Go 每日一库之 mapstructure

转载 dz45693 于 2021-10-12 11:54:35 发布 2833
分类专栏: GO 文章标签: golang restful http

简介

mapstructure用于将通用的map[string]interface{} 解码到对应的 Go 结构体中，或者执行相反的操作。很多时候，解析来自多种源头的数据流时，我们一般事先并不知道他们对应的具体类型。只有读取到一些字段之后才能做出判断。这时，我们可以先使用标准的encoding/json库将数据解码为map[string]interface{}类型，然后根据标识字段利用mapstructure库转为相应的 Go 结构体以便使用。

快速使用

本文代码采用 Go Modules。

首先创建目录并初始化：

```
$ mkdir mapstructure && cd mapstructure

$ go mod init github.com/darjun/go-daily-lib/mapstructure
```

下载mapstructure库：

```
$ go get github.com/mitchellh/mapstructure
```

使用：

```
package main

import (
  "encoding/json"
  "fmt"
  "log"

  "github.com/mitchellh/mapstructure"
)

type Person struct {
  Name string
  Age  int
  Job  string
}

type Cat struct {
  Name  string
  Age   int
  Breed string
}

func main() {
  datas := []string{`
    {
      "type": "person",
      "name":"dj",
      "age":18,
      "job": "programmer"
    }
  `,
    `
    {
      "type": "cat",
      "name": "kitty",
      "age": 1,
      "breed": "Ragdoll"
    }
  `,
```

```
  }
  41
  for _, data := range datas {
    var m map[string]interface{}
    err := json.Unmarshal([]byte(data), &m)
    if err != nil {
      log.Fatal(err)
    }

    switch m["type"].(string) {
    case "person":
      var p Person
      mapstructure.Decode(m, &p)
      fmt.Println("person", p)

    case "cat":
      var cat Cat
      mapstructure.Decode(m, &cat)
      fmt.Println("cat", cat)
    }
  }
}
```

运行结果：

```
$ go run main.go
person {dj 18 programmer}
cat {kitty 1 Ragdoll}
```

我们定义了两个结构体Person和Cat，他们的字段有些许不同。现在，我们约定通信的 JSON 串中有一个type字段。当type的值为person时，该 JSON 串表示的是Person类型的数据。当type的值为cat时，该 JSON 串表示的是Cat类型的数据。

上面代码中，我们先用json.Unmarshal将字节流解码为map[string]interface{}类型。然后读取里面的type字段。根据type字段的值，再使用mapstructure.Decode将该 JSON 串分别解码为Person和Cat类型的值，并输出。

实际上，Google Protobuf 通常也使用这种方式。在协议中添加消息 ID 或**全限定消息名**。接收方收到数据后，先读取协议 ID 或**全限定消息名**。然后调用 Protobuf 的解码方法将其解码为对应的Message结构。从这个角度来看，mapstructure也可以用于网络消息解码，**如果你不考虑性能的话**。

字段标签

默认情况下，mapstructure使用结构体中字段的名称做这个映射，例如我们的结构体有一个Name字段，mapstructure解码时会在map[string]interface{}中查找键名name。注意，这里的name是大小写不敏感的！

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

当然，我们也可以指定映射的字段名。为了做到这一点，我们需要为字段设置mapstructure标签。例如下面使用username代替上例中的name：

```
type Person struct {
  Name string `mapstructure:"username"`
}
```

看示例：

```
type Person struct {
  Name string `mapstructure:"username"`
  Age  int
  Job  string
}

type Cat struct {
  Name  string
  Age   int
  Breed string
```

```
}12
func main() {
  datas := []string{`
    {
      "type": "person",
      "username":"dj",
      "age":18,
      "job": "programmer"
    }
  `,
    `
    {
      "type": "cat",
      "name": "kitty",
      "Age": 1,
      "breed": "Ragdoll"
    }
  `,
    `
    {
      "type": "cat",
      "Name": "roooooose",
      "age": 2,
      "breed": "shorthair"
    }
  `,
  }

  for _, data := range datas {
    var m map[string]interface{}
    err := json.Unmarshal([]byte(data), &m)
    if err != nil {
      log.Fatal(err)
    }

    switch m["type"].(string) {
    case "person":
      var p Person
      mapstructure.Decode(m, &p)
      fmt.Println("person", p)

    case "cat":
      var cat Cat
      mapstructure.Decode(m, &cat)
      fmt.Println("cat", cat)
    }
  }
}
```

上面代码中，我们使用标签`mapstructure:"username"`将`Person`的`Name`字段映射为`username`，在 JSON 串中我们需要设置`username`才能正确解析。另外，注意到，我们将第二个 JSON 串中的`Age`和第三个 JSON 串中的`Name`首字母大写了，但是并没有影响解码结果。`mapstructure`处理字段映射是大小写不敏感的。

内嵌结构

结构体可以任意嵌套，嵌套的结构被认为是拥有该结构体名字的另一个字段。例如，下面两种`Friend`的定义方式对于`mapstructure`是一样的：

```
type Person struct {
  Name string
}

// 方式一
type Friend struct {
  Person
}

// 方式二
type Friend struct {
  Person Person
```

```
}
```

为了正确解码，Person结构的数据要在person键下：

```
map[string]interface{} {
  "person": map[string]interface{}{"name": "dj"},
}
```

我们也可以设置mapstructure:",squash"将该结构体的字段提到父结构中：

```
type Friend struct {
  Person `mapstructure:",squash"`
}
```

这样只需要这样的 JSON 串，无效嵌套person键：

```
map[string]interface{}{
  "name": "dj",
}
```

看示例：

```
type Person struct {
  Name string
}

type Friend1 struct {
  Person
}

type Friend2 struct {
  Person `mapstructure:",squash"`
}

func main() {
  datas := []string{`
    {
      "type": "friend1",
      "person": {
        "name":"dj"
      }
    }
  `,
    `
    {
      "type": "friend2",
      "name": "dj2"
    }
  `,
  }

  for _, data := range datas {
    var m map[string]interface{}
    err := json.Unmarshal([]byte(data), &m)
    if err != nil {
      log.Fatal(err)
    }

    switch m["type"].(string) {
    case "friend1":
      var f1 Friend1
      mapstructure.Decode(m, &f1)
      fmt.Println("friend1", f1)

    case "friend2":
```

```
        var f2 Friend2
        mapstructure.Decode(m, &f2)
        fmt.Println("friend2", f2)
    }
  }
}
```

注意对比Friend1和Friend2使用的 JSON 串的不同。

另外需要注意一点，如果父结构体中有同名的字段，那么mapstructure会将JSON 中对应的值**同时设置到这两个字段中**，即这两个字段有相同的值。

未映射的值

如果源数据中有未映射的值（即结构体中无对应的字段），mapstructure默认会忽略它。

我们可以在结构体中定义一个字段，为其设置mapstructure:",remain"标签。这样未映射的值就会添加到这个字段中。注意，这个字段的类型只能为map[string]interface{}或map[interface{}]interface{}。

看示例：

```
type Person struct {
  Name  string
  Age   int
  Job   string
  Other map[string]interface{} `mapstructure:",remain"`
}

func main() {
  data := `
    {
      "name": "dj",
      "age":18,
      "job":"programmer",
      "height":"1.8m",
      "handsome": true
    }
  `

  var m map[string]interface{}
  err := json.Unmarshal([]byte(data), &m)
  if err != nil {
    log.Fatal(err)
  }

  var p Person
  mapstructure.Decode(m, &p)
  fmt.Println("other", p.Other)
}
```

上面代码中，我们为结构体定义了一个Other字段，用于保存未映射的键值。输出结果：

```
other map[handsome:true height:1.8m]
```

逆向转换

前面我们都是将map[string]interface{}解码到 Go 结构体中。mapstructure当然也可以将 Go 结构体反向解码为map[string]interface{}。在反向解码时，我们可以为某些字段设置mapstructure:",omitempty"。这样当这些字段为默认值时，就不会出现在结构的map[string]interface{}中：

```
type Person struct {
  Name string
  Age  int
  Job  string `mapstructure:",omitempty"`
}

func main() {
  p := &Person{
```

```
    Name: "dj",10     Age:  18,
  }

  var m map[string]interface{}
  mapstructure.Decode(p, &m)

  data, _ := json.Marshal(m)
  fmt.Println(string(data))
}
```

上面代码中，我们为Job字段设置了mapstructure:",omitempty"，且对象p的Job字段未设置。运行结果：

```
$ go run main.go
{"Age":18,"Name":"dj"}
```

Metadata

解码时会产生一些有用的信息，mapstructure可以使用Metadata收集这些信息。Metadata结构如下：

```
// mapstructure.go
type Metadata struct {
  Keys   []string
  Unused []string
}
```

Metadata只有两个导出字段：

- Keys：解码成功的键名；
- Unused：在源数据中存在，但是目标结构中不存在的键名。

为了收集这些数据，我们需要使用DecodeMetadata来代替Decode方法：

```
type Person struct {
  Name string
  Age  int
}

func main() {
  m := map[string]interface{}{
    "name": "dj",
    "age":  18,
    "job":  "programmer",
  }

  var p Person
  var metadata mapstructure.Metadata
  mapstructure.DecodeMetadata(m, &p, &metadata)

  fmt.Printf("keys:%#v unused:%#v\n", metadata.Keys, metadata.Unused)
}
```

先定义一个Metadata结构，传入DecodeMetadata收集解码的信息。运行结果：

```
$ go run main.go
keys:[]string{"Name", "Age"} unused:[]string{"job"}
```

错误处理

mapstructure执行转换的过程中不可避免地会产生错误，例如 JSON 中某个键的类型与对应 Go 结构体中的字段类型不一致。Decode/DecodeMetadata会返回这些错误：

```
type Person struct {
  Name   string
```

```
  Age    int
  Emails []string
}

func main() {
  m := map[string]interface{}{
    "name":   123,
    "age":    "bad value",
    "emails": []int{1, 2, 3},
  }

  var p Person
  err := mapstructure.Decode(m, &p)
  if err != nil {
    fmt.Println(err.Error())
  }
}
```

上面代码中，结构体中Person中字段Name为string类型，但输入中name为int类型；字段Age为int类型，但输入中age为string类型；字段Emails为[]string类型，但输入中emails为[]int类型。故Decode返回错误。运行结果：

```
$ go run main.go
5 error(s) decoding:

* 'Age' expected type 'int', got unconvertible type 'string'
* 'Emails[0]' expected type 'string', got unconvertible type 'int'
* 'Emails[1]' expected type 'string', got unconvertible type 'int'
* 'Emails[2]' expected type 'string', got unconvertible type 'int'
* 'Name' expected type 'string', got unconvertible type 'int'
```

从错误信息中很容易看出哪里出错了。

弱类型输入

有时候，我们并不想对结构体字段类型和map[string]interface{}的对应键值做强类型一致的校验。这时可以使用WeakDecode/WeakDecodeMetadata方法，它们会尝试做类型转换：

```
type Person struct {
  Name   string
  Age    int
  Emails []string
}

func main() {
  m := map[string]interface{}{
    "name":   123,
    "age":    "18",
    "emails": []int{1, 2, 3},
  }

  var p Person
  err := mapstructure.WeakDecode(m, &p)
  if err == nil {
    fmt.Println("person:", p)
  } else {
    fmt.Println(err.Error())
  }
}
```

虽然键name对应的值123是int类型，但是在WeakDecode中会将其转换为string类型以匹配Person.Name字段的类型。同样的，age的值"18"是string类型，在WeakDecode中会将其转换为int类型以匹配Person.Age字段的类型。 需要注意一点，如果类型转换失败了，WeakDecode同样会返回错误。例如将上例中的age设置为"bad value"，它就不能转为int类型，故而返回错误。

解码器

除了上面介绍的方法外，mapstructure还提供了更灵活的解码器（Decoder）。可以通过配置DecoderConfig实现上面介绍的任何功能：

```go
// mapstructure.go
type DecoderConfig struct {
  ErrorUnused       bool
  ZeroFields        bool
  WeaklyTypedInput  bool
  Metadata          *Metadata
  Result            interface{}
  TagName           string
}
```

各个字段含义如下：

- ErrorUnused：为true时，如果输入中的键值没有与之对应的字段就返回错误；
- ZeroFields：为true时，在Decode前清空目标map。为false时，则执行的是map的合并。用在struct到map的转换中；
- WeaklyTypedInput：实现WeakDecode/WeakDecodeMetadata的功能；
- Metadata：不为nil时，收集Metadata数据；
- Result：为结果对象，在map到struct的转换中，Result为struct类型。在struct到map的转换中，Result为map类型；
- TagName：默认使用mapstructure作为结构体的标签名，可以通过该字段设置。

看示例：

```go
type Person struct {
  Name string
  Age  int
}

func main() {
  m := map[string]interface{}{
    "name": 123,
    "age":  "18",
    "job":  "programmer",
  }

  var p Person
  var metadata mapstructure.Metadata

  decoder, err := mapstructure.NewDecoder(&mapstructure.DecoderConfig{
    WeaklyTypedInput: true,
    Result:           &p,
    Metadata:         &metadata,
  })

  if err != nil {
    log.Fatal(err)
  }

  err = decoder.Decode(m)
  if err == nil {
    fmt.Println("person:", p)
    fmt.Printf("keys:%#v, unused:%#v\n", metadata.Keys, metadata.Unused)
  } else {
    fmt.Println(err.Error())
  }
}
```

这里用Decoder的方式实现了前面弱类型输入小节中的示例代码。实际上WeakDecode内部就是通过这种方式实现的，下面是WeakDecode的源码：

```go
// mapstructure.go
func WeakDecode(input, output interface{}) error {
  config := &DecoderConfig{
    Metadata:         nil,
    Result:           output,
    WeaklyTypedInput: true,
  }

```

```
  decoder, err := NewDecoder(config)    if err != nil {
    return err
  }

  return decoder.Decode(input)
}
```

再实际上，Decode/DecodeMetadata/WeakDecodeMetadata内部都是先设置DecoderConfig的对应字段，然后创建Decoder对象，最后调用其Decode方法实现的。

总结

mapstructure实现优雅，功能丰富，代码结构清晰，非常推荐一看！