Part 21: Goroutines

02 JULY 2017

Welcome to tutorial no. 21 in Golang tutorial series.

In the previous tutorial, we discussed about concurrency and how it is different from parallelism. In this tutorial we will discuss about how concurrency is achieved in Go using Goroutines.

What are Goroutines?

Goroutines are functions or methods that run concurrently with other functions or methods. Goroutines can be thought of as light weight threads. The cost of creating a Goroutine is tiny when compared to a thread. Hence its common for Go applications to have thousands of Goroutines running concurrently.

Advantages of Goroutines over threads

- Goroutines are extremely cheap when compared to threads. They are only a few kb in stack size and the stack can grow and shrink according to needs of the application whereas in the case of threads the stack size has to be specified and is fixed.
- The Goroutines are multiplexed to fewer number of OS threads. There might be only one thread in a program with thousands of Goroutines. If any Goroutine in that thread blocks say waiting for user input, then another OS thread is created and the remaining Goroutines are moved to the new OS thread. All these are taken care by the runtime and we as programmers are abstracted from these intricate details and are given a clean API to work with concurrency.
- Goroutines communicate using channels. Channels by design prevent race conditions from happening when accessing shared memory using Goroutines. Channels can be thought of as a pipe using which Goroutines communicate. We will discuss channels in detail in the next tutorial.

How to start a Goroutine?

Prefix the function or method call with the keyword `go` and you will have a new Goroutine running concurrently.

Lets create a Goroutine :)

```
1  package main
2
3  import (
4      "fmt"
5  )
6
7  func hello() {
8      fmt.Println("Hello world goroutine")
9  }
10 func main() {
11     go hello()
12     fmt.Println("main function")
13 }
```

Run program in playground

In line no. 11, `go hello()` starts a new Goroutine. Now the `hello()` function will run concurrently along with the `main()` function. The main function runs in its own Goroutine and its called the *main Goroutine*.

Run this program and you will have a surprise!

This program only outputs the text `main function`. What happened to the Goroutine we started? We need to understand two main properties of go routines to understand why this happens.

- **When a new Goroutine is started, the goroutine call returns immediately. Unlike functions, the control does not wait for the Goroutine to finish executing. The control returns immediately to the next line of code after the Goroutine call and any return values from the Goroutine are ignored.**
- **The main Goroutine should be running for any other Goroutines to run. If the main Goroutine terminates then the program will be terminated and no other Goroutine will run.**

I guess now you will be able to understand why our Goroutine did not run. After the call to `go hello()` in line no. 11, the control returned immediately to the next line of code without waiting for the hello goroutine to finish and printed `main function`. Then the main Goroutine terminated since there is no other code to execute and hence the `hello` Goroutine did not get a chance to run.

Lets fix this now.

```
1  package main
2
3  import (
4      "fmt"
5      "time"
6  )
7
8  func hello() {
9      fmt.Println("Hello world goroutine")
10 }
11 func main() {
12     go hello()
13     time.Sleep(1 * time.Second)
14     fmt.Println("main function")
15 }
```

Run program in playground

In line no.13 of the program above, we have called the Sleep method of the *time* package which sleeps the go routine in which it is being executed. In this case the main goroutine is put to sleep for 1 second. Now the call to `go hello()` has enough time to execute before the main Goroutine terminates. This program first prints `Hello world goroutine`, waits for 1 second and then prints `main function`.

This way of using sleep in the main Goroutine to wait for other Goroutines to finish their execution is a hack we are using to understand how Goroutines work. Channels can be used to block the main Goroutine until all other Goroutines finish their execution. We will discuss channels in the next tutorial.

Starting multiple Goroutines

Lets write one more program that starts multiple Goroutines to better understand Goroutines.

```
1  package main
2
3  import (
4      "fmt"
5      "time"
6  )
7
8  func numbers() {
9      for i := 1; i <= 5; i++ {
10         time.Sleep(250 * time.Millisecond)
11         fmt.Printf("%d ", i)
12     }
13 }
14 func alphabets() {
15     for i := 'a'; i <= 'e'; i++ {
16         time.Sleep(400 * time.Millisecond)
17         fmt.Printf("%c ", i)
18     }
19 }
20 func main() {
21     go numbers()
22     go alphabets()
23     time.Sleep(3000 * time.Millisecond)
24     fmt.Println("main terminated")
25 }
```

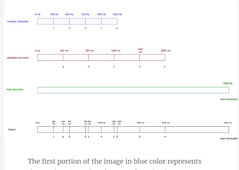
Run in playground

The above program starts two Goroutines in line nos. 21 and 22. These two Goroutines now run concurrently. The `numbers` Goroutine sleeps initially for 250 milliseconds and then prints `1`, then sleeps again and prints `2` and the same cycle happens till it prints 5. Similarly the `alphabets` Goroutine prints alphabets from `a` to `e` and has 400 milliseconds sleep time. The main Goroutine initiates the `numbers` and `alphabets` Goroutines, sleeps for 3000 milliseconds and then terminates.

This program outputs

```
1 a 2 3 b 4 c 5 d e main terminated
```

The following image depicts how this program works. Please open the image in a new tab for better visibility :)



The first portion of the image in blue color represents the *numbers Goroutine*, the second portion in maroon color represents the *alphabets Goroutine*, the third portion in green represents the *main Goroutine* and the final portion in black merges all the above three and shows us how the program works. The strings like *0 ms*, *250 ms* at the top of each box represent the time in milliseconds and the output is represented in the bottom of each box as *1*, *2*, *3* and so on. The blue box tells us that `1` is printed after `250 ms`, `2` is printed after `500 ms` and so on. The bottom of last black box has values `1 a 2 3 b 4 c 5 d e main terminated` which is the output of the program as well. The image is self explanatory and you will be able to understand how the program works.

That's it for Goroutines. Have a good day.

Next tutorial – Channels

Naveen Ramanathan
iOS developer at dietco.de and Golang enthusiast.