Go Struct Tags Explained In Color

REVISION 1

Hawthorne-Press.com

Go Struct Tags Explained in Color

Published by

Hawthorne-Press.com

10310 Moorberry Lane
Houston, Texas 77043, USA

TABLE OF CONTENTS

GO STRUCT TAGS EXPLAINED IN COLOR

INTRODUCTION

One of the less explored aspects of Golang is *Reflection*, the method by which a program can inspect its own code. It has many uses; one of these is inspecting *Struct Tags*. In the following example, the Struct Tags are in red.

```
type Exam1 struct {
    Name    string `json:"enc_name"`
    Address string `json:"enc_address"`
    City    string `json:"enc_city"`
}
```

JSON EXAMPLE PROGRAM

The **JSON** package uses Struct Tags, when present, to specify the data identifier used, whether to include a field, and to skip empty fields when marshalling a structure.

```
package main

import (
    "encoding/json"
    "fmt"
)

type Exam1 struct {
    Name    string `json:"enc_name"`
    Address string `json:"enc_address"`
    City    string `json:"enc_city"`
}

func main() {

    val1 := Exam1{
        Name:    "Charles",
        Address: "10 Main St",
        City:    "Houston",
}
    res1, _ := json.Marshal(val1)
    fmt.Println(string(res1))
}
```

The result of the program is:

{"enc_name":"Charles","enc_address":"10 Main St","enc_city":"Houston"}

When the structure encoded by the JSON package has Struct Tags of the form `**"json: "text"**` the "text" value is used for the data identifier in the marshalled data. If the above program did not have Struct Tags, the marshalled result would use the *Field Name* for the data identifier as shown below:

{"Name":"Charles","Address":"10 Main St","City":"Houston"}

XML Example Program

The following program is a modified version of an example program presented in the **XML** package.

```
package main

import (
    "encoding/xml"
    "fmt"
    "os"
)

func ExampleMarshalIndent() {
    type Address struct {
        City, State string
    }
    type Person struct {
        XMLName   xml.Name `xml:"person"`
        Id        int      `xml:"id,attr"`
        FirstName string   `xml:"name>first"`
        LastName  string   `xml:"name>last"`
        Age       int      `xml:"age"`
        Height    float32  `xml:"height,omitempty"`
        Married   bool
        Address
        Comment string `xml:",comment"`
    }

    v := &Person{Id: 13, FirstName: "John", LastName: "Doe", Age: 42}
    v.Comment = " Need more details. "
    v.Address = Address{"Hanga Roa", "Easter Island"}

    output, err := xml.MarshalIndent(v, "  ", "    ")
    if err != nil {
        fmt.Printf("error: %v\n", err)
    }

    os.Stdout.Write(output)
    // Output:
    //   <person id="13">
    //       <name>
    //           <first>John</first>
    //           <last>Doe</last>
    //       </name>
    //       <age>42</age>
    //       <Married>false</Married>
    //       <City>Hanga Roa</City>
    //       <State>Easter Island</State>
    //       <!-- Need more details. -->
    //   </person>
}
func main() {
    ExampleMarshalIndent()
    fmt.Println("")
}
```

Struct Tags And Reflection

The previous examples were just two of the more common packages that use Struct Tags. The value of these Tags is to add context and/or attributes to the fields in a structure. They can be used to set default values for a structure containing configuration data for example. The uses are endless and only depend on the imagination of the developer.

The primary thing to remember is that only *Exported Fields* of structures are seen by the reflection package. Reflection processing is done in the Reflection package, only exported fields in a structure can be seen by that package. *Non-structure* values can by processed by many reflection methods such as TypeOf() and ValueOf() methods. While structure values require "looking" inside the value, normal values do not require any deeper inspection.

Struct Tags, hereafter referred to just Tags, are strings that have no defined "Formal" format. However, developers have adopted a standard convention that is supported by the reflection package, which is a concatenation of optionally space-separated *key :" value"* pairs. Each Tag is a non-empty string of non-control characters other than space, quote, and colon. For example:

Field int `json:"gopher"`

While a developer can format Tags any way they wish, following the normal convention allows interaction between more than one package. For example, if we wish to specify Tags for both XML and JSON for the same structure, it is handled by the normal convention. See Below:

Field int `json:"gopher" xml:"name>first"`

When processed by a package function such as JSON *Marshall*, it will only use the "value" associated with the key "json:". The same applies for XML processing.

A Developer should follow the convention stated above unless there is an overriding reason to ignore the convention. As shown by the example programs the value string can contain a lot of information. The formats of values are defined in the packages that use Struct Tags.

Using Reflection

So far, we have used reflection as a by-product of other packages, such as *encoding/json* and *encoding/xml.* In this document we will primarily explore the *reflect* package in relation to Struct Tags. While reflect package provides a large number of methods for examining its own code, a smaller number relate to the subject of Tags.

The following are the primary reflection methods used for examining Struct Tags. There are variations of these basic methods that are not discussed.

- TypeOf (val)– Extracts the dynamic type information, usually from a static interface{} type.
- ValueOf (val)– Returns the value of the run-time data.
- ValueOf(val).NumberField() – Returns number of fields in structure.
- Field(i) – Returns i'th field of the structure. (See also FieldByName(name) and other combinations).
- Field(i).Tag – Returns Tag Field
- Field(i).Tag.Get("tag") – Returns Tag Value
- Field(i).Kind() – Returns Type of value

Extracting Basic Information From Non-Structure Values

Given a value, the reflection package can extract information about the variable or structure.

```
var x float64 = 3.4                       // Define a Variable

fmt.Println("type:", reflect.TypeOf(x))   // Returns dynamic reflection type "type: float64"
fmt.Println("value:", reflect.ValueOf(x)) // Returns "value: 3.4"

v := reflect.Value(x)                     // Captures the dynamic reflection information of 'x'
```

Given the value 'v', we can extract reflection information directly as show below:

```
fmt.Println("type:", v.Type())                                  // "type: float64"
fmt.Println("kind is float64:", v.Kind() == reflect.Float64)    // "kind is float64: true"
fmt.Println("Kind of value = ", v.Kind())                       // "kind of value: float64"
fmt.Println("value:", v.Float())                                // "value: 3.4"
```

Extracting Struct Tags Information

If you try to add a Struct Tag to a non-structure value, you will encounter the following error:

example.go:12: syntax error: unexpected string literal at end of statement

To extract Tag information from a structure, you must do the following:

- Get the dynamic type of the structure.
- Extract the field of interest.
- Extract the Tag from the field.
- Extract the value from the Tag of interest.

Let us create a structure 'S', with a Struct Tag and instantiate a variable of that type:

```
type S struct {
    F string `species:"gopher" color:"blue"`
}
s := S{"Animal"}
```

Next, we need the dynamic reflection type of the instantiated variable. This is necessary because the TypeOf Method parameter is an *interface value*. Thus, its dynamic type must be determined for further processing.

Func TypeOf (i interface{}) type

```
st := reflect.TypeOf(s)
fmt.Println("type: ", st)     // Returns "type: main.S"
```

Notice that the dynamic reflection type indicates the package as well as the structure name. This gives the reflection package the information needed to process the value. The reflection package cannot see "into" structures unless they are exported values. Only exported values can be seen outside another package.

To extract the field of interest we need to know its position index inside the structure. In the current example, we only have one field and fields indexes start at zero(0). The following extracts the field and prints its contents:

```
    field := st.Field(0)        // Extract Field Information
    fmt.Println(field)          // Returns `F string species:"gopher" color:"blue" 0 [0] false`
```

This response contains internal housekeeping information that is irrelevant to this discussion and is shown in red. However, it does show the information that is available. If we ask for the contents of the Struct Tag with the reflection.Tag() method we get:

```
    fmt.Println(field.Tag)      // Returns `species:"gopher" color:"blue"`
```

When we ask for the "values" of each Tag, the following is returned.

```
   fmt.Println(field.Tag.Get("color"), field.Tag.Get("species"))    // Returns `blue gopher`
```

Extracting Struct Tags From Complex Structures

The following example is from Matthew Conninghams blog on Struct Tags: http://mattcunningham.net/post/125740143590/tutorial-utilizing-custom-tags-in-struct-type.

This is an example of using Tags to create a map containing the specified URL name and the input values. The Encode method from the *net/url* package returns a URL encoded value in the form ("bar=baz&foo=quux") sorted by key.

The ReadAndTagValue() Method follows the following steps:

- Get the dynamic reflection type.
- Create a blank url.Values map.
- Loop over the number of fields in the structure
- Get Tag String
- Get the contents of Field String
- Create a map entry out of the URL Tag value and field string.
- Append the new entry to the url.Values map.
- Return populated url.Values map.

ReadAndTagValue() Example Program

```
package main

import (
    "fmt"
    "net/url"
    "reflect"
)

type ExampleType struct {
    Name    string `url:"name"`
    Address string `url:"address"`
    City    string `url:"city"`
}
```

```
func ReadTagAndValue(val interface{}) url.Values {
    value := reflect.ValueOf(val)                  // returns dynamic reflection type
    params := url.Values{}
    for i := 0; i < value.NumField(); i++ {        // iterates through every struct type field
        tag := value.Type().Field(i).Tag           // returns the tag string
        field := value.Field(i)                    // returns the content of the struct type field
        params.Set(tag.Get("url"), field.String()) // Append to Value Map
    }
    return params                                  // Return completed Value Map
}

func main() {
    params := ReadTagAndValue(ExampleType{         // Call Function with ExampleType Value
        Name:    "Example",
        Address: "1234 Apple Street",
        City:    "New York",
    })
    fmt.Println(params.Encode())                   // Print Encoded Value
}
```

Studying the program above should give a developer most of the tools you will need to incorporate Struct Tags into your programs.

A Final Word

Strive for simplicity in your code. The Occam 's razor principle applies as strongly for software as it does for many other occupations. In most situations, giving up a little in efficiency to simply 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
06/05/2016	C.E. Thornton	All	Initial Document