Go语言 标准库 encoding/json&encoding/xml包

encoding/json

JSON(JavaScript Object Notation, JS 对象标记) 是一种轻量级的数据交换格式。它基于ECMAScript规范的一个子集，采用完全独立于编程语言的文本格式来存储和表示数据。简洁和清晰的层次结构使得 JSON 成为理想的数据交换语言。 易于人阅读和编写，同时也易于机器解析和生成，并有效地提升网络传输效率。

encoding/json 是Go语言自带的JSON转换库，这个包可以实现json的编码和解码，就是将json字符串转换为struct，或者将struct转换为json。

基本使用

`encoding/json` 包中最常用的是 `Marshal()` 和 `Unmarshal()` 函数：

Marshal

将struct编码成json，可以接收任意类型

```
func Marshal(v interface{}) ([]byte, error)
```

一般来说，`Marshal()` 函数会使用以下的基于类型的默认编码格式：

- 布尔类型编码为 `json` 布尔类型；
- 浮点数、整数和 `json.Number` 类型编码为 `json` 数字类型；
- 字符串类型编码为 `json` 字符串；
- 数组和切片类型编码为 `json` 数组，但 `[]byte` 编码为 `base64` 编码字符串，`nil` 切片编码为 `null`；
- 结构体类型编码为 `json` 对象，每一个可导出字段（首字母大写）会变成该对象的一个成员。

示例：

```
package main

import (
    "encoding/json"
    "fmt"
)

type Person struct {
    Name  string
    Age   int
    Email string
}

func Marshal() {
    p := Person{
        Name:  "包子",
```

```
        Age:   20,
        Email: "baozi@gmail.com",
    }
    b, _ := json.Marshal(p)
    fmt.Printf("b: %v\n", string(b))
}

func main() {
    Marshal()
}
#结果
b: {"Name":"包子","Age":20,"Email":"baozi@gmail.com"}
```

Unmarshal

将json转码为struct结构体

```
func Unmarshal(data []byte, v interface{}) error
```

示例:

```
package main

import (
    "encoding/json"
    "fmt"
)

type Person struct {
    Name  string
    Age   int
    Email string
}

func Unmarshal() {
    // 常规解析
    b1 := []byte(`{"Name":"包子","Age":20,"Email":"baozi@gmail.com"}`)
    var m Person
    json.Unmarshal(b1, &m)
    fmt.Printf("m: %v\n", m)

    // 解析嵌套类型
    b := []byte(`{"Name":"包子","Age":20,"Email":"baozi@gmail.com","Pets":["白
胖","橘胖"]}`)
    var f interface{}
    json.Unmarshal(b, &f)
    fmt.Printf("f: %v\n", f)

    // 解析嵌套引用类型
    type Person struct {
        Name  string
        Age   int
        Email string
        Pets  []string
```

```
    }
    p := Person{
        Name:  "包子",
        Age:   20,
        Email: "baozi@gmail.com",
        Pets:  []string{"白胖", "橘胖"},
    }
    d, _ := json.Marshal(p)
    fmt.Printf("d: %v\n", string(d))
    var g interface{}
    json.Unmarshal(d, &g)
    fmt.Printf("g: %v\n", g)
}

func main() {
    Unmarshal()
}
#结果
m: {包子 20 baozi@gmail.com}
f: map[Age:20 Email:baozi@gmail.com Name:包子 Pets:[白胖 橘胖]]
d: {"Name":"包子","Age":20,"Email":"baozi@gmail.com","Pets":["白胖","橘胖"]}
g: map[Age:20 Email:baozi@gmail.com Name:包子 Pets:[白胖 橘胖]]
```

常用序列化/反序列化操作

指定字段名

```
// tag 是结构体的元信息，可以在运行的时候通过反射的机制读取出来
// 在 tag 中添加字段名，json 序列化/反序列化时会使用该字段名
type Person struct {
    Name   string `json:"name"` // 指定 json 序列化/反序列化时使用小写 name
    Age    int64  `json:"age"`  // 指定 json 序列化/反序列化时使用小写 age
    Email  string
}
```

忽略某个字段

```
// 在 tag 中添加 "-"，json 序列化/反序列化时会忽略该字段
type Person struct {
    Name   string  `json:"name"` // 指定 json 序列化/反序列化时使用小写 name
    Age    int64   `json:"age"`  // 指定 json 序列化/反序列化时使用小写 age
    Email  string  `json:"-"`    // 指定 json 序列化/反序列化时忽略此字段
}
```

忽略空值字段

```
type User struct {
    Name  string   `json:"name"`
    Email string   `json:"email"`
    Pets  []string `json:"pets"`
}

u1 := User{
    Name: "包子",
}
```

这时候我们序列化是不会忽略空字段的

```
// 空值字段不会被忽略
{"name":"包子","email":"","pets":null}
```

忽略空值字段使用 omitempty

```
// 在 tag 中添加 omitempty 会忽略空值
// 空值指的是 false、0、""、nil 指针、nil 接口、长度为 0 的数组、切片、映射
type User struct {
    Name  string   `json:"name"`
    Email string   `json:"email,omitempty"`
    Pets  []string `json:"pets,omitempty"`
}

u1 := User{
    Name: "包子",
}
// 添加 omitempty 后，空值字段会被忽略
{"name":"包子"}
```

忽略嵌套结构体空值字段

嵌套结构体

```
type User struct {
    Name  string   `json:"name"`
    Email string   `json:"email,omitempty"`
    Pets  []string `json:"pets,omitempty"`
    Phone
}
type Phone struct {
    Brand  string `json:"brand"`
    Model  string `json:"model"`
}

u1 := User{
    Name:  "包子",
    Pets: []string{"白胖", "橘胖"},
}
// 需要注意匿名嵌套 Phone 时即Phone没有tag，序列化后的 json 串为单层的。
{"name":"包子","pets":["白胖", "橘胖"],"brand":"","model":""}
```

如果想要变成嵌套的`json`串，需要改为具名嵌套或定义字段`tag`

```
type User struct {
    Name  string   `json:"name"`
    Email string   `json:"email,omitempty"`
    Pets  []string `json:"pets,omitempty"`
    Phone `json:"phone"` // 定义tag
}
type Phone struct {
    Brand  string `json:"brand"`
    Model  string `json:"model"`
}

u1 := User{
    Name:  "包子",
    Pets: []string{"白胖", "橘胖"},
}
// 定义字段 tag 后，序列化后的 json 串为双层的
{"name":"包子","pets":["白胖","橘胖"],"phone":{"brand":"","model":""}}
```

如果想要忽略嵌套结构体空值字段，仅添加`omitempty`是不够的，需要使用嵌套的结构体指针

```
type User struct {
    Name  string   `json:"name"`
    Email string   `json:"email,omitempty"`
    Pets  []string `json:"pets,omitempty"`
    *Phone `json:"phone,omitempty"` // 定义tag，添加omitempty，设置结构体指针*
}
type Phone struct {
    Brand  string `json:"brand"`
    Model  string `json:"model"`
}

u1 := User{
    Name:  "包子",
    Pets: []string{"白胖", "橘胖"},
}
#结果
{"name":"包子","pets":["白胖", "橘胖"]}
```

不修改原结构体忽略空值字段

```
// 如果不需要将 User 结构体的 Password 字段序列化，但是又不想修改 User 结构体，
// 可以创建另外一个结构体匿名嵌套原 User，同时指定 Password 字段为匿名结构体指针类型，并添加
omitempty 标签
package main

import (
    "encoding/json"
    "fmt"
)

type User struct {
    Name     string `json:"name"`
```

```
    Password string `json:"password"`
}

type NewUser struct {
    *User              // 匿名嵌套
    Password *struct{} `json:"password,omitempty"`
}

func main() {
    u := User{
        Name:     "包子",
        Password: "123456",
    }
    b, err := json.Marshal(NewUser{User: &u})
    if err != nil {
        fmt.Printf("json.Marshal u1 failed, err:%v\n", err)
        return
    }
    fmt.Printf("%s\n", b)
}
#结果
{"name":"包子"}
```

使用匿名结构体添加字段

```
// 如果想扩展结构体字段，但有时候又没有必要单独定义新的结构体，可以使用匿名结构体简化操作
package main

import (
    "encoding/json"
    "fmt"
)

type User struct {
    Name     string `json:"name"`
    Password string `json:"password"`
}

func main() {
    u := User{
        Name:     "包子",
        Password: "123456",
    }
    // 使用匿名结构体内嵌 User 并添加额外字段Token
    b, err := json.Marshal(struct {
        *User
        Token string `json:"token"`
    }{
        User:  &u,
        Token: "91je3a4s72d1da96h",
    })
    if err != nil {
        fmt.Printf("json.Marshal u1 failed, err:%v\n", err)
        return
    }
```

```
    fmt.Printf("%s\n", b)
}
#结果
{"name":"包子","password":"123456","token":"91je3a4s72d1da96h"}
```

使用匿名结构体组合多个结构体

```
// 同理，可以使用匿名结构体来组合多个结构体来序列化与反序列化数据
package main

import (
    "encoding/json"
    "fmt"
)

type User struct {
    Name     string `json:"name"`
    Password string `json:"password"`
}

type Post struct {
    ID    int    `json:"id"`
    Title string `json:"title"`
}

func main() {
    u := User{
        Name:     "包子",
        Password: "123456",
    }

    p := Post{
        ID:    123456,
        Title: "hello world",
    }

    b, err := json.Marshal(struct {
        *User
        *Post
    }{
        User: &u,
        Post: &p,
    })
    if err != nil {
        fmt.Printf("json.Marshal u1 failed, err:%v\n", err)
        return
    }
    fmt.Printf("%s\n", b) // {"name":"包
子","password":"123456","id":123456,"title":"hello world"}

    jsonStr := `{"name":"包子","password":"123456","id":123456,"title":"Hello
World"}`
    var (
        u2 User
        p2 Post
```

```
    )
    if err := json.Unmarshal([]byte(jsonStr), &struct {
        *User
        *Post
    }{&u2, &p2}); err != nil {
        fmt.Printf("json.Unmarshal failed, err:%v\n", err)
        return
    }
    fmt.Printf("%#v\n", u2) // main.User{Name:"包子", Password:"123456"}
    fmt.Printf("%#v\n", p2) // main.Post{ID:123456, Title:"Hello World"}
}
```

处理字符串格式数字

```
// 如果 json 串使用的是字符串格式的数字，可以在 tag 中添加 string 来告诉 json 包反序列化时从字符串解析相应字段
package main

import (
    "encoding/json"
    "fmt"
)

type Card struct {
    ID    int64   `json:"id,string"`    // 添加 string tag
    Score float64 `json:"score,string"` // 添加 string tag
}

func main() {
    // json 串中使用的是字符串格式的数字，不添加 string tag 反序列会报错
    jsonStr := `{"id": "1234567","score": "88.50"}`
    var c1 Card
    if err := json.Unmarshal([]byte(jsonStr), &c1); err != nil {
        fmt.Printf("json.Unmarsha jsonStr1 failed, err:%v\n", err)
        return
    }
    fmt.Printf("%#v\n", c1) // main.Card{ID:1234567, Score:88.5}
}
```

整数变为浮点数

```
// JSON 协议中没有整型和浮点型的区别，它们统称为 number
// 如果将 JSON 格式的数据反序列化为 map[string]interface{} 时，数字都变成科学计数法表示的浮点数
package main

import (
    "encoding/json"
    "fmt"
)

type student struct {
    ID   int64  `json:"id"`
    Name string `json:"q1mi"`
```

```
}

func main() {
    s := student{
        ID:   123456789,
        Name: "包子",
    }
    b, _ := json.Marshal(s)

    var m map[string]interface{}
    _ = json.Unmarshal(b, &m)
    fmt.Printf("%#v\n", m["id"]) // 1.23456789e+08
    fmt.Printf("%T\n", m["id"])  // float64
}
```

如果想更合理的处理数字，需要使用`decoder`去反序列化，使用`json.Number`类型，示例代码如下：

```
package main

import (
    "bytes"
    "encoding/json"
    "fmt"
)

type student struct {
    ID   int64  `json:"id"`
    Name string `json:"q1mi"`
}

func main() {
    s := student{
        ID:   123456789,
        Name: "包子",
    }
    b, _ := json.Marshal(s)

    // 使用功能 decoder 方式进行反序列，指定使用 number 类型
    var m map[string]interface{}
    decoder := json.NewDecoder(bytes.NewReader(b))
    decoder.UseNumber()
    _ = decoder.Decode(&m)

    // 反序列后，类型为 json.Number 类型
    fmt.Printf("%#v\n", m["id"]) // "123456789"
    fmt.Printf("%T\n", m["id"])  // json.Number

    // 根据字段的实际类型调用 Float64() 或 Int64() 函数获取对应类型
    id, _ := m["id"].(json.Number).Int64()
    fmt.Printf("%#v\n", id) // 123456789
    fmt.Printf("%T\n", id)  // int64
}
```

处理不确定层级的 json

```
// 如果 json 串没有固定格式导致不好定义与其相对应的结构体时，可以使用 json.RawMessage 将原始字节数据保存下来
package main

import (
    "encoding/json"
    "fmt"
)

type sendMsg struct {
    User string `json:"user"`
    Msg  string `json:"msg"`
}

func main() {
    jsonStr := `{"sendMsg":{"user":"包子","msg":"永远不要高估自己"},"say":"Hello"}`
    var data map[string]json.RawMessage
    if err := json.Unmarshal([]byte(jsonStr), &data); err != nil {
        fmt.Printf("json.Unmarshal jsonStr failed, err:%v\n", err)
        return
    }

    var msg sendMsg
    if err := json.Unmarshal(data["sendMsg"], &msg); err != nil {
        fmt.Printf("json.Unmarshal failed, err:%v\n", err)
        return
    }
    fmt.Printf("%#v\n", msg) // main.sendMsg{User:"包子", Msg:"永远不要高估自己"}
}
```

自定义序列化/反序列化

```
// 可以通过实现 json.Marshaler/json.Unmarshaler 接口来实现自定义的事件格式解析
type Marshaler interface {
    MarshalJSON() ([]byte, error)
}

type Unmarshaler interface {
    UnmarshalJSON([]byte) error
}
```

示例：

```
// 内置的 json 包不识别常用的字符串时间格式，如 2022-05-24 14:50:00
package main

import (
    "encoding/json"
    "fmt"
    "time"
)
```

```
type Post struct {
    ID         int       `json:"id"`
    Title      string    `json:"title"`
    CreateTime time.Time `json:"create_time"`
}

func main() {
    // 序列化
    p := Post{
        ID:         123456,
        Title:      "hello world",
        CreateTime: time.Now(),
    }
    b, err := json.Marshal(p)
    if err != nil {
        fmt.Printf("json.Marshal p1 failed, err:%v\n", err)
        return
    }
    fmt.Printf("%s\n", b) // {"id":123456,"title":"hello
world","create_time":"2022-05-24T14:58:45.3702937+08:00"}

    // 反序列化
    jsonStr := `{"id":123456,"title":"hello world","create_time":"2022-05-24
14:50:00"}`
    var p2 Post
    if err := json.Unmarshal([]byte(jsonStr), &p2); err != nil {
        fmt.Printf("json.Unmarshal failed, err:%v\n", err)
        return
    }
    fmt.Printf("%#v\n", p2) // json.Unmarshal failed, err:parsing time "\"2022-
05-24 14:50:00\"" as "\"2006-01-02T15:04:05Z07:00\"": cannot parse " 14:50:00\""
as "T"
}
```

```
// 通过实现 json.Marshaler/json.Unmarshaler 接口来自定义时间字段的事件格式解析
package main

import (
    "encoding/json"
    "fmt"
    "time"
)

const layout = "2006-01-02 15:04:05"

type Post struct {
    ID         int       `json:"id"`
    Title      string    `json:"title"`
    CreateTime time.Time `json:"create_time"`
}

// MarshalJSON 为 Post 类型自定义序列化方法
func (p Post) MarshalJSON() ([]byte, error) {
    type TempPost Post // 定义与 Post 字段一致的新类型
```

```
    return json.Marshal(struct {
        *TempPost         // 直接嵌套 Post 会进入死循环，需要使用新类型 TempPost
        CreateTime string `json:"create_time"`
    }{
        TempPost:   (*TempPost)(&p),
        CreateTime: p.CreateTime.Format(layout),
    })
}

// UnmarshalJSON 为 Post 类型自定义反序列化方法
func (p *Post) UnmarshalJSON(data []byte) error {
    type TempPost Post // 定义与 Post 字段一致的新类型
    tp := struct {
        *TempPost        // 直接嵌套 Post 会进入死循环，需要使用新类型 TempPost
        CreateTime string `json:"create_time"`
    }{
        TempPost: (*TempPost)(p),
    }

    if err := json.Unmarshal(data, &tp); err != nil {
        return err
    }

    var err error
    p.CreateTime, err = time.Parse(layout, tp.CreateTime)
    if err != nil {
        return err
    }
    return nil
}

func main() {
    // 序列化
    p := Post{
        ID:         123456,
        Title:      "hello world",
        CreateTime: time.Now(),
    }
    b, err := json.Marshal(p)
    if err != nil {
        fmt.Printf("json.Marshal p1 failed, err:%v\n", err)
        return
    }
    fmt.Printf("%s\n", b) // {"id":123456,"title":"hello
world","create_time":"2022-05-24 18:30:03"}

    // 反序列化
    jsonStr := `{"id":123456,"title":"hello world","create_time":"2022-05-24
14:50:00"}`
    var p2 Post
    if err := json.Unmarshal([]byte(jsonStr), &p2); err != nil {
        fmt.Printf("json.Unmarshal failed, err:%v\n", err)
        return
    }
```

```
    fmt.Printf("%#v\n", p2) // main.Post{ID:123456, Title:"hello world",
CreateTime:time.Date(2022, time.May, 24, 14, 50, 0, 0, time.UTC)}
}
```

encoding/xml

基本使用

`encoding/xml`包中最常用的是`Marshal()`和`Unmarshal()`函数：

Marshal

将struct编码成xml，可以接收任意类型

```
func Marshal(v interface{}) ([]byte, error)
```

示例：

```
package main

import (
    "encoding/xml"
    "fmt"
)

type Person struct {
    XMLName xml.Name `xml:"person"`
    Name    string   `xml:"name"`
    Age     int      `xml:"age"`
    Email   string   `xml:"email"`
}

func Marshal() {
    p := Person{
        Name:  "包子",
        Age:   20,
        Email: "baozi@gmail.com",
    }
    // b, _ := xml.Marshal(p)
    // 有缩进格式
    b, _ := xml.MarshalIndent(p, " ", "  ")
    fmt.Printf("%v\n", string(b))
}

func main() {
    Marshal()
}
#结果
 <person>
   <name>包子</name>
   <age>20</age>
   <email>baozi@gmail.com</email>
 </person>
```

Unmarshal

将xml转码为struct结构体

```
func Unmarshal(data []byte, v interface{}) error
```

示例：

```
package main

import (
    "encoding/xml"
    "fmt"
)

type Person struct {
    XMLName xml.Name `xml:"person"`
    Name    string   `xml:"name"`
    Age     int      `xml:"age"`
    Email   string   `xml:"email"`
}

func Unmarshal() {
    p := Person{
        Name:  "包子",
        Age:   20,
        Email: "baozi@gmail.com",
    }
    // b, _ := xml.Marshal(p)
    // 有缩进格式
    b, _ := xml.MarshalIndent(p, " ", "  ")
    var u Person
    xml.Unmarshal(b, &u)
    fmt.Printf("u: %v\n", u)
}

func main() {
    Unmarshal()
}
#结果
u: {{ person} 包子 20 baozi@gmail.com}
```