





**Go的json解析：Marshal与Unmarshal**

实现

Json Marshal：将数据编码成json字符串

看一个简单的例子

type Stu struct {

Name string `json:"name"`

Age int

HIgh bool

sex string

Class \*Class `json:"class"`

}

type Class struct {

Name string

Grade int

}

func main() {

//实例化一个数据结构，用于生成json字符串

stu := Stu{

Name: "张三",

Age: 18,

HIgh: true,

sex: "男",

}

//指针变量

cla := new(Class)

cla.Name = "1班"

cla.Grade = 3

stu.Class=cla

//Marshal失败时err!=nil

jsonStu, err := json.Marshal(stu)

if err != nil {

fmt.Println("生成json字符串错误")

}

//jsonStu是[]byte类型，转化成string类型便于查看

fmt.Println(string(jsonStu))

}

结果：

{"name":"张三","Age":18,"HIgh":true,"class":{"Name":"1班","Grade":3}}

1

从结果中可以看出

只要是可导出成员（变量首字母大写），都可以转成json。因成员变量sex是不可导出的，故无法转成json。

如果变量打上了json标签，如Name旁边的 `json:"name"` ，那么转化成的json key就用该标签“name”，否则取变量名作为key，如“Age”，“HIgh”。

bool类型也是可以直接转换为json的value值。Channel， complex 以及函数不能被编码json字符串。当然，循环的数据结构也不行，它会导致marshal陷入死循环。

指针变量，编码时自动转换为它所指向的值，如cla变量。

（当然，不传指针，Stu struct的成员Class如果换成Class struct类型，效果也是一模一样的。只不过指针更快，且能节省内存空间。）

最后，强调一句：json编码成字符串后就是纯粹的字符串了。

上面的成员变量都是已知的类型，只能接收指定的类型，比如string类型的Name只能赋值string类型的数据。

但有时为了通用性，或使代码简洁，我们希望有一种类型可以接受各种类型的数据，并进行json编码。这就用到了interface{}类型。

前言：

interface{}类型其实是个空接口，即没有方法的接口。go的每一种类型都实现了该接口。因此，任何其他类型的数据都可以赋值给interface{}类型。

type Stu struct {

Name interface{} `json:"name"`

Age interface{}

HIgh interface{}

sex interface{}

Class interface{} `json:"class"`

}

type Class struct {

Name string

Grade int

}

func main() {

//与前面的例子一样

......

}

结果：

{"name":"张三","Age":18,"HIgh":true,"class":{"Name":"1班","Grade":3}}

1

从结果中可以看出，无论是string，int，bool，还是指针类型等，都可赋值给interface{}类型，且正常编码，效果与前面的例子一样。

补充：

在实际项目中，编码成json串的数据结构，往往是切片类型。如下定义了一个[]StuRead类型的切片

//正确示范

//方式1：只声明，不分配内存

var stus1 []\*StuRead

//方式2：分配初始值为0的内存

stus2 := make([]\*StuRead,0)

//错误示范

//new()只能实例化一个struct对象，而[]StuRead是切片，不是对象

stus := new([]StuRead)

stu1 := StuRead{成员赋值...}

stu2 := StuRead{成员赋值...}

//由方式1和2创建的切片，都能成功追加数据

//方式2最好分配0长度，append时会自动增长。反之指定初始长度，长度不够时不会自动增长，导致数据丢失

stus1 := appen(stus1,stu1,stu2)

stus2 := appen(stus2,stu1,stu2)

//成功编码

json1,\_ := json.Marshal(stus1)

json2,\_ := json.Marshal(stus2)

解码时定义对应的切片接受即可

Json Unmarshal：将json字符串解码到相应的数据结构

我们将上面的例子进行解码

type StuRead struct {

Name interface{} `json:"name"`

Age interface{}

HIgh interface{}

sex interface{}

Class interface{} `json:"class"`

Test interface{}

}

type Class struct {

Name string

Grade int

}

func main() {

//json字符中的"引号，需用\进行转义，否则编译出错

//json字符串沿用上面的结果，但对key进行了大小的修改，并添加了sex数据

data:="{\"name\":\"张三\",\"Age\":18,\"high\":true,\"sex\":\"男\",\"CLASS\":{\"naME\":\"1班\",\"GradE\":3}}"

str:=[]byte(data)

//1.Unmarshal的第一个参数是json字符串，第二个参数是接受json解析的数据结构。

//第二个参数必须是指针，否则无法接收解析的数据，如stu仍为空对象StuRead{}

//2.可以直接stu:=new(StuRead),此时的stu自身就是指针

stu:=StuRead{}

err:=json.Unmarshal(str,&stu)

//解析失败会报错，如json字符串格式不对，缺"号，缺}等。

if err!=nil{

fmt.Println(err)

}

fmt.Println(stu)

}

结果：

{张三 18 true <nil> map[naME:1班 GradE:3] <nil>}

1

总结：

json字符串解析时，需要一个“接收体”接受解析后的数据，且Unmarshal时接收体必须传递指针。否则解析虽不报错，但数据无法赋值到接受体中。如这里用的是StuRead{}接收。

解析时，接收体可自行定义。json串中的key自动在接收体中寻找匹配的项进行赋值。匹配规则是：

先查找与key一样的json标签，找到则赋值给该标签对应的变量(如Name)。

没有json标签的，就从上往下依次查找变量名与key一样的变量，如Age。或者变量名忽略大小写后与key一样的变量。如HIgh，Class。第一个匹配的就赋值，后面就算有匹配的也忽略。

(前提是该变量必需是可导出的，即首字母大写)。

不可导出的变量无法被解析（如sex变量，虽然json串中有key为sex的k-v，解析后其值仍为nil,即空值）

当接收体中存在json串中匹配不了的项时，解析会自动忽略该项，该项仍保留原值。如变量Test，保留空值nil。

你一定会发现，变量Class貌似没有解析为我们期待样子。因为此时的Class是个interface{}类型的变量，而json串中key为CLASS的value是个复合结构，不是可以直接解析的简单类型数据（如“张三”，18，true等）。所以解析时，由于没有指定变量Class的具体类型，json自动将value为复合结构的数据解析为map[string]interface{}类型的项。也就是说，此时的struct Class对象与StuRead中的Class变量没有半毛钱关系，故与这次的json解析没有半毛钱关系。

让我们看一下这几个interface{}变量解析后的类型

func main() {

//与前边json解析的代码一致

...

fmt.Println(stu) //打印json解析前变量类型

err:=json.Unmarshal(str,&stu)

fmt.Println("--------------json 解析后-----------")

...

fmt.Println(stu) //打印json解析后变量类型

}

//利用反射，打印变量类型

func printType(stu \*StuRead){

nameType:=reflect.TypeOf(stu.Name)

ageType:=reflect.TypeOf(stu.Age)

highType:=reflect.TypeOf(stu.HIgh)

sexType:=reflect.TypeOf(stu.sex)

classType:=reflect.TypeOf(stu.Class)

testType:=reflect.TypeOf(stu.Test)

fmt.Println("nameType:",nameType)

fmt.Println("ageType:",ageType)

fmt.Println("highType:",highType)

fmt.Println("sexType:",sexType)

fmt.Println("classType:",classType)

fmt.Println("testType:",testType)

}

结果：

nameType: <nil>

ageType: <nil>

highType: <nil>

sexType: <nil>

classType: <nil>

testType: <nil>

--------------json 解析后-----------

nameType: string

ageType: float64

highType: bool

sexType: <nil>

classType: map[string]interface {}

testType: <nil>

从结果中可见

interface{}类型变量在json解析前，打印出的类型都为nil，就是没有具体类型，这是空接口（interface{}类型）的特点。

json解析后，json串中value，只要是”简单数据”，都会按照默认的类型赋值，如”张三”被赋值成string类型到Name变量中，数字18对应float64，true对应bool类型。

“简单数据”：是指不能再进行二次json解析的数据，如”name”:”张三”只能进行一次json解析。

“复合数据”：类似”CLASS\”:{\”naME\”:\”1班\”,\”GradE\”:3}这样的数据，是可进行二次甚至多次json解析的，因为它的value也是个可被解析的独立json。即第一次解析key为CLASS的value，第二次解析value中的key为naME和GradE的value

对于”复合数据”，如果接收体中配的项被声明为interface{}类型，go都会默认解析成map[string]interface{}类型。如果我们想直接解析到struct Class对象中，可以将接受体对应的项定义为该struct类型。如下所示：

type StuRead struct {

...

//普通struct类型

Class Class `json:"class"`

//指针类型

Class \*Class `json:"class"`

}

stu打印结果

Class类型：{张三 18 true <nil> {1班 3} <nil>}

\*Class类型：{张三 18 true <nil> 0xc42008a0c0 <nil>}

可以看出，传递Class类型的指针时，stu中的Class变量存的是指针，我们可通过该指针直接访问所属的数据，如stu.Class.Name/stu.Class.Grade

Class变量解析后类型

classType: main.Class

classType: \*main.Class

解析时，如果接受体中同时存在2个匹配的项，会发生什么呢？

测试1

type StuRead struct {

NAme interface{}

Name interface{}

NAMe interface{} `json:"name"`

}

结果1:

//当存在匹配的json标签时，其对应的项被赋值。

//切记：匹配的标签可以没有，但有时最好只有一个哦

{<nil> <nil> 张三}

测试2

type StuRead struct {

NAme interface{}

Name interface{}

NAMe interface{} `json:"name"`

NamE interface{} `json:"name"`

}

结果2

//当匹配的json标签有多个时，标签对应的项都不会被赋值。

//忽略标签项，从上往下寻找第一个没有标签且匹配的项赋值

{张三 <nil> <nil> <nil>}

测试3

type StuRead struct {

NAme interface{}

Name interface{}

}

结果3

//没有json标签时，从上往下，第一个匹配的项会被赋值哦

{张三 <nil>}

测试4

type StuRead struct {

NAMe interface{} `json:"name"`

NamE interface{} `json:"name"`

}

结果4

//当相同的json标签有多个，且没有不带标签的匹配项时，报错了哦

# command-line-arguments

src/test/b.go:48: stu.Name undefined (type \*StuRead has no field or method Name, but does have NAMe)

可见，与前边说过的匹配规则是一致的。

如果不想指定Class变量为具体的类型，仍想保留interface{}类型，但又希望该变量可以解析到struct Class对象中，这时候该怎么办呢？

这种需求是很可能存在的，例如笔者我就碰到了

办法还是有的，我们可以将该变量定义为json.RawMessage类型

type StuRead struct {

Name interface{}

Age interface{}

HIgh interface{}

Class json.RawMessage `json:"class"` //注意这里

}

type Class struct {

Name string

Grade int

}

func main() {

data:="{\"name\":\"张三\",\"Age\":18,\"high\":true,\"sex\":\"男\",\"CLASS\":{\"naME\":\"1班\",\"GradE\":3}}"

str:=[]byte(data)

stu:=StuRead{}

\_:=json.Unmarshal(str,&stu)

//注意这里：二次解析！

cla:=new(Class)

json.Unmarshal(stu.Class,cla)

fmt.Println("stu:",stu)

fmt.Println("string(stu.Class):",string(stu.Class))

fmt.Println("class:",cla)

printType(&stu) //函数实现前面例子有

}

结果

stu: {张三 18 true [123 34 110 97 77 69 34 58 34 49 231 143 173 34 44 34 71 114 97 100 69 34 58 51 125]}

string(stu.Class): {"naME":"1班","GradE":3}

class: &{1班 3}

nameType: string

ageType: float64

highType: bool

classType: json.RawMessage

从结果中可见

接收体中，被声明为json.RawMessage类型的变量在json解析时，变量值仍保留json的原值，即未被自动解析为map[string]interface{}类型。如变量Class解析后的值为：{“naME”:”1班”,”GradE”:3}

从打印的类型也可以看出，在第一次json解析时，变量Class的类型是json.RawMessage。此时，我们可以对该变量进行二次json解析，因为其值仍是个独立且可解析的完整json串。我们只需再定义一个新的接受体即可，如json.Unmarshal(stu.Class,cla)