Programming in Go

Matt Holiday
Christmas 2020

Closures

Scope vs lifetime

Scope is static, based on the code at compile time

Lifetime depends on program execution (*runtime*)

```
package xyz

func doIt() *int {
    var b int
    . . .

    return &b
}
```

Variable `b` can only be seen inside `doit`, but its value will live on

The value (object) will live so long as part of the program keeps a pointer to it

What is a closure?

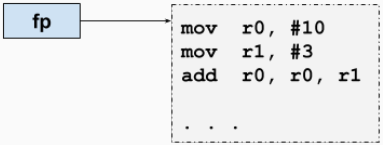
A *closure* is when a function inside another function "closes over" one or more local variables of the outer function

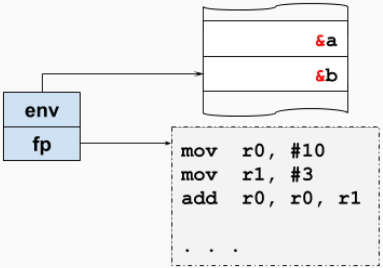
```
func fib() func() int {
    a, b := 0, 1

    return func() int {
        a, b = b, a+b
        return b
    }
}
```

The inner function gets a **reference** to the outer function's vars

Those variables may end up with a much longer *lifetime* than expected — as long as there's a reference to the inner function

Closures: how they work





Closures: scope vs lifetime

```
package main
import "fmt"

func fib() func() int {
    a, b := 0, 1
    return func() int {
        a, b = b, a+b
        return b
    }
}

func main() {
    f := fib()
    for x := f(); x < 100; x = f() {
        fmt.Println(x)                  // 1, 2, 3, 5, 8, 13, 21, 34, 55, 89
    }
}
```

Closures: scope vs lifetime

The inner variables continue to live on

```
func fib() func() int {
    a, b := 0, 1

    // return a closure over a & b
}

func main() {
    f := fib()

    // f keeps ahold of a and b and updates them

    fmt.Println(f(), f(), f(), f(), f(), f())
}
```

The inner function continues to mutate the variables it references

Closures: unique environment

The inner variables are unique to each closure

```
func fib() func() int {
    a, b := 0, 1
    // return a closure over a & b
}

func main() {
    f, g := fib(), fib()

    // f & g have their own copies of a & b
    fmt.Println(f(), f(), f(), f(), f(), f())
    fmt.Println(g(), g(), g(), g(), g(), g())
}
```

They print identical lines 1 2 3 ... (think a_1, b_1; a_2, b_2 etc.)

Closure gotcha

Avoid closing over a variable that is mutating (a loop index)

```
func do(d func()) {
    d()
}

func main() {
    for i := 0; i < 4; i++ {
        v := func() {
            fmt.Printf("%d %p\n", i, &i)
        }
        do(v)
    }
}
```

The program prints 0, 1, 2, 3; addresses all the same (**why?**)

Closure gotcha

Avoid closing over a variable that is mutating (a loop index)

```
func main() {
    s := make([]func(), 4)

    for i := 0; i < 4; i++ {
        s[i] = func() {
            // they all point to the same "i"
            fmt.Printf("%d %p\n", i, &i)
        }
    }
    for i := 0; i < 4; i++ {
        s[i]()
    }
}
```

The program prints 4 each time; addresses all the same (**why?**)

Closure gotcha

Avoid closing over a variable that is mutating (a loop index)

```
func main() {
    s := make([]func(), 4)

    for i := 0; i < 4; i++ {
        j := i // closure capture
        s[i] = func() {
            fmt.Printf("%d %p\n", j, &j)
        }
    }
    for i := 0; i < 4; i++ {
        s[i]()
    }
}
```

The program prints 0, 1, 2, 3 as expected; addresses different