Programming in Go

Matt Holiday
Christmas 2020

Functions

Functions in Go

Functions are “first class” objects; you can:

- Define them — even inside another function
- Create anonymous *function literals*
- Pass them as function parameters / return values
- Store them in variables
- Store them in slices and maps (but not as keys)
- Store them as fields of a structure type
- Send and receive them in channels
- Write methods against a function type
- Compare a function var against `nil`

Function scope

Almost anything can be defined inside a function

```
func Do() error {
    const a = 21

    type b struct {
        . . .
    }

    var c int

    reallyDoIt := func() {  // only anonymous funcs with assignment
        . . .
    }
}
```

Methods cannot be defined in a function (only at package scope)

Function signatures

The *signature* of a function is the order & type of its parameters and return values

It does not depend on the *names* of those parameters or returns

```
var try func(string, int) string

func Do(a string, b int) string {
    . . .
}

func NotDo(x string, y int) (a string) {}
    . . .
}
```

These functions have the same *structural* type

Parameter terms

A function declaration lists **formal** parameters

```
func do(a, b int) int { ... }
```

A function call has **actual** parameters (a/k/a “arguments”)

```
result := do(1, 2)
```

A parameter is passed **by value** if the function gets a copy;
the caller can’t see changes to the copy

A parameter is passed **by reference** if the function can modify
the actual parameter such that the caller sees the changes

Parameter passing

By value:

- numbers
- bool
- arrays
- structs

By reference:

- things passed by pointer (&x)
- strings (but they're immutable)
- slices
- maps
- channels

Parameter passing

Parameters may be passed *by value*

```
func do(b [3]int) int {
    b[0] = 0
    return b[1]
}

func main() {
    a := [3]int{1, 2, 3}
    v := do(a)

    fmt.Println(a, v)    // [1,2,3] 2
}
```

Here do gets a copy of the array so any change to it is not seen by the caller

Parameter passing

Parameters may be passed *by reference*

```
func do(b []int) int {
    b[0] = 0
    return b[1]
}

func main() {
    a := []int{1, 2, 3}
    v := do(a)

    fmt.Println(a, v)    // [0,2,3] 2
}
```

Here do gets a copy of the slice descriptor which *refers to* the same backing array, so the caller sees changes

Parameter passing

Parameters may be passed *by value* or *by reference*

```
func do(m1 map[int]int) {
    m1[3] = 1
    m1 = make(map[int]int)
    m1[4] = 4
    fmt.Println(m1)          // map[4:4]
}

func main() {
    m := map[int]int{4: 1}
    fmt.Println(m)           // map[4:1]
    do(m)
    fmt.Println(m)           // map[3:1 4:1]
}
```

We can re-assign `m1` because the formal parameter is a local variable

Parameter passing

Parameters may be passed *by value* or *by reference*

```
func do(m1 *map[int]int) {
    (*m1)[3] = 1
    *m1 = make(map[int]int)
    (*m1)[4] = 4
    fmt.Println(*m1)          // map[4:4]
}

func main() {
    m := map[int]int{4: 1}
    fmt.Println(m)            // map[4:1]
    do(&m)
    fmt.Println(m)            // map[4:4]
}
```

The map pointer `m` allows replacing the caller's entire map with a new one

Parameter passing: the ultimate truth

Parameters may be passed *by value* or ~~by reference~~

Actually, **all** parameters are passed by copying something
(i.e., by value)

If the thing copied is a pointer or descriptor, then the shared
backing store (array, hash table, etc.) can be changed through it

Thus we think of it as “by reference”

Return values

Functions can have multiple return values

```
func doIt(a int, b []int) int {
    . . .

    return 1
}

func doItAgain(a string) (int, error) {
    . . .

    return 1, nil
}
```

Every return statement must have all the values specified

Recursion

A function may call itself; the trick is knowing when to stop

```
func walk(node *tree.T) int {
    if node == nil {
        return 0
    }

    return node.value + walk(node.left) + walk(node.right)
}
```

This works because each function call adds context to the stack and unwinds it when done

If you don't have good stopping criteria, the program will crash

Defer

Deferred execution

How do we make sure something gets done?

- close a file we opened
- close a socket / HTTP request we made
- unlock a mutex we locked
- make sure something gets saved before we're done
- . . .

The `defer` statement captures a function *call* to run later

Defer

We need to ensure the file closes no matter what

```
func main() {
    f, err := os.Open("my_file.txt")

    if err != nil {
        . . .
    }

    defer f.Close()

    // and do something with the file
}
```

The call to `Close` is guaranteed to run at *function exit*
(don't defer closing the file until we know it really opened!)

Defer

We need to ensure the file closes no matter what

```
func main() {
    f := os.Stdin

    if len(os.Args) > 1 {
        if f, err := os.Open(os.Args[1]); err != nil {
            . . .
        }
        defer f.Close()
    }

    // and do something with the file
}
```

Notice that the `defer` will *not* execute when we leave the `if` block

Defer gotcha #1

The scope of a `defer` statement is the *function*

```
func main() {
    for i := 1; i < len(os.Args); i++ {
        f, err := os.Open(os.Args[i])

        . . .

        defer f.Close()

        . . .
    }
}
```

The deferred calls to `Close` must wait until function exit
(we might run out of file descriptors before that!)

Defer gotcha #2

Unlike a closure, `defer` copies arguments to the deferred call

```
func main() {
    a := 10

    defer fmt.Println(a)

    a = 11

    fmt.Println(a)
}

// prints 11, 10
```

The parameter `a` gets **copied** at the `defer` statement (not a reference)

Defer gotcha #2

A `defer` statement runs before the `return` is “done”

```
func doIt() (a int) {
    defer func() {
        a = 2
    }()

    a = 1
    return
}

// returns 2
```

We have a named return value and a “naked” return

The deferred anonymous function can update that variable