第十六章 JSON处理

本章重点为大家介绍如下的内容：

* 什么是JSON
* JSON语法规则
* JSON的优点和缺点
* 对象转换为JSON
* JSON转换为对象
* JSON转换为map
* JSON流解析

16.1 JSON简介

JSON是一种轻量级的数据交换格式(JavaScript Object Notation)，因其简单、可读性强广泛使用。

JSON语法规则

对象是一个无序的“‘名称/值’对”集合。一个对象以“{”（左括号）开始，“}”（右括号）结束。每个“名称”后跟一个“:”（冒号）；“‘名称/值’ 对”之间使用“,”（逗号）分隔。

数组是值（value）的有序集合。一个数组以“[”（左中括号）开始，“]”（右中括号）结束。值之间使用“,”（逗号）分隔。

值（value）可以是双引号括起来的字符串（string）、数值(number)、true、false、 null、对象（object）或者数组（array）。这些结构可以嵌套。

字符串（string）是由双引号包围的任意数量Unicode字符的集合，使用反斜线转义。一个字符（character）即一个单独的字符串（character string）。

简单的json：

{

name: "张三",

age: 23

}

复杂的json：

{

"employee": {

"firstName": "张",

"lastName": "三",

"employeeNumber": 101,

"age": 23

}

}

JSON的优点:

数据格式比较简单, 易于读写, 格式都是压缩的, 占用带宽小

因为JSON格式能够直接为服务器端代码使用, 大大简化了服务器端和客户端的代码开发量, 但是完成的任务不变, 且易于维护

16.2 对象转换为JSON

Go的标准包encoding/json对JSON的编解码提供了完整的支持。

将Go数据类型转换为JSON。用到的函数：

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

该函数递归遍历v的结构，生成对应的JSON。

字典转JSON

package main

import (

"encoding/json"

"fmt"

)

func main() {

m := map[string][]string{

"level": {"debug"},

"message": {"File not found", "Stack overflow"},

}

if data, err := json.Marshal(m); err == nil {

fmt.Printf("%s\n", data)

}

}

输出结果：

{"level":["debug"],"message":["File not found","Stack overflow"]}

大家可以看到Marshal函数返回的JSON字符串是没有空白字符和缩进的，这种紧凑的表示形式是最常用的传输形式，但是不好阅读。如果需要为前端生成便于阅读的格式，可以调json.MarshaIndent，该函数有两个参数表示每一行的前缀和缩进方式：

if data, err := json.MarshalIndent(m, "", " "); err == nil {

fmt.Printf("%s\n", data)

}

输出结果：

{

"level": [

"debug"

],

"message": [

"File not found",

"Stack overflow"

]

}

Go数据类型转换为JSON规则

在编码过程中，json包会将Go的类型转换为JSON类型，转换规则如下：

bool -> JSON boolean

浮点数, 整数, Number -> JSON number

string -> JSON string

数组、切片 -> JSON数组

[]byte -> base64 string

struct、map -> JSON object

结构体转JSON

经常会使用结构体来转换成JSON。json包是通过反射机制来实现编解码的，因此结构体必须导出所转换的字段，不导出的字段不会被json包解析：

package main

import (

"encoding/json"

"fmt"

)

type DebugInfo struct {

Level string

Msg string

author string // 未导出字段不会被json解析(首字母小写)

}

func main() {

dbgInfs := []DebugInfo{

DebugInfo{"debug", `File: "test.txt" Not Found`, "Cynhard"},

DebugInfo{"", "Logic error", "Gopher"},

}

if data, err := json.Marshal(dbgInfs); err == nil {

fmt.Printf("%s\n", data)

}

}输出结果如下：

[{"Level":"debug","Msg":"File: \"test.txt\" Not Found"},{"Level":"","Msg":"Logic error"}]

结构体字段标签：

json包在解析结构体时，如果遇到key为“json”的字段标签，则会按照一定规则解析该标签：第一个出现的是字段在JSON串中使用的名字，之后为其他选项，例如“omitempty”指定空值字段不出现在JSON中。如果整个value为'"-"'，则不解析该字段。例如将上例中的结构体改为如下：

type DebugInfo struct {

Level string `json:"level,omitempty"` // Level解析为level,忽略空值

Msg string `json:"message" ` // Msg解析为message

Author string `json:"-"` // 忽略Author

}

则输出为：

[{"level":"debug","message":"File: \"test.txt\" Not Found"},{"message":"Logic error"}]

结构体中匿名字段：

json包在解析匿名字段时，会将匿名字段的字段当成该结构体的字段处理：

package main

import (

"encoding/json"

"fmt"

)

type Point struct {

X, Y int

}

type Circle struct {

Point

Radius int

}

func main() {

if data, err := json.Marshal(Circle{Point{50, 50}, 25}); err == nil {

fmt.Printf("%s\n", data)

}

}

输出结果：

{"X":50,"Y":50,"Radius":25}

注意：

Marshal函数只有在转换成功的时候才会返回数据，在转换的过程中我们需要注意几点：

JSON对象只支持string作为key，所以要编码一个map，那么必须是map[string]T这种类型(T是Go语言中任意的类型)

Channel, complex和function是不能被编码成JSON的

指针在编码的时候会输出指针指向的内容，而空指针会输出null

16.3 JSON转换为对象

将JSON转换为Go数据类型。用到的函数：

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

JSON转数组

package main

import (

"encoding/json"

"fmt"

)

func main() {

data := `[{"Level":"debug","Msg":"File: \"test.txt\" Not Found"},{"Level":"","Msg":"Logic error"}]`

var dbgInfos []map[string]string

json.Unmarshal([]byte(data), &dbgInfos)

fmt.Println(dbgInfos)

}

输出为：

[map[Level:debug Msg:File: "test.txt" Not Found] map[Level: Msg:Logic error]]

在解码过程中，json包会将JSON类型转换为Go类型，转换规则如下：

JSON boolean -> bool

JSON number -> float64

JSON string -> string

JSON数组 -> []interface{}

JSON object -> map

null -> nil

JSON转结构体

JSON可以转换成结构体。同编码一样，json包是通过反射机制来实现解码的，因此结构体必须导出所转换的字段，不导出的字段不会被json包解析，另外解析时不区分大小写：

package main

import (

"encoding/json"

"fmt"

)

type DebugInfo struct {

Level string

Msg string

author string // 未导出字段不会被json解析

}

func (dbgInfo DebugInfo) String() string {

return fmt.Sprintf("{Level: %s, Msg: %s}", dbgInfo.Level, dbgInfo.Msg)

}

func main() {

data := `[{"level":"debug","msg":"File Not Found","author":"Cynhard"},{"level":"","msg":"Logic error","author":"Gopher"}]`

var dbgInfos []DebugInfo

json.Unmarshal([]byte(data), &dbgInfos)

fmt.Println(dbgInfos)

}

输出结果如下：

[{Level: debug, Msg: File Not Found} {Level: , Msg: Logic error}]

结构体字段标签

解码时依然支持结构体字段标签，规则和编码时一样：

package main

import (

"encoding/json"

"fmt"

)

type DebugInfo struct {

Level string `json:"level"` // level 解码为 Level

Msg string `json:"message"` // message 解码为 Msg

Author string `json:"-"` // 忽略Author

}

func (dbgInfo DebugInfo) String() string {

return fmt.Sprintf("{Level: %s, Msg: %s}", dbgInfo.Level, dbgInfo.Msg)

}

func main() {

data := `[{"level":"debug","message":"File Not Found","author":"Cynhard"},{"level":"","message":"Logic error","author":"Gopher"}]`

var dbgInfos []DebugInfo

json.Unmarshal([]byte(data), &dbgInfos)

fmt.Println(dbgInfos)

}

则结果为：

[{Level: debug, Msg: File Not Found} {Level: , Msg: Logic error}]

匿名字段

编码时，和解码类似，在解码JSON时，如果找不到字段，则查找字段的字段：

package main

import (

"encoding/json"

"fmt"

)

type Point struct{ X, Y int }

type Circle struct {

Point

Radius int

}

func main() {

data := `{"X":80,"Y":80,"Radius":40}`

var c Circle

json.Unmarshal([]byte(data), &c)

fmt.Println(c)

}

输出结果：

{{80 80} 40}

16.4 JSON转换为map

package main

mport (

"encoding/json"

"fmt"

)

func main() {

map2json2map()

}

func map2json2map() {

map1 := make(map[string]interface{})

map1["1"] = "hello"

map1["2"] = "world"

//return []byte

str, err := json.Marshal(map1)

if err != nil {

fmt.Println(err)

}

fmt.Println("map to json", string(str))

//json([]byte) to map

map2 := make(map[string]interface{})

err = json.Unmarshal(str, &map2)

if err != nil {

fmt.Println(err)

}

fmt.Println("json to map ", map2)

fmt.Println("The value of key1 is", map2["1"])

}

运行结果：

map to json {"1":"hello","2":"world"}

json to map map[1:hello 2:world]

The value of key1 is hello

16.5 JSON流解析

json 包提供了Decoder和Encoder用来支持JSON 数据流的读写。函数NewDecoder 和 NewEncoder封装了io.Reader和io.Writer 接口类型。

func NewDecoder(r io.Reader) \*Decoder

func NewEncoder(w io.Writer) \*Encoder

由于读写操作的普遍性，类型Encoder和Decoder可以用于多种场合，例如读写HTTP 链接，WebSockets或者文件。

func main() {

data := `{"name":"go语言学习", "Count":34, "student":["张三", "里斯"], "address":{"floor":2, "classroom":"201"}}`

//流式编码

dec := json.NewDecoder(bytes.NewBufferString(data))

var v map[string]interface{}

if err := dec.Decode(&v); err != nil {

log.Println(err)

return

}

for k := range v {

if k == "name" {

delete(v, k)

}

}

fmt.Printf("%T %v\n", v, v)

//流式解码

buffer := bytes.Buffer{}

enc := json.NewEncoder(&buffer)

if err := enc.Encode(&v); err != nil {

log.Println(err)

}

fmt.Printf("%v", buffer.String())

}

map[string]interface {} map[Count:34 address:map[classroom:201 floor:2] student:[张三 里斯]]

{"Count":34,"address":{"classroom":"201","floor":2},"student":["张三","里斯"]}