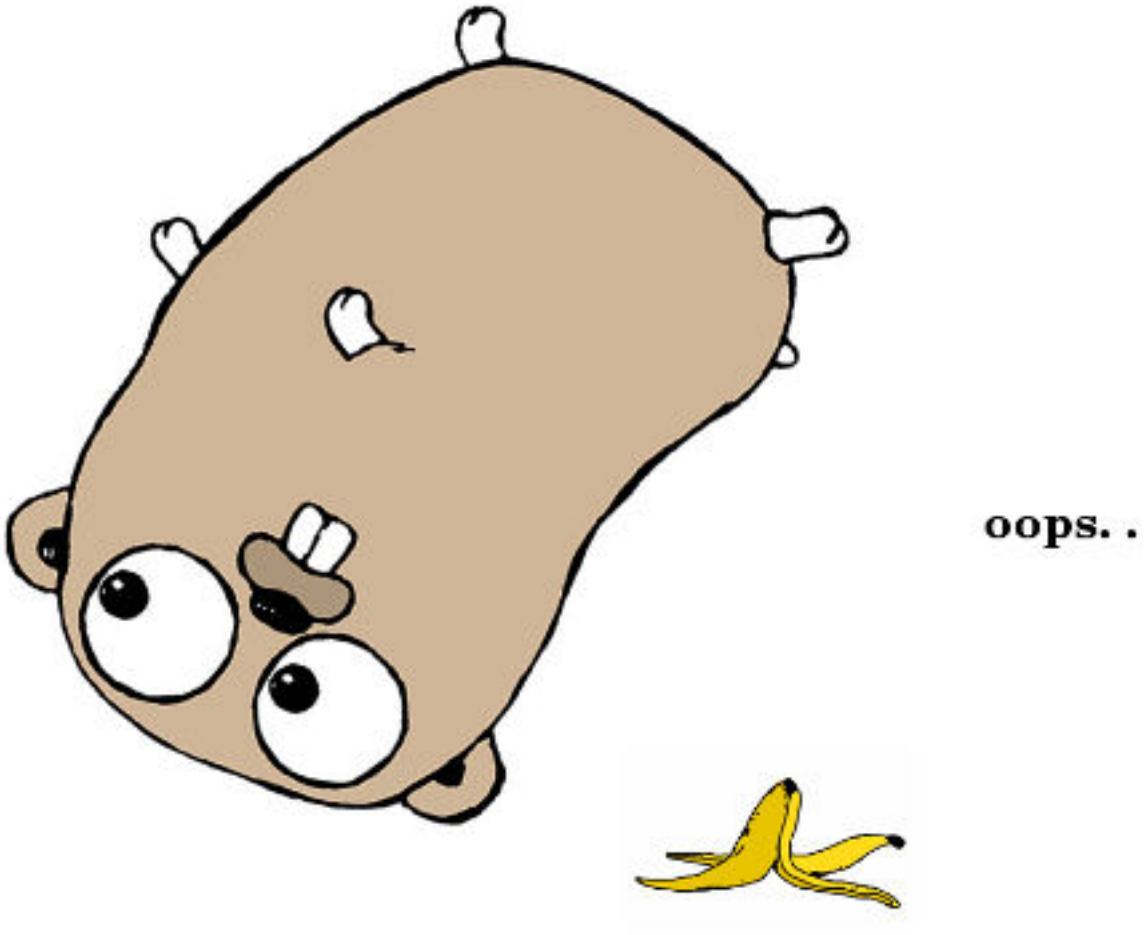
Learn from my Go mistakes, Part 1

Closing over loop variables



This post is about a mistake that's common and easy to make in Golang. I show a few examples, explain why it happens and how you can avoid it. This is explained in many places already (see links at the end). My goal is to be more thorough than the others.

There might be many of these blogposts. There's at least one more topic I want to write about. We'll see.

1 Closing over a loop variable

This mistake is very easy to make in some situations. If you please, take a look at the code below and try to figure out why Example B fails.

(Most code snipet has a link in the header to the Go Playground where you can run and play with the code. All links open in this window).

Print out the numbers from 0 to 2:

(On the playgroud)

```
package main

import (
```

```
  "fmt"
  "time"
)

func main() {

  // A
  fmt.Println("Example A - Count to 2, This works :) ")
  for a := 0; a < 3; a++ {
    fmt.Println("a is:", a)
  }

  // B
  fmt.Println("\n\nExample B - Count to 2, concurrently.")
  fmt.Println("This does not work :( ")
  for b := 0; b < 3; b++ {
    // Run fmt.Println(..) in the background
    go func() {
      fmt.Println("b is:", b)
    }()
  }

  // we sleep to let the go functions finish before the program quits
  time.Sleep(2 * time.Second)
}
```

Running the above code prints:

```
Example A - Count to 2, This works :)
a is: 0
a is: 1
a is: 2


Example B - Count to 2, concurrently.
This does not work :(
b is: 3
b is: 3
b is: 3
```

Huh? 3, 3, 3? Where's zero and one? And wasn't it suppose to stop at two? Bah! We can't even count to 2, what's going on? Two things:

1.1 Closures

Firstly, when we print variable *b* in Example B what are we even referring to? It's inside of a function, and there's no variable *b* inside this function body. If you did this in a normal function you would get a nice little error from the compiler complaining that there's no *b*.

This does not happen here. The function we have in Example B, *func(){..}*, is not a normal function: it's in the middle of another one and it lacks a name. These types of functions are often call *function literals* or *anonymous functions*. I like "anonymous functions". An anonymous function can "see" variables from the function it's inside (enclosed in). That means that in our example it can access variable *b*.

```
// B
fmt.Println("\n\nExample B - Count to 2, concurrently.")
fmt.Println("This does not work :( ")
for b := 0; b < 3; b++ {
        go func() {
                fmt.Println("b is:", b)
        }()
}
```

This is not a bug in Go, by the way. It's intended behavior that can be used to do some pretty advanced stuff. See the "function closures" links at the end.

1.2 Variable reuse

Our anonymous function can access variable *b*. But it's the same variable *b*, accessed 3 time in the loop. It's declared once before the loop starts to run. It then changes value from 0 to 1, then to 2 and finally to 3. Then the loop stops since 3 is not less than 3.

1.3 Goroutines runs out of order

With our *go func()..* statement we start a new goroutine. They run concurrently, which they do not run one after the other in an orderly fashion. They could in theory run one after the other, or they could all run at the same time (in parallel). Or maybe the "last one" runs first and then the "first one" and the "second one" runs last, or maybe.. think you get the point; it's unpredictable. Well this is kind of the point of using Goroutines in the first place. We want to be able to run many things at once. In this example it does not make a lot of sense, but imagine, if you will, that the printing took 10 seconds. Or we did something else that took 10 seconds. In that case Example A would take 30 seconds to complete and Example B would only take 10 seconds. Great!

We can think of the go statement as "run this code in the background at some point". It could run right away, or it could run later. Thinking like this it becomes more clear why Example B fails. It's possible, likely even, that the loop finishes before the first *fmt.Println("b is:", b)* ever runs. So the loops runs, increasing *b* to 3 and then stopping. **Then** the goroutines run and print variable b. They all look at the same variable b, the one declared at the start of the loop. They never got their own copy. At this point *b* is 3, so they all print 3.

2 Fixing it

There are several ways to change our code and fix this bug. I've listed the 3 I could come up with at this time, with the last one possibly being the most "idiomatic" go.

Each of these examples should print something like this if you run them. Note that the numbers can be out of order, since the goroutines can run in any possible order, or all at once.

```
Fix 2 for Example B
dd is: 1
dd is: 0
dd is: 2
```

2.1 Alternative 1

One easy way of fixing this is to change the anonymous function to a normal function call. Normal functions can not refer to variables "outside" itself. This forces us to pass the variable we want to use as an argument to the function (print() in our

example). When you call a normal function the arguments are copied, every time you call it. That is, the value of *c* (0, 1 or 2) is copied and given to the invoked function. Therefore this works as you would expect:

(On the playgroud)

```
package main

import (
  "fmt"
  "time"
)

func main() {

  // Fix 1
  fmt.Println("\n\nFix 1 for Example B")
  for c := 0; c < 3; c++ {
    // same code as earlier,
    // but no longer in an anonymous function:
    go print(c)

    // The value of c is copied when we call print()
    //   so it's no longer reffering to the same variable.

    // This copying happens before the code
    //   in print() actually runs
  }

  time.Sleep(2 * time.Second)
}

func print(c int) {
  // The value of c was copied.
  // So 'c' here is not the same as 'c' in the for loop above.
  // It's a "coincidence" that they are both named c
  // (it's irrelevant).
  fmt.Println("c is:", c)
}
```

2.2 Alternative 2

But let's say you wanted to keep the anonymous function? They can certainly be useful, creating a named function that is only used in one place is kind of ugly. This code fixes this by explicitly copying the variable before using it in the anonymous function. This works because each run (iteration) through the loop now gets its own variable, with its own value. So it does not matter when the goroutine actually runs. It will refer to its own variable *dd* that never changes.

(On the playgroud)

```
package main

import (
  "fmt"
  "time"
)

func main() {

  // Fix 2
  fmt.Println("Fix 2 for Example B")
  for d := 0; d < 3; d++ {
    // The value of d is copied into its own variable.
```

```
    // This happens for every iteration in the loop.
    dd := d
    go func() {
      fmt.Println("dd is:", dd)
    }()
  }

  time.Sleep(2 * time.Second)

}
```

2.3 Alternative 3

Finally this fix is similar to the first one (well, actually they're all pretty similar). Here we also copy the variable for every iteration of the **for** loop. Our anonymous function still sees the variable *e* but ignores it. Instead we send in the variable we want to access when we call the anonymous function.

Having both *e* and *ee* inside the function body can be a little confusing. We could for example use variable *e* by mistake and the compiler would not complain. Therefore some people prefer to name the second variable the same as the first one (*e* instead of *ee* in our example). This has the effect of hiding (shadowing) the first one and thus making it inaccessible.

(On the playgroud)

```
package main

import (
  "fmt"
  "time"
)

func main() {

  // Fix 3
  fmt.Println("Fix 3 for Example B")
  for e := 0; e < 3; e++ {
    go func(ee int) {
      // The value of e is copied into its own variable ee
      // (We could also call this new variable e)
      fmt.Println("ee is:", ee)
    }(e) // e becomes ee inside the function
  }

  time.Sleep(2 * time.Second)

}
```

3 That's it

Ok, that's it =) Hit me up on twitter (@oyvindsk) if you have any questions or corrections. All honest feedback is welcome. You can also enter your email underneath to get future articles.

4 Links

4.1 Closures in Go

- https://gobyexample.com/closures

- https://tour.golang.org/moretypes/25

4.2 Other

- Entry at the 50 shades of Go for this gotcha