуска клиент подключится к серверу и получит от него сообщение.

Взаимодействие клиента и сервера

Последнее обновление: 18.01.2018

В прошлой теме было рассмотрено создание простейшего сервера TCP, к которому подключается клиент, и этому клиенту отправляется некоторое сообщение. Теперь рассмотрим, как клиент может посылать сообщения и получать ответ.

Для начала определим следующий сервер:

```
package main
import (
    "fmt"
    "net"
)
var dict = map[string]string{
    "red": "красный",
    "green": "зеленый",
    "blue": "синий",
    "yellow": "желтый",
}

func main() {
    listener, err := net.Listen("tcp", ":4545")

    if err != nil {
        fmt.Println(err)
        return
    }
    defer listener.Close()
    fmt.Println("Server is listening...")
    for {
        conn, err := listener.Accept()
        if err != nil {
            fmt.Println(err)
            conn.Close()
            continue
        }
        go handleConnection(conn) // запускаем горутину для обработки запроса
    }
}
// обработка подключения
func handleConnection(conn net.Conn) {
    defer conn.Close()
    for {
        // считываем полученные в запросе данные
        input := make([]byte, (1024 * 4))
        n, err := conn.Read(input)
        if n == 0 || err != nil {
            fmt.Println("Read error:", err)
            break
        }
        source := string(input[0:n])
        // на основании полученных данных получаем из словаря перевод
        target, ok := dict[source]
        if ok == false {
            target = "undefined" // если данные не найдены в словаре
        }
        // выводим на консоль сервера диагностическую информацию
        fmt.Println(source, "-", target)
        // отправляем данные клиенту
        conn.Write([]byte(target))
    }
}
```

Сервер имитирует поведение программы для перевода слов. Для этого определен словарь dit, который содержит англоязычные слова и их перевод.

В бесконечном цикле сервер принимает подключения. Однако вместо прямой обработки подключения сервер запускает горутину в виде функции handleConnection, в которой и обрабатывается подключение. Это позволит входящим клиентам не ждать, пока первый из них будет обработан. Таким образом, все входящие клиенты в определенной степени будут обрабатываться одновременно.

В функции handleConnection получаем запрос от клиента. Для этого выделяем буфер достаточной длины в 4096 байт.

```
input := make([]byte, (1024 * 4))
n, err := conn.Read(input)
```

В данном случае мы ожидаем, что запрос от клиента не превысит 4096 байт, однако точный размер запроса и его максимальный размер не всегда бывают известны. В этом случае мы можем применять различные техники, в частности, в бесконечном цикле считывать данные запроса от клиента и только потом их обрабатывать. Но в данном случае мы разберем более простую ситуацию.

Получив запрос, преобразовав его строку, получаем значение из словаря и отправляем его обратно клиенту:

```
conn.Write([]byte(target))
```

Для взаимодействия с этим сервером определим следующий клиент:

```
package main
import (
    "fmt"
    "net"
)
func main() {
    conn, err := net.Dial("tcp", "127.0.0.1:4545")
    if err != nil {
        fmt.Println(err)
        return
    }
}
```

```

defer conn.Close()
for{
    var source string
    fmt.Print("Введите слово: ")
    _, err := fmt.Scanln(&source)
    if err != nil {
        fmt.Println("Некорректный ввод", err)
        continue
    }
    // отправляем сообщение серверу
    if n, err := conn.Write([]byte(source));
    n == 0 || err != nil {
        fmt.Println(err)
        return
    }
    // получем ответ
    fmt.Print("Перевод:")
    buff := make([]byte, 1024)
    n, err := conn.Read(buff)
    if err != nil{ break}
    fmt.Print(string(buff[0:n]))
    fmt.Println()
}
}

```

На клиенте в бесконечном цикле вводим слово для перевода и отправляем серверу сообщение:

```
if n, err := conn.Write([]byte(source));
```

И затем получаем от сервера ответ и выводим его на консоль. Так как ответ от сервера может быть переменной длины, то для получения ответа в бесконечном цикле считываем данные с помощью метода Read:

```

buff := make([]byte, 1024)
n, err := conn.Read(buff)
if err != nil{ break}
fmt.Print(string(buff[0:n]))

```

Запустим сервер.

Затем запустим программу клиента и введем какое-либо значение и сервер возвратит перевод слова:

Для выхода из программ сервера и клиента необходимо нажать комбинацию клавиш Ctrl+C.

Установка таймута

Последнее обновление: 18.01.2018

При взаимодействии клиента и сервера мы можем устанавливать таймаут, по истечении которого соединение между сервером и клиентом при отсутствии взаимодействия будет разорвано. Для этого у типа net.Conn определены следующие методы:

- SetDeadline(t time.Time) error : устанавливает таймаут на все операции ввода-вывода. Для установки времени применяется структура time.Time
- SetReadDeadline(t time.Time) error : устанавливает таймаут на операции ввода в поток
- SetWriteDeadline(t time.Time) error : устанавливает таймаут на операции вывода из потока

В каком случае они могут пригодиться? В прошлой теме было рассмотрено взаимодействие сервера и клиента. Для чтения данных от клиента сервер использовал буфер фиксированного размера:

```

input := make([]byte, (1024 * 4))
n, err := conn.Read(input)

```

Однако в ряде ситуаций это не лучший способ, особенно когда размер передаваемых данных превышает размер буфера. Мы можем точно не знать, сколько данных возвратит нам сервер. Поэтому определим следующий код клиента:

```

package main
import (
    "fmt"
    "net"
    "time"
)
func main() {

```

```

conn, err := net.Dial("tcp", "127.0.0.1:4545")
if err != nil {
    fmt.Println(err)
    return
}
defer conn.Close()
for{
    var source string
    fmt.Print("Введите слово: ")
    _, err := fmt.Scanln(&source)
    if err != nil {
        fmt.Println("Некорректный ввод", err)
        continue
    }
    // отправляем сообщение серверу
    if n, err := conn.Write([]byte(source));
    n == 0 || err != nil {
        fmt.Println(err)
        return
    }
    // получем ответ
    fmt.Print("Перевод:")
    conn.SetReadDeadline(time.Now().Add(time.Second * 5))
    for{
        buff := make([]byte, 1024)
        n, err := conn.Read(buff)
        if err != nil{ break}
        fmt.Print(string(buff[0:n]))
        conn.SetReadDeadline(time.Now().Add(time.Millisecond * 700))
    }
    fmt.Println()
}
}

```

Теперь получение данных выделено в отдельный цикл for:

```

for{
    buff := make([]byte, 1024)
    n, err := conn.Read(buff)
    if err != nil{ break}
    fmt.Print(string(buff[0:n]))
    conn.SetReadDeadline(time.Now().Add(time.Millisecond * 700))
}

```

Поэтому даже если сервер передал больше 1024 байт, все они все равно будут обработаны. Но кроме того, здесь также устанавливается таймаут на чтение данных. Перед самим циклом устанавливается таймаут в 5 секунд:

```
conn.SetReadDeadline(time.Now().Add(time.Second * 5))
```

Это значит, что клиент может ожидать данные на чтение от сервера в течении 5 секунд. По истечении этого времени операция чтения генерирует ошибку и соответственно происходит выход из цикла, где мы пытаемся прочитать данные от сервера. 5 секунд - довольно большой период, но в начале перед первым взаимодействием лучше устанавливать период побольше. И после прочтения первых 1024 байт таймаут сбрасывается до 700 миллисекунд. То есть если в течение последующих 700 миллисекунд сервер не пришлет никаких данных, то происходит выход из цикла и соответственно чтение данных заканчивается.

Важно понимать роль подобных задержек, так как они позволяют сгенерировать ошибку при чтении данных. А значит мы можем получить эту ошибку и должным образом обработать ее, например, выйти из бесконечного цикла. Если бы мы не использовали установку таймаута, то могла бы сложиться ситуация, когда сервер ожидал данных от клиента в операции чтения, а клиент ожидал данных от сервера также в операции чтения. И была бы своего рода блокировка.

Код сервера остается тем же, что и в прошлой теме:

```

package main
import (
    "fmt"
    "net"
)
var dict = map[string]string{
    "red": "красный",
    "green": "зеленый",
    "blue": "синий",
    "yellow": "желтый",
}

func main() {
    listener, err := net.Listen("tcp", ":4545")
    if err != nil {
        fmt.Println(err)
        return
    }
    defer listener.Close()
    fmt.Println("Server is listening...")
    for {
        conn, err := listener.Accept()
        if err != nil {
            fmt.Println(err)
            conn.Close()
            continue
        }
        go handleConnection(conn) // запускаем горутину для обработки запроса
    }
}
// обработка подключения
func handleConnection(conn net.Conn) {
    defer conn.Close()
    for {
        // считываем полученные в запросе данные
        input := make([]byte, (1024 * 4))
        n, err := conn.Read(input)
    }
}

```

```

    if n == 0 || err != nil {
        fmt.Println("Read error:", err)
        break
    }
    source := string(input[0:n])
    // на основании полученных данных получаем из словаря перевод
    target, ok := dict[source]
    if ok == false{
        target = "undefined" // если данные не найдены в словаре
    }
    // выводим на консоль сервера диагностическую информацию
    fmt.Println(source, "-", target)
    // отправляем данные клиенту
    conn.Write([]byte(target))
}
}

```

Отправка запросов по HTTP

Последнее обновление: 20.01.2018

Особую область применения Go представляют запросы по протоколу HTTP. Протокол HTTP работает поверх TCP, и технически мы можем написать приложение, которое принимает или отправляет запросы по протоколу TCP и тем самым отправлять и получать запросы и по протоколу HTTP. Однако в связи с тем, что данный протокол и в целом сфера веб играет большую роль, то все соответствующие функции по работе с http были выделены в отдельный пакет net/http.

Для отправки запросов в пакете net/http определен ряд функций:

```

func Get(url string) (resp *Response, err error)
func Head(url string) (resp *Response, err error)
func Post(url string, contentType string, body io.Reader) (resp *Response, err error)
func PostForm(url string, data url.Values) (resp *Response, err error)

```

- Get() : отправляет запрос GET
- Head() : отправляет запрос HEAD
- Post() : отправляет запрос POST
- PostForm() : отправляет форму в запросе POST

Рассмотрим выполнение самого простого запроса - запроса GET, для которого применяется одноименный метод:

```

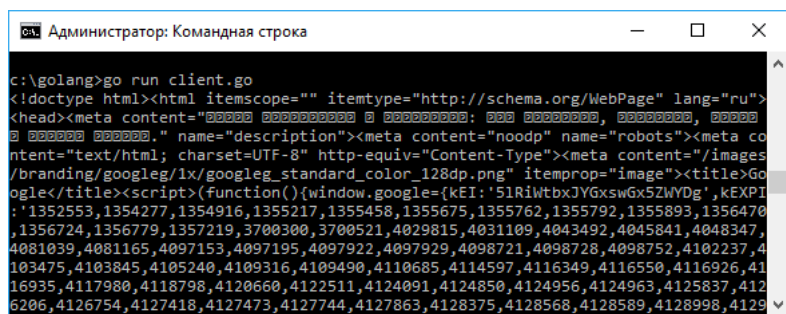
package main
import (
    "fmt"
    "net/http"
)
func main() {
    resp, err := http.Get("https://google.com")
    if err != nil {
        fmt.Println(err)
        return
    }
    defer resp.Body.Close()
    for true {

        bs := make([]byte, 1014)
        n, err := resp.Body.Read(bs)
        fmt.Println(string(bs[:n]))

        if n == 0 || err != nil{
            break
        }
    }
}

```

Метод Get() в качестве параметра принимает адрес ресурса, к которому надо выполнить запрос, и возвращает объект *http.Response, который инкапсулирует ответ. Поле Body структуры http.Response представляет ответ от веб-ресурса и при этом также представляет интерфейс io.ReadCloser. А это значит, что это поле по сути является потоком для чтения, и мы можем считать пришедшие данные через метод Read. И кроме того, для того, чтобы закрыть поток, необходимо вызвать метод Close. Поэтому после запроса вызывается метод defer resp.Body.Close() и в цикле считываем через метод Read данные и выводим на консоль.



Поскольку в данном случае ответ от веб-ресурса все равно выводится на консоль, то мы можем сократить код:

```

package main
import (
    "fmt"
    "net/http"
    "io"
    "os"
)

```

```
func main() {
    resp, err := http.Get("https://google.com")
    if err != nil {
        fmt.Println(err)
        return
    }
    defer resp.Body.Close()
    io.Copy(os.Stdout, resp.Body)
}
```

http.Client

Последнее обновление: 20.01.2018

Для осуществления HTTP-запросов также может применяться структура `http.Client`. Чтобы отправить запрос к веб-ресурсу, можно использовать один из ее методов:

```
func (c *Client) Do(req *Request) (*Response, error)
func (c *Client) Get(url string) (resp *Response, err error)
func (c *Client) Head(url string) (resp *Response, err error)
func (c *Client) Post(url string, contentType string, body io.Reader) (resp *Response, err error)
func (c *Client) PostForm(url string, data url.Values) (resp *Response, err error)
```

Во многом они аналогичны тем одноименным функциям (за исключением метода `Do`), которые определены в пакете `net/http` и которые были рассмотрены в прошлой теме. Например, выполнение самого простого запроса `GET`:

```
package main
import (
    "fmt"
    "net/http"
    "io"
    "os"
)
func main() {
    client := http.Client{}
    resp, err := client.Get("https://google.com")
    if err != nil {
        fmt.Println(err)
        return
    }
    defer resp.Body.Close()
    io.Copy(os.Stdout, resp.Body)
}
```

Настройка клиента

Структура `http.Client` имеет ряд полей, которые позволяют настроить ее поведение:

- `Timeout` : устанавливает таймаут для запроса
- `Jar` : устанавливает куки, отправляемые в запросе
- `Transport` : определяет механизм выполнения запроса

Установка таймаута:

```
package main
import (
    "fmt"
    "net/http"
    "io"
    "os"
    "time"
)
func main() {
    client := http.Client{
        Timeout: 6 * time.Second,
    }
    resp, err := client.Get("https://google.com")
    if err != nil {
        fmt.Println(err)
        return
    }
    defer resp.Body.Close()
    io.Copy(os.Stdout, resp.Body)
}
```

Свойство `Timeout` представляет объект `time.Duration`, и в данном случае оно равно 6 секундам.

Request

Для управления запросом и его параметрами в Go используется объект `http.Request`. Он позволяет установить различные настройки, добавить куки, заголовки, определить тело запроса. Для создания объекта `http.Request` применяется функция `http.NewRequest()`:

```
func NewRequest(method, url string, body io.Reader) (*Request, error)
```

Функция принимает три параметра. Первый параметр - тип запроса в виде строки ("GET", "POST"). Второй параметр - адрес ресурса. Третий параметр - тело запроса.

Для отправки объекта `Request` можно применять метод `Do()`:

```
Do(req *http.Request) (*http.Response, error)
```

Например:

```
package main
import (
    "fmt"
    "net/http"
    "io"
    "os"
)
func main() {
    client := &http.Client{}
    req, err := http.NewRequest(
        "GET", "https://google.com", nil,
    )
    // добавляем заголовки
    req.Header.Add("Accept", "text/html") // добавляем заголовок Accept
    req.Header.Add("User-Agent", "MSIE/15.0") // добавляем заголовок User-Agent

    resp, err := client.Do(req)
    if err != nil {
        fmt.Println(err)
        return
    }
    defer resp.Body.Close()
    io.Copy(os.Stdout, resp.Body)
}
```

Базы данных

Работа с реляционными база данных

Последнее обновление: 31.01.2018

Для работы с реляционными базами данных в языке Go применяется встроенный пакет [database/sql](#). Однако он не используется сам по себе. Он лишь предоставляет универсальный интерфейс для работы с базами данных. Для работы с конкретной СУБД нам также необходим драйвер. Список доступных драйверов можно найти [здесь](#). Однако поскольку драйвера реализуют одни и те же интерфейсы, то в принципе работа с различными СУБД будет идентична.

Для того, чтобы начать работу с базой данных, необходимо открыть подключение с помощью функции `Open()` :

```
func Open(driverName, dataSourceName string) (*DB, error)
```

Эта функция принимает в качестве параметров имя драйвера и имя источника данных, к которому надо подключаться. Возвращает функция объект `DB` - по сути базу данных, с которой мы можем работать. Если не удалось подключить к источнику данных, то в объекте `error` мы сможем найти сведения об ошибке.

Затем взаимодействие с базой данных осуществляется посредством методов объекта `DB`.

```
func (db *DB) Exec(query string, args ...interface{}) (Result, error)
func (db *DB) Query(query string, args ...interface{}) (*Rows, error)
func (db *DB) QueryRow(query string, args ...interface{}) *Row
func (db *DB) Close() error // закрывает подключение
```

- Метод `Exec()` выполняет некоторое sql-выражение, которое передается через первый параметр, не возвращая никакого результата. Метод также принимает дополнительные параметры, с помощью которых можно передать значения в выполняемое sql-выражение. Например, абстрактная операция добавления данных в БД, которая предполагает выполнение команды `INSERT`:

```
result, err := db.Exec("INSERT INTO Products (model, company, price) VALUES ('iPhone X', 'Apple', 72000)")
```

Способ вставки дополнительных параметров в SQL-выражение зависит от конкретного драйвера. Также этот метод подходит для выполнения команд `UPDATE` (обновление) и `DELETE` (удаление).

Метод возвращает объект `Result`. Это интерфейс определяет два метода:

```
LastInsertId() (int64, error) // возвращает id последней строки, которая была добавлена/обновлена/удалена
RowsAffected() (int64, error) // возвращает количество затронутых строк
```

- Метод `Query()` для выполнения запроса, который возвращает какие-либо данные. Обычно это запросы, которые содержат команду `SELECT`.

```
rows, err := db.Query("SELECT name FROM users WHERE age=23")
```

Результатом запроса является объект `*Rows` - по сути набор строк. С помощью ряда его методов можно извлечь полученные данные:

```
func (rs *Rows) Columns() ([]string, error) // возвращает названия столбцов набора
func (rs *Rows) Next() bool // возвращает true если в наборе есть еще одна строка и пер
func (rs *Rows) Scan(dest ...interface{}) error // считывает данные строки в переменные
func (rs *Rows) Close() error // закрывает объект Rows для дальнейшего чтения
```

Общий принцип чтения набора строк выглядит примерно следующим образом:

```
rows, err := db.Query("SELECT ...")
...
defer rows.Close()
for rows.Next() {
    var id int
    var name string
    rows.Scan(&id, &name)
    fmt.Println(id, name)
}
```

Вначале выполняем запрос к базе данных с помощью метода `db.Query`, затем с помощью метода `Next()` последовательно считываем все строки из набора. Если строк в наборе нет, то метод возвращает `false`, и происходит выход из цикла. Если строки еще есть, то указатель `*Rows` переходит к следующей строке. И затем мы можем считать в переменные с помощью метода `Scan()` данные из текущей строки.

- Метод `QueryRow()` возвращает одну строку в виде объекта `*Row`. Как правило, этот метод применяется для получения единичного объекта, например, по `id`. Этот объект имеет метод `Scan()`, который позволяет извлечь данные из строки:


```
func (r *Row) Scan(dest ...interface{}) error
```

Также стоит отметить, что язык Go поддерживает создание запросов с помощью объекта Stmt, в который можно вводить различные данные и который повышает производительность. И также в Go имеется поддержка транзакций в виде объекта Tx.

Все эти вещи по разному реализуются в различных драйверах для конкретных систем управления базами данных. Но общие принципы будут одни и те же. То есть общая структура работы с различными базами данных благодаря единому интерфейсу будут совпадать.

MySQL

Последнее обновление: 31.01.2018

Для работы с MySQL будем использовать драйвер [Go MySQL Driver](#). Прежде всего нам надо добавить данный драйвер к переменной GOPATH. Для этого выполним в командной строке/терминале следующую команду:

```
go get github.com/go-sql-driver/mysql
```

Вначале создадим на сервере MySQL базу данных productdb и в ней таблицу products. Для этого можно использовать следующие выражения SQL

```
create database productdb;
use productdb;
create table products (
    id int auto_increment primary key,
    model varchar(30) not null,
    company varchar(30) not null,
    price int not null
)
```

То есть база данных productdb, в ней есть таблица products, которая будет хранить информацию о товарах, будет 4 столбца: id - идентификатор каждой записи, model - название товара, company - производитель товара и price - цена товара.

Добавление данных

Созданная база данных пуста, поэтому добавим в нее какие-нибудь данные:

```
package main
import (
    "database/sql"
    "fmt"
    _ "github.com/go-sql-driver/mysql"
)

func main() {
    db, err := sql.Open("mysql", "root:password@productdb")

    if err != nil {
        panic(err)
    }
    defer db.Close()

    result, err := db.Exec("insert into productdb.Products (model, company, price) values (?, ?, ?)",
        "iPhone X", "Apple", 72000)
    if err != nil {
        panic(err)
    }
    fmt.Println(result.LastInsertId()) // id добавленного объекта
    fmt.Println(result.RowsAffected()) // количество затронутых строк
}
```

Вначале подключаем все нужные нам пакеты. Для работы с реляционной базой данных необходим пакет "database/sql". И так как мы используем mysql, то также подключаем пакет "github.com/go-sql-driver/mysql", причем обратите внимание, что перед ним стоит знак подчеркивания. Этот знак позволяет при загрузке пакета инициализировать его с помощью вызова функции init.

Далее открываем подключение функцией Open:

```
sql.Open("mysql", "root:password@productdb")
```

Первый аргумент функции - название драйвера, в данном случае это "mysql". Второй параметр имеет форму "логин:пароль@база_данных". Логин и пароль должны быть те, которые были указаны для mysql при его установке. В моем случае логин - root, а пароль - password. Название базы данных - productdb - та, которая была создана выше.

Эта функция возвращает объект базы данных - DB. И для добавления данных у этого объекта вызывается метод Exec() :

```
result, err := db.Exec("insert into productdb.Products (model, company, price) values (?, ?, ?)",
    "iPhone X", "Apple", 72000)
```

Первый аргумент функции - это sql-выражение, которое добавляет строку в таблицу Products. Знаки вопроса в этом выражении представляют плейсхолдеры, вместо которых вставляются значения, которые передаются через второй, третий и последующие параметры. То есть в таблице Products четыре столбца, но один из них автогенерируемый - id. Поэтому передаем три значения - для столбцов model, company и price. Поэтому в выражении три знака вопроса и соответственно функция принимает три дополнительных параметра. Все дополнительные параметры передаются в sql-выражение на место плейсхолдеров по позиции - первый параметр вместо первого плейсхолдера и так далее.

Результат выполнения функции попадает в переменную result, которая хранит результат выполнения операции в базе данных. В частности, через метод result.LastInsertId() мы можем получить id последнего добавленного объекта, а с помощью метода result.RowsAffected() - количество добавленных строк.

И при выполнении данного скрипта получим следующие результаты:

```

C:\golang>go run main.go
1 <nil>
1 <nil>
C:\golang>

```

При этом необязательно определять все добавляемые данные через параметры, можно ввести их напрямую в sql-выражение:

```
result, err := db.Exec("insert into productdb.Products (model, company, price) values ('Pixel 2', 'Google', 64000)")
```

Получение данных

Теперь получим ранее добавленные данные:

```

package main
import (
    "database/sql"
    "fmt"
    _ "github.com/go-sql-driver/mysql"
)

type product struct{
    id int
    model string
    company string
    price int
}

func main() {
    db, err := sql.Open("mysql", "root:password@/productdb")

    if err != nil {
        panic(err)
    }
    defer db.Close()
    rows, err := db.Query("select * from productdb.Products")
    if err != nil {
        panic(err)
    }
    defer rows.Close()
    products := []product{}

    for rows.Next(){
        p := product{}
        err := rows.Scan(&p.id, &p.model, &p.company, &p.price)
        if err != nil{
            fmt.Println(err)
            continue
        }
        products = append(products, p)
    }
    for _, p := range products{
        fmt.Println(p.id, p.model, p.company, p.price)
    }
}

```

Для работы с данными здесь определена структура product, которая соответствует данным в таблице Products.

Для получения данных вызывается метод Query() :

```
rows, err := db.Query("select * from productdb.Products")
```

Этот метод в качестве параметра принимает sql-выражение SELECT на получение всех данных из таблицы Products. Результат выборки попадает в переменную rows, которая представляет указатель на структуру Rows . И с помощью метода rows.Next() мы можем последовательно перебрать все строки в полученном наборе:

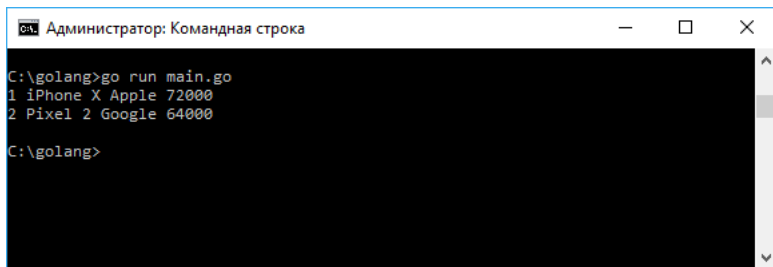
```

for rows.Next(){
    p := product{}
    err := rows.Scan(&p.id, &p.model, &p.company, &p.price)
    if err != nil{
        fmt.Println(err)
        continue
    }
    products = append(products, p)
}

```

Тип Rows определяет метод Scan, с помощью которого можно считать все полученные данные в переменные. Например, здесь считываем данные в структуру Product и затем добавляем ее в срез. Поскольку мы получаем все данные - все четыре столбца, то соответственно в Scan передается адреса четырех переменных.

После прочтения данных из среза мы можем делать с ними все что угодно, например, вывести на консоль:



```

Администратор: Командная строка
C:\golang>go run main.go
1 iPhone X Apple 72000
2 Pixel 2 Google 64000
C:\golang>

```

В методе Query мы можем указывать дополнительные параметры. Например, получим товары, у которых цена больше 70000:

```
rows, err := db.Query("select * from productdb.Products where price > ?", 70000)
```

Если надо получить только одну строку, то можно использовать метод QueryRow() :

```

row := db.QueryRow("select * from productdb.Products where id = ?", 2)
prod := product{}
err = row.Scan(&prod.id, &prod.model, &prod.company, &prod.price)
if err != nil {
    panic(err)
}
fmt.Println(prod.id, prod.model, prod.company, prod.price)

```

Обновление

Для обновления данных применяется метод Exec:

```

package main
import (
    "database/sql"
    "fmt"
    _ "github.com/go-sql-driver/mysql"
)

func main() {
    db, err := sql.Open("mysql", "root:password@productdb")
    if err != nil {
        panic(err)
    }
    defer db.Close()
    // обновляем строку с id=1
    result, err := db.Exec("update productdb.Products set price = ? where id = ?", 69000, 1)
    if err != nil {
        panic(err)
    }
    fmt.Println(result.LastInsertId())
    fmt.Println(result.RowsAffected())
}

```

Удаление

Для удаления также применяется метод Exec:

```

package main
import (
    "database/sql"
    "fmt"
    _ "github.com/go-sql-driver/mysql"
)

func main() {
    db, err := sql.Open("mysql", "root:password@productdb")
    if err != nil {
        panic(err)
    }
    defer db.Close()
    result, err := db.Exec("delete from productdb.Products where id = 1")
    if err != nil {
        panic(err)
    }
    fmt.Println(result.LastInsertId()) // id последнего удаленного объекта
    fmt.Println(result.RowsAffected()) // количество затронутых строк
}

```

PostgreSQL

Последнее обновление: 31.01.2018

Для работы с PostgreSQL в Go мы можем применять различные драйверы, но в данном случае мы будем использовать [Pure Go Postgres driver](#).

Поэтому начле перейдем в командную строку/терминал и установим данный драйвер с помощью команды

```
go get github.com/lib/pq
```

Пусть на сервере PostgreSQL будет база данных productdb, в которой есть таблица Products, описываемая следующим скриптом:

```

CREATE TABLE Products (
    id integer PRIMARY KEY GENERATED BY DEFAULT AS IDENTITY,
    model varchar(30) NOT NULL,
    company varchar(30) NOT NULL,

```

```
    price integer NOT NULL
);
```

То есть в таблице будет четыре столбца: id, model, company, price.

Открытие подключения

Для открытия соединения с базой данных в функцию `sql.Open()` передается имя драйвера "postgres" и строка подключения:

```
connStr := "user=postgres password=mypass dbname=productdb sslmode=disable"
db, err := sql.Open("postgres", connStr)
```

Строка подключения инкапсулирует следующие параметры: `user` (логин на сервере PostgreSQL), `password` (пароль этого пользователя), `dbname` (имя базы данных), `sslmode` (режим работы с SSL). В моем случае логин - postgres (логин по умолчанию на сервере), пароль - mypass, имя базы данных - productdb (которая была создана выше) и ssl отключен (значение disable).

В результате установки подключения метод `db.Open` возвращает объект `*DB`, через который можно будет взаимодействовать в базой данных.

Это не все возможные параметры. Полный список параметров для строки подключения и их значения можно посмотреть в [документации](#).

Добавление данных

Для добавления используется метод `Exec()`:

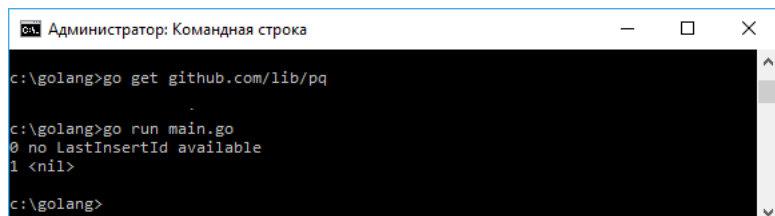
```
package main
import (
    "database/sql"
    "fmt"
    _ "github.com/lib/pq"
)

func main() {
    connStr := "user=postgres password=mypass dbname=productdb sslmode=disable"
    db, err := sql.Open("postgres", connStr)
    if err != nil {
        panic(err)
    }
    defer db.Close()

    result, err := db.Exec("insert into Products (model, company, price) values ('iPhone X', $1, $2)",
        "Apple", 72000)
    if err != nil {
        panic(err)
    }
    fmt.Println(result.LastInsertId()) // не поддерживается
    fmt.Println(result.RowsAffected()) // количество добавленных строк
}
```

Выполняемое sql-выражение может получать значения через дополнительные параметры метода `db.Exec()`. В самом sql-выражении такие значения представлены плейсхолдерами \$1, \$2 и так далее, вместо которых вставляются значения дополнительных параметров метода `db.Exec`.

Стоит обратить внимание, что этот драйвер не поддерживает метод `result.LastInsertId()`, который возвращает id последнего добавленного объекта:



```
Администратор: Командная строка

c:\golang>go get github.com/lib/pq

c:\golang>go run main.go
0 no LastInsertId available
1 <nil>

c:\golang>
```

Если нам обязательно нужно получить id добавленного объекта, то мы можем использовать метод `db.QueryRow()`, который выполняет запрос и возвращает определенный объект:

```
var id int
db.QueryRow("insert into Products (model, company, price) values ('Mate 10 Pro', $1, $2) returning id",
    "Huawei", 35000).Scan(&id)
fmt.Println(id)
```

В само sql выражение вводится подвыражение "returning id". И с помощью метода `Scan()` полученное значение считывается в переменную `id`.

Получение данных

Для получения данных применяется метод `db.Query()`, который возвращает набор строк, либо `db.QueryRow()`, который возвращает одну строку:

```
package main
import (
    "database/sql"
    "fmt"
    _ "github.com/lib/pq"
)

type product struct{
    id int
    model string
    company string
    price int
}

func main() {
    connStr := "user=postgres password=mypass dbname=productdb sslmode=disable"
    db, err := sql.Open("postgres", connStr)
    if err != nil {
```

```

        panic(err)
    }
    defer db.Close()

    rows, err := db.Query("select * from Products")
    if err != nil {
        panic(err)
    }
    defer rows.Close()
    products := []product{}

    for rows.Next(){
        p := product{}
        err := rows.Scan(&p.id, &p.model, &p.company, &p.price)
        if err != nil{
            fmt.Println(err)
            continue
        }
        products = append(products, p)
    }
    for _, p := range products{
        fmt.Println(p.id, p.model, p.company, p.price)
    }
}

```

Для работы с данными здесь определена структура product, которая соответствует данным в таблице Products.

Для получения данных вызывается метод Query() :

```
rows, err := db.Query("select * from Products")
```

Этот метод в качестве параметра принимает sql-выражение SELECT на получение всех данных из таблицы Products. Результат выборки попадает в переменную rows, которая представляет указатель на структуру Rows . И с помощью метода rows.Next() мы можем последовательно перебрать все строки в полученном наборе:

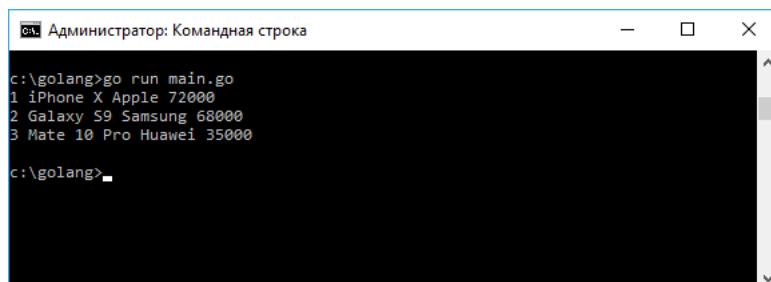
```

for rows.Next(){
    p := product{}
    err := rows.Scan(&p.id, &p.model, &p.company, &p.price)
    if err != nil{
        fmt.Println(err)
        continue
    }
    products = append(products, p)
}

```

Тип Rows определяет метод Scan, с помощью которого можно считать все полученные данные в переменные. Например, здесь считываем данные в структуру Product и затем добавляем ее в срез. Поскольку мы получаем все данные - все четыре столбца, то соответственно в Scan передается адреса четырех переменных.

После прочтения данных в срез мы можем делать с ними все что угодно, например, вывести на консоль:



```

c:\golang>go run main.go
1 iPhone X Apple 72000
2 Galaxy S9 Samsung 68000
3 Mate 10 Pro Huawei 35000
c:\golang>

```

В методе Query мы можем указывать дополнительные параметры. Например, получим товары, у которых цена больше 70000:

```
rows, err := db.Query("select * from Products where price > $1", 70000)
```

Если надо получить только одну строку, то можно использовать метод QueryRow() :

```

row := db.QueryRow("select * from Products where id = $1", 2)
prod := product{}
err = row.Scan(&prod.id, &prod.model, &prod.company, &prod.price)
if err != nil{
    panic(err)
}
fmt.Println(prod.id, prod.model, prod.company, prod.price)

```

Обновление

Для обновления данных применяется метод Exec:

```

package main
import (
    "database/sql"
    "fmt"
    _ "github.com/lib/pq"
)
func main() {

    connStr := "user=postgres password=mypass dbname=productdb sslmode=disable"
    db, err := sql.Open("postgres", connStr)
    if err != nil {
        panic(err)
    }
    defer db.Close()

    // обновляем строку с id=1

```

```

    result, err := db.Exec("update Products set price = $1 where id = $2", 69000, 1)
    if err != nil {
        panic(err)
    }
    fmt.Println(result.RowsAffected())    // количество обновленных строк
}

```

Удаление

Для удаления также применяется метод Exec:

```

package main
import (
    "database/sql"
    "fmt"
    _ "github.com/lib/pq"
)
func main() {

    connStr := "user=postgres password=mypass dbname=productdb sslmode=disable"
    db, err := sql.Open("postgres", connStr)
    if err != nil {
        panic(err)
    }
    defer db.Close()

    // удаляем строку с id=2
    result, err := db.Exec("delete from Products where id = $1", 2)
    if err != nil {
        panic(err)
    }
    fmt.Println(result.RowsAffected())    // количество удаленных строк
}

```

SQLite

Последнее обновление: 31.01.2018

Для работы с SQLite в Go нам потребуется драйвер [go-sqlite3](#). Для использования драйвера вначале установим его, выполнив в командной строке/терминале следующую команду:

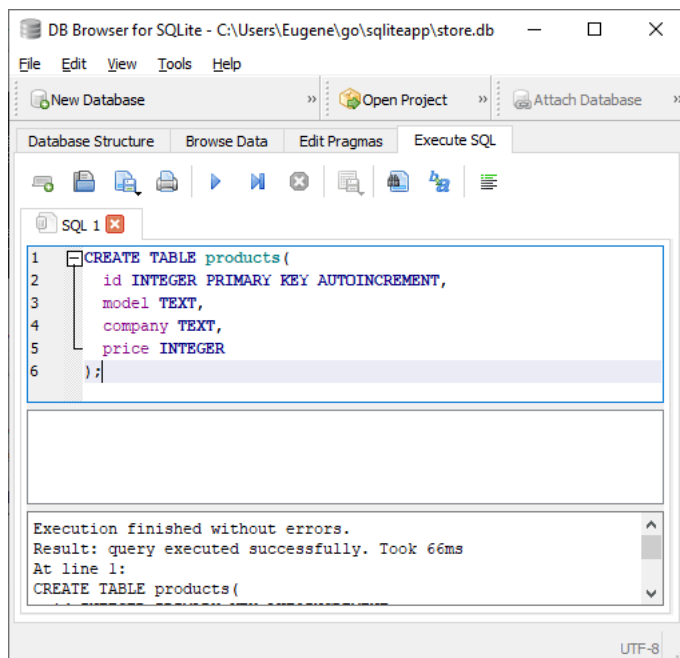
```
go get github.com/mattn/go-sqlite3
```

Пусть в папке со скриптом программы на Go у нас будет создана база данных SQLite, которая называется store.db, и в ней будет таблица products, которая описывается следующим скриптом:

```

CREATE TABLE products(
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    model TEXT,
    company TEXT,
    price INTEGER
);

```



Открытие подключения

Для открытия соединения с базой данных в функцию sql.Open() передается имя драйвера "sqlite3" и путь к файлу базы данных:

```
db, err := sql.Open("sqlite3", "store.db")
```

В результате установки подключения метод db.Open возвращает объект *DB, через который можно будет взаимодействовать в базой данных.

Добавление данных

Для добавления используется метод Exec() объекта DB:

```
package main
import (
    "database/sql"
    "fmt"
    _ "github.com/mattn/go-sqlite3"
)
func main() {
    db, err := sql.Open("sqlite3", "store.db")
    if err != nil {
        panic(err)
    }
    defer db.Close()
    result, err := db.Exec("insert into products (model, company, price) values ('iPhone X', $1, $2)",
        "Apple", 72000)
    if err != nil {
        panic(err)
    }
    fmt.Println(result.LastInsertId()) // id последнего добавленного объекта
    fmt.Println(result.RowsAffected()) // количество добавленных строк
}
```

Через дополнительные параметры метода `db.Exec()` можно передавать значения выполняемому sql-выражению через плейхолдеры `$1`, `$2` и так далее, вместо которых вставляются значения дополнительных параметров метода `db.Exec`.

У результата операции есть метод `result.LastInsertId()`, который возвращает id последнего добавленного объекта, и метод `result.RowsAffected()`, который позволяет получить количество добавленных строк:

```
Администратор: Командная строка
c:\Users\Eugene\go\sqliteapp>go get github.com/mattn/go-sqlite3
c:\Users\Eugene\go\sqliteapp>go run main.go
1 <nil>
1 <nil>
c:\Users\Eugene\go\sqliteapp>
```

Получение данных

Для получения данных применяется метод `db.Query()`, который возвращает набор строк, либо `db.QueryRow()`, который возвращает одну строку:

```
package main
import (
    "database/sql"
    "fmt"
    _ "github.com/mattn/go-sqlite3"
)
type product struct{
    id int
    model string
    company string
    price int
}
func main() {
    db, err := sql.Open("sqlite3", "store.db")
    if err != nil {
        panic(err)
    }
    defer db.Close()
    rows, err := db.Query("select * from Products")
    if err != nil {
        panic(err)
    }
    defer rows.Close()
    products := []product{}

    for rows.Next(){
        p := product{}
        err := rows.Scan(&p.id, &p.model, &p.company, &p.price)
        if err != nil {
            fmt.Println(err)
            continue
        }
        products = append(products, p)
    }
    for _, p := range products{
        fmt.Println(p.id, p.model, p.company, p.price)
    }
}
```

Для работы с данными здесь определена структура `product`, которая соответствует данным в таблице `Products`.

Для получения данных вызывается метод `Query()`:

```
rows, err := db.Query("select * from Products")
```

Этот метод в качестве параметра принимает sql-выражение `SELECT` на получение всех данных из таблицы `Products`. Результат выборки попадает в переменную `rows`, которая представляет указатель на структуру `Rows`. И с помощью метода `rows.Next()` мы можем последовательно перебрать все строки в полученном наборе:

```
for rows.Next(){
    p := product{}
    err := rows.Scan(&p.id, &p.model, &p.company, &p.price)
```

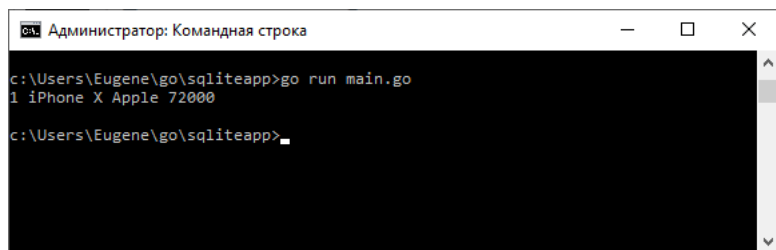
```

        if err != nil{
            fmt.Println(err)
            continue
        }
        products = append(products, p)
    }
}

```

Тип Rows определяет метод Scan, с помощью которого можно считать все полученные данные в переменные. Например, здесь считываем данные в структуру Product и затем добавляем ее в срез. Поскольку мы получаем все данные - все четыре столбца, то соответственно в Scan передается адреса четырех переменных.

После прочтения данных в срез мы можем делать с ними все что угодно, например, вывести на консоль:



```

c:\Users\Eugene\go\sqliteapp>go run main.go
1 iPhone X Apple 72000
c:\Users\Eugene\go\sqliteapp>

```

В методе Query мы можем указывать дополнительные параметры. Например, получим товары, у которых цена больше 70000:

```
rows, err := db.Query("select * from Products where price > $1", 70000)
```

Если надо получить только одну строку, то можно использовать метод QueryRow() :

```

row := db.QueryRow("select * from Products where id = $1", 2)
prod := product{}
err = row.Scan(&prod.id, &prod.model, &prod.company, &prod.price)
if err != nil{
    panic(err)
}
fmt.Println(prod.id, prod.model, prod.company, prod.price)

```

Обновление

Для обновления данных применяется метод Exec:

```

package main
import (
    "database/sql"
    "fmt"
    _ "github.com/mattn/go-sqlite3"
)
func main() {

    db, err := sql.Open("sqlite3", "store.db")
    if err != nil {
        panic(err)
    }
    defer db.Close()

    // обновляем строку с id=1
    result, err := db.Exec("update Products set price = $1 where id = $2", 69000, 1)
    if err != nil{
        panic(err)
    }
    fmt.Println(result.RowsAffected())    // количество обновленных строк
}

```

Удаление

Для удаления также применяется метод Exec:

```

package main
import (
    "database/sql"
    "fmt"
    _ "github.com/mattn/go-sqlite3"
)
func main() {

    db, err := sql.Open("sqlite3", "store.db")
    if err != nil {
        panic(err)
    }
    defer db.Close()

    // удаляем строку с id=1
    result, err := db.Exec("delete from Products where id = $1", 1)
    if err != nil{
        panic(err)
    }
    fmt.Println(result.RowsAffected())    // количество удаленных строк
}

```

MongoDB

Последнее обновление: 31.01.2018

MongoDB не является реляционный СУБД, но тем не менее это тоже довольно распространенная система управления базами данных, которую можно использовать в Go.

Для работы с MongoDB нам потребуется драйвер [mgo](#).

Вначале установим драйвер, выполнив в командной строке/терминале следующую команду:

```
go get gopkg.in/mgo.v2
```

Подключение

Для подключения к серверу MongoDB необходимо использовать функцию `mgo.Dial()`, в которую передается адрес сервера:

```
func Dial(url string) (*Session, error)
```

Например, подключение к серверу на локальном компьютере:

```
session, err := mgo.Dial("mongodb://127.0.0.1")
```

Функция возвращает указатель на объект `Session`, который представляет текущую сессию.

Используя метод `DB` данного объекта, мы можем получить указатель на объект `Database`, который представляет конкретную базу данных на сервере.

```
func (s *Session) DB(name string) *Database
```

Все данные в базах данных MongoDB структурированы по коллекциям. Фактически коллекция - это аналог таблицы в реляционных базах данных, представлена типом `Collection`. И чтобы получить обратиться к нужной коллекции, необходимо использовать метод `C()`:

```
func (db *Database) C(name string) *Collection
```

Например, получим коллекцию "products", которая расположена в базе данных "productdb":

```
// открываем соединение
session, err := mgo.Dial("mongodb://127.0.0.1")
if err != nil {
    panic(err)
}
defer session.Close()

// получаем коллекцию products в базе данных productdb
productCollection := session.DB("productdb").C("products")
```

Получив коллекцию, мы сможем добавлять, получать данные и проводить с ними иные операции. И по завершении работы с сервером необходимо закрыть подключение методом `Close()`.

Добавление данных

Для добавления данных в коллекцию применяется метод `Insert()` объекта `Collection`:

```
func (c *Collection) Insert(docs ...interface{}) error
```

Этот метод принимает неопределенное количество добавляемых в коллекцию объектов.

```
package main
import (
    "fmt"
    "gopkg.in/mgo.v2"
    "gopkg.in/mgo.v2/bson"
)
type Product struct{
    Id bson.ObjectId `bson:"_id"`
    Model string `bson:"model"`
    Company string `bson:"company"`
    Price int `bson:"price"`
}
func main() {
    // открываем соединение
    session, err := mgo.Dial("mongodb://127.0.0.1")
    if err != nil {
        panic(err)
    }
    defer session.Close()

    // получаем коллекцию
    productCollection := session.DB("productdb").C("products")

    p1 := []bson.ObjectId{bson.NewObjectId(), Model:"iPhone 8", Company:"Apple", Price:64567}
    // добавляем один объект
    err = productCollection.Insert(p1)
    if err != nil {
        fmt.Println(err)
    }

    p2 := []bson.ObjectId{bson.NewObjectId(), Model:"Pixel 2", Company:"Google", Price:58000}
    p3 := []bson.ObjectId{bson.NewObjectId(), Model:"Xplay7", Company:"Vivo", Price:49560}
    // добавляем два объекта
    err = productCollection.Insert(p2, p3)
    if err != nil {
        fmt.Println(err)
    }
}
```

Прежде всего вначале импортируем два пакета драйвера, которые содержат весь необходимый нам функционал:

```
"gopkg.in/mgo.v2"
"gopkg.in/mgo.v2/bson"
```

Для представления данных здесь определена структура `Product`. Определение каждой переменной структуры кроме названия и типа данных содержит название поля в коллекции, с которым данная переменная будет сопоставляться. Например,

```
Model string `bson:"model"`
```

Переменная Model будет сопоставляться с полем "model" в коллекции. Причем между названием переменной и поля коллекции необязательно должно быть соответствие.

Также стоит отметить, что идентификатор объекта в MongoDB представляет специальный тип bson.ObjectId, а в базе данных ему соответствует поле "_id".

Для добавления создаем три объекта - фактически три указателя на объекты Product. Для создания уникального идентификатора применяется функция bson.ObjectId(). Затем добавляем объекты в коллекцию:

```
err = productCollection.Insert(p1)
err = productCollection.Insert(p2, p3)
```

Получение данных

Для получения данных из коллекции применяется метод Find() :

```
func (c *Collection) Find(query interface{}) *Query
```

В качестве параметра он принимает критерий выборки и возвращает объект *Query. Среди методов этого объекта следует выделить методы All() и One, которые возвращают соответственно все объекты выборки и один объект из выборки:

```
func (q *Query) All(result interface{}) error
func (q *Query) One(result interface{}) (err error)
```

В качестве параметра оба метода принимают указатель на объект, в который будет сохраняться результат выборки.

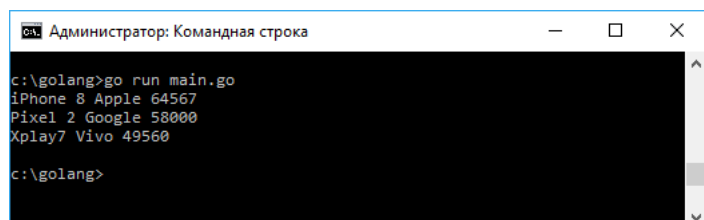
Например, получим ранее сохраненные объекты:

```
package main
import (
    "fmt"
    "gopkg.in/mgo.v2"
    "gopkg.in/mgo.v2/bson"
)
type Product struct{
    Id bson.ObjectId `bson:"_id"`
    Model string `bson:"model"`
    Company string `bson:"company"`
    Price int `bson:"price"`
}
func main() {
    // открываем соединение
    session, err := mgo.Dial("mongodb://127.0.0.1")
    if err != nil {
        panic(err)
    }
    defer session.Close()

    // получаем коллекцию
    productCollection := session.DB("productdb").C("products")
    // критерий выборки
    query := bson.M{}
    // объект для сохранения результата
    products := []Product{}
    productCollection.Find(query).All(&products)

    for _, p := range products{
        fmt.Println(p.Model, p.Company, p.Price)
    }
}
```

Критерий выборки представляет объект bson.M{}. Пустой объект bson.M{} охватывает все документы в коллекции. Все полученные документы передаются в объект products. И затем данные выводятся на консоль:



```

c:\golang>go run main.go
iPhone 8 Apple 64567
Pixel 2 Google 58000
Xplay7 Vivo 49560
c:\golang>

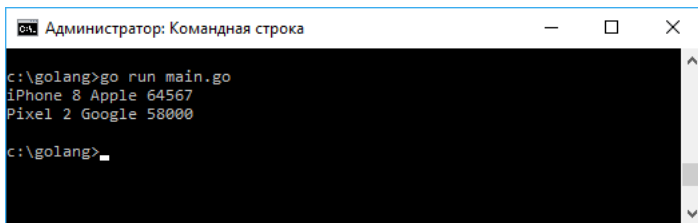
```

Также мы можем конкретизировать выборку:

```
query := bson.M{
    "price" : bson.M{
        "$gt":50000,
    },
}
products := []Product{}
productCollection.Find(query).All(&products)

for _, p := range products{
    fmt.Println(p.Model, p.Company, p.Price)
}
```

В данном случае ищем все документы, у которых поле "price" имеет значение больше 50000.



```

c:\golang>go run main.go
iPhone 8 Apple 64567
Pixel 2 Google 58000

c:\golang>

```

Обновление данных

Для обновления данных применяются методы Update()/UpdateAll() объекта Collection:

```

func (c *Collection) Update(selector interface{}, update interface{}) error
func (c *Collection) UpdateAll(selector interface{}, update interface{}) (info *ChangeInfo, err error)

```

Первый параметр методов выборки представляет критерий выборки документов, которые будут обновляться. Второй параметр указывает, как эти документы будут обновляться. Оба параметра задаются с помощью объекта bson.M. Однако если метод Update обновляет только один документ, который соответствует первому параметру, то метод UpdateAll - обновляет все элементы.

Например, изменим цену смартфона "iPhone 8":

```

package main
import (
    "fmt"
    "gopkg.in/mgo.v2"
    "gopkg.in/mgo.v2/bson"
)
type Product struct{
    Id bson.ObjectId `bson:"_id"`
    Model string `bson:"model"`
    Company string `bson:"company"`
    Price int `bson:"price"`
}
func main() {

    // открываем соединение
    session, err := mgo.Dial("mongodb://127.0.0.1")
    if err != nil {
        panic(err)
    }
    defer session.Close()

    // получаем коллекцию
    productCollection := session.DB("productdb").C("products")

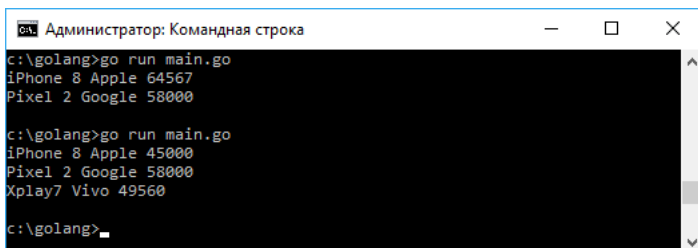
    // обновляем данные
    err = productCollection.Update(bson.M{"model": "iPhone 8"}, bson.M{"$set":bson.M{"price":45000}})
    if err != nil{
        fmt.Println(err)
    }

    products := []Product{}
    productCollection.Find(bson.M{}).All(&products)

    for _, p := range products{
        fmt.Println(p.Model, p.Company, p.Price)
    }
}

```

Первый аргумент метода Update - bson.M{"model": "iPhone 8"} указывает, что выбираются все элементы, у которых поле "model" равно "iPhone 8". Второй аргумент - bson.M{"\$set":bson.M{"price":45000}} с помощью параметра \$set устанавливает, какие значения будут иметь те или иные поля (в данном случае поле "price").



```

c:\golang>go run main.go
iPhone 8 Apple 45000
Pixel 2 Google 58000
Xplay7 Vivo 49560

c:\golang>

```

Удаление документов

Для удаления документов из коллекции применяется методы Remove()/RemoveAll() объекта Collection:

```

func (c *Collection) Remove(selector interface{}) error
func (c *Collection) RemoveAll(selector interface{}) (info *ChangeInfo, err error)

```

Оба метода в качестве параметра принимают критерий выборки документов, которые будут удаляться. Только метод Remove удаляет только один документ из выборки, а метод RemoveAll удаляет все элементы выборки.

Например, удалим все смартфоны компании Vivo:

```

package main
import (
    "fmt"
    "gopkg.in/mgo.v2"
    "gopkg.in/mgo.v2/bson"
)

```

```
type Product struct{
    Id bson.ObjectId `bson:"_id"`
    Model string `bson:"model"`
    Company string `bson:"company"`
    Price int `bson:"price"`
}
func main() {
    // открываем соединение
    session, err := mgo.Dial("mongodb://127.0.0.1")
    if err != nil {
        panic(err)
    }
    defer session.Close()

    // получаем коллекцию
    productCollection := session.DB("productdb").C("products")

    // удаляем все документы с company = "Vivo"
    _, err = productCollection.RemoveAll(bson.M{"company": "Vivo"})
    if err != nil{
        fmt.Println(err)
    }

    products := []Product{}
    productCollection.Find(bson.M{}).All(&products)

    for _, p := range products{
        fmt.Println(p.Model, p.Company, p.Price)
    }
}
```