Table of Contents

- Introduction
- Using reference to loop iterator variable
- Using goroutines on loop iterator variables

Introduction

When new programmers start using Go or when old Go programmers start using a new concept, there are some common mistakes that many of them make. Here is a non-exhaustive list of some frequent mistakes that show up on the mailing lists and in IRC.

Using reference to loop iterator variable

In Go, the loop iterator variable is a single variable that takes different values in each loop iteration. This is very efficient, but might lead to unintended behavior when used incorrectly. For example, see the following program:

```
func main() {
    var out []*int
    for i := 0; i < 3; i++ {
        out = append(out, &i)
    }
    fmt.Println("Values:", *out[0], *out[1], *out[2])
    fmt.Println("Addresses:", out[0], out[1], out[2])
}
```

It will output unexpected results:

```
Values: 3 3 3
Addresses: 0x40e020 0x40e020 0x40e020
```

Explanation: in each iteration we append the address of `i` to the `out` slice, but since it is the same variable, we append the same address which eventually contains the last value that was assigned to `i`. One of the solutions is to copy the loop variable into a new variable:

```
 for i := 0; i < 3; i++ {
+        i := i // Copy i into a new variable.
     out = append(out, &i)
 }
```

The new output of the program is what was expected:

```
Values: 0 1 2
Addresses: 0x40e020 0x40e024 0x40e028
```

Explanation: the line `i := i` copies the loop variable `i` into a new variable scoped to the for loop body block, also called `i`. The address of the new variable is the one that is appended to the array, which makes it outlive the for loop body block. In each loop iteration a new variable is created.

While this example might look a bit obvious, the same unexpected behavior could be more hidden in some other cases. For example, the loop variable can be an array and the reference can be a slice:

```
func main() {
    var out [][]int
    for _, i := range [][1]int{{1}, {2}, {3}} {
        out = append(out, i[:])
    }
    fmt.Println("Values:", out)
}
```

Output:

```
Values: [[3] [3] [3]]
```

The same issue can be demonstrated also when the loop variable is being used in a Goroutine (see the following section).

Using goroutines on loop iterator variables

When iterating in Go, one might attempt to use goroutines to process data in parallel. For example, you might write something like this, using a closure:

```
for _, val := range values {
    go func() {
        fmt.Println(val)
    }()
}
```

The above for loops might not do what you expect because their `val` variable is actually a single variable that takes on the value of each slice element. Because the closures are all only bound to that one variable, there is a very good chance that when you run this code you will see the last element printed for every iteration instead of each value in sequence, because the goroutines will probably not begin executing until after the loop.

The proper way to write that closure loop is:

```
for _, val := range values {
    go func(val interface{}) {
        fmt.Println(val)
    }(val)
}
```

By adding val as a parameter to the closure, `val` is evaluated at each iteration and placed on the stack for the goroutine, so each slice element is available to the goroutine when it is eventually executed.

It is also important to note that variables declared within the body of a loop are not shared between iterations, and thus can be used separately in a closure. The following code uses a common index variable `i` to create separate `val` s, which results in the expected behavior:

```
for i := range valslice {
    val := valslice[i]
    go func() {
        fmt.Println(val)
    }()
}
```

Note that without executing this closure as a goroutine, the code runs as expected. The following example prints out the integers between 1 and 10.

```
for i := 1; i <= 10; i++ {
    func() {
        fmt.Println(i)
    }()
}
```

Even though the closures all still close over the same variable (in this case, `i`), they are executed before the variable changes, resulting in the desired behavior. https://go.dev/doc/faq#closures_and_goroutines

You may find another, similar situation like the following:

```
for _, val := range values {
    go val.MyMethod()
}

func (v *val) MyMethod() {
    fmt.Println(v)
}
```

The above example also will print last element of values, the reason is same as closure. To fix the issue declare another variable inside the loop.

```
for _, val := range values {
    newVal := val
    go newVal.MyMethod()
}

func (v *val) MyMethod() {
    fmt.Println(v)
}
```