Mat Ryer Follow
Founder at MachineBox.io — Gopher, developer, speaker, author — BitBar app https://getbitbar.com — Author of Go Programming Blueprints
Nov 18, 2015 · 6 min read

Very basic concurrency for beginners in Go

Computers can do things very quickly, and if you can make them do many things at the same time, jobs get finished even sooner. Modern computers have processors with many cores, and spreading load across those cores maximises performance, and therefore speed of execution. Coding such multi-threaded code was hard, until Go came along and ruined it by making it so easy.

In this article, we will explore:

- The `go` keyword
- A common gotcha when writing concurrent code
- How to use the `sync.WaitGroup` to make sure our program doesn't terminate prematurely
- What kind of impact concurrent code can have on performance
- How making code run concurrently introduces some unpredictability

The go keyword

To make a function run in the background, insert the keyword `go` before the call (like you do with `defer`).

So this:

```
func main(){
  doSomething()
}
```

Becomes:

```
func main(){
  go doSomething()
}
```

Now, the `doSomething` function will run in the background in a *goroutine*.

> *If you want to play along with this article but don't have your Go environment setup yet, you can always use the Go playground over at http://play.golang.org.*

Simple example

Let's explore this in a little more detail, and look at some common gotchas that can trip us up.

```
package main

import (
  "fmt"
)

func main() {
  fmt.Println("start")
  doSomething()
  fmt.Println("end")
}

func doSomething() {
  fmt.Println("do something")
}
```

> *To run this code save it into a file called `one.go`, open a command line and do `go run one.go`*

Executing the above code will produce the following output:

```
start
do something
end
```

It's pretty predictable, because everything happens in the order in which we have written it.

Now let's make the `doSomething` function run in the background by modifying the main function:

```
func main() {
  fmt.Println("start")
  go doSomething()
  fmt.Println("end")
}
```

> *I really love that adding concurrency in Go takes only three key presses, g, o and a space.*

Running this code now (probably) produces the following output:

```
start
end
```

Oh no—what happened? And why only "probably?"

> *In Go, when the main function exits, the program stops.*

In our above code, the background task doesn't get chance to write "do something," before the program has ended—at which point, all goroutines are terminated.

To solve this, we could add a sleep operation at the bottom of our main function (with `time.Sleep`) but that's not a very nice solution—because we don't know how long our `doSomething` function might need to run.

Wait groups

The standard library gives us a package called `sync`, that has some great features which we can use to solve this problem properly.

A `sync.WaitGroup` is essentially a counter that we can increase (to indicate we want to wait for things), and decrease (to indicate things are done). Then we can tell code to wait until the WaitGroup counter reaches zero, which would mean all things have finished.

Update your code to look like this:

```
package main

import (
  "fmt"
  "sync"
)

var wg sync.WaitGroup

func main() {
  fmt.Println("start")
  wg.Add(1) // indicate we are going to wait for one thing
  go doSomething()
  fmt.Println("end")
  wg.Wait() // wait for all things to be done
  // end of program
}

func doSomething() {
  fmt.Println("do something")
  wg.Done() // this is done
}
```

Some key pieces explained:

- var wg sync.WaitGroup—defines a WaitGroup that is ready to use
- wg.Add(1)—indicates that there is 1 thing to wait for (our doSomething function)
- wg.Wait()—indicates that code should block until the WaitGroup counter reaches zero
- wg.Done()—indicates that 1 thing has finished

> *Notice that we don't ever create a new WaitGroup, this is because in its default zero state (the state you get just by defining it), a WaitGroup is*

> *ready to use. Check out the sync.WaitGroup documentation for more about how to use them.*

Now re-run your code, and you'll (probably) see the following output:

```
start
end
do something
```

What's with all this "probably" talk? When we ask Go to run code concurrently, we can't tell it exactly how and when to run the instructions. So it introduces some unpredictability in our code. As long as we understand this, it's not a problem—and we can use things like WaitGroup objects to introduce synchronisation points between our goroutines.

> *If we move our wg.Wait() instruction to above the line where we write "end," we can be sure that things happen in the order in which we intend. But then there wouldn't be much point in running the code in the background, since we'd just be waiting in our 'main goroutine' anyway.*

How quickly can we do our multiplication tables?

Create a new file called `times.go` with the following code:

```
package main

import (
  "fmt"
  "time"
)

func main() {
  timestable(2)
}

func timestable(x int) {
  for i := 1; i <= 12; i++ {
    fmt.Printf("%d x %d = %d\n", i, x, x*i)
    time.Sleep(100 * time.Millisecond)
  }
}
```

> *The weird %d symbols in the Printf argument are special verbs that you can learn more about in the fmt documentation. Essentially, it formats the numbers in a nice human readable way.*

Running the above code will print out the multiplication table (we called it our “times table” at school) for the number 2:

```
$ go run times.go
1 x 2 = 2
2 x 2 = 4
3 x 2 = 6
4 x 2 = 8
5 x 2 = 10
6 x 2 = 12
7 x 2 = 14
8 x 2 = 16
9 x 2 = 18
10 x 2 = 20
11 x 2 = 22
12 x 2 = 24
```

> *The `time.Sleep` instruction tells the code to block (or wait) for 100ms, this is to slow things down so we can really see the impact of making code run concurrently—you would probably never do this in real code.*

Let’s enhance our main function to give us the times tables for all numbers that we needed to learn at school:

```
func main() {
  for n := 2; n <= 12; n++ {
    timestable(n)
  }
}
```

Now, our code will call the timestable function for all numbers between 2 and 12 inclusively.

How long does it take?

On unix systems (including Macs), running a command prefixed by the `time` command will tell us how long our code took to execute. In my case it was just over 13 seconds:

```
$ time go run times.go
real 0m13.762s
user 0m0.261s
sys 0m0.059s
```

> *13 seconds to do this kind of work is embarassing if it weren't for our time.Sleep instructions. But remember, it's just to simulate some resource intensive task.*

Multiplying concurrency

Now we are going to make the code run concurrently, to see if we can speed things up. Update the code:

```
package main

import (
  "fmt"
  "sync"
  "time"
)

var wg sync.WaitGroup

func main() {
  for n := 2; n <= 12; n++ {
    wg.Add(1)
    go timestable(n)
  }
  wg.Wait()
}

func timestable(x int) {
  for i := 1; i <= 12; i++ {
    fmt.Printf("%d x %d = %d\n", i, x, x*i)
    time.Sleep(100 * time.Millisecond)
  }
  wg.Done()
}
```

see http://play.golang.org/p/NkNzaewRjH

As we did before, we've made the `timestable` function run concurrently, and used a `sync.WaitGroup` to make sure our program

doesn't end before all calculations have finished.

> *This time, we have the wg.Add function call inside the for loop. This is more common, and allows us to calculate the times table for a variable number of things.*

Running this code with the `time` command shows us the impact that making our code run concurrently has had:

```
$ time go run times.go
real 0m1.431s
user 0m0.253s
sys 0m0.053s
```

We've gone from over 13 seconds, to just under 1.5 seconds. Your results may differ, but they will be significantly improved.

The order is confusing

Taking a closer look at the output will reveal again that the order the instructions get executed in is unpredictable:

```
12 x 7 = 84
12 x 6 = 72
12 x 3 = 36
12 x 8 = 96
12 x 12 = 144
12 x 4 = 48
12 x 9 = 108
12 x 10 = 120
12 x 2 = 24
12 x 5 = 60
12 x 11 = 132
```

Even though we are counting sensibly upwards, the order the operations actually occur varies. This is because each goroutine runs at its own pace, depending on things out of our control (like other things going on in the processor at the time.)

If we care about the order, then we'd need to use Go's inbuilt Channels to send data safely across the goroutines. But that's out of scope for this

article.

Conclusion

We saw that making code run concurrently in Go is as easy as three key presses, and provided we are careful about how that behaves, we can get some pretty staggering results.

- Next, you should learn about Channels.