Reflection

You can find all the code for this chapter here

From Twitter

> golang challenge: write a function `walk(x interface{}, fn func(string))` which takes a struct `x` and calls `fn` for all strings fields found inside. difficulty level: recursively.

To do this we will need to use *reflection*.

> Reflection in computing is the ability of a program to examine its own structure, particularly through types; it's a form of metaprogramming. It's also a great source of confusion.

From The Go Blog: Reflection

What is `interface`?

We have enjoyed the type-safety that Go has offered us in terms of functions that work with known types, such as `string`, `int` and our own types like `BankAccount`.

This means that we get some documentation for free and the compiler will complain if you try and pass the wrong type to a function.

You may come across scenarios though where you want to write a function where you don't know the type at compile time.

Go lets us get around this with the type `interface{}` which you can think of as just *any* type.

So `walk(x interface{}, fn func(string))` will accept any value for `x`.

So why not use `interface` for everything and have really flexible functions?

- As a user of a function that takes `interface` you lose type safety. What if you meant to pass `Foo.bar` of type `string` into a function but instead did `Foo.baz` which is an `int`? The compiler won't be able to inform you of your mistake. You also have no idea *what* you're

allowed to pass to a function. Knowing that a function takes a `UserService` for instance is very useful.

- As a writer of such a function, you have to be able to inspect *anything* that has been passed to you and try and figure out what the type is and what you can do with it. This is done using *reflection*. This can be quite clumsy and difficult to read and is generally less performant (as you have to do checks at runtime).

In short only use reflection if you really need to.

If you want polymorphic functions, consider if you could design it around an interface (not `interface`, confusingly) so that users can use your function with multiple types if they implement whatever methods you need for your function to work.

Our function will need to be able to work with lots of different things. As always we'll take an iterative approach, writing tests for each new thing we want to support and refactoring along the way until we're done.

Write the test first

We'll want to call our function with a struct that has a string field in it (`x`). Then we can spy on the function (`fn`) passed in to see if it is called.

```
1  func TestWalk(t *testing.T) {
2
3          expected := "Chris"
4          var got []string
5
6          x := struct {
7                  Name string
8          }{expected}
9
10         walk(x, func(input string) {
11                 got = append(got, input)
12         })
13
14         if len(got) != 1 {
15                 t.Errorf("wrong number of function calls, got %d want %d", len(got),
```

```
        }
}
```

- We want to store a slice of strings (`got`) which stores which strings were passed into `fn` by `walk`. Often in previous chapters, we have made dedicated types for this to spy on function/method invocations but in this case, we can just pass in an anonymous function for `fn` that closes over `got`.
- We use an anonymous `struct` with a `Name` field of type string to go for the simplest "happy" path.
- Finally, call `walk` with `x` and the spy and for now just check the length of `got`, we'll be more specific with our assertions once we've got something very basic working.

Try to run the test

```
./reflection_test.go:21:2: undefined: walk
```

Write the minimal amount of code for the test to run and check the failing test output

We need to define `walk`

```
func walk(x interface{}, fn func(input string)) {

}
```

Try and run the test again

```
=== RUN   TestWalk
--- FAIL: TestWalk (0.00s)
    reflection_test.go:19: wrong number of function calls, got 0 want 1
```

```
4 FAIL
```

Write enough code to make it pass

We can call the spy with any string to make this pass.

```
1 func walk(x interface{}, fn func(input string)) {
2         fn("I still can't believe South Korea beat Germany 2-0 to put them last in th
3 }
```

The test should now be passing. The next thing we'll need to do is make a more specific assertion on what our `fn` is being called with.

Write the test first

Add the following to the existing test to check the string passed to `fn` is correct

```
1 if got[0] != expected {
2         t.Errorf("got %q, want %q", got[0], expected)
3 }
```

Try to run the test

```
1 === RUN   TestWalk
2 --- FAIL: TestWalk (0.00s)
3     reflection_test.go:23: got 'I still can't believe South Korea beat Germany 2-0 to
4 FAIL
```

Write enough code to make it pass

```
1 func walk(x interface{}, fn func(input string)) {
2         val := reflect.ValueOf(x)
3         field := val.Field(0)
4         fn(field.String())
5 }
```

This code is *very unsafe and very naive*, but remember: our goal when we are in "red" (the tests failing) is to write the smallest amount of code possible. We then write more tests to address our concerns.

We need to use reflection to have a look at `x` and try and look at its properties.

The reflect package has a function `ValueOf` which returns us a `Value` of a given variable. This has ways for us to inspect a value, including its fields which we use on the next line.

We then make some very optimistic assumptions about the value passed in

- We look at the first and only field, there may be no fields at all which would cause a panic
- We then call `String()` which returns the underlying value as a string but we know it would be wrong if the field was something other than a string.

Refactor

Our code is passing for the simple case but we know our code has a lot of shortcomings.

We're going to be writing a number of tests where we pass in different values and checking the array of strings that `fn` was called with.

We should refactor our test into a table based test to make this easier to continue testing new scenarios.

```
1 func TestWalk(t *testing.T) {
2
3         cases := []struct {
```

```
                Name          string
                Input         interface{}
                ExpectedCalls []string
        }{
                {
                        "Struct with one string field",
                        struct {
                                Name string
                        }{"Chris"},
                        []string{"Chris"},
                },
        }

        for _, test := range cases {
                t.Run(test.Name, func(t *testing.T) {
                        var got []string
                        walk(test.Input, func(input string) {
                                got = append(got, input)
                        })

                        if !reflect.DeepEqual(got, test.ExpectedCalls) {
                                t.Errorf("got %v, want %v", got, test.ExpectedCalls)
                        }
                })
        }
}
```

Now we can easily add a scenario to see what happens if we have more than one string field.

Write the test first

Add the following scenario to the `cases`.

```
{
    "Struct with two string fields",
    struct {
        Name string
        City string
    }{"Chris", "London"},
    []string{"Chris", "London"},
```

```
}
```

Try to run the test

```
=== RUN   TestWalk/Struct_with_two_string_fields
    --- FAIL: TestWalk/Struct_with_two_string_fields (0.00s)
        reflection_test.go:40: got [Chris], want [Chris London]
```

Write enough code to make it pass

```
func walk(x interface{}, fn func(input string)) {
        val := reflect.ValueOf(x)

        for i := 0; i < val.NumField(); i++ {
                field := val.Field(i)
                fn(field.String())
        }
}
```

`val` has a method `NumField` which returns the number of fields in the value. This lets us iterate over the fields and call `fn` which passes our test.

Refactor

It doesn't look like there's any obvious refactors here that would improve the code so let's press on.

The next shortcoming in `walk` is that it assumes every field is a `string`. Let's write a test for this scenario.

Write the test first

Add the following case

```
{
    "Struct with non string field",
    struct {
        Name string
        Age  int
    }{"Chris", 33},
    []string{"Chris"},
},
```

Try to run the test

```
=== RUN   TestWalk/Struct_with_non_string_field
    --- FAIL: TestWalk/Struct_with_non_string_field (0.00s)
        reflection_test.go:46: got [Chris <int Value>], want [Chris]
```

Write enough code to make it pass

We need to check that the type of the field is a `string`.

```
func walk(x interface{}, fn func(input string)) {
        val := reflect.ValueOf(x)

        for i := 0; i < val.NumField(); i++ {
                field := val.Field(i)

                if field.Kind() == reflect.String {
                        fn(field.String())
                }
```

```
        }
}
```

We can do that by checking its `Kind`.

Refactor

Again it looks like the code is reasonable enough for now.

The next scenario is what if it isn't a "flat" `struct`? In other words, what happens if we have a `struct` with some nested fields?

Write the test first

We have been using the anonymous struct syntax to declare types ad-hocly for our tests so we could continue to do that like so

```
{
    "Nested fields",
    struct {
        Name string
        Profile struct {
            Age  int
            City string
        }
    }{"Chris", struct {
        Age  int
        City string
    }{33, "London"}},
    []string{"Chris", "London"},
},
```

But we can see that when you get inner anonymous structs the syntax gets a little messy. There is a proposal to make it so the syntax would be nicer.

Let's just refactor this by making a known type for this scenario and reference it in the test. There is a little indirection in that some of the code for our test is outside the test but readers should be able to infer the structure of the `struct` by looking at the initialisation.
Add the following type declarations somewhere in your test file

```
type Person struct {
        Name    string
        Profile Profile
}

type Profile struct {
        Age  int
        City string
}
```

Now we can add this to our cases which reads a lot clearer than before

```
{
    "Nested fields",
    Person{
        "Chris",
        Profile{33, "London"},
    },
    []string{"Chris", "London"},
},
```

Try to run the test

```
=== RUN   TestWalk/Nested_fields
    --- FAIL: TestWalk/Nested_fields (0.00s)
        reflection_test.go:54: got [Chris], want [Chris London]
```

The problem is we're only iterating on the fields on the first level of the type's hierarchy.

Write enough code to make it pass

```
func walk(x interface{}, fn func(input string)) {
        val := reflect.ValueOf(x)

        for i := 0; i < val.NumField(); i++ {
                field := val.Field(i)

                if field.Kind() == reflect.String {
                        fn(field.String())
                }

                if field.Kind() == reflect.Struct {
                        walk(field.Interface(), fn)
                }
        }
}
```

The solution is quite simple, we again inspect its `Kind` and if it happens to be a `struct` we just call `walk` again on that inner `struct`.

Refactor

```
func walk(x interface{}, fn func(input string)) {
        val := reflect.ValueOf(x)

        for i := 0; i < val.NumField(); i++ {
                field := val.Field(i)

                switch field.Kind() {
                case reflect.String:
                        fn(field.String())
                case reflect.Struct:
                        walk(field.Interface(), fn)
                }
        }
```

```
14 }
```

When you're doing a comparison on the same value more than once *generally* refactoring into a `switch` will improve readability and make your code easier to extend.

What if the value of the struct passed in is a pointer?

Write the test first

Add this case

```
{
    "Pointers to things",
    &Person{
        "Chris",
        Profile{33, "London"},
    },
    []string{"Chris", "London"},
},
```

Try to run the test

```
=== RUN   TestWalk/Pointers_to_things
panic: reflect: call of reflect.Value.NumField on ptr Value [recovered]
    panic: reflect: call of reflect.Value.NumField on ptr Value
```

Write enough code to make it pass

```
func walk(x interface{}, fn func(input string)) {
        val := reflect.ValueOf(x)
```

```
        if val.Kind() == reflect.Ptr {
                val = val.Elem()
        }

        for i := 0; i < val.NumField(); i++ {
                field := val.Field(i)

                switch field.Kind() {
                case reflect.String:
                        fn(field.String())
                case reflect.Struct:
                        walk(field.Interface(), fn)
                }
        }
}
```

You can't use `NumField` on a pointer `Value`, we need to extract the underlying value before we can do that by using `Elem()`.

Refactor

Let's encapsulate the responsibility of extracting the `reflect.Value` from a given `interface{}` into a function.

```
func walk(x interface{}, fn func(input string)) {
        val := getValue(x)

        for i := 0; i < val.NumField(); i++ {
                field := val.Field(i)

                switch field.Kind() {
                case reflect.String:
                        fn(field.String())
                case reflect.Struct:
                        walk(field.Interface(), fn)
                }
        }
}

```

```
func getValue(x interface{}) reflect.Value {
        val := reflect.ValueOf(x)

        if val.Kind() == reflect.Ptr {
                val = val.Elem()
        }

        return val
}
```

This actually adds *more* code but I feel the abstraction level is right.

- Get the `reflect.Value` of `x` so I can inspect it, I don't care how.
- Iterate over the fields, doing whatever needs to be done depending on its type.

Next, we need to cover slices.

Write the test first

```
{
    "Slices",
    []Profile {
        {33, "London"},
        {34, "Reykjavík"},
    },
    []string{"London", "Reykjavík"},
},
```

Try to run the test

```
=== RUN   TestWalk/Slices
panic: reflect: call of reflect.Value.NumField on slice Value [recovered]
    panic: reflect: call of reflect.Value.NumField on slice Value
```

Write the minimal amount of code for the test to run and check the failing test output

This is similar to the pointer scenario before, we are trying to call `NumField` on our `reflect.Value` but it doesn't have one as it's not a struct.

Write enough code to make it pass

```
func walk(x interface{}, fn func(input string)) {
	val := getValue(x)

	if val.Kind() == reflect.Slice {
		for i := 0; i < val.Len(); i++ {
			walk(val.Index(i).Interface(), fn)
		}
		return
	}

	for i := 0; i < val.NumField(); i++ {
		field := val.Field(i)

		switch field.Kind() {
		case reflect.String:
			fn(field.String())
		case reflect.Struct:
			walk(field.Interface(), fn)
		}
	}
}
```

Refactor

This works but it's yucky. No worries, we have working code backed by tests so we are free to tinker all we like.

If you think a little abstractly, we want to call `walk` on either

- Each field in a struct
- Each *thing* in a slice

Our code at the moment does this but doesn't reflect it very well. We just have a check at the start to see if it's a slice (with a `return` to stop the rest of the code executing) and if it's not we just assume it's a struct.

Let's rework the code so instead we check the type *first* and then do our work.

```
func walk(x interface{}, fn func(input string)) {
        val := getValue(x)

        switch val.Kind() {
        case reflect.Struct:
                for i := 0; i < val.NumField(); i++ {
                        walk(val.Field(i).Interface(), fn)
                }
        case reflect.Slice:
                for i := 0; i < val.Len(); i++ {
                        walk(val.Index(i).Interface(), fn)
                }
        case reflect.String:
                fn(val.String())
        }
}
```

Looking much better! If it's a struct or a slice we iterate over its values calling `walk` on each one. Otherwise, if it's a `reflect.String` we can call `fn`.

Still, to me it feels like it could be better. There's repetition of the operation of iterating over fields/values and then calling `walk` but conceptually they're the same.

```
func walk(x interface{}, fn func(input string)) {
        val := getValue(x)

```

```
        numberOfValues := 0
        var getField func(int) reflect.Value

        switch val.Kind() {
        case reflect.String:
                fn(val.String())
        case reflect.Struct:
                numberOfValues = val.NumField()
                getField = val.Field
        case reflect.Slice:
                numberOfValues = val.Len()
                getField = val.Index
        }

        for i := 0; i < numberOfValues; i++ {
                walk(getField(i).Interface(), fn)
        }
}
```

If the `value` is a `reflect.String` then we just call `fn` like normal.

Otherwise, our `switch` will extract out two things depending on the type

- How many fields there are
- How to extract the `Value` (`Field` or `Index`)

Once we've determined those things we can iterate through `numberOfValues` calling `walk` with the result of the `getField` function.

Now we've done this, handling arrays should be trivial.

Write the test first

Add to the cases

```
{
    "Arrays",
    [2]Profile {
        {33, "London"},
        {34, "Reykjavík"},
```

```
    },
    []string{"London", "Reykjavík"},
},
```

Try to run the test

```
=== RUN   TestWalk/Arrays
    --- FAIL: TestWalk/Arrays (0.00s)
        reflection_test.go:78: got [], want [London Reykjavík]
```

Write enough code to make it pass

Arrays can be handled the same way as slices, so just add it to the case with a comma

```
func walk(x interface{}, fn func(input string)) {
        val := getValue(x)

        numberOfValues := 0
        var getField func(int) reflect.Value

        switch val.Kind() {
        case reflect.String:
                fn(val.String())
        case reflect.Struct:
                numberOfValues = val.NumField()
                getField = val.Field
        case reflect.Slice, reflect.Array:
                numberOfValues = val.Len()
                getField = val.Index
        }

        for i := 0; i < numberOfValues; i++ {
                walk(getField(i).Interface(), fn)
        }
}
```

The next type we want to handle is `map`.

Write the test first

```
{
    "Maps",
    map[string]string{
        "Foo": "Bar",
        "Baz": "Boz",
    },
    []string{"Bar", "Boz"},
},
```

Try to run the test

```
=== RUN   TestWalk/Maps
    --- FAIL: TestWalk/Maps (0.00s)
        reflection_test.go:86: got [], want [Bar Boz]
```

Write enough code to make it pass

Again if you think a little abstractly you can see that `map` is very similar to `struct`, it's just the keys are unknown at compile time.

```
func walk(x interface{}, fn func(input string)) {
        val := getValue(x)

        numberOfValues := 0
        var getField func(int) reflect.Value
```

```
6
7         switch val.Kind() {
8         case reflect.String:
9                 fn(val.String())
10        case reflect.Struct:
11                numberOfValues = val.NumField()
12                getField = val.Field
13        case reflect.Slice, reflect.Array:
14                numberOfValues = val.Len()
15                getField = val.Index
16        case reflect.Map:
17                for _, key := range val.MapKeys() {
18                        walk(val.MapIndex(key).Interface(), fn)
19                }
20        }
21
22        for i := 0; i < numberOfValues; i++ {
23                walk(getField(i).Interface(), fn)
24        }
25 }
```

However, by design you cannot get values out of a map by index. It's only done by *key*, so that breaks our abstraction, darn.

Refactor

How do you feel right now? It felt like maybe a nice abstraction at the time but now the code feels a little wonky.

This is OK! Refactoring is a journey and sometimes we will make mistakes. A major point of TDD is it gives us the freedom to try these things out.

By taking small steps backed by tests this is in no way an irreversible situation. Let's just put it back to how it was before the refactor.

```
1 func walk(x interface{}, fn func(input string)) {
2         val := getValue(x)
3
4         walkValue := func(value reflect.Value) {
5                 walk(value.Interface(), fn)
```

```
        }

        switch val.Kind() {
        case reflect.String:
                fn(val.String())
        case reflect.Struct:
                for i := 0; i < val.NumField(); i++ {
                        walkValue(val.Field(i))
                }
        case reflect.Slice, reflect.Array:
                for i := 0; i < val.Len(); i++ {
                        walkValue(val.Index(i))
                }
        case reflect.Map:
                for _, key := range val.MapKeys() {
                        walkValue(val.MapIndex(key))
                }
        }
}
```

We've introduced `walkValue` which DRYs up the calls to `walk` inside our `switch` so that they only have to extract out the `reflect.Value`s from `val`.

One final problem

Remember that maps in Go do not guarantee order. So your tests will sometimes fail because we assert that the calls to `fn` are done in a particular order.

To fix this, we'll need to move our assertion with the maps to a new test where we do not care about the order.

```
t.Run("with maps", func(t *testing.T) {
        aMap := map[string]string{
                "Foo": "Bar",
                "Baz": "Boz",
        }

        var got []string
        walk(aMap, func(input string) {
                got = append(got, input)
        })

```

```
        assertContains(t, got, "Bar")
        assertContains(t, got, "Boz")
})
```

Here is how `assertContains` is defined

```
func assertContains(t testing.TB, haystack []string, needle string) {
        t.Helper()
        contains := false
        for _, x := range haystack {
                if x == needle {
                        contains = true
                }
        }
        if !contains {
                t.Errorf("expected %+v to contain %q but it didn't", haystack, needle
        }
}
```

The next type we want to handle is `chan`.

Write the test first

```
t.Run("with channels", func(t *testing.T) {
        aChannel := make(chan Profile)

        go func() {
                aChannel <- Profile{33, "Berlin"}
                aChannel <- Profile{34, "Katowice"}
                close(aChannel)
        }()

        var got []string
        want := []string{"Berlin", "Katowice"}

        walk(aChannel, func(input string) {
                got = append(got, input)
        })
```

```
16
17          if !reflect.DeepEqual(got, want) {
18                  t.Errorf("got %v, want %v", got, want)
19          }
20  })
```

Try to run the test

```
1  --- FAIL: TestWalk (0.00s)
2      --- FAIL: TestWalk/with_channels (0.00s)
3          reflection_test.go:115: got [], want [Berlin Katowice]
```

Write enough code to make it pass

We can iterate through all values sent through channel until it was closed with Recv()

```
1  func walk(x interface{}, fn func(input string)) {
2          val := getValue(x)
3
4          walkValue := func(value reflect.Value) {
5                  walk(value.Interface(), fn)
6          }
7
8          switch val.Kind() {
9          case reflect.String:
10                 fn(val.String())
11         case reflect.Struct:
12                 for i := 0; i < val.NumField(); i++ {
13                         walkValue(val.Field(i))
14                 }
15         case reflect.Slice, reflect.Array:
16                 for i := 0; i < val.Len(); i++ {
17                         walkValue(val.Index(i))
18                 }
19         case reflect.Map:
```

```
                for _, key := range val.MapKeys() {
                        walkValue(val.MapIndex(key))
                }
        case reflect.Chan:
                for v, ok := val.Recv(); ok; v, ok = val.Recv() {
                        walkValue(v)
                }
        }
}
```

The next type we want to handle is `func`.

Write the test first

```
t.Run("with function", func(t *testing.T) {
        aFunction := func() (Profile, Profile) {
                return Profile{33, "Berlin"}, Profile{34, "Katowice"}
        }

        var got []string
        want := []string{"Berlin", "Katowice"}

        walk(aFunction, func(input string) {
                got = append(got, input)
        })

        if !reflect.DeepEqual(got, want) {
                t.Errorf("got %v, want %v", got, want)
        }
})
```

Try to run the test

```
--- FAIL: TestWalk (0.00s)
```

```
    --- FAIL: TestWalk/with_function (0.00s)
        reflection_test.go:132: got [] want [Berlin Katowice]
```

Write enough code to make it pass

Non zero-argument functions do not seem to make a lot of sense in this scenario. But we should allow for arbitrary return values.

```go
func walk(x interface{}, fn func(input string)) {
	val := getValue(x)

	walkValue := func(value reflect.Value) {
		walk(value.Interface(), fn)
	}

	switch val.Kind() {
	case reflect.String:
		fn(val.String())
	case reflect.Struct:
		for i := 0; i < val.NumField(); i++ {
			walkValue(val.Field(i))
		}
	case reflect.Slice, reflect.Array:
		for i := 0; i < val.Len(); i++ {
			walkValue(val.Index(i))
		}
	case reflect.Map:
		for _, key := range val.MapKeys() {
			walkValue(val.MapIndex(key))
		}
	case reflect.Chan:
		for v, ok := val.Recv(); ok; v, ok = val.Recv() {
			walkValue(v)
		}
	case reflect.Func:
		valFnResult := val.Call(nil)
		for _, res := range valFnResult {
			walkValue(res)
		}
	}
```

```
33 }
```

Wrapping up

- Introduced some concepts from the `reflect` package.
- Used recursion to traverse arbitrary data structures.
- Did an in retrospect bad refactor but didn't get too upset about it. By working iteratively with tests it's not such a big deal.
- This only covered a small aspect of reflection. The Go blog has an excellent post covering more details.
- Now that you know about reflection, do your best to avoid using it.