The Go Blog

The Laws of Reflection

Rob Pike
6 September 2011

Introduction

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.

In this article we attempt to clarify things by explaining how reflection works in Go. Each language's reflection model is different (and many languages don't support it at all), but this article is about Go, so for the rest of this article the word "reflection" should be taken to mean "reflection in Go".

Types and interfaces

Because reflection builds on the type system, let's start with a refresher about types in Go.

Go is statically typed. Every variable has a static type, that is, exactly one type known and fixed at compile time: `int`, `float32`, `*MyType`, `[]byte`, and so on. If we declare

```
type MyInt int

var i int
var j MyInt
```

then `i` has type `int` and `j` has type `MyInt`. The variables `i` and `j` have distinct static types and, although they have the same underlying type, they cannot be assigned to one another without a conversion.

One important category of type is interface types, which represent fixed sets of methods. An interface variable can store any concrete (non-interface) value as long as that value implements the interface's methods. A well-known pair of examples is `io.Reader` and `io.Writer`, the types `Reader` and `Writer` from the io package:

```
// Reader is the interface that wraps the basic Read method.
type Reader interface {
    Read(p []byte) (n int, err error)
}

// Writer is the interface that wraps the basic Write method.
type Writer interface {
    Write(p []byte) (n int, err error)
}
```

Any type that implements a `Read` (or `Write`) method with this signature is said to implement `io.Reader` (or `io.Writer`). For the purposes of this discussion, that means that a variable of type `io.Reader` can hold any value whose type has a `Read` method:

```
var r io.Reader
r = os.Stdin
r = bufio.NewReader(r)
r = new(bytes.Buffer)
// and so on
```

It's important to be clear that whatever concrete value `r` may hold, `r`'s type is always `io.Reader`: Go is statically typed and the static type of `r` is `io.Reader`.

An extremely important example of an interface type is the empty interface:

```
interface{}
```

It represents the empty set of methods and is satisfied by any value at all, since any value has zero or more methods.

Some people say that Go's interfaces are dynamically typed, but that is misleading. They are statically typed: a variable of interface type always has the same static type, and even though at run time the value stored in the interface variable may change type, that value will always satisfy the interface.

We need to be precise about all this because reflection and interfaces are closely related.

The representation of an interface

Russ Cox has written a detailed blog post about the representation of interface values in Go. It's not necessary to repeat the full story here, but a simplified summary is in order.

A variable of interface type stores a pair: the concrete value assigned to the variable, and that value's type descriptor. To be more precise, the value is the underlying concrete data item that implements the interface and the type describes the full type of that item. For instance, after

```
var r io.Reader
tty, err := os.OpenFile("/dev/tty", os.O_RDWR, 0)
if err != nil {
    return nil, err
}
r = tty
```

`r` contains, schematically, the (value, type) pair, (`tty`, `*os.File`). Notice that the type `*os.File` implements methods other than `Read`; even though the interface value provides access only to the `Read` method, the value inside carries all the type information about that value. That's why we can do things like this:

```
var w io.Writer
w = r.(io.Writer)
```

The expression in this assignment is a type assertion; what it asserts is that the item inside `r` also implements `io.Writer`, and so we can assign it to `w`. After the assignment, `w` will contain the pair (`tty`, `*os.File`). That's the same pair as was held in `r`. The static type of the interface determines what methods may be invoked with an interface variable, even though the concrete value inside may have a larger set of methods.

Continuing, we can do this:

```
var empty interface{}
empty = w
```

and our empty interface value `empty` will again contain that same pair, (`tty`, `*os.File`). That's handy: an empty interface can hold any value and contains all the information we could ever need about that value.

(We don't need a type assertion here because it's known statically that `w` satisfies the empty interface. In the example where we moved a value from a `Reader` to a `Writer`, we needed to be explicit and use a type assertion because `Writer`'s methods are not a subset of `Reader`'s.)

One important detail is that the pair inside an interface always has the form (value, concrete type) and cannot have the form (value, interface type). Interfaces do not hold interface values.

Now we're ready to reflect.

The first law of reflection

1. Reflection goes from interface value to reflection object.

At the basic level, reflection is just a mechanism to examine the type and value pair stored inside an interface variable. To get started, there are two types we need to know about in package reflect: Type and Value. Those two types give access to the contents of an interface variable, and two simple functions, called `reflect.TypeOf` and `reflect.ValueOf`, retrieve `reflect.Type` and `reflect.Value` pieces out of an interface value. (Also, from the `reflect.Value` it's easy to get to the `reflect.Type`, but let's keep the `Value` and `Type` concepts separate for now.)

Let's start with `TypeOf`:

```
package main

import (
    "fmt"
    "reflect"
)

func main() {
    var x float64 = 3.4
    fmt.Println("type:", reflect.TypeOf(x))
}
```

This program prints

```
type: float64
```

You might be wondering where the interface is here, since the program looks like it's passing the `float64` variable `x`, not an interface value, to `reflect.TypeOf`. But it's there; as godoc reports, the signature of `reflect.TypeOf` includes an empty interface:

```
// TypeOf returns the reflection Type of the value in the interface{}.
func TypeOf(i interface{}) Type
```

When we call `reflect.TypeOf(x)`, `x` is first stored in an empty interface, which is then passed as the argument; `reflect.TypeOf` unpacks that empty interface to recover the type information.

The `reflect.ValueOf` function, of course, recovers the value (from here on we'll elide the boilerplate and focus just on the executable code):

```
var x float64 = 3.4
fmt.Println("value:", reflect.ValueOf(x).String())
```

prints

```
value: <float64 Value>
```

(We call the `String` method explicitly because by default the `fmt` package digs into a `reflect.Value` to show the concrete value inside. The `String` method does not.)

Both `reflect.Type` and `reflect.Value` have lots of methods to let us examine and manipulate them. One important example is that `Value` has a `Type` method that returns the `Type` of a `reflect.Value`. Another is that both `Type` and `Value` have a `Kind` method that returns a constant indicating what sort of item is stored: `Uint`, `Float64`, `Slice`, and so on. Also methods on `Value` with names like `Int` and `Float` let us grab values (as `int64` and `float64`) stored inside:

```
var x float64 = 3.4
v := reflect.ValueOf(x)
fmt.Println("type:", v.Type())
fmt.Println("kind is float64:", v.Kind() == reflect.Float64)
fmt.Println("value:", v.Float())
```

prints

```
type: float64
kind is float64: true
value: 3.4
```

There are also methods like `SetInt` and `SetFloat` but to use them we need to understand settability, the subject of the third law of reflection, discussed below.

The reflection library has a couple of properties worth singling out. First, to keep the API simple, the "getter" and "setter" methods of `Value` operate on the largest type that can hold the value: `int64` for all the signed integers, for instance. That is, the `Int` method of `Value` returns an `int64` and the `SetInt` value takes an `int64`; it may be necessary to convert to the actual type involved:

```
var x uint8 = 'x'
v := reflect.ValueOf(x)
fmt.Println("type:", v.Type())                            // uint8.
fmt.Println("kind is uint8: ", v.Kind() == reflect.Uint8) // true.
x = uint8(v.Uint())                                       // v.Uint returns a uint64.
```

The second property is that the `Kind` of a reflection object describes the underlying type, not the static type. If a reflection object contains a value of a user-defined integer type, as in

```
type MyInt int
var x MyInt = 7
v := reflect.ValueOf(x)
```

```
v := reflect.ValueOf(x)
```

the `Kind` of `v` is still `reflect.Int`, even though the static type of `x` is `MyInt`, not `int`. In other words, the `Kind` cannot discriminate an int from a `MyInt` even though the `Type` can.

The second law of reflection

2. Reflection goes from reflection object to interface value.

Like physical reflection, reflection in Go generates its own inverse.

Given a `reflect.Value` we can recover an interface value using the `Interface` method; in effect the method packs the type and value information back into an interface representation and returns the result:

```
// Interface returns v's value as an interface{}.
func (v Value) Interface() interface{}
```

As a consequence we can say

```
y := v.Interface().(float64) // y will have type float64.
fmt.Println(y)
```

to print the `float64` value represented by the reflection object `v`.

We can do even better, though. The arguments to `fmt.Println`, `fmt.Printf` and so on are all passed as empty interface values, which are then unpacked by the `fmt` package internally just as we have been doing in the previous examples. Therefore all it takes to print the contents of a `reflect.Value` correctly is to pass the result of the `Interface` method to the formatted print routine:

```
fmt.Println(v.Interface())
```

(Why not `fmt.Println(v)`? Because `v` is a `reflect.Value`; we want the concrete value it holds.) Since our value is a `float64`, we can even use a floating-point format if we want:

```
fmt.Printf("value is %7.1e\n", v.Interface())
```

and get in this case

```
3.4e+00
```

Again, there's no need to type-assert the result of `v.Interface()` to `float64`; the empty interface value has the concrete value's type information inside and `Printf` will recover it.

In short, the `Interface` method is the inverse of the `ValueOf` function, except that its result is always of static type `interface{}`.

Reiterating: Reflection goes from interface values to reflection objects and back again.

The third law of reflection

3. To modify a reflection object, the value must be settable.

The third law is the most subtle and confusing, but it's easy enough to understand if we start from first principles.

Here is some code that does not work, but is worth studying

Here is some code that does not work, but is worth studying.

```
var x float64 = 3.4
v := reflect.ValueOf(x)
v.SetFloat(7.1) // Error: will panic.
```

If you run this code, it will panic with the cryptic message

```
panic: reflect.Value.SetFloat using unaddressable value
```

The problem is not that the value `7.1` is not addressable; it's that `v` is not settable. Settability is a property of a reflection `Value`, and not all reflection `Values` have it.

The `CanSet` method of `Value` reports the settability of a `Value`; in our case,

```
var x float64 = 3.4
v := reflect.ValueOf(x)
fmt.Println("settability of v:", v.CanSet())
```

prints

```
settability of v: false
```

It is an error to call a `Set` method on an non-settable `Value`. But what is settability?

Settability is a bit like addressability, but stricter. It's the property that a reflection object can modify the actual storage that was used to create the reflection object. Settability is determined by whether the reflection object holds the original item. When we say

```
var x float64 = 3.4
v := reflect.ValueOf(x)
```

we pass a copy of `x` to `reflect.ValueOf`, so the interface value created as the argument to `reflect.ValueOf` is a copy of `x`, not `x` itself. Thus, if the statement

```
v.SetFloat(7.1)
```

were allowed to succeed, it would not update `x`, even though `v` looks like it was created from `x`. Instead, it would update the copy of `x` stored inside the reflection value and `x` itself would be unaffected. That would be confusing and useless, so it is illegal, and settability is the property used to avoid this issue.

If this seems bizarre, it's not. It's actually a familiar situation in unusual garb. Think of passing `x` to a function:

```
f(x)
```

We would not expect `f` to be able to modify `x` because we passed a copy of `x`'s value, not `x` itself. If we want `f` to modify `x` directly we must pass our function the address of `x` (that is, a pointer to `x`):

```
f(&x)
```

This is straightforward and familiar, and reflection works the same way. If we want to modify `x` by reflection, we must give the reflection library a pointer to the value we want to modify.

Let's do that. First we initialize `x` as usual and then create a reflection value that points to it, called `p`.

```
var x float64 = 3.4
p := reflect.ValueOf(&x) // Note: take the address of x.
fmt.Println("type of p:", p.Type())
fmt.Println("settability of p:", p.CanSet())
```

The output so far is

```
type of p: *float64
settability of p: false
```

The reflection object `p` isn't settable, but it's not `p` we want to set, it's (in effect) `*p`. To get to what `p` points to, we call the `Elem` method of `Value`, which indirects through the pointer, and save the result in a reflection `Value` called `v`:

```
v := p.Elem()
fmt.Println("settability of v:", v.CanSet())
```

Now `v` is a settable reflection object, as the output demonstrates,

```
settability of v: true
```

and since it represents `x`, we are finally able to use `v.SetFloat` to modify the value of `x`:

```
v.SetFloat(7.1)
fmt.Println(v.Interface())
fmt.Println(x)
```

The output, as expected, is

```
7.1
7.1
```

Reflection can be hard to understand but it's doing exactly what the language does, albeit through reflection `Types` and `Values` that can disguise what's going on. Just keep in mind that reflection Values need the address of something in order to modify what they represent.

Structs

In our previous example `v` wasn't a pointer itself, it was just derived from one. A common way for this situation to arise is when using reflection to modify the fields of a structure. As long as we have the address of the structure, we can modify its fields.

Here's a simple example that analyzes a struct value, `t`. We create the reflection object with the address of the struct because we'll want to modify it later. Then we set `typeOfT` to its type and iterate over the fields using straightforward method calls (see package reflect for details). Note that we extract the names of the fields from the struct type, but the fields themselves are regular `reflect.Value` objects.

```
type T struct {
    A int
    B string
}
t := T{23, "skidoo"}
s := reflect.ValueOf(&t).Elem()
typeOfT := s.Type()
for i := 0; i < s.NumField(); i++ {
    f := s.Field(i)
    fmt.Printf("%d: %s %s = %v\n", i,
        typeOfT.Field(i).Name, f.Type(), f.Interface())
}
```

The output of this program is

```
0: A int = 23
1: B string = skidoo
```

There's one more point about settability introduced in passing here: the field names of `T` are upper case (exported) because only exported fields of a struct are settable.

Because `s` contains a settable reflection object, we can modify the fields of the structure.

```
s.Field(0).SetInt(77)
s.Field(1).SetString("Sunset Strip")
fmt.Println("t is now", t)
```

And here's the result:

```
t is now {77 Sunset Strip}
```

If we modified the program so that `s` was created from `t`, not `&t`, the calls to `SetInt` and `SetString` would fail as the fields of `t` would not be settable.

Conclusion

Here again are the laws of reflection:

- Reflection goes from interface value to reflection object.
- Reflection goes from reflection object to interface value.
- To modify a reflection object, the value must be settable.

Once you understand these laws reflection in Go becomes much easier to use, although it remains subtle. It's a powerful tool that should be used with care and avoided unless strictly necessary.

There's plenty more to reflection that we haven't covered — sending and receiving on channels, allocating memory, using slices and maps, calling methods and functions — but this post is long enough. We'll cover some of those topics in a later article.

Related articles

- A new Go API for Protocol Buffers
- Working with Errors in Go 1.13
- Debugging what you deploy in Go 1.12
- HTTP/2 Server Push
- Introducing HTTP Tracing
- Generating code
- Introducing the Go Race Detector
- Go maps in action
- go fmt your code
- Organizing Go code
- Debugging Go programs with the GNU Debugger
- The Go image/draw package
- The Go image package
- Error handling and Go
- First Class Functions in Go
- Profiling Go Programs
- A GIF decoder: an exercise in Go interfaces
- Introducing Gofix
- Godoc: documenting Go code
- Gobs of data
- C? Go? Cgo!
- JSON and Go
- Go Slices: usage and internals
- Go Concurrency Patterns: Timing out, moving on
- Defer, Panic, and Recover
- Share Memory By Communicating
- JSON-RPC: a tale of interfaces