Go Reflection Basics Explained In Color

REVISION 1

Hawthorne-Press.com

Go Reflection Basics Explained in Color

Published by

Hawthorne-Press.com
916 Adele Street
Houston, Texas 77009, USA

Table of Contents

Go Reflection Basics Explained in Color

Introduction

This paper will introduce the reader to the basic concepts of using the ***reflection package,*** which allows a program to inspect itself. One of the primary aspects of this package is the handling of *Struct Tags* and this is covered in a separate document under the title "*Go Structure Tags Explained In Color*".

This document will cover other reflection concepts and examples.

What Is Reflection?

Reflection is the ability of a program to manipulate objects with arbitrary types. The typical use is to determine the static type and value of an item within an *interface{}* value. It can also be used on normal variables and structures as shown by the basic operations below.

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

    v := reflect.ValueOf(x)              // Extract the Reflection interface of the GO Value
    fmt.Println("\ntype:", v.Type())
    fmt.Println("kind is float64:", v.Kind() == reflect.Float64)
    fmt.Println("value:", v.Float())
```

Given the variable x, the code above produces:

type: float64
value: 3.4

type: float64
kind is float64: true
value: 3.4

The reflection interface allows access to the type and value pair of an interface, further once you have the reflection interface of a value you may obtain the type also. See 'v' above!

The other thing to remember is that when *reflect.ValueOf(x)* is called in your code, the value *'x'* has been implicitly converted to an *interface{}*. The input parameter of both *ValueOf* and *TypeOf* are always type interface{}. Types **Value** and **Type** are both structures containing control information and data.

The number of methods in the Reflection Package is impressive and we cannot cover them all. However, we will present a variety of examples to explore many of the main concepts.

For more information about the underlying mechanics of interfaces and reflection read the following:

http://research.swtch.com/interfaces
https://blog.golang.org/laws-of-reflection

Examining The Attributes Of A Data Model Instance

When you are given the address of a Data Object Instance, how do you determine its attributes? This is another useful operation facilitated by reflection.

```
//Example From blog:   http://merbist.com/2011/06/27/golang-reflection-example
package main

import (
    "fmt"
    "reflect"
)

func main() {
    // iterate through the attributes of a Data Model instance
    for name, mtype := range attributes(&Dish{}) {     // Instantiate Data Model and get attributes
        fmt.Printf("Name:  %s,    Type:  %s\n", name, mtype)
    }
}

// Data Model
type Dish struct {
    Id_Val int
    Name   string
    Origin string
    Query  func()
}

// Example of how to use Go's reflection
// Print the attributes of a Data Model
func attributes(m interface{}) map[string]reflect.Type {
    // create an attribute data structure as a map of types keyed by a string.
    attrs := make(map[string]reflect.Type)

    typ := reflect.TypeOf(m)
    // if a pointer to a struct is passed, get the type of the dereferenced object
    if typ.Kind() == reflect.Ptr {
        typ = typ.Elem()
    }

    // Only structs are supported so return an empty result if the passed object is not a struct
    if typ.Kind() != reflect.Struct {
        fmt.Printf("%v type can't have attributes inspected\n", typ.Kind())
        return attrs
    }

    // loop through the struct's fields and set the map
    for i := 0; i < typ.NumField(); i++ {
        p := typ.Field(i)
        if !p.Anonymous {
            attrs[p.Name] = p.Type
        }
    }
    return attrs
}
```

The results running the above program:

Name: Query, Type: func()
Name: Id_Val, Type: int
Name: Name, Type: string
Name: Origin, Type: string

The Attributes Method

This method expects an interface parameter and returns a map. The signature of the function is as follows:

attributes(m interface{}) map[string]reflect.Type

First, we create an empty map for the results; each name field will have its reflection type. Next, we extract the reflection type object.

Before we can determine the attributes of each structure field, we should determine that the interface submitted meets the requirements of the function.

- If it is a pointer instead of a value, it "normalized" by dereferencing.
- If the normalized type is not a structure, an error is reported and an empty map is returned.

Given that there are no errors, the structure is scanned for each field's reflection type.

We have the reflection type object and from that, we can determine the number of fields and return a pointer to each field. However, we have one more check to make. When the reflection object's Anonymous flag is set, the field will be ignored.

By ignoring error checks, *which must be made*, the code simplifies to the following:

```
func attributes(m interface{}) map[string]reflect.Type {
    attrs := make(map[string]reflect.Type)     // Make empty map
    typ := reflect.TypeOf(m)                   // Extract Reflection Type Object
    for i := 0; i < typ.NumField(); i++ {      // Loop over typ.NumField()number of fields
        p := typ.Field(i)                      // Get pointer to nth field
        attrs[p.Name] = p.Type                 // Load Map entry with Reflection Type and Field Name
   }
    return attrs                               // Return Competed Map
}
```

The Main Function

The main function merely creates, in place, a pointer to an instantiated interface of type Dish.

&Dish{}

Then is simply steps through the map returned by the *attributes()* function printing out the name and type fields.

```
    for name, mtype := range attributes(&Dish{}) {     // Instantiate Data Model and get attributes
        fmt.Printf("Name:  %s,    Type:  %s\n", name, mtype)
    }
```

As the reader should notice, reflection is a straightforward process and not that mysterious.

Handling Pointers

So far, we have been handling values for the most part. If the interface value is a pointer, then we can use the function *reflect.Indirect()* to convert a pointer value into its actual value.

```
package main

import "fmt"
```

```
import "reflect"

type A struct {
    Foo string
}

func (a *A) PrintFoo() {
    fmt.Println("Foo value is " + a.Foo)
}

func main() {
    a := &A{Foo: "afoo"}
    val := reflect.Indirect(reflect.ValueOf(a))   // Return the Reflection value of pointer‘a’
    fmt.Println(val.Type().Field(0).Name)         // Print field name of structure ‘A’
}
```

Manipulating Fields – Setting And Getting

In the following short example, we demonstrate some different ways of getting field values and setting field values. To change the value of a field, the field name must be *exportable* (i.e. the first letter of the name must be a capital letter).

```
package main

import (
    "fmt"
    "reflect"
)

func main() {

    type MyStruct struct {                  // Define Sructure
        N int
    }
    n := MyStruct{1}                        // Instantiate Variable MyStruct

    // get
    immutable := reflect.ValueOf(n)
    val := immutable.FieldByName("N").Int() // Accesses field by Field Name not index
    fmt.Printf("N=%d\n", val)               // prints 1

    // set
    mutable := reflect.ValueOf(&n).Elem()   // Returns a value if parameter is interface or pointer
    mutable.FieldByName("N").SetInt(7)
    fmt.Printf("N=%d\n", n.N)               // prints 7

}
```

Reflect Method MakeFunc()

This particular reflection function is Go language’s minimalist approach to generic programming. However, even putting that way is misleading. This function does allow creating functions that can operate without knowing the “type” of data they will be processing, albeit in a more limited fashion. The following is from the reflection package. We have removed some of the comments to make following the code easier. The original program can be found at https://golang.org/pkg/reflect/#example_MakeFunc .

```
package main

import (
       "fmt"
```

```
        "reflect"
)

func main() {
// swap is the implementation passed to MakeFunc.
        swap := func(in []reflect.Value) []reflect.Value {
                return []reflect.Value{in[1], in[0]}
        }

// makeSwap expects fptr to be a pointer to a nil function.
        makeSwap := func(fptr interface{}) {
                fn := reflect.ValueOf(fptr).Elem()     // fptr is a pointer to a function
                v := reflect.MakeFunc(fn.Type(), swap) // Make a function of the right type
                fn.Set(v)                              // Assign it to the value fn represents
        }

        // Make and call a swap function for ints.
        var intSwap func(int, int) (int, int)
        makeSwap(&intSwap)
        fmt.Println(intSwap(0, 1))

        // Make and call a swap function for float64s.
        var floatSwap func(float64, float64) (float64, float64)
        makeSwap(&floatSwap)
        fmt.Println(floatSwap(2.72, 3.14))

}
```

The result of the program above is:

1 0
3.14 2.74

What are the problems with this approach? It is not “generic”, is it! You still have to create a function for each type you process. You are simply providing an automatic way to provision the results. On the plus side, it does provide a solution for a narrower set of problems. You could use it to launch functions from a dispatch table for example.

The function variable specification supplies the *function signature* for the function created by the reflect.MakeFunc method. The method must know the type and number of values in the both input and the output. The function signature determines these parameters.

The function signature is defined by:

```
        var intSwap func(int , int) (int, int)
```

The user function itself is defined in terms of reflection values. It expects a slice of Values for both the input and output. See below:

```
        swap := func(in []reflect.Value) []reflect.Value {
                return []reflect.Value{in[1], in[0]}
        }
```

While somewhat confusing at first, the function wrapping the user supplied function does the following:

- Converts the input arguments supplied into a slice of reflection Values

- Executes the supplied function with those arguments: **results := user_func([]reflect.Value)**
- Returns the results as a slice of values, one per formal result

In the above example, swap(), the function simply returns two values based on the input slice values. The function returned by MakeFunc, when executed, "appears" to operate exactly as the variable function signature specifies, even though a little magic is going on behind the scenes.

The following modification of the original intswap() function call, shown in red, is intended to show this more clearly. The call to the intswap() function illustrates that it "operates" just as the function signature defines.

```
        var intSwap func(int, int) (int, int)
        makeSwap(&intSwap)
        one, zero := intSwap(0, 1)
        fmt.Println("one and zero = ", one, zero)   // Prints one and zero = 1 0
```

Variadic Inputs

The documentation on reflection mentions *variadic inputs* only in passing. The following shows the form:

```
        var intSwap func(...int) (int, int)
```

Let us modify the previous example:

```
        var intSwap func(…int) (int, int)
        makeSwap(&intSwap)
        one, zero := intSwap(0, 1)
        fmt.Println("one and zero = ", one, zero)   // Prints one and zero = 1 0
```

This is correct as far as it goes, but it will fail. The reason is not in this code, but in the code for the swap function. The original swap code expects a slice of Values, but the variadic input is already a slice. To access the input values of a variadic input, the slice must be indexed. In this case, you are actually processing a slice of a slice. For the program to work, swap() must be modified as shown in red:

```
    swap := func(in []reflect.Value) []reflect.Value {
        return []reflect.Value{in[0].Index(1), in[0].Index(0)}
    }
```

Reflect Method DeepEqual

This method is a strange beast. It attempts to be a recursive "==" operator. However, because of some particularities in the Go Language, it is not always possible. The following are some simple rules:

Array values are comparable if the values of the array element type are comparable. Two array values are equal if their corresponding elements are equal.

For example, pointers are always equal themselves, even they point to problematic values. The same is true of slices and maps, if 'x' and 'y' point the same slice or same map; they are deeply equal regardless of content.

It is possible for a value to be unequal to itself, either because it is of func type (uncomparable in general) or because it is a floating-point NaN value (not equal to itself in a floating-point comparison).

The complete description of this method contains a lot of qualifications and caveats that apply is special cases, please read the documentation. See: https://golang.org/pkg/reflect/#DeepEqual

The following is an example of comparing slices.

```
package main

import (
	"fmt"
	"reflect"
)

func main() {
	var oldSlice string
	var newSlice string

	bool := reflect.DeepEqual(oldSlice, newSlice)  // Both slices are empty
	fmt.Println(bool)                              // Prints true

	oldSlice = "abc"
	newSlice = "def"

	bool = reflect.DeepEqual(oldSlice, newSlice)    // Slices have different values
	fmt.Println(bool)                               // Prints false

	newSlice = "abc"                                // Updates newSlice value

	bool = reflect.DeepEqual(oldSlice, newSlice)    // Slices now have the same values
	fmt.Println(bool)                               // Prints true
}
```

Reflect Method Select

We often use the Go language *select* statement to multiplex the selection of active channels. What do we select on when we do not have a native channel? What if we are processing a reflection interface, for example, that contains channels? *The normal select statement expects a standard channel, not a reflection value.*

The section is going to explore several ways to handle this problem. The following examples are a modified version of the code from: https://www.socketloop.com/references/golang-reflect-select-and-selectcase-function-example. Additionally, members of **StackOverflow.com** provided insights that I missed. Thanks.

Basic Example

The following program uses **reflect.Select** to handle the receive channel instead of the **select statement**. Since the channel value is passed as **chan** value, the use of the reflect.Select statement is not actually necessary.

```
package main

import (
    "fmt"
    "reflect"
)

func main() {
    var sendCh = make(chan int)                    // Create channel to use

    var increaseInt = func(c chan int) {           // Counter Function
        for i := 0; i < 8; i++ {
            c <- i
        }
        close(c)
    }
    go increaseInt(sendCh)                         // Launch Counter Routine as a GO Routine

    jobRun(sendCh)                                 // Call Channel Processing Function
}

func jobRun(sendCh chan int) {
    var selectCase = make([]reflect.SelectCase, 1)  // Create selectCase Table
    selectCase[0].Dir = reflect.SelectRecv
    selectCase[0].Chan = reflect.ValueOf(sendCh)

    counter := 0
    for counter < 1 {
        chosen, recv, recvOk := reflect.Select(selectCase)
        if recvOk {
            fmt.Println(chosen, recv.Int(), recvOk)

        } else {
            fmt.Println("Exit Condition Detected:  ", chosen, recv.Int(), recvOk)
            counter++                              // Forces Loop Exit
        }
    }
}
```

Results:

0 0 true
: : :
0 7 true
Exit Condition Detected: 0 0 false

Interface Example

The following modification of the program assumes that the channel is from an interface input parameter. We will create a structure with one channel field and initialize it. We also change the parameter of jobRun to an interface. The changes to the previous program are shown in red.

```
package main

import (
    "fmt"
    "reflect"
)
// Add Type for Struct
type Foo struct {
   Ch chan int
}

func main() {
//  -----  following replaces  " var sendCh := make(chan int) " ----
    sendCh := Foo{make(chan int)}
```

```
//  ----- End of replacement code -----

    var increaseInt = func(c chan int) {
        for i := 0; i < 8; i++ {
            c <- i
        }
        close(c)
    }

    go increaseInt(sendCh.Ch)
    runJob(sendCh)
}

func runJob(f interface{}) {
    var selectCase = make([]reflect.SelectCase, 1)

//  ----- Create "sendCh" by extracting from the interface -----
    val := reflect.ValueOf(f)
    valueField := val.FieldByName("Ch")
    sendCh := valueField.Interface()
//  ----- End of replacement code -----

    selectCase[0].Dir = reflect.SelectRecv
    selectCase[0].Chan = reflect.ValueOf(sendCh)

    counter := 0
    for counter < 1 {
        chosen, recv, recvOk := reflect.Select(selectCase) // <--- here
        if recvOk {
            fmt.Println(chosen, recv.Int(), recvOk)

        } else {
            fmt.Println("Exit Condition Detected:  ", chosen, recv.Int(), recvOk)
            counter++
        }
    }
}
```

As was pointed out to me, if I know what the interface contains and its type is available, the following replacement can be made. The removed code is shown in gray and replacement code is in red.!

```
//  ----- Create "sendCh" by extracting it from the interface -----

    val := reflect.ValueOf(f)                // Using Reflection to create sendCh
    valueField := val.FieldByName("Ch")
    sendCh := valueField.Interface()

    val, ok := f.(Foo)                       // Using Type Assertion to create sendCh
    if !ok {                                 // Check for conversion failure
        fmt.Println("Interface is not Foo")
    }
    sendCh := val.Ch                         // “val” is an instantiation of structure Foo

//  ----- End of replacement code -----
```

The result is the same. The reflection version is necessary if you do not have access to its type or you must use reflection to find the value you need. For example, you may be passed several different structures that all have channel information in different locations. The developer may need to look for a specific field name or specific struct tag!

The examples above should give a leg up on handling these situations.

Type Inference And Conversion

The type system is at the heart of the Go language. Whereas reflection handles the manipulation of types and values in a run-time environment, built-in type language elements handle both static and dynamic conversions of value types at compile time and run-time.

When declaring a variable without an explicit type, the type may be inferred by the value on the right hand side. If the compiler can determine the type of the right hand side value, it automatically uses that type for the variable being defined provided that a type was not specified. The right-hand value may be a constant, variable, or an expression. Let us look some examples.

```
var k int = 88                  // Variable definition with initialization
var g float64 = float64(k)      // Variable definition with conversion and initialization

j := 88                         // Type Inference from right-hand values
i := k
f := float64(i) * 0.1           // Type can be inferred from the value of an expression
u := uint32(f)

fmt.Println(k, g, j, i, f, u)
```

Results

88 88 76 88 8.8 8

Interface Values Type Assertions and Switches

The Go Language as has a method for converting the dynamic type of an interface values to the appropriate static type. If the developer knows the type of the value stored in an interface, a *type assertion* can be uses to recover the static type and data.

```
var myData interface{} = “string msg”
var moreData intergace{} = 123
```

Given the definitions above, we can get the string value with a type assertion in the form: **value(type).**

```
Mystring  :=  myData.(string)               // Converts interface value to static string
```

However, this assumes the conversion is possible. While the code above will work if all goes right, it will crash the program if there is a problem. The way to prevent this is to use the following form.

```
Mystring, ok  :=  myData.(string)           // Converts interface value to static string
```

The error return value ‘ok’ will be true if the conversion is successful. The following is an example from a working program. Notice that the type specified can be any custom type in addition to built-types.

```
  type Foo struct {
      Ch chan int
  }
    ::   ::   ::
  val, ok := f.(Foo)      // Check Type Directly
```

```
  if !ok {
      fmt.Println("Interface value does not contain Foo")
  }
```

This is acceptable code when the type is known. If the interface value is not known to contain a specific type, the type can be discovered using a *type switch*. Instead of inserting a type name within assertion expression, use the key word **type**. For Example:

```
var myData interface{} = "string msg"

switch v := myData.(type) {
case string:
       fmt.Println(v)
case int32, int64:
       fmt.Println(v)
default:
       fmt.Println("unknown")
}
```

The type of an interface value may also be displayed using the fmt package's ***%T*** operator as shown below:

```
fmt.Printf("Static type = %T\n", myData)
```

A Final Word

As stated elsewhere in this and previous documents, this is only the beginning of what can be accomplished with Go Reflections.

Strive for simplicity in your code. The *Occam's Razor principle* applies as strongly for software as it does for many other walks of life. In most situations, giving up a little in efficiency to simplify your code is often the right decision. Code Simplicity reduces the chances of subtle errors.

As always, comments are welcome at hawthornepresscom@gmail.com.

As to why I have an email address on **gmail** instead of on my website, has to do with AT&T and their inability to handle the concept that not everyone is running a Microsoft OS and my stubborn insistence on running Linux.

Published by

Hawthorne-Press.com
10310 Moorberry Lane
Houston, Texas 77043, USA

Revision History

Date	By	Section	Changes
05/06/2016	C.E. Thornton	All	Initial Document
05/27/2018	C.E. Thornton	CopyRight	Update Address