Packt

MASTERING GO FOR UNIX ADMINISTRATORS, UNIX DEVELOPERS AND WEB DEVELOPERS

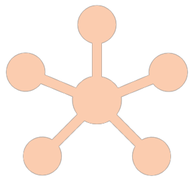
Day 1

Mastering Go for UNIX administrators, UNIX developers and Web Developers

3h 45m

Topics

1. Go Garbage Collector
2. Go functions
3. Channels + Shared Memory
4. Benchmarking
5. Go Package versioning



Downloading the Go code

GitHub repository: https://github.com/mactsouk/mGoLC

Execute the next command on your Terminal to get the code for this course:

git clone git@github.com:mactsouk/mGoLC.git

In this course, the code will be in color – if you have troubles viewing the code, let me know and I will turn it back to *black and white*.

Section 1

The Go Garbage Collector

The required theory

The purpose of the Go GC

The purpose of each Garbage Collection performed by the Garbage Collector (GC) is to free memory that is not being referenced any more without delaying the execution of the program.

This is a *simple and rational* requirement. However, back in the old days when computers where slow, running a GC used to be a pretty slow process.

I think that the first **popular** programming language that came with a GC is Java.

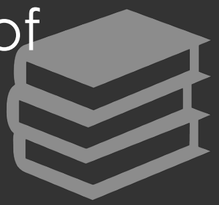
Two algorithms

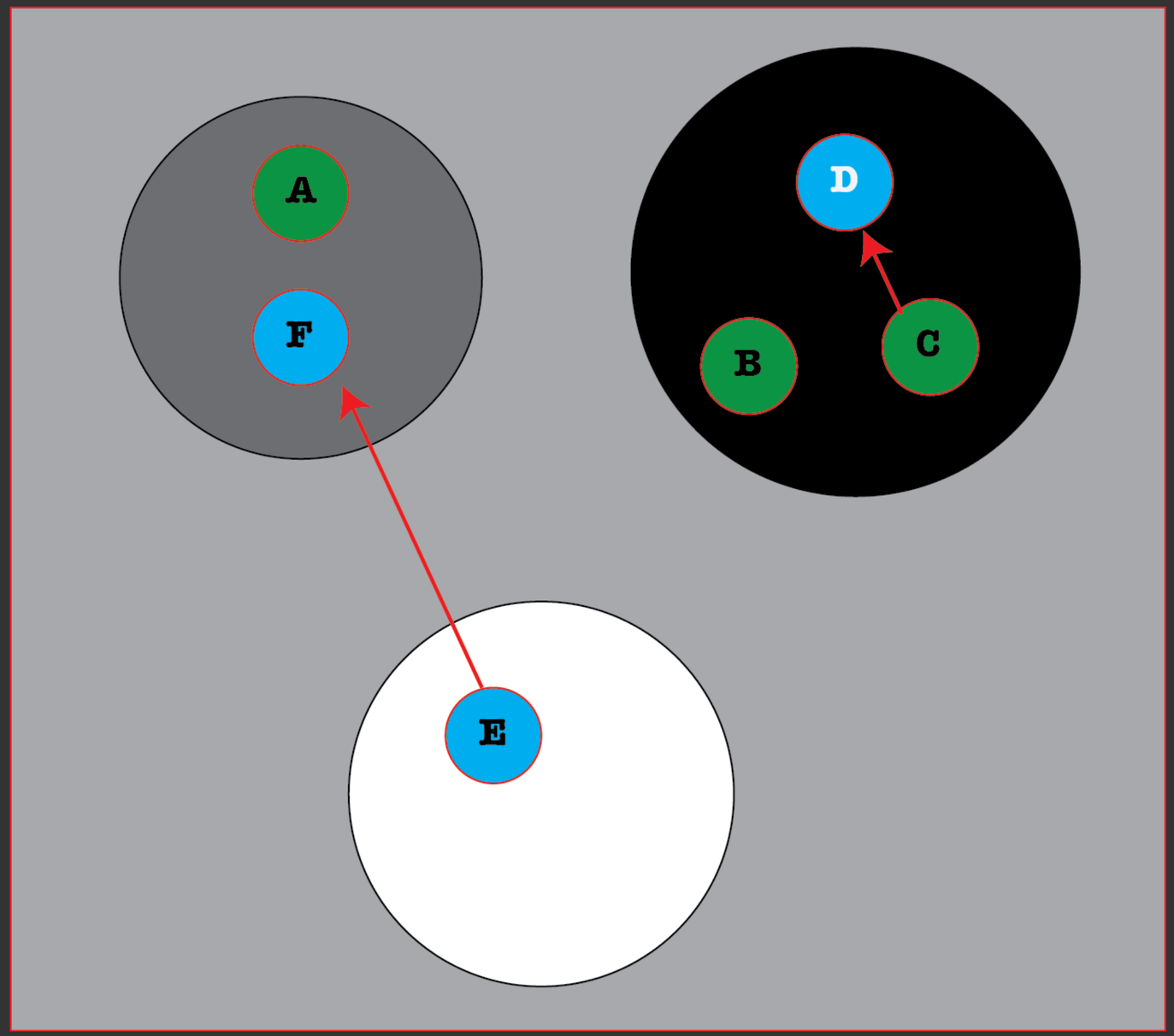
The Tricolor Algorithm

Tricolor Algorithm

The primary principle behind the tricolor mark-and-sweep algorithm is that it divides the objects of the heap into three different sets according to their color, which is assigned by the algorithm. The heap is where the various variables are located.

The objects of the **black** set are guaranteed to have no pointers to any object of the white set. However, an object in the white set can have a pointer to an object of the black set, because this has no effect on the operation of the garbage collector! The objects of the **grey** set might have pointers to some objects of the white set. Also, the objects of the **white** set are candidates for garbage collection.
Note that no object can go directly from the black set to the white set, which allows the algorithm to operate and be able to clear the objects in the white set. Additionally, no object of the black set can directly point to an object of the white set.




A
F
D
B
C
E

The mark and sweep algorithm

mark-and-sweep

The way that the **mark-and-sweep** algorithm works is pretty simple and easy to understand: the algorithm *stops the program execution* (stop-the-world garbage collector) in order to visit all of the accessible objects of the heap of a program and marks them. After that, it sweeps the inaccessible objects. During the mark phase of the algorithm, each object is marked as white, grey, or black. The children of a grey object are colored grey, whereas the original grey object is now colored black. The sweep phase begins when there are no more grey objects to examine. This technique works because there are no pointers from the black set to the white set, which is a fundamental invariant of the algorithm.

Although the mark-and-sweep algorithm is simple, it *suspends* the execution of the program while it is running, which means that it adds latency to the actual process.

Go Garbage Collector

Go GC

Strictly speaking, the official name for the algorithm used in Go is the **tricolor mark-and-sweep** algorithm. It can work concurrently with the program and uses a **write barrier**. This means that when a Go program runs, the Go scheduler is responsible for the scheduling of the application and the garbage collector as if the Go scheduler had to deal with a regular application with multiple goroutines!

The Go garbage collector is always being improved by the Go team, mainly by trying to make it faster by *lowering the number of scans* it needs to perform over the data of the three sets . However, despite the various optimizations, the central idea behind the algorithm remains the *same*.

You can say that the grey set acts like a barrier between the white set and the black set. Last, each time a pointer is moved, some Go code gets automatically executed, which is the **write barrier** mentioned earlier that does some recoloring.

Go GC

It is really important to remember that the Go garbage collector is a *real-time* garbage collector, which runs concurrently with the other goroutines of a Go program and only optimizes for *low latency*.

You can find the long and relatively advanced Go code of the garbage collector at https://github.com/golang/go/blob/master/src/runtime/mgc.go.

GC stats

Go allows you to inspect the operation of the GC. The example code can be found in **gColl.go**.

We will execute the program in two ways:

- go run gColl.go
- *GODEBUG=gctrace=1* go run gColl.go

The end!

Packt›

The tricolor algorithm was first illustrated on a paper called *On-the-fly Garbage Collection: An Exercise in Cooperation* by Edsger W. Dijkstra, Leslie Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens.

Go Scheduler

The Go scheduler

The *UNIX kernel scheduler* is responsible for the execution of the threads of a program. The *Go runtime* has its own scheduler, which is responsible for the execution of the goroutines using a technique known as **m:n scheduling**, where m goroutines are executed using n OS threads using *multiplexing*.

The Go scheduler is the Go component responsible for the way and the order in which the goroutines of a Go program get executed. This makes the Go scheduler a really important part of the Go programming language, as everything in a Go program is executed as a goroutine.
Go uses the **fork-join concurrency** model. The fork part of the model states that a child branch can be created at any point of a program. Analogously, the join part of the Go concurrency model is where the child branch will end and join with its parent. Among other things, both *sync.Wait()* statements and *channels* that collect the results of goroutines are join points, whereas any new goroutine creates a child branch.

The Go scheduler

The fair scheduling strategy, which is pretty straightforward and has a simple implementation, shares evenly all the load between the available processors. At first, this might look like the perfect strategy because it does not have to take many things into consideration while keeping all processors equally occupied. It turns out that this is not exactly the case because most of the distributed tasks usually depend on other tasks. Therefore, at the end of the day, some processors are underutilized, or equivalently, some processors are utilized more than others.

The Go scheduler

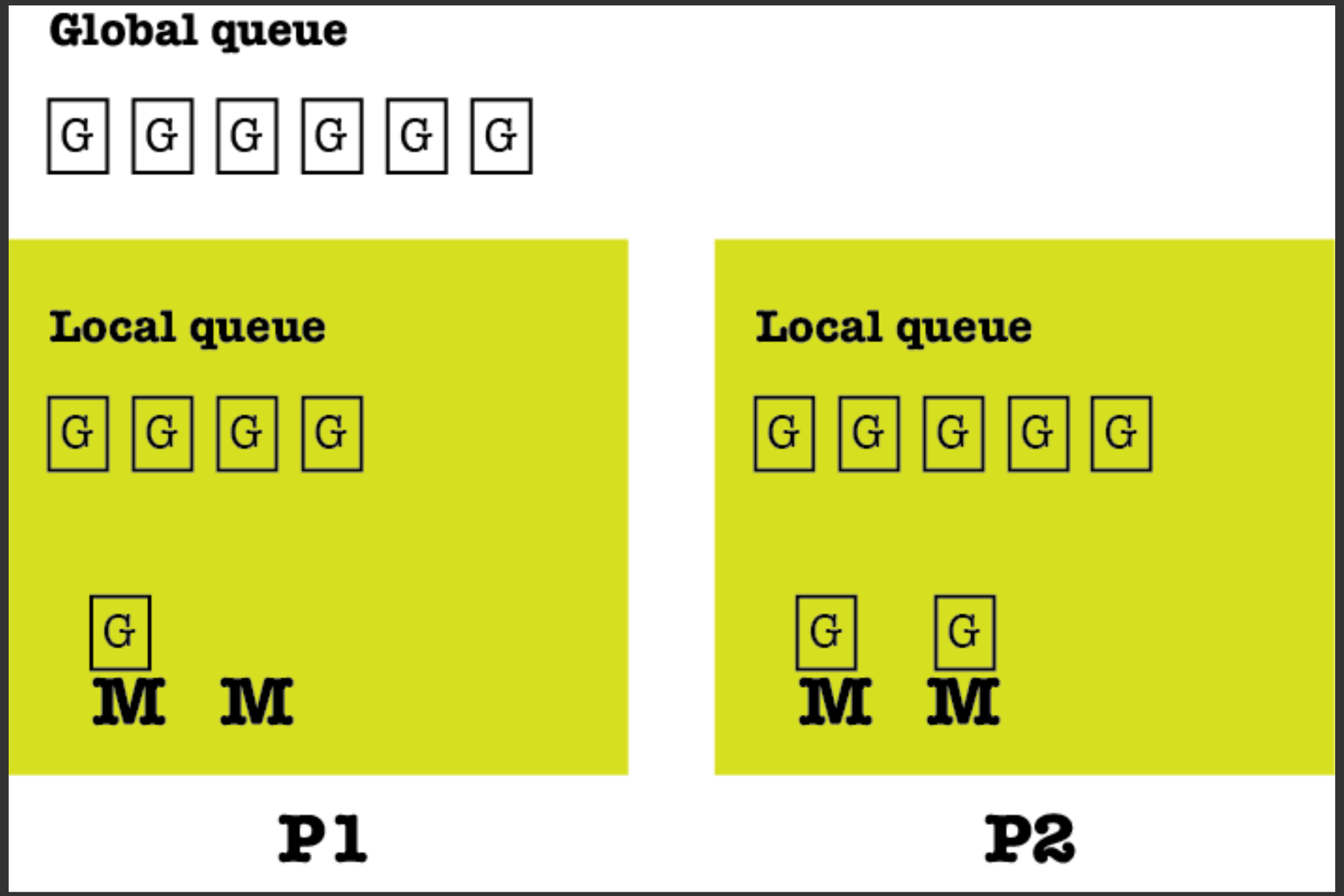
A goroutine in Go is a task, whereas everything after the calling statement of a goroutine is a **continuation**. In the work stealing strategy used by Go scheduler, a (logical) processor that is underutilized looks for additional work from other processors. When it finds such jobs, it steals them from the other processor(s), hence the name, **work stealing strategy**. Additionally, the work-stealing algorithm of Go queues and steals **continuations**. A **stalling join**, as is suggested by its name, is a point where a thread of execution stalls at a join and starts looking for other work to do. Although both task stealing and continuation stealing have stalling joins, continuations happen more often than tasks; therefore, the Go algorithm works with continuations rather than tasks.

The main disadvantage of **continuation stealing** is that it requires extra work from the compiler of the programming language. Fortunately, Go provides that extra help and therefore uses continuation stealing in its work-stealing algorithm.

The Go scheduler

One of the benefits of continuation stealing is that you get the same results when using just functions or a single thread with multiple goroutines. This makes perfect sense as only one thing is executed at any given time in both cases.

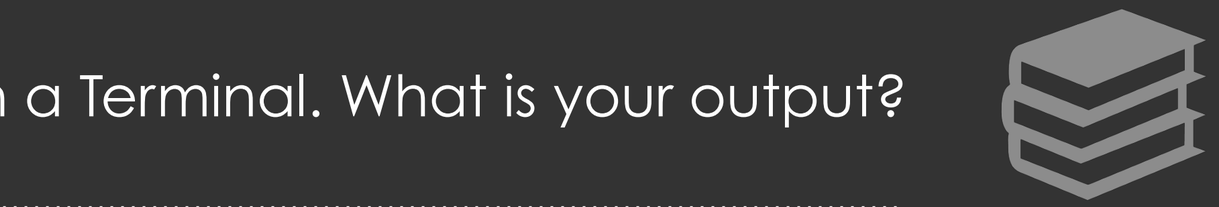
The Go scheduler works using three main kinds of entities: OS threads (M) that are related to the operating system in use, goroutines (G), and logical processors (P). The number of processors that can be used by a Go program is specified by the GOMAXPROCS environment variable—at any given time there are most GOMAXPROCS processors.
Now, let's return back to the m:n scheduling algorithm used in Go. Strictly speaking, at any time, you have m goroutines that are executed, and therefore scheduled to run, on n OS threads, using at most **GOMAXPROCS** number of logical processors.


Global queue
G G G G G G
Local queue
G G G G
G
M M
Local queue
G G G G G
G G
M M
P1
P2

GOMAXPROCS

```
package main

import (
    "fmt"
    "runtime"
)

func getGOMAXPROCS() int {
    return runtime.GOMAXPROCS(0)
}

func main() {
    fmt.Printf("GOMAXPROCS: %d\n", getGOMAXPROCS())
}
```

Now, let us execute **gomaxprocs.go** on a Terminal. What is your output?



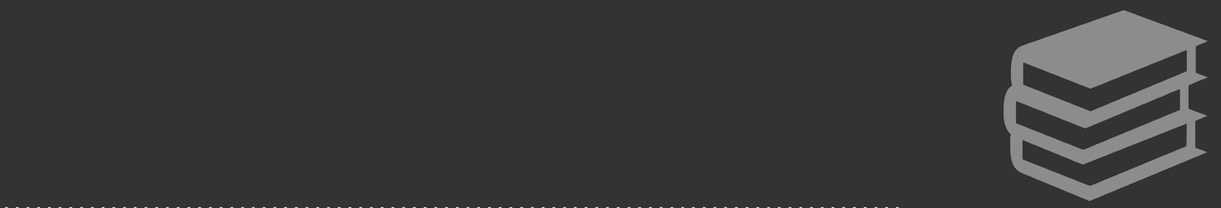
Section 2

Go functions revisited

Go functions

Go functions are quite powerful and can do the following:

- Functions can be anonymous. Anonymous functions can be defined inline without the need for a name, and they are usually used for implementing things that require a *small amount of code. Anonymous functions are also called closures.*
- Function can have pointer arguments.
- Functions can return pointers. A very common use of functions in Go is for returning *pointers to structures.*
- Functions can accept other functions as arguments.
- Function can return functions.



Functions


```
square := func(s int) int {
    return s * s
}
fmt.Println("The square of", y, "is", square(y))
double := func(s int) int {
    return s + s
}
fmt.Println("The double of", y, "is", double(y))

func getPtr(v *float64) float64 {
    return *v * *v
}

func returnPtr(x int) *int {
    y := x * x
    return &y
}
```

```
func funReturnFun() func() int {
    i := 0
    return func() int {
        i++
        return i * i
    }
}

func funFun(f func(int) int, v int) int {

      return f(v)

}

func createStruct(n, s string, h int32) *st {

    if h > 300 {

        h = 0

    }

    return &st{n, s, h}

}
```

The return value of **funReturnFun()** is an anonymous function!

The **funFun()** function accepts two parameters, a function parameter named f and an int value. The f parameter should be a function that takes one int argument and returns an int value.

For those with a C or C++ background, it is perfectly legal for a Go function to return the memory address of a local variable. Nothing gets lost, so everybody is happy!

The flag Go package

The flag package

The **flag** package does the dirty work of parsing command-line arguments and options for us. Additionally, it supports various data types, including strings, integers, and Boolean, which saves you time as you do not have to perform any data type conversions.
This means that there is no need for writing complicated and perplexing Go code for reading, converting and verifying command line arguments and values.

Using flag

```
package main

import (
    "flag"
    "fmt"
)

func main() {
    minusO := flag.Bool("o", false, "o")
    minusC := flag.Bool("c", false, "c")
    minusK := flag.Int("k", 0, "an int")

    flag.Parse()

    fmt.Println("-o:", *minusO)
    fmt.Println("-c:", *minusC)
    fmt.Println("-k:", *minusK)

    for index, val := range flag.Args() {
        fmt.Println(index, ":", val)
    }
}
```

The viper package

The viper package

The **viper** package is more advanced than the **flag** package – however, you do not always need that extra functionality.

https://github.com/spf13/viper

- You should download it first.
- Can read JSON, TOML, YAML, HCL, and Java properties config files!
- It is more powerful than flag. Do you need that power?

Now, let me go to my Terminal and present you **usingViper.go**. The first task will be to download some packages.

Go interfaces

About Go Interfaces

Strictly speaking, a Go **interface** type defines the behavior of other types by specifying a set of methods that need to be implemented. For a type to satisfy an interface, it needs to implement all of the methods required by that interface. Put simply, interfaces are abstract types that define a set of functions that need to be implemented so that a type can be considered an instance of the interface. When this happens, we say that the type satisfies this interface. So, an interface is two things: a set of methods and a type, and it is used for defining the behavior of other types.
The biggest advantage that you receive from having and using an interface is that you can pass a variable of a type that implements that particular interface to any function that expects a parameter of that specific interface. Without that amazing capability, interfaces would only be a formality without any practical or real benefit.

Remember: **interface{}** says nothing!

A simple example

The definition of io.Reader, as found in https://golang.org/src/io/io.go, is as follows:

```
   type Reader interface {
       Read(p []byte) (n int, err error)
}
```

In order for a type to satisfy the io.Reader interface, you will need to implement the Read() method as described in the interface.

```
func tellInterface(x interface{}) {
    switch v := x.(type) {
    case square:
        fmt.Println("This is a square!")
    case circle:
        fmt.Printf("%v is a circle!\n", v)
    case rectangle:
        fmt.Println("This is a rectangle!")
    default:
        fmt.Printf("Unknown type %T!\n", v)
    }
}
```

The **switch** statement can be used with Go Interfaces.

Interfaces + switch

```
package main

import (
    "fmt"
)

type square struct {
    X float64
}

type circle struct {
    R float64
}

type rectangle struct {
    X float64
    Y float64
}
```

```
func tellInterface(x interface{}) {
    switch v := x.(type) {
    case square:
        fmt.Println("This is a square!")
    case circle:
        fmt.Printf("%v is a circle!\n", v)
    case rectangle:
        fmt.Println("This is a rectangle!")
    default:
        fmt.Printf("Unknown type %T!\n", v)
    }
}

func main() {
    x := circle{R: 10}
    tellInterface(x)
    y := rectangle{X: 4, Y: 1}
    tellInterface(y)
    z := square{X: 4}
    tellInterface(z)
    tellInterface(10)
}
```

Interfaces

```
$ cat myInterface.go
package myInterface

type Shape interface {
      Area() float64
      Perimeter() float64
}
```

Now let me share my Terminal to see **s2/myInterface.go** in action!

A little Reflection

The reflect package

Reflection is an advanced Go feature that allows you to *dynamically* learn the type of an arbitrary object as well as information about its structure.

Go offers the **reflect** package for working with reflection. What you should remember is that you will most likely not need to use reflection in each one of your Go programs.

Let me share my Terminal and see the Go code of **reflection.go**.

Disadvantages of Reflection

- Extensive use of reflection will make your programs hard to read and maintain. A potential solution to this problem is good documentation; however, developers are famous for not having the time to write the required documentation.
- Go code which uses reflection will make your programs slower. Generally speaking, Go code that is made to work with a particular data type will always be faster than Go code that uses reflection to work dynamically with any Go data type. Additionally, such dynamic code will make it difficult for tools to refactor or analyze your code.
- Reflection related errors cannot be caught at **build time** and are reported at **runtime** as a panic. This means that reflection errors can potentially crash your programs! This can happen months or even years after the development of a Go program! One solution to this problem is extensive testing before a dangerous function call. However, this will add even more Go code to your programs, which will make them even slower.

The init() function

The init() function

Every Go package can optionally have a function named **init()** that is automatically executed at the beginning of the execution time.

The init() function is a private function by design, which means that it cannot be called from outside the package it belongs to. Additionally, as the user of a package has no control over the init() function, you should think carefully before using an init() function in public packages.

init() in action

Packt>

```
package a

import (
    "fmt"
)

func init() {
    fmt.Println("init() a")
}

func FromA() {
    fmt.Println("fromA()")
}
```

```
package b

import (
    "a"
    "fmt"
)

func init() {
    fmt.Println("init() b")
}

func FromB() {
    fmt.Println("fromB()")
    a.FromA()
}
```

The init() function

Let me share my Terminal to illustrate the use of the **init()** function with the help of the **manyInit.go** program that uses package **a** and package **b**.

Lab 1

Downloading the Go code

Let me share my terminal window to get the Go code using the next **git** command:

git clone git@github.com:mactsouk/mGoLC.git

Go functions

Exercise

- Write a function that accepts a pointer to a structure
- Write a function that returns a pointer to a structure
- Write a function that returns a **pointer** to a structure and an **error** variable

Do not forget to define your own structures when needed!

More about functions

Exercise

- Write a Go function that returns multiple values

- Write an anonymous Go function

- Write a function that returns another function

QUESTIONS?

Lab 2

The flag package

Using flag

Create a Go program that uses **flag** with the following command line parameters:

- A boolean named **subdirs**.
- An integer named **count**.
- A string named **name**.
- A float named **limit**.

lab2/usingFlag.go

Packt>

Now, let me show you the implementation of **lab2/funWithFlag.go**, which illustrates how can a flag accept multiple values separated by commas.

I will need to share my Terminal window.

The viper package

Using viper

Create the previous example using **viper** instead of **flag**.

Go Interfaces

Create an Interface

Defining a Go Interface can be tricky, especially when you try to add too much functionality to it – try to create simple but **not too simple** interfaces.

Let **io.Reader** and **io.Writer** inspire you!
Each one of these two interfaces includes just a single function!

Now, define an interface and use it!

If you are creating an Interface in a separate Go package, which is usually the case, do not forget to install that package first!

Go Reflection

Reflection

I will now present you another Go example that uses Reflection, which is more advanced than the one you saw in Section 2.
The name of the program is *lab2/advRefl.go*.

Let me show you the code, explain it and execute advRefl.go.

Section 3

Benchmarking Go code

Introduction

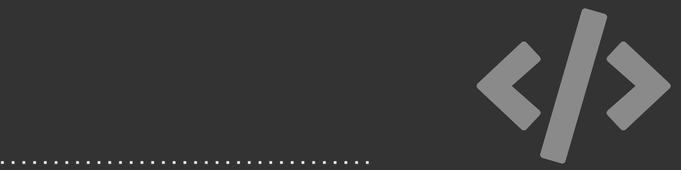
Benchmarking can give you information about the performance of a function or a program in order to understand better how much faster or slower a function is compared to another function, or compared to the rest of the application. Using that information, you can easily reveal the part of the Go code that needs to be rewritten in order to improve its performance.

Go follows certain conventions regarding benchmarking. The most important convention is that the name of a benchmark function must begin with **Benchmark**.

A package

```
package main

import (
    "fmt"
)

func fibo1(n int) int {
    if n == 0 {
        return 0
    } else if n == 1 {
        return 1
    } else {
        return fibo1(n-1) + fibo1(n-2)
    }
}

func fibo2(n int) int {
    if n == 0 || n == 1 {
        return n
    }
    return fibo2(n-1) + fibo2(n-2)
}
```

```
func fibo3(n int) int {
    fn := make(map[int]int)
    for i := 0; i <= n; i++ {
        var f int
        if i <= 2 {
            f = 1
        } else {
            f = fn[i-1] + fn[i-2]
        }
        fn[i] = f
    }
    return fn[n]
}

func main() {
    fmt.Println(fibo1(40))
    fmt.Println(fibo2(40))
    fmt.Println(fibo3(40))
}
```



Benchmarking!

```
package main

import (
    "testing"
)

var result int

func benchmarkfibo1(b *testing.B, n int) {
    var r int
    for i := 0; i < b.N; i++ {
        r = fibo1(n)
    }
    result = r
}

func benchmarkfibo2(b *testing.B, n int) {
    var r int
    for i := 0; i < b.N; i++ {
        r = fibo2(n)
    }
    result = r
}

func benchmarkfibo3(b *testing.B, n int) {
    var r int
    for i := 0; i < b.N; i++ {
        r = fibo3(n)
    }
    result = r
}
```

```
func Benchmark30fibo1(b *testing.B) {
    benchmarkfibo1(b, 30)
}

func Benchmark30fibo2(b *testing.B) {
    benchmarkfibo2(b, 30)
}

func Benchmark30fibo3(b *testing.B) {
    benchmarkfibo3(b, 30)
}

func Benchmark50fibo1(b *testing.B) {
    benchmarkfibo1(b, 50)
}

func Benchmark50fibo2(b *testing.B) {
    benchmarkfibo2(b, 50)
}

func Benchmark50fibo3(b *testing.B) {
    benchmarkfibo3(b, 50)
}
```

Two wrong benchmark functions

Remember that each benchmark is executed for at least 1 second by default. If the benchmark function returns in a time that is less than 1 second, the value of b.N is increased and the function is run again. The first time the value of b.N is 1, then it becomes 2, 5, 10, 20, 50, and so on. This happens because the faster the function, the more times you need to run it to get accurate results.

```
func BenchmarkFiboI(b *testing.B) {
  for i := 0; i < b.N; i++ {
      _ = fibo1(i)
  }
}
```

The **BenchmarkFibo1()** function has a valid name and the correct signature. The bad news, however, is that this benchmark function is wrong, and you will not get any output from it after executing the **go test** command!
The reason for this is that as the b.N value grows according to the way we described earlier, the run time of the benchmark function will also increase because of the for loop. This fact prevents **BenchmarkFibo1()** from converging to a stable number, which prevents the function from completing and therefore returning.

```
func BenchmarkFiboII(b *testing.B) {
for i := 0; i < b.N; i++ {
    _ = fibo2(b.N)
  }
}
```

The **BenchmarkFiboII()** benchmark function is also wrongly implemented for the same reasons as before.

What about these functions?

```
func BenchmarkFiboIV(b *testing.B) {
	for i := 0; i < b.N; i++ {
		_ = fibo3(10)
	}
}

func BenchmarkFiboIII(b *testing.B) {
	_ = fibo3(b.N)
}
```

Hint: they are fine!

Benchmarking File I/O operations

```
func benchmarkCreate(b *testing.B, buffer, filesize int) {
    var err error
    for i := 0; i < b.N; i++ {
        err = Create("/tmp/random", buffer, filesize)
    }
    ERR = err

    err = os.Remove("/tmp/random")
    if err != nil {
        fmt.Println(err)
    }
}
```

Now, let me show and explain the code of **writingBU.go** and **writingBU_test.go**.

Lab 3

Executing Benchmark functions

You can execute one or more Benchmarking functions as follows:

$ go test -bench=. writingBU.go writingBU_test.go

The following output also checks the memory allocations of the benchmark functions:

$ go test -bench=. writingBU.go writingBU_test.go -benchmem

Please execute them on your machine.

Examining the results

Explaining the Results

It is obvious that using a write buffer with a size of 1 byte is totally inefficient and slows everything down. Additionally, such a buffer size requires too many memory operations, which slows down the program even more!
Using a write buffer with 2 bytes makes the entire program twice as fast, which is a good thing. However, it is still very slow. The same applies to a write buffer with a size of 4 bytes.

Where things get faster is when we decide to use a write buffer with a size of 10 bytes. Finally, the results show that using a write buffer with a size of 1,000 bytes does not make things 100 faster than when using a buffer size of 10 bytes, which means that the sweet spot between speed and write buffer size is between these two values.

QUESTIONS?

SEE YOU TOMORROW!

THANK YOU!