Concurrency

You can find all the code for this chapter here

Here's the setup: a colleague has written a function, `CheckWebsites`, that checks the status of a list of URLs.

```
package concurrency

type WebsiteChecker func(string) bool

func CheckWebsites(wc WebsiteChecker, urls []string) map[string]bool {
        results := make(map[string]bool)

        for _, url := range urls {
                results[url] = wc(url)
        }

        return results
}
```

It returns a map of each URL checked to a boolean value - `true` for a good response, `false` for a bad response.

You also have to pass in a `WebsiteChecker` which takes a single URL and returns a boolean. This is used by the function to check all the websites.

Using dependency injection has allowed them to test the function without making real HTTP calls, making it reliable and fast.

Here's the test they've written:

```
package concurrency

import (
        "reflect"
        "testing"
)

```

```
func mockWebsiteChecker(url string) bool {
        if url == "waat://furhurterwe.geds" {
                return false
        }
        return true
}

func TestCheckWebsites(t *testing.T) {
        websites := []string{
                "http://google.com",
                "http://blog.gypsydave5.com",
                "waat://furhurterwe.geds",
        }

        want := map[string]bool{
                "http://google.com":          true,
                "http://blog.gypsydave5.com": true,
                "waat://furhurterwe.geds":    false,
        }

        got := CheckWebsites(mockWebsiteChecker, websites)

        if !reflect.DeepEqual(want, got) {
                t.Fatalf("Wanted %v, got %v", want, got)
        }
}
```

The function is in production and being used to check hundreds of websites. But your colleague has started to get complaints that it's slow, so they've asked you to help speed it up.

Write a test

Let's use a benchmark to test the speed of `CheckWebsites` so that we can see the effect of our changes.

```
package concurrency

import (
        "testing"
```

```
)       "time"

func slowStubWebsiteChecker(_ string) bool {
        time.Sleep(20 * time.Millisecond)
        return true
}

func BenchmarkCheckWebsites(b *testing.B) {
        urls := make([]string, 100)
        for i := 0; i < len(urls); i++ {
                urls[i] = "a url"
        }
        b.ResetTimer()
        for i := 0; i < b.N; i++ {
                CheckWebsites(slowStubWebsiteChecker, urls)
        }
}
```

The benchmark tests `CheckWebsites` using a slice of one hundred urls and uses a new fake implementation of `WebsiteChecker`. `slowStubWebsiteChecker` is deliberately slow. It uses `time.Sleep` to wait exactly twenty milliseconds and then it returns true. We use `b.ResetTimer()` in this test to reset the time of our test before it actually runs

When we run the benchmark using `go test -bench=.` (or if you're in Windows Powershell `go test -bench="."`):

```
pkg: github.com/gypsydave5/learn-go-with-tests/concurrency/v0
BenchmarkCheckWebsites-4               1        2249228637 ns/op
PASS
ok      github.com/gypsydave5/learn-go-with-tests/concurrency/v0        2.268s
```

`CheckWebsites` has been benchmarked at 2249228637 nanoseconds - about two and a quarter seconds.

Let's try and make this faster.

Write enough code to make it pass

Now we can finally talk about concurrency which, for the purposes of the following, means 'having more than one thing in progress'. This is something that we do naturally everyday.

For instance, this morning I made a cup of tea. I put the kettle on and then, while I was waiting for it to boil, I got the milk out of the fridge, got the tea out of the cupboard, found my favourite mug, put the teabag into the cup and then, when the kettle had boiled, I put the water in the cup.

What I *didn't* do was put the kettle on and then stand there blankly staring at the kettle until it boiled, then do everything else once the kettle had boiled.

If you can understand why it's faster to make tea the first way, then you can understand how we will make `CheckWebsites` faster. Instead of waiting for a website to respond before sending a request to the next website, we will tell our computer to make the next request while it is waiting.

Normally in Go when we call a function `doSomething()` we wait for it to return (even if it has no value to return, we still wait for it to finish). We say that this operation is *blocking* - it makes us wait for it to finish. An operation that does not block in Go will run in a separate *process* called a *goroutine*. Think of a process as reading down the page of Go code from top to bottom, going 'inside' each function when it gets called to read what it does. When a separate process starts it's like another reader begins reading inside the function, leaving the original reader to carry on going down the page.

To tell Go to start a new goroutine we turn a function call into a `go` statement by putting the keyword `go` in front of it: `go doSomething()`.

```
1  package concurrency
2
3  type WebsiteChecker func(string) bool
4
5  func CheckWebsites(wc WebsiteChecker, urls []string) map[string]bool {
6          results := make(map[string]bool)
7
8          for _, url := range urls {
9                  go func() {
10                         results[url] = wc(url)
11                 }()
12         }
13
14         return results
15 }
```

Because the only way to start a goroutine is to put `go` in front of a function call, we often use *anonymous functions* when we want to start a goroutine. An anonymous function literal looks just

the same as a normal function declaration, but without a name (unsurprisingly). You can see one above in the body of the `for` loop.

Anonymous functions have a number of features which make them useful, two of which we're using above. Firstly, they can be executed at the same time that they're declared - this is what the `()` at the end of the anonymous function is doing. Secondly they maintain access to the lexical scope they are defined in - all the variables that are available at the point when you declare the anonymous function are also available in the body of the function.

The body of the anonymous function above is just the same as the loop body was before. The only difference is that each iteration of the loop will start a new goroutine, concurrent with the current process (the `WebsiteChecker` function) each of which will add its result to the results map.

But when we run `go test`:

```
--- FAIL: TestCheckWebsites (0.00s)
        CheckWebsites_test.go:31: Wanted map[http://google.com:true http://blog.gypsy
FAIL
exit status 1
FAIL    github.com/gypsydave5/learn-go-with-tests/concurrency/v1        0.010s
```

A quick aside into a parallel(ism) universe...

You might not get this result. You might get a panic message that we're going to talk about in a bit. Don't worry if you got that, just keep running the test until you *do* get the result above. Or pretend that you did. Up to you. Welcome to concurrency: when it's not handled correctly it's hard to predict what's going to happen. Don't worry - that's why we're writing tests, to help us know when we're handling concurrency predictably.

... and we're back.

We are caught by the original tests `CheckWebsites` is now returning an empty map. What went wrong?

None of the goroutines that our `for` loop started had enough time to add their result to the `results` map; the `WebsiteChecker` function is too fast for them, and it returns the still empty map.

To fix this we can just wait while all the goroutines do their work, and then return. Two seconds ought to do it, right?

```go
package concurrency

import "time"

type WebsiteChecker func(string) bool

func CheckWebsites(wc WebsiteChecker, urls []string) map[string]bool {
        results := make(map[string]bool)

        for _, url := range urls {
                go func() {
                        results[url] = wc(url)
                }()
        }

        time.Sleep(2 * time.Second)

        return results
}
```

Now when we run the tests you get (or don't get - see above):

```
--- FAIL: TestCheckWebsites (0.00s)
        CheckWebsites_test.go:31: Wanted map[http://google.com:true http://blog.gypsy
FAIL
exit status 1
FAIL    github.com/gypsydave5/learn-go-with-tests/concurrency/v1        0.010s
```

This isn't great - why only one result? We might try and fix this by increasing the time we wait - try it if you like. It won't work. The problem here is that the variable `url` is reused for each iteration of the `for` loop - it takes a new value from `urls` each time. But each of our goroutines have a reference to the `url` variable - they don't have their own independent copy. So they're *all* writing the value that `url` has at the end of the iteration - the last url. Which is why the one result we have is the last url.

To fix this:

```
package concurrency

import (
        "time"
)

type WebsiteChecker func(string) bool

func CheckWebsites(wc WebsiteChecker, urls []string) map[string]bool {
        results := make(map[string]bool)

        for _, url := range urls {
                go func(u string) {
                        results[u] = wc(u)
                }(url)
        }

        time.Sleep(2 * time.Second)

        return results
}
```

By giving each anonymous function a parameter for the url - `u` - and then calling the anonymous function with the `url` as the argument, we make sure that the value of `u` is fixed as the value of `url` for the iteration of the loop that we're launching the goroutine in. `u` is a copy of the value of `url`, and so can't be changed.

Now if you're lucky you'll get:

```
PASS
ok      github.com/gypsydave5/learn-go-with-tests/concurrency/v1        2.012s
```

But if you're unlucky (this is more likely if you run them with the benchmark as you'll get more tries)

```
fatal error: concurrent map writes

goroutine 8 [running]:
```

```
runtime.throw(0x12c5895, 0x15)
        /usr/local/Cellar/go/1.9.3/libexec/src/runtime/panic.go:605 +0x95 fp=0xc42003

runtime.mapassign_faststr(0x1271d80, 0xc42007acf0, 0x12c6634, 0x17, 0x0)
        /usr/local/Cellar/go/1.9.3/libexec/src/runtime/hashmap_fast.go:783 +0x4f5 fp=
github.com/gypsydave5/learn-go-with-tests/concurrency/v3.WebsiteChecker.func1(0xc4200
        /Users/gypsydave5/go/src/github.com/gypsydave5/learn-go-with-tests/concurrenc
runtime.goexit()
        /usr/local/Cellar/go/1.9.3/libexec/src/runtime/asm_amd64.s:2337 +0x1 fp=0xc42
created by github.com/gypsydave5/learn-go-with-tests/concurrency/v3.WebsiteChecker
        /Users/gypsydave5/go/src/github.com/gypsydave5/learn-go-with-tests/concurrenc

            many more scary lines of text
```

This is long and scary, but all we need to do is take a breath and read the stacktrace: `fatal error: concurrent map writes`. Sometimes, when we run our tests, two of the goroutines write to the results map at exactly the same time. Maps in Go don't like it when more than one thing tries to write to them at once, and so `fatal error`.

This is a *race condition*, a bug that occurs when the output of our software is dependent on the timing and sequence of events that we have no control over. Because we cannot control exactly when each goroutine writes to the results map, we are vulnerable to two goroutines writing to it at the same time.

Go can help us to spot race conditions with its built in *race detector*. To enable this feature, run the tests with the `race` flag: `go test -race`.

You should get some output that looks like this:

```
==================
WARNING: DATA RACE
Write at 0x00c420084d20 by goroutine 8:
  runtime.mapassign_faststr()
      /usr/local/Cellar/go/1.9.3/libexec/src/runtime/hashmap_fast.go:774 +0x0
  github.com/gypsydave5/learn-go-with-tests/concurrency/v3.WebsiteChecker.func1()
      /Users/gypsydave5/go/src/github.com/gypsydave5/learn-go-with-tests/concurrency/

Previous write at 0x00c420084d20 by goroutine 7:
  runtime.mapassign_faststr()
      /usr/local/Cellar/go/1.9.3/libexec/src/runtime/hashmap_fast.go:774 +0x0
  github.com/gypsydave5/learn-go-with-tests/concurrency/v3.WebsiteChecker.func1()
      /Users/gypsydave5/go/src/github.com/gypsydave5/learn-go-with-tests/concurrency/

Goroutine 8 (running) created at:
```

```
  github.com/gypsydave5/learn-go-with-tests/concurrency/v3.WebsiteChecker()
      /Users/gypsydave5/go/src/github.com/gypsydave5/learn-go-with-tests/concurrency/
  github.com/gypsydave5/learn-go-with-tests/concurrency/v3.TestWebsiteChecker()
      /Users/gypsydave5/go/src/github.com/gypsydave5/learn-go-with-tests/concurrency/
  testing.tRunner()
      /usr/local/Cellar/go/1.9.3/libexec/src/testing/testing.go:746 +0x16c

Goroutine 7 (finished) created at:
  github.com/gypsydave5/learn-go-with-tests/concurrency/v3.WebsiteChecker()
      /Users/gypsydave5/go/src/github.com/gypsydave5/learn-go-with-tests/concurrency/
  github.com/gypsydave5/learn-go-with-tests/concurrency/v3.TestWebsiteChecker()
      /Users/gypsydave5/go/src/github.com/gypsydave5/learn-go-with-tests/concurrency/
  testing.tRunner()
      /usr/local/Cellar/go/1.9.3/libexec/src/testing/testing.go:746 +0x16c
==================
```

The details are, again, hard to read - but `WARNING: DATA RACE` is pretty unambiguous. Reading into the body of the error we can see two different goroutines performing writes on a map:

`Write at 0x00c420084d20 by goroutine 8:`

is writing to the same block of memory as

`Previous write at 0x00c420084d20 by goroutine 7:`

On top of that we can see the line of code where the write is happening:

`/Users/gypsydave5/go/src/github.com/gypsydave5/learn-go-with-tests/concurrency/v3/websiteChecker.go:12`

and the line of code where goroutines 7 an 8 are started:

`/Users/gypsydave5/go/src/github.com/gypsydave5/learn-go-with-tests/concurrency/v3/websiteChecker.go:11`

Everything you need to know is printed to your terminal - all you have to do is be patient enough to read it.

Channels

We can solve this data race by coordinating our goroutines using *channels*. Channels are a Go data structure that can both receive and send values. These operations, along with their details, allow communication between different processes.

In this case we want to think about the communication between the parent process and each of the goroutines that it makes to do the work of running the `WebsiteChecker` function with the url.

```
package concurrency

type WebsiteChecker func(string) bool
type result struct {
        string
        bool
}

func CheckWebsites(wc WebsiteChecker, urls []string) map[string]bool {
        results := make(map[string]bool)
        resultChannel := make(chan result)

        for _, url := range urls {
                go func(u string) {
                        resultChannel <- result{u, wc(u)}
                }(url)
        }

        for i := 0; i < len(urls); i++ {
                r := <-resultChannel
                results[r.string] = r.bool
        }

        return results
}
```

Alongside the `results` map we now have a `resultChannel`, which we `make` in the same way. `chan result` is the type of the channel - a channel of `result`. The new type, `result` has been made to associate the return value of the `WebsiteChecker` with the url being checked - it's a struct of `string` and `bool`. As we don't need either value to be named, each of them is anonymous within the struct; this can be useful in when it's hard to know what to name a value.

Now when we iterate over the urls, instead of writing to the `map` directly we're sending a `result` struct for each call to `wc` to the `resultChannel` with a *send statement*. This uses the `<-` operator, taking a channel on the left and a value on the right:

```
// Send statement
resultChannel <- result{u, wc(u)}
```

The next `for` loop iterates once for each of the urls. Inside we're using a *receive expression*, which assigns a value received from a channel to a variable. This also uses the `<-` operator, but with the two operands now reversed: the channel is now on the right and the variable that we're assigning to is on the left:

```
// Receive expression
r := <-resultChannel
```

We then use the `result` received to update the map.

By sending the results into a channel, we can control the timing of each write into the results map, ensuring that it happens one at a time. Although each of the calls of `wc`, and each send to the result channel, is happening in parallel inside its own process, each of the results is being dealt with one at a time as we take values out of the result channel with the receive expression.

We have parallelized the part of the code that we wanted to make faster, while making sure that the part that cannot happen in parallel still happens linearly. And we have communicated across the multiple processes involved by using channels.

When we run the benchmark:

```
pkg: github.com/gypsydave5/learn-go-with-tests/concurrency/v2
BenchmarkCheckWebsites-8             100          23406615 ns/op
PASS
ok      github.com/gypsydave5/learn-go-with-tests/concurrency/v2        2.377s
```

23406615 nanoseconds - 0.023 seconds, about one hundred times as fast as original function. A great success.

Wrapping up

This exercise has been a little lighter on the TDD than usual. In a way we've been taking part in one long refactoring of the `CheckWebsites` function; the inputs and outputs never changed, it just got faster. But the tests we had in place, as well as the benchmark we wrote, allowed us to refactor

`CheckWebsites` in a way that maintained confidence that the software was still working, while demonstrating that it had actually become faster.

In making it faster we learned about

- *goroutines*, the basic unit of concurrency in Go, which let us check more than one website at the same time.
- *anonymous functions*, which we used to start each of the concurrent processes that check websites.
- *channels*, to help organize and control the communication between the different processes, allowing us to avoid a *race condition* bug.
- *the race detector* which helped us debug problems with concurrent code

Make it fast

One formulation of an agile way of building software, often misattributed to Kent Beck, is:

> Make it work, make it right, make it fast

Where 'work' is making the tests pass, 'right' is refactoring the code, and 'fast' is optimizing the code to make it, for example, run quickly. We can only 'make it fast' once we've made it work and made it right. We were lucky that the code we were given was already demonstrated to be working, and didn't need to be refactored. We should never try to 'make it fast' before the other two steps have been performed because

> Premature optimization is the root of all evil -- Donald Knuth