УДК 004.415

INTRODUCTION TO GOROUTINES IN GO

Alchangian O.A.
Scientific director – Senior Lecturer, Oleinik O.V.
Kharkiv National University of Radio Electronics, dept. PI
Kharkiv, Ukraine
tel. +38(099) 975-27-09, e-mail: oleksandra.alchangian@gmail.com

This work is devoted to goroutines in Go, which are a key feature of the language for providing parallel programming. They are easy to use and efficient, making them a powerful tool for developers who want to take advantage of parallel programming. This article explains what goroutines are and how they can be created and used in Go. It also explains all the advantages of using goroutines. In addition to explaining the basics of goroutines, this article will also cover some practical examples of how to use goroutines in real-world scenarios. This article serves as a useful introduction to Goroutines for programmers who want to learn parallel programming in Go.

Multithreading enables the execution of multiple tasks concurrently. For instance, web browsers utilize multithreading. Go offers numerous tools, such as goroutines and channels, to facilitate multithreading. Goroutines are lightweight threads that enable functions or methods to execute concurrently with others. In Go applications, thousands of goroutines can run simultaneously, with each one representing a function call and executing instructions sequentially.

Advantages of goroutines over threads:

– The goroutines in Go use fewer operating system threads, which means that a program with many goroutines can use only one thread. When a goroutine is blocked, a new OS thread can be created and the remaining goroutines are moved into it, which is controlled by the Go runtime environment.

– Using goroutines in Go is much cheaper than working with threads. goroutines are small, and their stack can be adjusted to the needs of the application, while the size of the thread stack is predefined and cannot be changed.

– Goroutines in Go communicate with each other using channels, which are designed to prevent race conditions from happening when accessing shared memory using goroutines.

In Golang, the main function is called first goroutine when starting a program. To create new goroutine you must use the keyword "go" before calling the function. Without it, the program will execute the function and then return to where it was called from. With the "go" keyword, the function will be executed in a new goroutine, asynchronously with the piece of code that called it. The Go runtime environment, running in the background, starts a set of goroutines, with a scheduler distributing them to the machines. A thread is then created that

processes all the goroutines and the maximum number of threads is determined by the variable GOMAXPROCS.

Let's look at an example for a better understanding: A variable i is initialized with the value 5 inside the main function. The keyword "go" creates a goroutine that outputs the value of variable i using the function fmt.Printf(). The value of variable i is incremented by 1 using the i++ line. An anonymous goroutine is created that increments the value of variable i by 1, and then creates another goroutine that outputs the value of variable i. The function time.Sleep(100000) is called to allow the goroutines to execute before the program ends.

```
1.    package main

2.    import (
3.    "fmt"
4.    "time"
5.    )

6.    func main() {
7.    i := 5
8.    go fmt.Printf("1. The value of variable i is %d\n", i)
9.    i++
10.   go fmt.Printf("2. The value of variable i is %d\n", i)
11.   go func() {
12.   i++
13.   go fmt.Printf("3. The value of variable i is %d\n", i)
14.   }()
15.   i++
16.   go fmt.Printf("4. The value of variable i is %d\n", i)
17.   time.Sleep(100000)
18.   }
```

When using goroutines to execute a program, the output values of the variable i in each goroutine may be different depending on how the goroutines are scheduled, similar to the behavior of threads. Anonymous functions can also be used to create goroutines. However, without the sleep() command, output may not be displayed because the main goroutine will end before the spawned goroutine can output anything. Using goroutines can make the output unpredictable because of the compiler's limitation on the order of memory accesses during parallel execution. Not using goroutines will result in more consistent output.

References:

1. Donovan, A. A. A., & Kernighan, B. W. (2015). The Go Programming Language. Addison-Wesley Professional.

2. Cox-Buday, K. (2017). Concurrency in Go: Tools and Techniques for Developers. O’Reilly Media, Inc.

3. St Martin, E., Kennedy, W., & Ketelsen, B. (2015). Go in Action. Simon and Schuster.