Why Golang is so Sick

Jason Miller

October 7, 2022

Contents

Multiple Return

In golang it is very common to have multiple return values for functions. This is really nice in two ways. The first is for error handling. In a lot of languages you might see something like this.

```
File myFile = os.Open("notes.txt")
if(myFile == NULL){
    return
}
```

This is kinda wacky because you now have weird cases where the space of all files overlaps with the space of all errors. This is VERY annoying if you return a primitive, like -1 being the error code for a file descriptor in C. What would your error code be for a integer overflow or a divide by zero. Here is how golang handles it.

```
myFile, err := os.Open("notes.txt")
if err != nil){
    log.Println(err)
    return
}
```

par Wow. Now we aren't causing collisions in the space of return values and don't have dumb things like error codes. Python has this but it is not super standard and java has the weird `catch(exception e)` system, which is a mess.

But wait. It gets even better than that. This combines a ton of functions into a handful of common functions. Do you want to find if a key is in a hashmap. It is super easy. Just lookup the value and throw it away. (Note, _ means to throw away whatever is assigned to it).

```
if _, err := map[key]; err != nil{
    The key is in the map
} else {
    The key is not in the map
}
```

Or here is another one. Have you ever wanted to do for each on a list, but couldn't because you needed the index for one part. Golang has you covered.

```
for index, value := range(list){
    ...
}
```

Anonymous Types and Functions

One thing that I am a big fan of in golang is that you don't need to assign a name to little things. Have a struct or function that you only use once, you don't need to clutter your code with them. For example, lets say you want to swap two ints but don't want a temp variable to be in scope forever. Check this out.

```
a := 4
b := 5
func(){
    temp := a
    a = b
    b = temp
}()
```

And this anonymous function, and any function for that matter, can be assigned to a variable, which adds a ton of interesting opportunities.

Now in programming there are basically 4 big datatypes. Lists, trees, maps, and sets. Now I am ashamed to say, I often don't bother trying to memorize the syntax for sets. On more than one occasion I have done the following in python.

```
mySet = {}
mySet["SetItem"] = 0
```

I will shamefully waste the 4 bytes that make up that zero to avoid learning new syntax. If only there were a way to put no data into the map and get the same result.

```
mySet := make(map[string]struct{})
mySet["SetItem"] = struct{}{}
```

And all of the tools to iterate through the keys of a map are EXACTLY the same as with sets.

Multi-threading

But here is the real meat and potatoes of golang. Where the go in golang comes from. Normally multithreading is a gigantic pain in the ass. They are super simple in golang, using the ”go” keyword.

```
go fmt.Println(”Hello from the other thread”)
```

The go keyword goes in front of any function call and it will create a new thread and call the function there. All are super lightweight and automatic. But wait, how can we do synchronization primitives? By using something called a channel. A channel (denoted by $<-$) is used for all critical regions. Writing to a channel is atomic, and there is a set volume to the channel, typically 1. If you try to write to a full channel, your thread will be paused until the channel is cleared. If you try to read from a empty channel then the thread will be paused until there is a item in the channel. Here is a example.

```
c := make(chan int)
go func(d chan int){
    time.Sleep(time.Second)
    d <- 5
}(c)
fmt.Println(<- c)
```

This correctly prints 5 because the main thread waits for the channel to have a int in it before it can read and print it. If you have multiple channels you are waiting on, you can even do a switch statement where the cases are which one to have an item in it first.

```
select{
    case msg1 := <- c1:
        fmt.Println(msg1)
    case msg2 := <- c2:
        fmt.Println(msg2)
}
```

Polymorphisim

Here is another cool thing about golang. You know how a lot of languages have very explicit ways to handle polymorphism, where a dog and a cat both have to inherit from the animal class. Golang does it automatically. What you do is describe some interfaces that are a collection of functions. Then at compile time, golang will look through all of the objects and see which of them implements all of the methods of that interface, and automatically handle the polymorphisim. This makes polymorphism between packages super easy. The best part is that you can throw any struct into a function, and the compiler will tell you what methods it is missing to be a member of that interface, so you don't even need to look at the documentation. I have written code that polymorphizes complex objects, like those used to handle http requests, and have gotten it to work perfectly. AND the interface doesn't even need to be named, it can be treated as a type. If you want to pass random data into a function, you can do so by setting the parameter type to interface{}. While this is a neat trick, it is useless because once in the function body, there are no methods to use the data with.

Misc

A few cool random things. There are no keywords for public and private. If a variable is private it will start with a lowercase. If it is public it starts with an uppercase. On top of this being a great way to see the scope of variables, it also enforces coding consistency.

One of the best compilers I have ever worked with. I think I have only gotten a runtime error once. All errors are compile time errors, which are exact to the line number and column number.

If you import unused packages or declare unused variables, your code will not compile. While this is a bit annoying to deal with, it makes things a lot cleaner.

There is a special keyword called defer. What this does is takes a line of code, and will execute it at the end of the function instead. This is super nice for some weird cases like files closing so you don't have to keep track of if you left a file open.

```
file, err := os.OpenFile("notes.txt")
defer file.Close()
The rest of the function
```

Built-in code formatter with the compiler.
Extremely good documentation for the standard libraries.
By far the easiest networking I have ever dealt with. Here is a basic http server in 9 lines of code.

```
package main
import ("fmt"; "net/http")
func main(){
    http.HandleFunc("/", Hello)
    http.ListenAndServe(":8080", nil)
}
func Hello(w http.ResponseWriter, r *http.Request){
    fmt.Fprintf(w, "Hello World")
}
```

Conclusion

Use Golang. 5/5