练识课堂 -- Go语言资深工程师培训课程

第一章 JSON操作

1 JSON介绍

> JSON （JavaScript Object Notation）是一种比XML更轻量级的数据交换格式，在易于人们阅读和编写的同时，也易于程序解析和生成。尽管JSON是JavaScript的一个子集，但JSON采用完全独立于编程语言的文本格式，且表现为键/值对集合的文本描述形式（类似一些编程语言中的字典结构），这使它成为较为理想的、跨平台、跨语言的数据交换语言。

```
{
 "Company":"Lianshi",
 "Subjects":[
     "Go",
     "Web",
     "Python",
     "C++"
 ],
 "IsOk":true,
 "Price":8980.88
}
```

> 开发者可以用 JSON 传输简单的字符串、数字、布尔值，也可以传输一个数组，或者一个更复杂的复合结构。在 Web 开发领域中， JSON被广泛应用于 Web 服务端程序和客户端之间的数据通信。

2 JSON编码

> Go语言内建对JSON的支持。使用Go语言内置的encoding/json 标准库，开发者可以轻松使用Go程序生成和解析JSON格式的数据。
>
> 使用json.Marshal()函数可以对一组数据进行JSON格式的编码。
>
> json.Marshal()函数的声明如下：

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

> 格式化输出：

```
// MarshalIndent 很像 Marshal，只是用缩进对输出进行格式化
func MarshalIndent(v interface{}, prefix, indent string) ([]byte, error)
```

2、1 通过结构体生成JSON

示例代码:

```
package main

import (
  "encoding/json"
  "fmt"
)

type IT struct {
  Company  string
  Subjects []string
  IsOk     bool
  Price    float64
}

func main() {
  t1 := IT{"Lianshi", []string{"Go", "Web", "Python", "C++"}, true,
8980.88}

  //生成一段JSON格式的文本
  //如果编码成功， err 将赋于零值 nil，变量b 将会是一个进行JSON格式化之后的[]byte类
型
  //b, err := json.Marshal(t1)

  b, err := json.MarshalIndent(t1, "", "    ")

  if err != nil {
    fmt.Println("json err:", err)
  }
  fmt.Println(string(b))
}
```

2、2 通过map生成JSON

示例代码:

```
// 创建一个保存键值对的映射
t1 := make(map[string]interface{})
t1["company"] = "Lianshi"
t1["subjects "] = []string{"Go", "Web", "Python", "C++"}
t1["isok"] = true
t1["price"] = 8980.88

b, err := json.Marshal(t1)
//json.MarshalIndent(t1, "", "    ")
if err != nil {
 fmt.Println("json err:", err)
}
fmt.Println(string(b))
```

3 JSON解码

> 可以使用json.Unmarshal()函数将JSON格式的文本解码为Go里面预期的数据结构。
>
> json.Unmarshal()函数的原型如下：
>
> ```
> func Unmarshal(data []byte, v interface{}) error
> ```
>
> 该函数的第一个参数是输入，即JSON格式的文本（比特序列），第二个参数表示目标输出容器，用于存放解码后的值。

3、1 解析JSON到结构体

> 示例代码：
>
> ```
> package main
>
> import (
>   "encoding/json"
>   "fmt"
> )
>
> type IT struct {
>   Company  string   `json:"company"`
>   Subjects []string `json:"subjects"`
>   IsOk     bool     `json:"isok"`
>   Price    float64  `json:"price"`
> }
>
> func main() {
>   b := []byte(`{
>  "company": "Lianshi",
>  "subjects": [
>      "Go",
>      "Web",
> ```

```
    "Python",
    "C++"
 ],
 "isok": true,
 "price": 8980.88
}`)

  var t IT
  err := json.Unmarshal(b, &t)
  if err != nil {
    fmt.Println("json err:", err)
  }
  fmt.Println(t)
}
```

3、2 解析JSON到map

示例代码：

```
package main

import (
  "encoding/json"
  "fmt"
)


func main() {
  b := []byte(`{
 "company": "Lianshi",
 "subjects": [
    "Go",
    "Web",
    "Python",
    "C++"
 ],
 "isok": true,
 "price": 8980.88
}`)
  //默认返回值类型为interface类型 以map类型进行格式存储
 //可以理解为：json的key为map的key json的value为map的value
 //格式：map[string]interface{}
  var t interface{}
  err := json.Unmarshal(b, &t)
  if err != nil {
    fmt.Println("json err:", err)
  }
  fmt.Println(t)
```

```
  //使用断言判断类型
  m := t.(map[string]interface{})
  for k, v := range m {
    switch val := v.(type) {
    case string:
      fmt.Println(k, "is string", val)
    case int:
      fmt.Println(k, "is int", val)
    case float64:
      fmt.Println(k, "is float64", val)
    case bool:
      fmt.Println(k, "is bool", val)
    case []interface{}:
      fmt.Println(k, "is an array:")
      for i, u := range val {
        fmt.Println(i, u)
      }
    default:
      fmt.Println(k, "is of a type I don't know how to handle")
    }
  }
}
```

第二章 文件操作

1 创建与打开文件

新建文件可以通过如下两个方法：

```
//根据提供的文件名创建新的文件，返回一个文件对象，默认权限是0666的文件，返回的文件对象是可读写的。
func Create(name string) (file *File, err Error)

//根据文件描述符创建相应的文件，返回一个文件对象
func NewFile(fd uintptr, name string) *File
```

示例代码：

```
func main()  {
 //创建文件
 f, err := os.Create("C:/Lianshi/test.txt")
 if err != nil {
     fmt.Println("Create err:", err)
     return
 }
 //关闭文件
 defer f.Close()

 fmt.Println("create successful")
}
```

通过如下两个方法来打开文件：

```
//该方法打开一个名称为name的文件，但是是只读方式，内部实现其实调用了OpenFile。
func Open(name string) (file *File, err Error)
```

Open() 是以只读权限打开文件名为name的文件，得到的文件指针file，只能用来对文件进行“读”操作。如果有 “写” 文件的需求，就需要借助OpenFile函数来打开了。

```
//打开名称为name的文件，flag是打开的方式，只读、读写等，perm是权限
func OpenFile(name string, flag int, perm uint32) (file *File, err Error)
```

参数介绍

```
OpenFile()可以选择打开name文件的读写权限。这个函数有三个默认参数：
参1：name，表示打开文件的路径。可使用相对路径 或 绝对路径
参2：flg，表示读写模式，常见的模式有：
O_RDONLY(只读模式)，O_WRONLY(只写模式)，O_RDWR(可读可写模式)，O_APPEND(追加模式)。
如果读取目录只能指定O_RDONLY模式
参3：perm，表权限取值范围（0-7），表示如下：
0[---]：没有任何权限
1[--x]：执行权限(如果是可执行文件，是可以运行的)
2[-w-]：写权限
3[-wx]：写权限与执行权限
4[r--]：读权限
5[r-x]：读权限与执行权限
6[rw-]：读权限与写权限
7[rwx]：读权限，写权限，执行权限
```

示例代码：

```
func main()  {
 //以可读可写方式打开文件 如果文件不存在则创建新文件 如果文件存在会覆盖其内容
 f, err := os.OpenFile("C:/Lianshi/test.txt", os.O_RDWR | os.O_CREATE,
0600)
 if err != nil {
     fmt.Println("OpenFile err: ", err)
     return
 }
 defer f.Close()
 //将字符串写入到文件
 f.WriteString("hello world")

 fmt.Println("open successful")
}
```

2 写入文件

```
//写入byte类型的信息到文件
func (file *File) Write(b []byte) (n int, err Error)

//在指定位置开始写入byte类型的信息
func (file *File) WriteAt(b []byte, off int64) (n int, err Error)

//写入string信息到文件
func (file *File) WriteString(s string) (ret int, err Error)
```

```
/*
获取文件读写位置
offset:  矢量。 正数，向文件末尾偏移。负数，向文件开头偏移
whence:  偏移的起始位置。
io.SeekStart : 文件起始位置。
io.SeekCurrent : 文件当前位置。
io.SeekEnd: 文件末尾位置。
返回值 ret:  从文件起始位置到，读写位置的偏移量。
*/
func (f *File) Seek(offset int64, whence int) (ret int64, err error)
```

示例代码:

```
package main

import (
  "fmt"
  "os"
)

func main() {
```

```
 //新建文件
  f, err := os.Create("C:/Lianshi/test.txt")
  //f, err := os.OpenFile("C:/Lianshi/test.txt",os.O_RDWR | os.O_CREATE,
0666)
  if err != nil {
    fmt.Println(err)
    return
  }
  //关闭文件
  defer f.Close()

  //写入byte类型的信息到文件
  f.Write([]byte("hello world\r\n"))
  //写入指定位置byte类型的信息到文件
  f.WriteAt([]byte("hello world\r\n"), 5)
  //写入string信息到文件
  f.WriteString("hello world\r\n")
}
```

3 读取文件

```
//读取数据到b中
func (file *File) Read(b []byte) (n int, err Error)

//从off开始读取数据到b中
func (file *File) ReadAt(b []byte, off int64) (n int, err Error)
```

示例代码:

```
package main

import (
  "fmt"
  "os"
)

func main() {
  f, err := os.Open("C:/Lianshi/test.txt")
  if err != nil {
    fmt.Println(err)
    return
  }
  //关闭文件
  defer f.Close()

  buf := make([]byte, 1024)
  //读取文件，返回值为读取有效字符个数和错误信息
```

```
    //n, _ := f.Read(buf)
    //读取指定位置
    n,_:=f.ReadAt(buf,5)
    fmt.Println(string(buf[:n]))
}
```

4 案例：大文件拷贝

示例代码：

```
package main

import (
 "fmt"
 "io"
 "os"
)

func main() {
 args := os.Args //获取用户输入的所有参数

 //如果用户没有输入,或参数个数不够,则调用该函数提示用户
 if args == nil || len(args) != 3 {
     fmt.Println("useage : xxx srcFile dstFile")
     return
 }

 srcPath := args[1] //获取输入的第一个参数
 dstPath := args[2] //获取输入的第二个参数
 fmt.Printf("srcPath = %s, dstPath = %s\n", srcPath, dstPath)

 if srcPath == dstPath {
     fmt.Println("源文件和目的文件名字不能相同")
     return
 }

 srcFile, err1 := os.Open(srcPath) //打开源文件
 if err1 != nil {
     fmt.Println(err1)
     return
 }

 dstFile, err2 := os.Create(dstPath) //创建目的文件
 if err2 != nil {
     fmt.Println(err2)
     return
 }

 buf := make([]byte, 1024) //切片缓冲区
```

```
 for {
     //从源文件读取内容，n为读取文件内容的长度
     n, err := srcFile.Read(buf)
     if err != nil && err != io.EOF {
         fmt.Println(err)
         break
     }

     if n == 0 {
         fmt.Println("文件处理完毕")
         break
     }

     //切片截取
     tmp := buf[:n]
     //把读取的内容写入到目的文件
     dstFile.Write(tmp)
 }

 //关闭文件
 srcFile.Close()
 dstFile.Close()
}
```

第三章 表格处理

1 下载和使用xlsx包

使用的包：

```
github.com/tealeg/xlsx
```

get下载：

```
go get github.com/tealeg/xlsx
```

mod下载：

```
go mod download github.com/tealeg/xlsx
```

```
package main

import (
  "fmt"
  "github/tealeg/xlsx"
)
func main() {
```

```
  //新建表格
  file = xlsx.NewFile()
 //保存文件
  err = file.Save("文件路径.xlsx")
  if err != nil {
    fmt.Printf(err.Error())
  }
}
```

2 创建和写入表格

示例代码:

```
package main

import (
  "fmt"
  "github/tealeg/xlsx"
)

//列
type Cell struct {
  Cname []string
}

//行
type Row struct {
  Cell
}

//表
type Sheet struct {
  SheetName string
  Row
}

//创建表格
func CreateExcel(data Sheet) {

  var file *xlsx.File
  var sheet *xlsx.Sheet
  var row *xlsx.Row
  var cell *xlsx.Cell
  var err error
  //新建表格
  file = xlsx.NewFile()
  //设置当前页名称
  sheet, err = file.AddSheet(data.SheetName)
```

```
  if err != nil {
    fmt.Printf(err.Error())
  }
  //添加行
  row = sheet.AddRow()
  //设置行高
  row.SetHeightCM(1.0)
  //循环添加列 并设置内容
  for _,v:=range data.Cname  {
    cell = row.AddCell()
    cell.Value=v
  }
  //保存文件
  err = file.Save("D:/学员信息表.xlsx")
  if err != nil {
    fmt.Printf(err.Error())
  }

}
func main() {
  //设置表头信息
  sheet := Sheet{"学员信息表",
    Row{Cell{[]string{"姓名", "性别", "年龄", "成绩", "住址"}}}}
  CreateExcel(sheet)
}
```

3 读取和操作表格

示例代码:

```
package main

import (
  "fmt"
  "github/tealeg/xlsx"
)

type Person struct {
  Name       string //微信昵称
  Education  string //学历
  University string //高校
  Industry   string //行业
  Workyear   string //工作年限
  Position   string //职位
  Salary     string //薪资
  Language   string //编程语言
}

func Getxlsx() {
```

```
  var per []Person
  //读取Excel文件
  file, err := xlsx.OpenFile("D:/Go语言工程师信息表.xlsx")
  if err != nil {
    fmt.Printf("open failed: %s\n", err)
    return
  }
  //页
  for _, sheet := range file.Sheets {
    //行
    for _, row := range sheet.Rows {
      //列
      var temp Person
      //将Excel每一列文件读取放在字符串切片中
      var str []string
      for _, cell := range row.Cells {
        str = append(str, cell.String())
      }
      //按照列顺序将数据存储在结构体中
      temp.Name = str[0]
      temp.Education = str[1]
      temp.University = str[2]
      temp.Industry = str[3]
      temp.Workyear = str[4]
      temp.Position = str[5]
      temp.Salary = str[6]
      temp.Language = str[7]

      //将结构体放在结构体切片per中
      per = append(per, temp)
    }
  }
  fmt.Println(per)
}

func main() {
  Getxlsx()
}
```