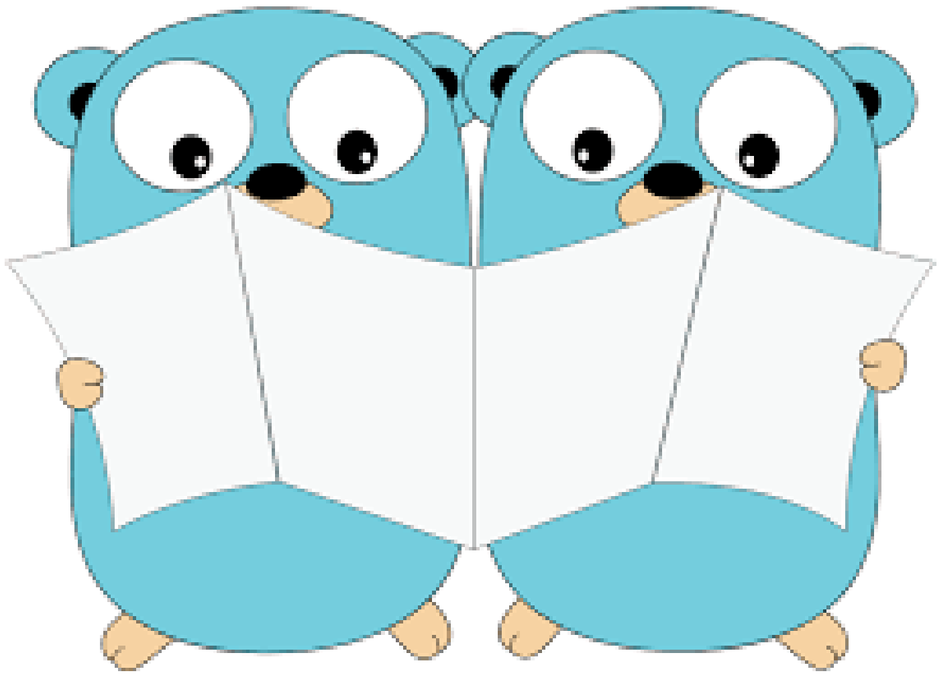
Concurrency in GO

golangbyte.com

Contents



Goroutiune

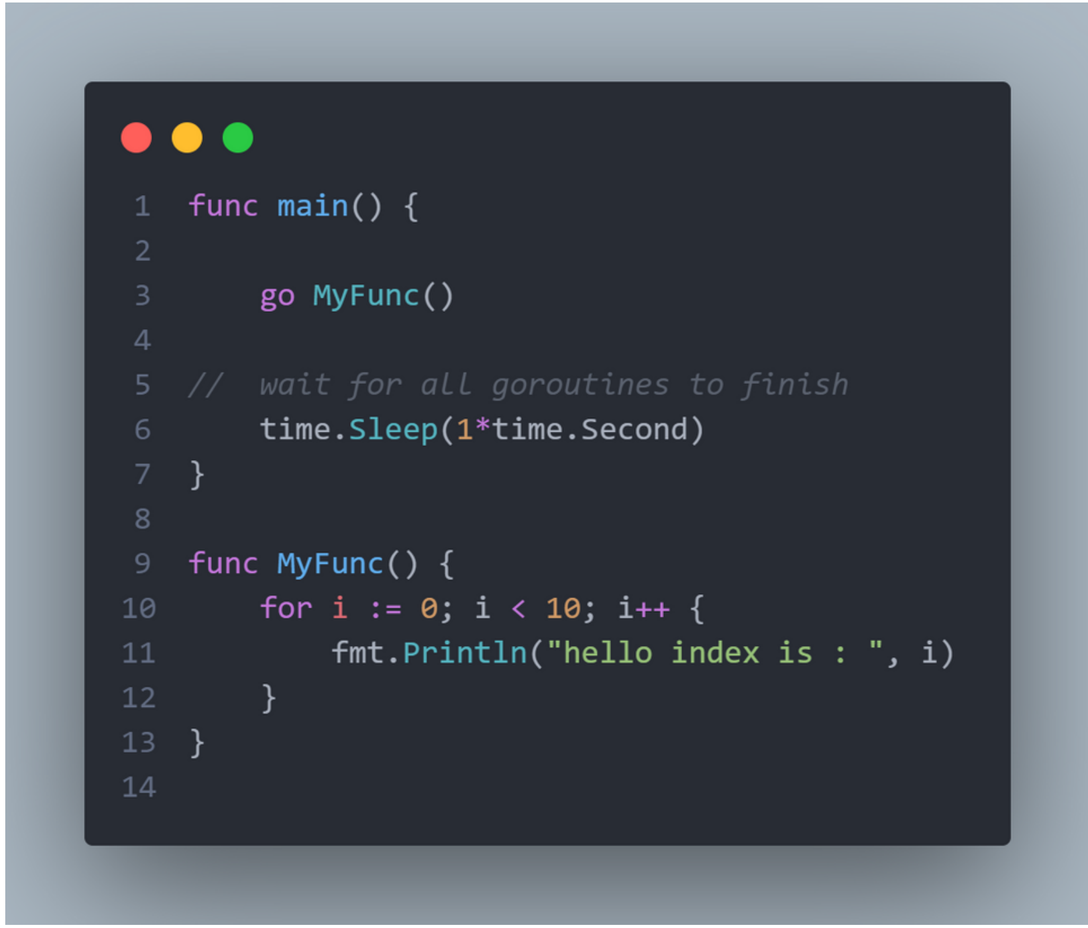
Goroutines are functions that run concurrently in their own runtime. they are extremely lightweight just 2 Kb compared to 8 Mb os thread.

Any function can be run as a goroutine by adding o keyword in-front of it.

Thousands of goroutines can be run at once without heavy memory or cpu time cost.

They can run in sync or concurrently.

They run in their own forked runtime.



```go
func main() {

    go MyFunc()

//  wait for all goroutines to finish
    time.Sleep(1*time.Second)
}

func MyFunc() {
    for i := 0; i < 10; i++ {
        fmt.Println("hello index is : ", i)
    }
}

```

Waitgroup

Goroutines gets executed in a fork join pattern. It means main routine has no knowledge of other goroutines. It can get closed and goroutines can still run. This is called goroutine leak.

To let main wait for all goroutines to execute successfully before exiting we use waitgroups that tells main routine how many goroutines to wait for and also gives done signal on each goroutine completion.

So all goroutines execute successfully then the program gets closed.

```go
func main() {
    var wg sync.WaitGroup
    //  add number of goroutines to track for
    wg.Add(1)
    //  run goroutine with wg as parameter
    go MyFunc(&wg)
    //  wait for all goroutines to finish
    wg.Wait()

    fmt.Println("all done now exiting")
}

func MyFunc(wg *sync.WaitGroup) {
//  call wg.done after function is executed
    defer wg.Done()
    for i := 0; i < 10; i++ {
        fmt.Println("hello index is : ", i)
    }
}

```

Channels Unbuffered

Channels are used to communicate between goroutines, they work as queues of specific types. they can be buffered or unbuffered.

Unbuffered channels have zero capacity so after pushing value into them pulling should be done immediately. This keeps goroutine communication in synchronization.

```go
func main() {
    var wg sync.WaitGroup
    wg.Add(2)

    // unbuffered channel
    myChan := make(chan int)

    go Generator(&wg,myChan)

    go func(wg *sync.WaitGroup) {
    defer wg.Done()
        for val := range myChan {
            fmt.Println("value is ", val)
        }
    }(&wg)

    wg.Wait()
    fmt.Println("all done now exiting")
}

func Generator(wg *sync.WaitGroup,myChan chan int) {
    defer wg.Done()
    for i := 0; i < 10; i++ {
        myChan <- i
    }
    close(myChan)
}
```

Channels Buffered

Buffered channels have a capacity as assigned to them. They can hold value in them before pulling till their set capacity.

They provide asynchronous communication between goroutines.

They may hold some value in them if all values were not consumed. So they should be checked for any residual value in them after execution.

```go
func main() {

    // buffered channel
    myChan := make(chan int, 10)

    myChan <- 1
    myChan <- 88
    myChan <- 99

    //  values are present in channel
    //check for values if present
    val, ok := <-myChan
    fmt.Println(val, " ", ok)

    fmt.Println(<-myChan)
    fmt.Println(<-myChan)

    fmt.Println("all done now exiting")
}
```

Channels Buffered 2

Buffered channels works as an asynchronous communication pipeline between two goroutines.

This makes them continue their operations after value is pushed into the channel so they donot block waiting for value to be pulled from channel.

```go
func main() {
    var wg sync.WaitGroup
    wg.Add(2)

    // buffered channel
    myChan := make(chan int,10)

    go Generator(&wg,myChan)

    go func(wg *sync.WaitGroup) {
    defer wg.Done()
        for val := range myChan {
            fmt.Println("value is ", val)
        }
    }(&wg)

    wg.Wait()
    fmt.Println("all done now exiting")
}

func Generator(wg *sync.WaitGroup,myChan chan int) {
    defer wg.Done()
    for i := 0; i < 15; i++ {
        myChan <- i
    }
    close(myChan)
}
```

Directed Channels

Channels can be directed as

- read only
- write only
- read and write channel

As required by the program to prevent any unwanted write or read to a channel.

```go
func main() {
//  example sendonly channel
    sendOnly := make(chan<- int, 5)

//  example recieveonly channel
    receiveOnly := make(<-chan int)

    biDirectional := make(chan int)

    go func(ch chan<- int) {
        for i := 1; i <= 5; i++ {
            ch <- i
        }
        close(ch)
    }(biDirectional)

    go func(ch <-chan int) {
        for val := range ch {
            fmt.Println("Received value:", val)
        }
    }(biDirectional)

    fmt.Println("done execution")

}

```

mutex

If two goroutines access same share resource and update it simultaneously, it can cause race conditions or unstable value assignment to the resource. to protect from this , we use mutual exclusion lock or mutex. it locks the resource before updating and releases after it is done for other goroutines to read or update its value.

Mutex lock are of two types :-

- Mutex lock which does not allow read or write to other goroutines.
- RWLock mutex that allows read but not write to other goroutines during operation.

```go
func main() {

    var wg sync.WaitGroup
    var mu sync.Mutex
    wg.Add(2)

    go Increment(&wg, &mu)
    go Decrement(&wg, &mu)

    wg.Wait()
    fmt.Println("counter is :", Counter)
    fmt.Println("all done now exiting")
}

func Increment(wg *sync.WaitGroup, mu *sync.Mutex) {
    defer wg.Done()
    for i := 0; i < 10; i++ {
        mu.Lock()
        Counter ++
        mu.Unlock()
    }
}
func Decrement(wg *sync.WaitGroup, mu *sync.Mutex) {
    defer wg.Done()
    for i := 0; i < 10; i++ {
        mu.Lock()
        Counter--
        mu.Unlock()
    }
}

```

select

Select is used to wait on multiple channel operations without blocking execution when working with multiple channels and goroutines.

It helps in coordinating communication between goroutines and is a essential block for many concurrency patterns.

It is like switch statement for concurrency.

```go
func main() {
    myChan := make(chan int)
    done := make(chan bool)

    go Increment(myChan,done)
    for {
        select {
        case msg1 :=  <-myChan:
            fmt.Println("value is ", msg1)
        case <-done:
            fmt.Println("exiting ")
            return
        default:
            fmt.Println("no activity")
        }
    }

}

func Increment(myChan chan int,done chan bool) {
    for i := 0; i < 10; i++ {
        myChan <- 99
    }
    done <- true
}
```

Context

Context is used to manage cancellation, deadlines and request scoped values across multiple goroutines.

It helps in propagating signals through concurrent operations to multiple goroutines to change their behavior or return as defined.

```go
func main() {

    ctx, cancel := context.WithTimeout(context.Background(), 1*time.Second)
    defer cancel()
    done := make(chan bool)

    go Worker(ctx, done)

    // Wait for the worker to complete or timeout
    if <-done {
        fmt.Println("Main: Worker has completed its task.")
    } else {
        fmt.Println("Main: Worker did not complete. timeout or was canceled.")
    }
    fmt.Println("all done now exiting")

}

func Worker(ctx context.Context, done chan bool) {
    for {
        select {
        case <-time.After(2 * time.Second):
            fmt.Println("woring good")
            done <- true
        case <-ctx.Done():
            fmt.Println("timeout")
            done <- false
        }
    }
}

```

github.com/subpxl/golangbyte

golangbyte.com