An Introduction To Programming In Go

Fragoso Pérez, Jonathan

November 16, 2014

Abstract

This document, describes the basics in Go programming language. The aims of this report are far from making people to become an expert in this language, but to give an overview of the initial steps, so all the people that have programmed for some time in other languages can introduce quickly to Go, without the need of going through many web pages or reading a book.

Note: Some of the information and examples have been directly copied from the book [1].

Contents

1. Introduction

Why learning Go? "Go was born out of frustration with existing languages and environments for systems programming."[3]

Authors → Go, also commonly referred to as golang, is a programming language initially developed at Google in 2007 by Robert Griesemer, Rob Pike, and Ken Thompson.

Advantages:

- Compiles to machine code and it is possible to compile a large Go program very quickly in a single computer.
- No hierachy → No time is spent defining the relationships between types.
- Garbage-collected.
- cgo program → Allows calling C libraries from Go code. SWIG extends this capability to C++ libraries.
- Easy to learn, write, mantain and scale.
- Supports concurrency at the language level. Concurrency is based on Hoare's Communicating Sequential Processes (CSP), the communication between processes is handled with messaging like Erlang.
- Open Source project.

Disadvantages:

- Is a relatively new language. It will need some time till more and more companies start to use it.
- Although there is a big community and a lot of packages, there are still some libraries you will miss, like a UI toolkit.
- No support for generics.

2. Basic Types

2.1. Numbers

- Integer
  - uint8 → same as byte
  - uint32
  - uint64
  - int8
  - int16
  - int32 → same as rune
  - int64

  Machine dependent → their size is dependent on the type of architecture of the machine -> uint, int, uintptr

  Go allows to increment/decrement by a unit using the operator ++/−. The language also enables to increment/decrement using the operator +=/-= .
- Float
  - float32 → single precision
  - float64 → double precision
- Complex
  - complex64
  - complex128

2.2. Strings

Some operations:

- Length → len("Hello world")
- Char. accessing -> "Hello World"[1] → returns 101 instead of e as a character is represented as a byte.
- Concatenation → "Hello " + " world!"

2.3. Booleans

1 bit integer representing true or false.

Operations:

- &&
- ||
- !

3. Other Types

3.1. Arrays

```
var integerArray [10]int

x := [5]float64{ 2, 5, 3, 1}
```

3.2. Slices

Are like arrays, but their size is allowed to change.

```
//slice associated with an underlying
//float64 array of length 5
x := make([]float64, 5)

//slice associated with an underlying
//float64 array of length 5,
//where 10 is the capacity of the underlying
//array which the slice points to
x := make([]float64, 5, 10)

arr := []float64{1,2,3,4,5}
x := arr[0:4] // this will assign to x values [1,2,3,4] because the high
              // index is not included
```

Built-in functions:

- Append → creates a new slice by taking an existing one.

```
slice1 := []int{1,2,3}
slice2 := append(slice1, 4, 5)

// RESULT:
// slice 1 value is [1,2,3]
// slice 2 value is [1,2,3,4,5]
```

- Copy

```
slice1 := []int{1,2,3}
slice2 := make([]int, 2)
copy(slice2, slice1)

// RESULT:
// slice 2 now will have values [1,2] because slice2 has room for only two
   elements
```

3.3. Maps

Examples:

- Assigning

```
x := make(map[string]int)
x["key"] = 10

elements := map[string]string{
  "H": "Hydrogen",
  "He": "Helium",
  "Li": "Lithium",
  "Be": "Beryllium",
  "B": "Boron",
  "C": "Carbon",
  "N": "Nitrogen",
  "O": "Oxygen",
  "F": "Fluorine",
  "Ne": "Neon",
}
```

- Accessing

```
x["key"]
```

- Delete op.

```
delete(x,1)
```

IMPORTANT!!!

- In go, if we try to search for a key that is not into the map, it returns the zero value for its value type. For example, if value is a String it will return "", or if values are integers will return 0.
- Due to the fact that Go can return serveral elements, the lookup informs us knowing if the lookup was successful and the result. Example:

```
elements := make(map[string]string)
name, ok := elements["Hello"]
```

so in Go is usual to do things like:

```
if name, ok := elements["Un"]; ok {
  //if lookup was successful
  fmt.Println(name, ok)
}
```

4. Vars

Declaration and assignation:

```
var x string = "Hello"

var y string

var (a = 5
     b = 10
     c = 15 )

y = "What's up?"

x := "Hello world"

y := 5 //No needed to add keyword var" and the type. Go compiler is able to
    infer
       //the type based on the literal value you assign the variable.
       //Use this form whenever possible
```

Constants:

```
const x String = "Hello world"
```

5. Control Structures

5.1. For

```
for i:=1; i <= 10; i++ {
  //stuff inside the loop
}

x := [5]float64{
  98,
  93,
  77,
  82,
  83,
}

var total float64 = 0
for _, value := range x {
  total += value
}
fmt.Println(total / float64(len(x)))
```

5.2. If

```
if i % 2 == 0 {
  // even
} else if i % 3 == 0 {
  // divisible by 3
} else if i % 4 == 0 {
  // divisible by 4
}
```

5.3. Switch

```
switch i {
  case 0: fm.Println("Zero")
  case 1: fm.Println("One")
  default: fm.Println("Unknown")
}
```

6. Functions

Multiple returning values: Go allows to return multiple values from a function.

```
func performTransaction(...) (boolean, float64) {
  // all the stuff related to performing the transaction

  currentBalance := ... //incrementing or decrementing the current balance of
      the account

  //informing that the transaction has been performed successfully and returning
      the //current balance
  return true, currentBalance
}

func main() {
  transactionOk, accountBalance := performTransaction(...)
}
```

Variadic functions: By using "..." before the type name of the last parameter we are indicating that this functions takes 0 to more of these parameters.

```
func Println(a ...interface{}) (n int, err error)
```

Closure: Go allows to create functions inside functions.

```
func main() {
  sum := func(x, y int) int {
    return x + y
  }
  fmt.Println(sum(3+4))
}
```

Recursion

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

6.1. Special functions

- defer → schedules a function to be run after the function completes.

```
f, _ := os.Open(fileName)
  defer f.Close()
  //work with the file
```

This will make to close the file at the very end of the function in which this code sequence is called.
By this way, we can have the opening of the file and its closing very closely, so we do not forget to close the file, and we can work with the file without worring about closing the file at the end.

- panic → Causes a run time error. Generally indicates a programmer error or an exceptional condition that there's no easy way to recover from.
- panic → Handling a run-time panic.

IMPORTANT!!!
The call to panic immediately stops execution of the functions, so if we want to recover from a panic, the recover must be in a defer function.

```
func main() {
  defer func() {
    str := recover()
    fmt.Println(str)
  }()
  panic("PANIC")
}
```

7. Pointers

Pointers reference a location in memory where a values is stored rather than the value itself, by this way we are able to modify the original variable.

```
func zero(xPtr *int) {
  *xPtr = 0
}
func main() {
  x := 5
  zero(&x)
  fmt.Println(x) // x is 0
}
```

A pointer can also be created using the built-in new function. Example: xPtr := new(int).

Because of being a garbage collected programming language, everything will be cleaned up automatically when nothing refers anymore.

Pointers can be really helpful when paired with structs.

8. Structs and interfaces

8.1. Structs

A struct is a type which contians named fields and introduces a new type.

```
type Circle struct {
  x, y, r float64
}
```

Initialization → when it is a created a new instance of the type, it initializes to 0 state, which means all its fields are by default set to their 0 state, which becomes into for example 0 for ints, 0.0 for floats, "" for strings, nil for pointers, ...

```
var c Circle    //creates a new instance and puts all its fields to 0.0 as all
    of them belong to the float type.

c := Circle{x: 0, y: 0, r: 5}
c := Circle{0, 0, 5}

c := new(Circle)  //allocating memory for all the fields, sets each of them to
    its 0 state and returning a pointer.
```

IMPORTANT!!!

Remember that in Go, arguments of a function are always copied, if we want to modify fields of the object inside a function, we need to work with pointers.

```
func circleArea(c *Circle) float64 {
    return math.Pi * c.r*c.r
}
```

8.2. Methods

If we add a "receiver" instead of passing the object as a parameter, we can call then the function using the '.' operator.

```
type Rectangle struct {
  x1, y1, x2, y2 float64
}
func (r *Rectangle) area() float64 {
  l := distance(r.x1, r.y1, r.x1, r.y2)
  w := distance(r.x1, r.y1, r.x2, r.y1)
  return l * w
}

main() will call:
    r := Rectangle{0, 0, 10, 10}
    r.are()
```

8.3. Embedded Types

Struct fields usually represent the has-a relationship. Anonymous fields (embedded types) → type has a field that depends on another type.

```
type Person struct {
  Name string
}
func (p *Person) Talk() {
  fmt.Println("Hi, my name is", p.Name)
}
```

And we want an Android type:

```
type Android struct {
  Person
  Model string
}
```

As we see we define the type and don't give it a name. Then we can call:

```
a := new(Android)
a.Person.Talk()
```

or directly

```
a.Talk()
```

8.4. Interfaces

Like a struct, an interface is created using the type keyworkd, bollowed by a name and the keyword interface. Instead of defining fields, we define a "method set" which is the list of methods or behaviour that a type must have in order to "implement" the interface.

```
type Shape interface {
  area() float64
  perimeter() float64
}

func totalArea(shapes ...Shape) float64 {
  var area float64
  for _, s := range shapes {
      area += s.area()
  }
  return area
}
```

Example of call:

```
totalArea(&c, &r) //calculating the total area of the sum of the area of a
                  //circle and of a rectangle
```

Interfaces can also be used as fields.

```
type MultiShape struct{
  shapes []Shape
}
```

9. Concurrency

Go has rich support for concurrency using goroutines and channels.

9.1. Goroutines

A goroutine is a function that is capable of running concurrently with other functions:

- To create a goroutines we use the keyword "go" followed by a function invocation.
- Goroutines are lightweight and we can easily create thousands of them.

```
package main

import (
  "fmt"
  "time"
  "math/rand"
)

func f(n int) {
  for i := 0; i < 10; i++ {
    fmt.Println(n, ":", i)
    amt := time.Duration(rand.Intn(250))
    time.Sleep(time.Millisecond * amt)
  }
}

func main() {
    // creating 10 subroutines that perform each one the f function
    for i := 0; i < 10; i++ {
      go f(i)
    }
    var input string
    fmt.Scanln(&input) //forcing the program to not exit, because we are calling
        subroutines.
                       //Else the program would exit without waiting for the
                          subroutines to finish.
                       //We can stop if we hit although any subroutine is being
                          executed at
                       //the time we hit the inter.
}
```

9.2. Channels

Allow two goroutines to communicate and synchronize their execution.

- Representation → keyword "chan" followed by the type of the things that are passed on the channel
- The left arrow ← operator is used to send and receive messages on the channel.

```
package main

import (
  "fmt"
  "time"
)

func pinger(c chan string) {
  for i := 0; ; i++ {
    c <- "ping"
  }
}
func ponger(c chan string) {
  for i := 0; ; i++ {
    c <- "pong"
  }
}
func printer(c chan string) {
  for {
    msg := <- c
    fmt.Println(msg)
    time.Sleep(time.Second * 1)
  }
}
func main() {
  var c chan string = make(chan string)  //creating the channel declaring that
      the messages that will communicate it, will be strings

  go pinger(c)  // first sender subroutine
  go ponger(c)  // second sender subroutine
  go printer(c) // receiver subroutine

  var input string
  fmt.Scanln(&input)
}

  // this program prints "ping" and "pong" in turns till we hit the enter key.
```

A channel can be bi-directional, or we can specify a direction on a channel type. If the channel can only be sent to, attempting to receive from the channel will result in a compiler error.

```
func pinger(c chan<- string) //the channel c can only be sent to SPECIAL SWITCH
    STATEMENT FOR MESSAGING: select
```

Example: This program prints "from 1" every 2 seconds and "from 2" every 3 seconds. select picks first channel that is ready and receives from it (or sends to it). If more than one are ready, then it randomly picks which one to receive from.

```
func main() {
  c1 := make(chan string)
  c2 := make(chan string)
  go func() {
    for {
      time.Sleep(time.Second * 2)
    }
  }()
  go func() {
    for {
```

```
        c2 <- "from 2"
        time.Sleep(time.Second * 3)
    }
  }()
  go func() {
    for {
      select {
        case msg1 := <- c1:
          fmt.Println(msg1)
        case msg2 := <- c2:
          fmt.Println(msg2)
        case <- time.After(time.Second): //time.After creates a channel and
            after the given duration will send the current time on it.
          fmt.Println("timeout")
        default: //happens immediately if none of the channels are ready
          fmt.Println("nothing ready")
      }
    }
  }()
  var input string
  fmt.Scanln(&input)
}
```

9.3. Buffered Channels

It is also possible to pass a second parameter to the make function when creating a channel

```
c := make(chan int, 1)  //creates a buffered channel with capacity 1.
```

Normally channels are synchronous; both sizes of the channel will wait until the other side is ready. A buffered channel is asynchronous; sending or receiving a message will not wait unless the channel is already full.

10. Packages

- Go was designed to be a language that encourages good software engineering practices and one of them is code reuse ("Don't Repeat Yourself.")
- Improve code organization, to find quickly the code we want to reuse.
- In Go if something starts with a capital letter means other packages (and programs) are able to see it. If we had named a function with first letter as lowercase instead of uppercase, this functions won't be accessible from outside. It's a good practice to only expose the parts of our package that we want other packages using and hide everything else.
- Package names match the folders they fall in. There are ways around this, but it's a lot easier if you stay within this pattern.

10.1. Documentation

Go has the ability to autmoatically generate doc. for packages we write in a similar way to the standard package documentation.

To generate it, in a terminal run:

godoc $packageName $functionName

(where $packageName and $functionName are the names for the package and the function respectively)

Then doc will be also avilable in web form by running:

godoc -http=":6060"

Then in the browser: http://localhost:6060/pkg/ we will be able to browse through all the packages installed in our system.

11. Testing

Creating tests is really easy in Go.

If we created a math package which contains a function to calculate the average, we can create a test for it creating a file, let's say `math_test.go` in the math folder, and adding this code:

```
package math

import "testing"

func TestAverage(t *testing.T) {
    var v float64
    v = Average([]float64{1,2})
    if v != 1.5 {
            t.Error("Expected 1.5, got ", v)
    }
}
```

After that, running the command "go test" (inside the math folder, of course) the command will look for any tests in any of the files in the current folder and run them. Tests are identified by starting a function with the word Test and taking one argument of type "*testing.T".

12. The core packages

12.1. Bytes

package name → bytes

Contains the Buffer struct. A buffer does not need to be initialized and supports Reader and Writer interfaces. A buffer can be converted into a []byte by calling buf.Bytes().

12.2. Strings

package name → strings

```
// true
strings.Contains("test", "es"),
// 2
strings.Count("test", "t"),
// true
strings.HasPrefix("test", "te"),
// true
strings.HasSuffix("test", "st"),
// 1
strings.Index("test", "e"),
// "a-b"
strings.Join([]string{"a","b"}, "-"),
// == "aaaaa"
strings.Repeat("a", 5),
// "bbaa"
strings.Replace("aaaa", "a", "b", 2),
// []string{"a","b","c","d","e"}
strings.Split("a-b-c-d-e", "-"),
// "test"
strings.ToLower("TEST"),
// "TEST"
strings.ToUpper("test"),
```

To convert string to a slice of bytes (and viceversa):

```
arr := []byte("test")
str := string([]byte{'t','e','s','t'})
```

12.3. Input/Output

package name → io

```
func Copy(dst Writer, src Reader) (written
int64, err error)
```

```
var buf bytes.Buffer
buf.Write([]byte("test"))
```

12.4. Files and folders

package name → os
Reading file contents and displaying them on the terminal:

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

func main() {
    file, err := os.Open("test.txt")
    if err != nil {
        // handle the error here
        return
    }
    defer file.Close()

    // get the file size
    stat, err := file.Stat()
    if err != nil {
        return
    }

    // read the file
    bs := make([]byte, stat.Size())
    _, err = file.Read(bs)
    if err != nil {
        return
    }

    str := string(bs)
    fmt.Println(str)
}
```

Reading files is very common. Shorter way:

```
package main

import (
     "fmt"
     "io/ioutil"
)

func main() {
    bs, err := ioutil.ReadFile("test.txt")
    if err != nil {
        return
    }
    str := string(bs)
    fmt.Println(str)
}
```

Create file:

```
file, err := os.Create("test.txt")
if err != nil {
    // handle the error here
    return
}
defer file.Close()

file.WriteString("test")
```

To get the contents of a directory we use the same os.Open function but giving a directory path instead a file name. Then we call the Readdir method.

```
dir, err := os.Open(".")
if err != nil {
    return
}
defer dir.Close()
fileInfos, err := dir.Readdir(-1)
if err != nil {
    return
}
for _, fi := range fileInfos {
    fmt.Println(fi.Name())
}
```

To recursively walk a folder, there's a Walk function in package "path/filepath". Example of usage:

```
filepath.Walk(".", func(path string, info os.FileInfo, err error) error {
    fmt.Println(path)
    return nil
})
```

The function passed to Walk is called for every file and folder in the root folder (in this case ".")

12.5. Errors

package name → errors

```
err := errors.New("This is an error message")
```

12.6. Containers and Sort

- List → package name → container/list → implementation of a doubly-linked list
- Sort → package name → sort → Functions for sorting arbitrary data. There are serveral predefined sorting functions (for slices of ints and floats)

The Sort function in sort takes a sort.Interface and sorts it. A sort.Interface requires 3 methods: Len, Less and Swap. Example of usage:

```
package main

import ("fmt" ; "sort")

type Person struct {
```

```
    Name string
    Age int
}

type ByName []Person

func (this ByName) Len() int {
    return len(this)
}
func (this ByName) Less(i, j int) bool {
    return this[i].Name < this[j].Name
}
func (this ByName) Swap(i, j int) {
    this[i], this[j] = this[j], this[i]
}
func main() {
     kids := []Person{
            {"Jill",9},
            {"Jack",10},
     }
     sort.Sort(ByName(kids))
     fmt.Println(kids)
}
```

12.7. Hashes and Cryptography

Just to remember that a hash function takes a set of data and reduces it to a smaller fixed size. Hash functions in Go are broken into two categories: cryptographic and non-cryptographic.

Cryptographic hash functions are similar to their non-cryptographic counterparts, but they have the added property of being hard to reverse. Give the cryptographic hash of a set of data, it's extremely difficult to determine what made the hash. These kind of hashes are often used in security applications (one common cryptographic hash function is SHA-1).

```
import (
    "fmt"
    "crypto/sha1"
)

func main() {
    h := sha1.New()
    h.Write([]byte("test"))
    bs := h.Sum([]byte{})
    fmt.Println(bs)
}

//sha1 computes a 160 bit hash. There is no native type to
//represent a 160 bit number, so we use a slice of 20 bytes instead.
```

12.8. Servers

Encodings needed are in package "encoding/"+ name, for example encoding/json

Example of HTTP server setup and usage:

```
package main

import ("net/http" ; "io")
```

```
func hello(res http.ResponseWriter, req *http.Request) {
      res.Header().Set(
            "Content-Type",
            "text/html",
      )
      io.WriteString(
            res,
            `<doctype html>
<html>
  <head>
    <title>Hello World</title>
  </head>
     <body>
            Hello World!
     </body>
</html>`,
     )
}
func main() {
     http.HandleFunc("/hello", hello)  //handles a URL route (/hello) by calling
        the given func.
     http.ListenAndServe(":9000", nil)
}
```

To handle static files we can use FileServer:

```
http.Handle(
     "/assets/",
     http.StripPrefix(
           "/assets/",
           http.FileServer(http.Dir("assets")),
     ),
)
```

12.9. Parsing Command Line Arguments

The flag package allows us to parse arguments and flags sent to our program.

```
package main

import ("fmt";"flag";"math/rand")

func main() {
     // Define flags
     maxp := flag.Int("max", 6, "the max value")
     // Parse
     flag.Parse()
     // Generate a number between 0 and max
     fmt.Println(rand.Intn(*maxp))
}
```

12.10. Synchronization Primitives

Appart from goroutines and channels, Go does provide more traditional multithreading routines in the "sync" and "sync/atomic" packages. Includes for instance, mutexes.

Bibliography

[1] C. Doxsey. *An Introduction to Programming in Go*. CreateSpace Independent Publishing Platform, 2012.

[2] Google. Google Group: golang-nuts. `https://groups.google.com/forum/#!forum/golang-nuts/`.

[3] Google. The Go Programming Language: faq. `https://golang.org/doc/faq/`.

[4] Google. The Go Programming Language: packages list. `http://golang.org/src/pkg/`.