Tips on Maps & Errors

Useful tips and tricks for programming in Go.

WE'LL COVER THE FOLLOWING

Sets

You might want to find a way to extract unique value from a collection. In other languages, you often have a set data structure not allowing duplicates. Go doesn't have that built in, however it's not too hard to implement (due to a lack of generics, you do need to do that for most types, which can be cumbersome).

```
// UniqStr returns a copy if the passed slice with only unique string results.
func UniqStr(col []string) []string {
	m := map[string]struct{}{}
	for _, v := range col {
		if _, ok := m[v]; !ok {
			m[v] = struct{}{}
		}
	}
	list := make([]string, len(m))

	i := 0
	for v := range m {
		list[i] = v
		i++
	}
	return list
}
```

I used a few interesting tricks that are interesting to know. First, the map of empty structs:

```
m := map[string]struct{}{}
```

We create a map with the keys being the values we want to be unique, the associated value doesn't really matter much so it could be anything. For instance:

```
m := map[string]bool{}
```

However I chose an empty structure because it will be as fast as a boolean but doesn't allocate as much memory.

The second trick can been seen a bit further:

```
if _, ok := m[v]; !ok {
  m[v] = struct{}{}
}
```

What we are doing here, is simply check if there is a value associated with the key `v` in the map `m`, we don't care about the value itself, but if we know that we don't have a value, then we add one.

Once we have a map with unique keys, we can extract them into a new slice of strings and return the result.

Here is the test for this function, as you can see, I used a table test, which is the idiomatic Go way to run unit tests:

Environment Variables

Key:	Value:
GOPATH	/go

```
func TestUniqStr(t *testing.T) {

	data := []struct{ in, out []string }{
		{[]string{}, []string{}},
		{[]string{"", "", ""}, []string{""}},
		{[]string{"a", "a"}, []string{"a"}},
		{[]string{"a", "b", "a"}, []string{"a", "b"}},
		{[]string{"a", "b", "a", "b"}, []string{"a", "b"}},
		{[]string{"a", "b", "b", "a", "b"}, []string{"a", "b"}},
		{[]string{"a", "a", "b", "b", "a", "b"}, []string{"a", "b"}},
		{[]string{"a", "b", "c", "a", "b", "c"}, []string{"a", "b", "c"}},
	}
	for _, exp := range data {
		res := UniqStr(exp.in)
		if !reflect.DeepEqual(res, exp.out) {

			t.Fatalf("%q didn't match %q\n", res, exp.out) //produce an error if
		}
	}

}
```

Given below is the implementation of the `UniqueStr()` method and the test code that logs a runtime error if the output from the method does not match the test cases:

Environment Variables

Key:	Value:
GOPATH	/go

```
package main

import (
	"log"
	"reflect"
)

// UniqStr returns a copy if the passed slice with only unique string results.
func UniqStr(col []string) []string {
	m := map[string]struct{}{}
	for _, v := range col {
		_, ok := m[v]
		if !ok {
			m[v] = struct{}{}
		}
	}
	list := make([]string, len(m))

	i := 0
	for v := range m {
		list[i] = v
		i++
	}
	return list
}

func main() {
	data := []struct{ in, out []string }{
		{[]string{}, []string{}},
		{[]string{"", "", ""}, []string{""}},
		{[]string{"a", "a"}, []string{"a"}},
		{[]string{"a", "b", "a"}, []string{"a", "b"}},
		{[]string{"a", "b", "a", "b"}, []string{"a", "b"}},
		{[]string{"a", "b", "b", "a", "b"}, []string{"a", "b"}},
		{[]string{"a", "a", "b", "b", "a", "b"}, []string{"a", "b"}},
		{[]string{"a", "b", "c", "a", "b", "c"}, []string{"a", "b", "c"}},
	}
```

```
    for _, exp := range data {
        res := UniqStr(exp.in)
        if !reflect.DeepEqual(res, exp.out) {
            log.Fatalf("%q didn't match %q\n", res, exp.out) //log a runtime erro
        }
    }
}
```

Dependency package management

Unfortunately, Go doesn't ship with its own dependency package management system. Probably due to its roots in the C culture, packages aren't versioned and explicit version dependencies aren't addressed.

The challenge is that if you have multiple developers on your project, you want all of them to be on the same version of your dependencies. Your dependencies might also have their own dependencies and you want to make sure everything is in a good state. It gets even tricker when you have multiple projects using different versions of the same dependency. This is typically the case in a CI environment.

The Go community came up with a lot of different solutions for these problems. But for me, none are really great so at Splice we went for the simplest working solution we found: gpm

Gpm is a simple bash script, we end up modifying it a little so we could drop the script in each repo. The bash script uses a custom file called `Godeps` which lists the packages to install.

When switching to a different project, we run the project `gpm` script to pull down or set the right revision of each package.

In our CI environment, we set `GOPATH` to a project specific folder before running the test suite so packages aren't shared between projects.

Using errors

Errors are very important pattern in Go and at first, new developers are surprised by the amount of functions returning a value and an error.

Go doesn't have a concept of an exception like you might have seen in other

Go doesn't have a concept of an exception like you might have seen in other programming languages. Go does have something called `panic` but as its

name suggests they are really exceptional and shouldn't be rescued (that said, they can be).

The error handling in Go seems cumbersome and repetitive at first, but quickly becomes part of the way we think. Instead of creating exceptions that bubble up and might or might not be handled or passed higher, errors are part of the response and designed to be handled by the caller. Whenever a function might generate an error, its response should contain an error param.

Andrew Gerrand from the Go team wrote a great blog post on errors I strongly recommend you read it.

Effective Go section on errors