Concurrency in Golang 101

29 January 2022

Disclaimers:

- We won't explain more about basic Go.
- Go environment must be set up before you can execute any of the example given
- Code example can be found here: https://github.com/fakihariefnoto/goconcurrency101

Handbook

Introduction to Material

Concurrency 101

In computer science, concurrency is the ability of different parts or units of a program, algorithm, or problem to be executed out-of-order or in a partial order, without affecting the final outcome.

So in short it's about effectively handling multiple tasks at the same time.

Sequential vs Concurrency vs Parallelism

Instead of saying that one method beats the other, everything

	Sequential	**Parallelism**	**Concurrency**
Task execution	1 task at a time	Multiple task at a time	1 task at a time
Easiness to debug	Easiest	Hardest due to multiple failure point: Code error, race condition, resource management	Harder to debug: Code error, race condition
Time needed	Longest	Shortest	Shorter than Sequential
Easiness of implementation	Simplest and straightforward	Complicated (need to manually set the resource and close it after use)	Simple (no need to manually set the resource)
Resources usage	Use 1 core	Use multiple core	Use 1 core
Example of usage	Basic programming function	Bitcoin mining, neural network	Task manager in OS

Table 1.1 comparison of sequential, parallelism, and concurrency

Implementation of Concurrency in Golang

Goroutine

Goroutines are **functions** or **methods** that run **concurrently** with other functions or methods. Goroutines can be thought of as **lightweight threads**. The cost of creating a Goroutine is **tiny** when compared to a **thread**. Every program contains at least a single Goroutine and that Goroutine is known as the **main Goroutine**. All the Goroutines are working under the main Goroutines if the main Goroutine terminated, then all the goroutine present in the program also terminated. Goroutine always works in the **background**.

Goroutine vs **Thread**

- Goroutines are extremely cheap when compared to threads. They are only a few kb (2 kb (2048 bytes) source:https://go.dev/doc/go1.4#runtime & https://github.com/golang/go/blob/master/src/runtime/stack.go#L75) in stack size and the stack can grow and shrink according to the 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 a fewer number of OS threads. There might be only one thread in a program with thousands of Goroutines.
- Goroutines communicate using channels. Channels by design prevent race conditions from happening when accessing shared memory using Goroutines

Example of goroutine implementation can be found here:
https://github.com/fakihariefnoto/goconcurrency101/tree/main/1_goroutine

Channel

Channels are the pipes that connect concurrent goroutines. You can send values into channels from one goroutine and receive those values into another goroutine, in short, this is the way each goroutine communicates with each other.

Technically, a channel is a data transfer pipe where data can be passed into or read from. Hence one goroutine can send data into a channel, while other goroutines can read that data from the same channel.

Example of channel implementation can be found here:
https://github.com/fakihariefnoto/goconcurrency101/tree/main/2_channel

Here is the detailed explanation for the channel:
https://github.com/gophercon/2017-talks/blob/master/KavyaJoshi-UnderstandingChannels/Kavya%20Joshi%20-%20Understanding%20Channels.pdf

You can watch the presentation on youtube - Kavya Joshi Gopher Con Channel

Advanced: Buffered Channel

By default channels are *unbuffered*, meaning that they will only accept sends (*chan* <-) if there is a corresponding receive (<- *chan*) ready to receive the sent value. *Buffered channels* accept a limited number of values without a corresponding receiver for those values.

An example of buffered channel implementation can be found here:
https://github.com/fakihariefnoto/goconcurrency101/tree/main/3_channelbuff

Advanced: Select

The *select* statement is just like a switch statement without any input parameter
The *select* statement lets a goroutine wait on multiple communication operations.

A "*select"* blocks until one of its cases can run, then it executes that case. It chooses one at random if multiple are ready. Combining goroutines and channels with select is a powerful feature of Go.

Here are an example of "select" implementations:

```
select {
  case msg1 := <-channel1:
        fmt.Println("received", msg1)
  case msg2 := <-channel2:
       fmt.Println("received", msg2)
  case <-quit:
      return
}
```

Handling Race Condition

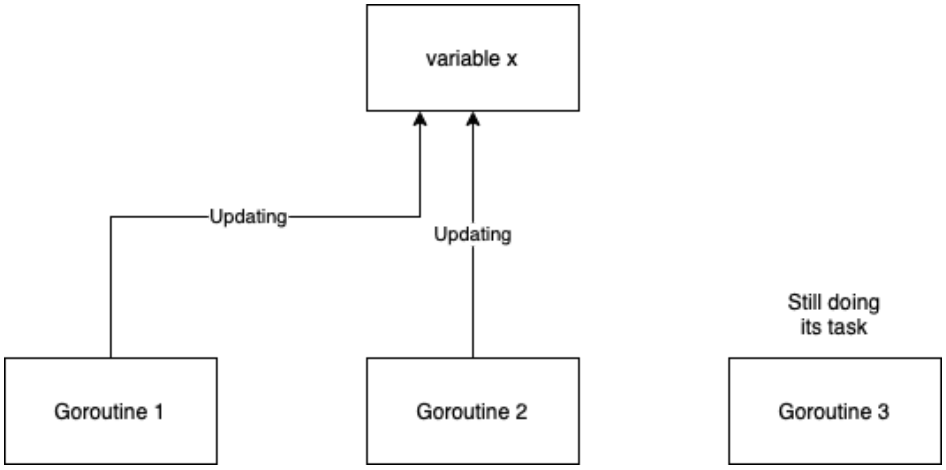

Fig 1.1 Race condition

With concurrency in place, we need to be aware that race conditions might happen. Race condition happens when 2 or more concurrent processes try to update the same data (in technical terms, it happens when 2 concurrent process tries to edit the value of 1 variable).
There are several approaches that we can implement to ensure our program is free of race conditions

Waitgroup

Waitgroup is one of many way to solve concurrency problem in go. It waits the already specified number of goroutine to finish before continuing to the next line of code. The main goroutine calls `*Add*` to set the number of goroutines to wait for, and then each of the goroutines runs and calls `*Done*` when finished. At the same time, Wait can be used to block until all goroutines have finished.

Common use case

- Each Goroutine process have different execution time
- We don't know when goroutine finish
- The results from multiple goroutines process is required for the next process

Example of Waitgroup implementation can be found here:

https://github.com/fakihariefnoto/goconcurrency101/tree/main/4_waitgroup

Mutex

Go is a language known for how simple it is to run concurrent routines (called goroutines). However, because of this, it's easy to run into errors when concurrent goroutines have to access the same piece of data. Some issue that might be occurs is **data race** when multiple goroutine write to the same resource.

Mutex provides **mutual exclusion to a resource**, which means that only one goroutine can hold it at a time. Any goroutine must first acquire the mutex and then access the resource. If a goroutine tries to acquire the mutex already held by another goroutine, it will be blocked and will wait for the mutex to be released.

Example of Mutex implementation can be found in:
https://github.com/fakihariefnoto/goconcurrency101/tree/main/5_mutex

Advanced: RWMutex

In some cases, a Mutex can ends up being a problem to overall process with all the exclusive locks that will halt the resource usages, in this case, **RWMutex** can be the optimized solution for more comprehensive resource management.

RWMutex is short of Read/Write Mutex, this is a type of Mutex that can be exclusively used to lock specific read/write process.
So when you need to lock resource exclusively for read operations or write operations without halting its counterpart, **RWMutex** can be the solution

Example of RWMutex implementation can be found in:
https://medium.com/golangspec/sync-rwmutex-ca6c6c3208a0

Use Case Discussion

red - product service https://go-routine-alpine.herokuapp.com/product/123

green - insight service https://go-routine-alpine.herokuapp.com/insight/123

purple - flashsale service https://go-routine-alpine.herokuapp.com/flashsale/123

blue - media service https://go-routine-alpine.herokuapp.com/media/123

Code example can be found in : https://github.com/fakihariefnoto/goconcurrency101/tree/main/8_case1

In the example above you can find the example of real case implementation for concurrency compared to sequential implementation.