BaseFunctionsGo_EN

Introduction

In this guide you can get the basic knowledge of Go. The reason for writing the guide was the complexity and confusion that I have noticed in other guides and guides. Of course, I'm not an expert in Go, but at the same time I have some experience and I think that I can help others with this knowledge :)

Useful links

Since this guide is quite subjective, I'm not sure that I will cover all the topics and I'm not sure that we won't have questions after reading, so in this block I will attach good links that can be used in such cases.

First of all, you can email me and I will try to help you:

boyarkin.gleb@gmail.com

source.boar@gmail.com

The mail is also duplicated in my profile.

Besides me, you can use:

https://www.w3schools.com/go/

These sources I guarantee that they are verified, I myself studied from them, but as for me, many things are painted there with too complex words.

Briefly about Go

Go is a very young static programming language. It was created in order to combine simplicity and speed, which, in principle, he did. Industries of use are most often small applications, microservices, and sometimes the Web.

Syntax

Go's syntax is mostly C-like, but with minor differences that we'll encounter along the way.

** If you go through the basic rules of syntax then: **

- there is no line ending character
- `{}` - characters for opening and closing code blocks
- `camelCase` – naming style variable
- `PascalCase` - naming style for functions, methods and structures
- `//` – character for single line comment
- `/* */` - characters for commenting a block of code

Program structure

The structure of a Go program consists of 2 mandatory things:

1. Declaring the package name

```
package /*package name*/
```

If your program is not a library then the package is declared as `main`

```
package main
```

2. The `main` function is the program entry function

```
func main() {

}
```

Working with the console

The `fmt` module is used to work with the console

`fmt.Print(/*message text*/)` - text output to the console

`fmt.Println(/*message text*/)` - output text to the console with the addition of a line break

`fmt.Printf(/*mask*/, /*variable 1*/, /*variable 2*/, ... /*variable n*/)` - formatted text output to the console

`fmt.Scan(/*reference to a variable where to write the result*/)` – text input from the console

Variables

As I think you know, or will become known now, in fact the entire programming language is based on variables and operations on them.

In Go, a variable is a pointer to a region or location in memory that holds a value. For now, the fact about memory can simply be put into your head, we will come to it later.

Go is also a static strongly and explicitly typed language, so when declaring variables, you must also pass the type or immediately pass the value, then the type will be determined automatically.

The `var` keyword is used to declare variables.

Ad example:

```
var a1 string = "Hello"
var a2 = 1
```

Also in Go there are constants. Constants are ordinary variables, except that they cannot be changed in the future after they are declared. Specifying the type of the constant is optional.

The `const` keyword is used to declare constants.

Ad example:

```
const a1 string = "Hello"
const a2 = 1
```

Variables can also be declared briefly: without using the `var` keyword. Instead, you just need to use the `:=` symbol.

Ad example:

```
a1 := "Hello"
```

When naming variables, as mentioned above, the camelCase style is used.

What does it include?

camelCase:

- first letter is lowercase
- each first letter of the next word is uppercase

It is also worth clarifying that Go is case sensitive, that is, in it `a` and `A` are different names.

Data types

Data type - types of what type of value the variable can store.

Built-in data types in Go:

- `bool` – boolean data type, stores 2 types of values: `true` or `false`
- `int8/int16/int32/int64` are numeric data types that can store 8, 16, 32 and 64 bit integers respectively
- `int` is a numeric data type that, depending on the platform, stores an integer of 32 or 64 bits, that is, equivalent to `int32` or `int64`
- `uint8/uint16/uint32/uint64` are numeric data types that can store positive integers of 8, 16, 32 and 64 bits respectively
- `uint` is a numeric data type that, depending on the platform, stores a positive integer of 32 or 64 bits, that is, equivalent to `uint32` or `uint64`
- `byte` is a synonym for type `uint8`
- `rune` is a synonym for type `int32`
- `float32/float64` - numeric data types that can store fractional numbers with dimensions of 32 and 64 bits, respectively
- `complex64/complex128` are numeric data types that can store complex numbers with bases `float32` and `float64` respectively
- `string` - string data type, stores a string of text of almost unlimited size
- `nil` is a keyword that means null or "empty" value

Pointers

Pointers are objects whose values are the addresses of other objects (such as variables). The pointer is defined as a regular variable, only the asterisk character `*` is placed before the data type

***Example of pointer to type int*:**

```
var a *int
```

To get the address of a pointer or any variable, use the `&` character

Operators

Assignment operators

`=` - assignment

`+=` - addition assignment

`-=` - assignment with subtraction

`*=` - multiplication assignment

`/=` - assignment with division

`%=` - division assignment with remainder

`++` is an increment, same as `+= 1`

`--` - decrement, same as `-= 1`

`<<=` - left shift assignment

`>>=` - right shift assignment

`&=` - assignment with bitwise `AND`

`|=` - assignment with bitwise `OR`

`^=` - assignment with bitwise exclusive `OR`

Boolean operators

`==` - equality

`!=` - inequality

`>` - more

`>=` - greater than or equal to

`<` - less than

`<=` - less than or equal to

`&&` is a logical `AND`, i.e. both conditions in the construction `(condition 1)&&(condition 2)` must be `true`

`||` is a logical `OR`, i.e. at least one condition in the construction `(condition 1)||(condition 2)` must be `true`

Arithmetic operators

`+` - plus

`-` - minus

`*` - multiplication

`/` - division

`%` - division with remainder

Bitwise operators

`&` - bitwise `AND`

`|` - bitwise `OR`

`^` - bitwise exclusive `OR`

`~` - bitwise `NOT`

`<<` - shift left

`>>` - right shift

Conditional constructs

Variables are, of course, good, but you have to work with them somehow, and conditional constructions are one of the ways to write program logic.

if...else

The most famous and commonly used design.

A condition is passed to `if`, if the condition is true, then what is in the `if` block is executed, otherwise what is in the `else` block

Syntax looks something like this:

```
if /*condition: either a logical expression or a variable of type bool*/ {
    // action if condition == true
} else {
    // action if condition == false
}
```

You can also add `else if` to this construction

Application example:

```
if /*condition 1: either boolean or bool*/ {
    // action if condition 1 == true
} else if /*condition 2: either boolean or bool*/ {
    // action if condition 1 == false, but condition 2 == true
} else {
    // action if condition 1 == false and condition 2 == false
}
```

Such `else if` constructs can be added any number of times

The syntax must be exactly this, other options for the location of the brackets are incorrect and are not accepted by the compiler

switch..case...default

Also a conditional construction that shortens the code that can be written through `if...else if...else`

`switch...case...default` is different in that we pass some variable to switch, and then the construction compares this variable with each value of `case` and if no match with `case` is found, the `default` block is executed

Syntax looks something like this:

```
switch /*variable to compare*/ {
    case /*value 1*/:
        // action if value 1 matches
    case /*value 2*/:
        // action if value 2 matches
    ...
    default:
        // action if no value matches
}
```

Loops

for

Type of loop with an action that is performed before the start of the bypass, an execution condition and an action that is performed at the end of each pass

Syntax looks something like this:

```
for /*action at the beginning*/; /*execution condition*/; /*action at the end of
each pass*/ {
    // action while condition is true
}
```

If actions at the beginning and end are not needed, then you can use the construction only with the condition

```
for /*execution condition*/ {
    // action while condition is true
}
```

`break` and `continue` keywords

`break` is used to exit the loop early, i.e. to terminate the loop regardless of the condition

Usage example:

```go
for i := 0; i < 10; i++ {
    if i%3 == 0 { // if the number is divisible by 3 without a remainder
        break
    }
    fmt.Println(i)
}
```

That is, the loop was executed until it came across the first number that is divisible by 3

`continue` is used to terminate the loop pass early and move to the next pass, that is, it is used when you need everything after its call not to be executed in this pass, but to be executed immediately making the pass

```go
for i := 0; i < 10; i++ {
    if i%3 == 0 { // if the number is divisible by 3 without a remainder
        continue
    }
    fmt.Println(i)
}
```

That is, for numbers that are divisible by 3, `fmt.Println(i)` was not executed, but the next loop run was immediately performed

Practical tasks on the given knowledge

For better consolidation, I propose to perform a few simple tasks that will check how much you understand this level of material, all the knowledge that may be needed to complete the tasks is above.

All problem solutions will be in the "Problems" folder and will be numbered, but I do not recommend you use them until you yourself succeed

1. Write a loop that will print to the console all numbers from 1 to 100 inclusive, which are divisible by 5 or 7 (`n%/*number*/ == 0` - the condition for divisibility of `n` by any number)

2. Write a loop that will print to the console all even numbers from 1 to 100 inclusive

3. Write a loop that will print to the console all numbers from 1 to 100 inclusive that contain 1

Functions

Functions are another important programming tool. If in simple words, this is the union of any section of code for its further call. In Go, functions are an object and you can work with them like an object, that is, write them to variables, pass them as an argument, write methods, and so on.

Every function in any programming language has:

- Name
- List of arguments (parameters) – variables whose values are set when calling the function. They are declared in `()` separated by commas. Also, when calling, their values are specified when calling in the same way in `()`. In the code block of our function, they play the role of ordinary block-scoped variables, but they do not need to be declared, since they are declared when the function is called.
- A block of code that will be executed when called. Written to `{}`

Syntax looks something like this:

```
func /*function name*/(/*argument1*/, /*argument2*/.../*argumentN*/) /*return type
(if needed)*/{
    // block of executable code
}
```

`func` is a keyword that means a function declaration

To call a function, it is enough to write its name and `()` in which you can list the arguments if they were declared

Function example:

```
func f(a1 int, a2 int, a3 int){
    fmt.Println(a1)
    fmt.Println(a2)
    fmt.Println(a3)
}

f(1, 2, 3)
```

Functions and procedures

If we move away from the topic of Go, where there is no syntactic difference between these concepts, in fact, but this is a very important issue.

What's the difference?

Function is a block of code that will return some value as a result of its execution.

Procedure is a function that does not return a value

The `return` keyword is responsible for returning a value in Go

Let's see the difference with examples:

```
func fnc(a1 int, a2 int, a3 int, a4 int) int { // function
    fmt.Println(a1)
    fmt.Println(a2)
    fmt.Println(a3)
    return a4
}

func proc(b1 int, b2 int, b3 int, b4 int) { // procedure
```

```
    fmt.Println(b1)
    fmt.Println(b2)
    fmt.println(b3)
}

a := fnc(1, 2, 3, 4) // == 4(a4)
b := proc(1, 2, 3, 4) // == error
```

What happened?

When declaring a **function**, we used `return`, so the function call can be assigned to a variable and it will have the value that we specified in `return`

Ordering data types

Variables are certainly good, but sometimes you have to store a lot of any data by combining them into one structure. For such things, arrays and dictionaries are invented.

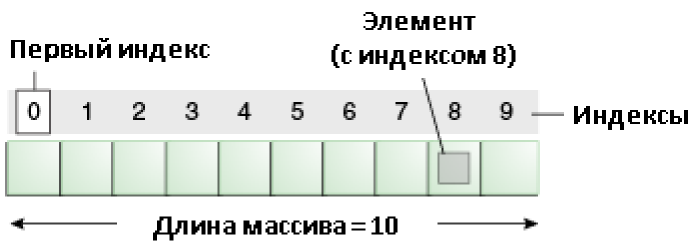
Arrays

Arrays are an ordered set of values that can be obtained by referring to the index.

An array in Go is a piece of memory with a fixed length.

We will also consider the way it is declared:

```
var a1[5]int = [5]int{1, 2, 3, 4, 5}
var a2 = [5]int{1, 2, 3, 4, 5}
var a3 := [5]int{1, 2, 3, 4, 5}
```



For a better understanding, let's use the picture above. As you can see, an array is, roughly speaking, a set of "cells" with values to which indices are attached in parallel, by which you can get the value to which they correspond.

To get a value or assign by index, it is enough to specify the index after the array name in `[]`

```
a := arr[0] // get
arr[0] = 1 // assignment
```

!!!Indexing in programming languages starts from 0!!!

Slices

Slices are sequences of elements of the same type of variable length. Unlike arrays, the length in slices is not fixed and can change dynamically, that is, you can add new elements or remove existing ones.

We will also consider the way it is declared:

```
var a1[]int = []int{1, 2, 3, 4, 5}
var a2 = []int{1, 2, 3, 4, 5}
var a3 := []int{1, 2, 3, 4, 5}
```

Adding elements

To add elements, use the function `append(/*array name*/, /*append element*/)`

Taking a slice

To take a slice, use the construction `/*array name*/[/*start index*/:/*end index*/]`

Cut length

To get the length of an array, use the `len(/*array name*/)` function

Iterate over an array/slice

for

```
for i := 0; i < /*array/slice length*/; i++ {
    // array/slice element arr[i]
}
```

for...range

```
for i, a := range arr {
    // array/slice element arr[i]
    // array/slice element a
}
```

Displays

Sometimes there are situations when it is necessary that the indices were not numbers, but something of their own. For this, mappings were created.

Displays is an array with self-specified indices.

Declaration syntax:

```
mp := map[/*key type*/]/*value type*/ {
    "/*index name 1*/": /*value 1*/,
    "/*index name 2*/": /*value 2*/,
    ...
    "/*index name N*/": /*value N*/}
```

Practical tasks on the given knowledge

4. Figure out how to reverse a string
5. Find a way to turn the string `"abcdef"` into `"cdefgh"`
6. Write a function that will return a string created according to the `camelCase` rule (the separator must be passed as the second argument)

   Example:

   `hello world` -> `helloWorld`
7. Create a function to populate an array of mappings that contain information about a person

   Structure:

   ```
   people := []map[string]string{} // main array
   /* dictionary structure
   {
       "firstName": "",
       "secondName": "",
       "age": "",
       "country": "",
       "city": ""
   }
   */
   ```
8. - Think about how you can sort the array

"*" - difficult task

Structures, methods and interfaces

Structures

Structures - data types created by the user to represent any non-standard objects

Structures are declared using the keywords `type` and `struct`

Syntax looks something like this:

```
type /*struct name*/ struct {
    /*structure fields*/
}
```

Each field has a name and data type, just like a variable

The initialization of structures might look something like this:

```
var s /*structure name*/ = /*structure name*/{/*values that are passed to fields*/}
```

Structure fields are accessed via `.`

Methods

Methods - functions associated with certain types (structures)

Syntax looks something like this:

```
func (/*variable name that will denote the type*/ /*type name*/) /*method
name*/(/*argument1*/, /*argument2*/.../*argumentN*/) /*return type values (if
needed)*/{
    // block of executable code
}
```

Calling methods is done as getting the fields of the structure, through `.`, the only thing is that parentheses are added with arguments

Interfaces

Interfaces represent an abstraction of the behavior of other types. Interfaces allow you to define functions that are not tied to a specific implementation. That is, interfaces define some functionality, but do not implement it.

Syntax looks something like this:

```
type /*interface name*/ interface {
    /*interface methods*/
}
```

Go has Duck typing ("Everything that quacks is a duck"), which means that interfaces do not require initialization. All struct objects with methods that correspond to interface methods are automatically objects of that interface.

Practical tasks on the given knowledge

9. Write the structure of a person where his name, age, country and city will be stored
10. Implement methods for this structure using interface

Everything!

It's all about the prince! I gave you everything I wanted. Hope this guide helped you.

I also have a more difficult guide with more complex topics, but it's more just a collection of solutions, but for those who are interested, you can read: https://github.com/s0urce18/AdditionalFunctions

Thanks to all! Hope to see you again :)