字符串处理

字符串在开发中经常用到，包括用户的输入，数据库读取的数据等，我们经常需要对字符串进行分割、连接、转换等操作，我们可以通过Go标准库中的strings和strconv两个包中的函数进行相应的操作。

字符串操作

下面这些函数来自于strings包，这里介绍一些平常经常用到的函数，更详细的请参考官方的文档。

Contains

```
func Contains(s, substr string) bool
功能：字符串s中是否包含substr，返回bool值
```

示例代码：

```
    fmt.Println(strings.Contains("seafood", "foo"))
    fmt.Println(strings.Contains("seafood",
"bar"))
    fmt.Println(strings.Contains("seafood", ""))
    fmt.Println(strings.Contains("", ""))
    //运行结果：
    //true
    //false
    //true
```

```
    //true
```

Join

```
func Join(a []string, sep string) string
功能：字符串链接，把slice a通过sep链接起来
```

示例代码：

```
s := []string{"foo", "bar", "baz"}
    fmt.Println(strings.Join(s, ", "))
    //运行结果:foo, bar, baz
```

Index

```
 func Index(s, sep string) int
功能：在字符串s中查找sep所在的位置，返回位置值，找不到返回-1
```

示例代码：

```
    fmt.Println(strings.Index("chicken", "ken"))
    fmt.Println(strings.Index("chicken", "dmr"))
    //运行结果:
    //    4
```

```
    //    -1
```

Repeat

```
func Repeat(s string, count int) string
功能：重复s字符串count次，最后返回重复的字符串
```

示例代码：

```
  fmt.Println("ba" + strings.Repeat("na", 2))

    //运行结果:banana
```

Replace

```
func Replace(s, old, new string, n int) string
功能：在s字符串中，把old字符串替换为new字符串，n表示替换的次数，小于0表示全部替换
```

示例代码：

```
fmt.Println(strings.Replace("oink oink oink", "k", "ky", 2))

    fmt.Println(strings.Replace("oink oink oink", "oink", "moo", -1))
```

```
    //运行结果:

    //oinky oinky oink

    //moo moo moo
```

split

```
func Split(s, sep string) []string
功能：把s字符串按照sep分割，返回slice
```

示例代码：

```
fmt.Printf("%q\n", strings.Split("a,b,c", ","))
    fmt.Printf("%q\n", strings.Split("a man a plan a canal panama", "a "))
    fmt.Printf("%q\n", strings.Split(" xyz ", ""))
    fmt.Printf("%q\n", strings.Split("", "Bernardo O'Higgins"))
    //运行结果:
    //["a" "b" "c"]
    //["" "man " "plan " "canal panama"]
    //[" " "x" "y" "z" " "]
    //[""]
```

Trim

```
func Trim(s string, cutset string) string
功能：在s字符串的头部和尾部去除cutset指定的字符串
```

示例代码：

```
fmt.Printf("[%q]", strings.Trim(" !!! Achtung !!!
", "! "))
    //运行结果:["Achtung"]
```

Fields

```
func Fields(s string) []string
功能：去除s字符串的空格符，并且按照空格分割返回slice
```

示例代码：

```
    fmt.Printf("Fields are: %q", strings.Fields("
foo bar  baz   "))
    //运行结果:Fields are: ["foo" "bar" "baz"]
```

字符串转换

字符串转化的函数在strconv中，如下也只是列出一些常用的。

Append

Append 系列函数将整数等转换为字符串后，添加到现有的字节数组中。

示例代码:

```
 str := make([]byte, 0, 100)
    str = strconv.AppendInt(str, 4567, 10) //以10进制方式追加
    str = strconv.AppendBool(str, false)
    str = strconv.AppendQuote(str, "abcdefg")
    str = strconv.AppendQuoteRune(str, '单')

    fmt.Println(string(str))
//4567false"abcdefg"'单'
```

Format

Format 系列函数把其他类型的转换为字符串。

示例代码:

```
    a := strconv.FormatBool(false)

    b := strconv.FormatInt(1234, 10)

    c := strconv.FormatUint(12345, 10)

    d := strconv.Itoa(1023)
```

```
    fmt.Println(a, b, c, d) //false 1234 12345
1023
```

Parse

Parse 系列函数把字符串转换为其他类型。
示例代码：

```
package main

import (
    "fmt"
    "strconv"
)

func checkError(e error) {
    if e != nil {
        fmt.Println(e)
    }
}
func main() {
    a, err := strconv.ParseBool("false")
    checkError(err)
    b, err := strconv.ParseFloat("123.23", 64)
    checkError(err)
    c, err := strconv.ParseInt("1234", 10, 64)
    checkError(err)
    d, err := strconv.ParseUint("12345", 10, 64)
    checkError(err)
```

```
    e, err := strconv.Atoi("1023")
    checkError(err)
    fmt.Println(a, b, c, d, e) //false 123.23 1234
12345 1023
}
```

正则表达式

正则表达式是一种进行模式匹配和文本操纵的复杂而又强大的工具。虽然正则表达式比纯粹的文本匹配效率低，但是它却更灵活。按照它的语法规则，随需构造出的匹配模式就能够从原始文本中筛选出几乎任何你想要得到的字符组合。

Go语言通过regexp标准包为正则表达式提供了官方支持，如果你已经使用过其他编程语言提供的正则相关功能，那么你应该对Go语言版本的不会太陌生，但是它们之间也有一些小的差异，因为Go实现的是RE2标准，除了\C，详细的语法描述参考：http://code.google.com/p/re2/wiki/Syntax

其实字符串处理我们可以使用strings包来进行搜索(Contains、Index)、替换(Replace)和解析(Split、Join)等操作，但是这些都是简单的字符串操作，他们的搜索都是大小写敏感，而且固定的字符串，如果我们需要匹配可变的那种就没办法实现了，当然如果strings包能解决你的问题，那么就尽量使用它来解决。因为他们足够简单、而且性能和可读性都会比正则好。

示例代码：

```
package main
```

```
import (
    "fmt"
    "regexp"
)

func main() {
    context1 := "3.14 123123 .68 haha 1.0 abc 6.66
123."

    //MustCompile解析并返回一个正则表达式。如果成功返
回，该Regexp就可用于匹配文本。
    //解析失败时会产生panic
    // \d 匹配数字[0-9]，d+ 重复>=1次匹配d，越多越好
（优先重复匹配d）
    exp1 := regexp.MustCompile(`\d+\.\d+`)

    //返回保管正则表达式所有不重叠的匹配结果的[]string切
片。如果没有匹配到，会返回nil。
    //result1 := exp1.FindAllString(context1, -1)
//[3.14 1.0 6.66]
    result1 :=
exp1.FindAllStringSubmatch(context1, -1) //[[3.14]
[1.0] [6.66]]

    fmt.Printf("%v\n", result1)
    fmt.Printf("\n-------------------------------
----\n\n")
```

```
    context2 := `
        <title>标题</title>
        <div>你过来啊</div>
        <div>hello mike</div>
        <div>你大爷</div>
        <body>呵呵</body>
    `
    //(.*?)被括起来的表达式作为分组
    //匹配<div>xxx</div>模式的所有子串
    exp2 := regexp.MustCompile(`<div>(.*?)</div>`)
    result2 :=
exp2.FindAllStringSubmatch(context2, -1)

    //[[<div>你过来啊</div> 你过来啊] [<div>hello
mike</div> hello mike] [<div>你大爷</div> 你大爷]]
    fmt.Printf("%v\n", result2)
    fmt.Printf("\n-------------------------------
----\n\n")

    context3 := `
        <title>标题</title>
        <div>你过来啊</div>
        <div>hello
        mike
        go</div>
        <div>你大爷</div>
        <body>呵呵</body>
    `
```

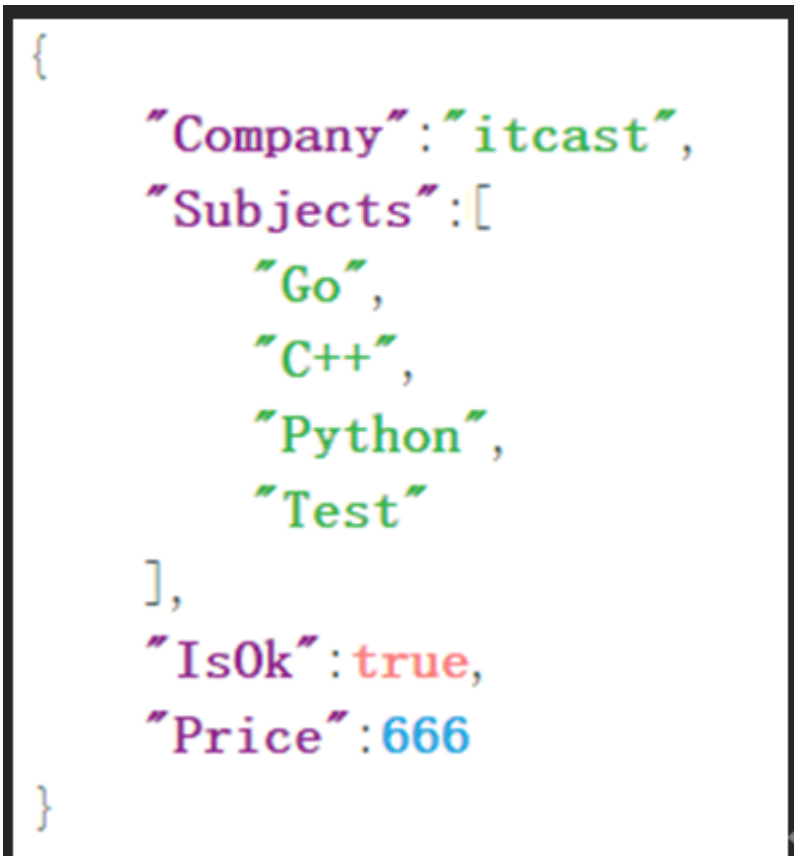
```
    exp3 := regexp.MustCompile(`<div>(.*?)</div>`)
    result3 :=
exp3.FindAllStringSubmatch(context3, -1)

    //[[<div>你过来啊</div> 你过来啊] [<div>你大爷
</div> 你大爷]]
    fmt.Printf("%v\n", result3)
    fmt.Printf("\n-------------------------------
----\n\n")

    context4 := `
        <title>标题</title>
        <div>你过来啊</div>
        <div>hello
        mike
        go</div>
        <div>你大爷</div>
        <body>呵呵</body>
    `
    exp4 := regexp.MustCompile(`<div>(?s:(.*?))
</div>`)
    result4 :=
exp4.FindAllStringSubmatch(context4, -1)

    /*
        [[<div>你过来啊</div> 你过来啊] [<div>hello
            mike
            go</div> hello
```

```
            mike
            go] [<div>你大爷</div> 你大爷]]
    */
    fmt.Printf("%v\n", result4)
    fmt.Printf("\n-------------------------------
----\n\n")

    for _, text := range result4 {
        fmt.Println(text[0]) //带有div
        fmt.Println(text[1]) //不带带有div
        fmt.Println("================\n")
    }
}
```

JSON处理

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。



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。

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。

JSON官方网站：http://www.json.org/

在线格式化：http://www.json.cn/

编码JSON

通过结构体生成JSON

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)
```

还有一个格式化输出：

```
// 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)
```

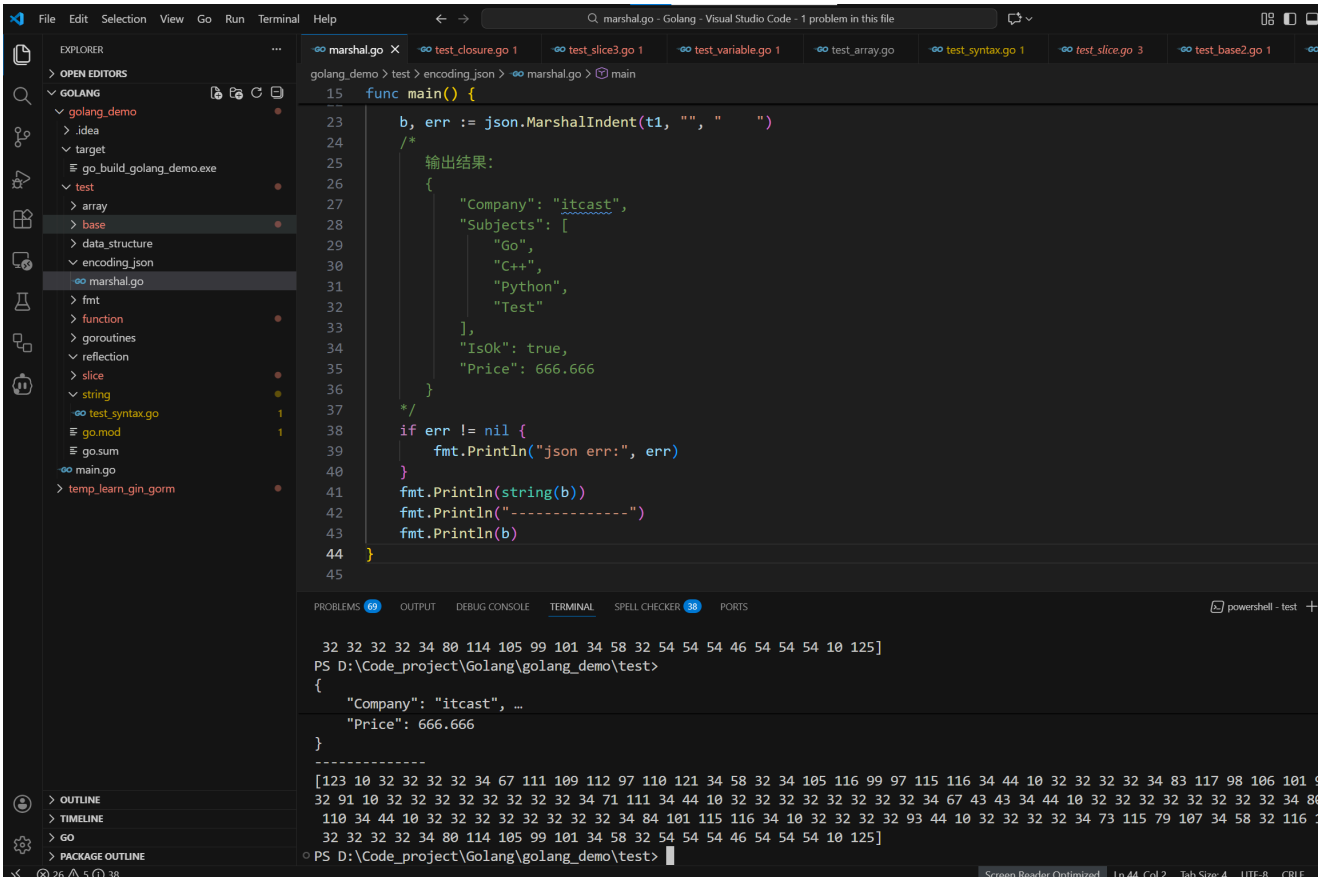
编码JSON

示例代码：

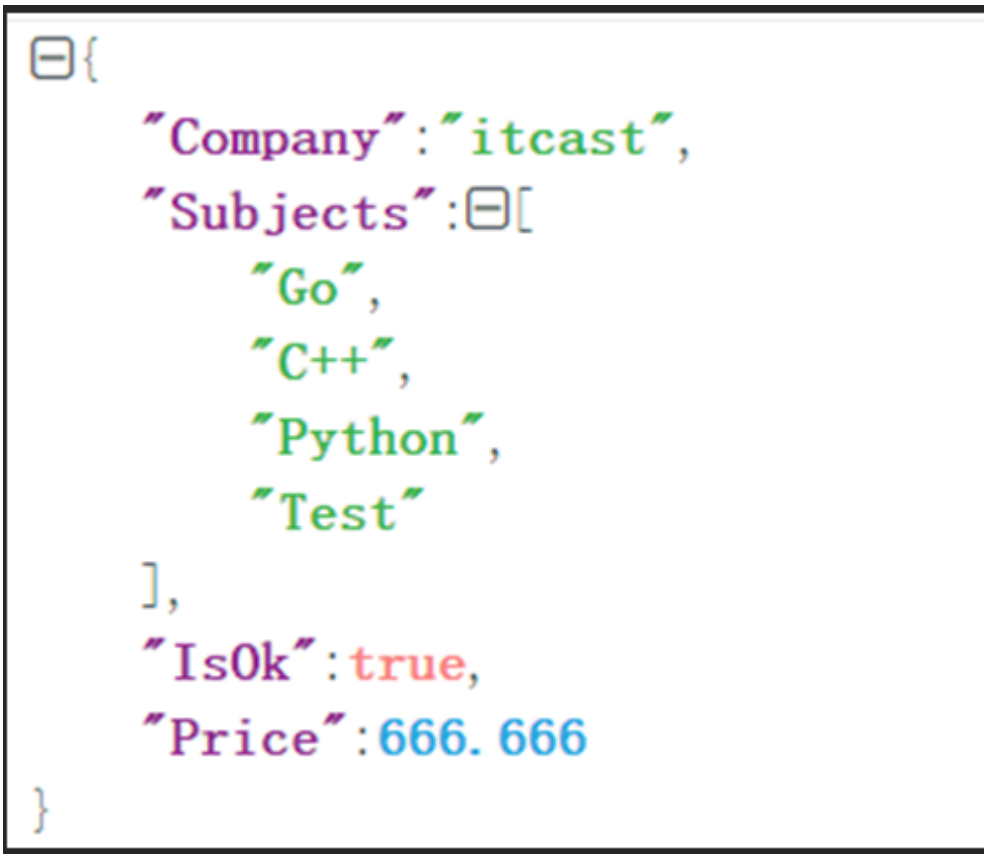
```
package main

import (
    "encoding/json"
    "fmt"
)

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

```
}

func main() {
    t1 := IT{"itcast", []string{"Go", "C++",
"Python", "Test"}, true, 666.666}

    //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)
    //输出结果：{"Company":"itcast","Subjects":
["Go","C++","Python","Test"],"IsOk":true,"Price":6
66.666}

    b, err := json.MarshalIndent(t1, "", "    ")
    /*
        输出结果:
        {
            "Company": "itcast",
            "Subjects": [
                "Go",
                "C++",
                "Python",
                "Test"
            ],
            "IsOk": true,
            "Price": 666.666
        }
```

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



struct tag



可以看到上面的输出字段名的首字母都是大写的，如果你想用小写的首字母怎么办呢？把结构体的字段名改成首字母小写的？JSON输出的时候必须注意，只有导出的字段(首字母是大写)才会被输出，如果修改字段名，那么就会发现什么都不会输出，所以必须通过struct tag定义来实现。

针对JSON的输出，我们在定义struct tag的时候需要注意的几点是：

- 字段的tag是"-"，那么这个字段不会输出到JSON
    - tag中带有自定义名称，那么这个自定义名称会出现在JSON的字段名中
- tag中如果带有"omitempty"选项，那么如果该字段值为空，就不会输出到JSON串中

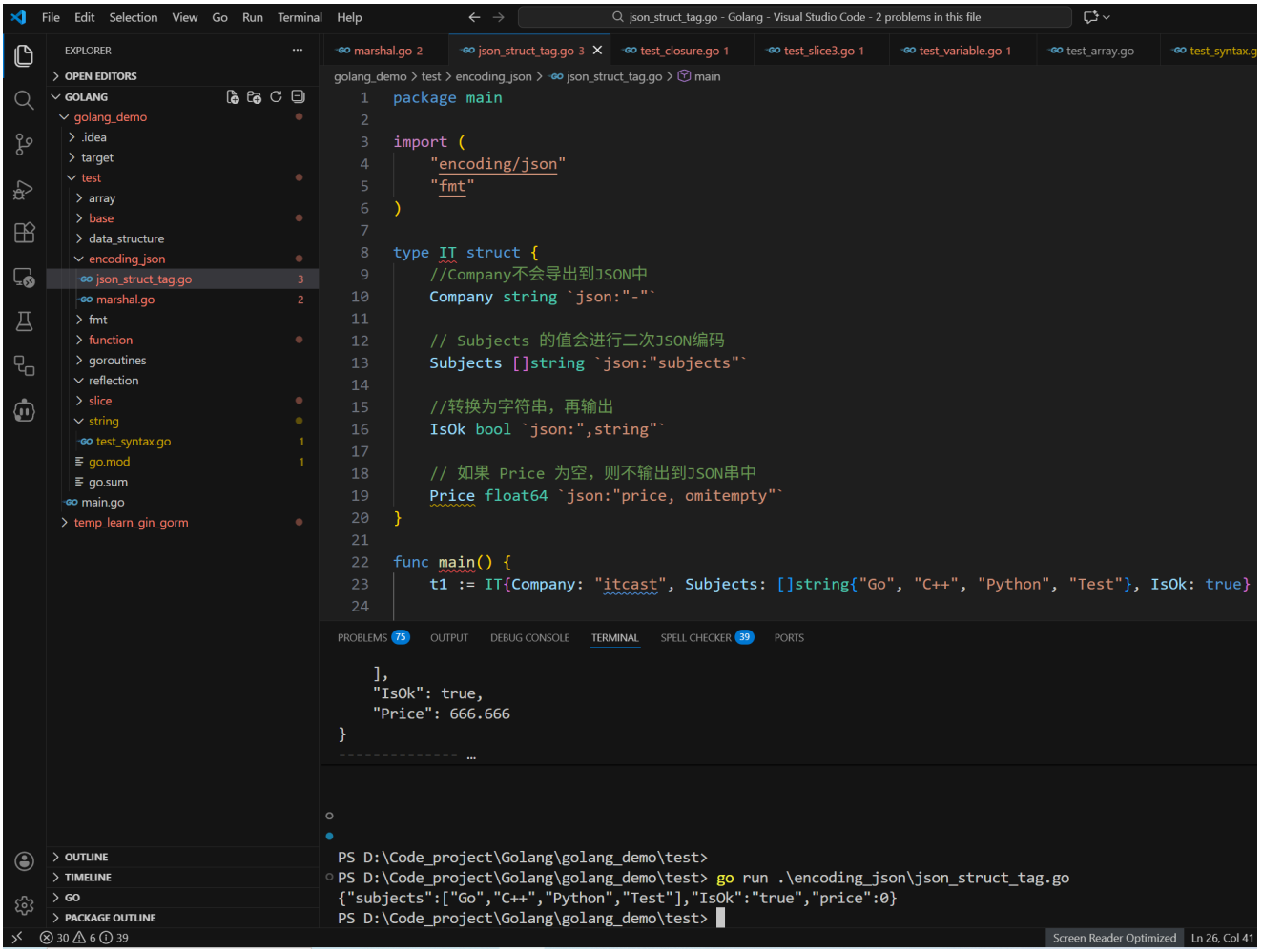
如果字段类型是bool, string, int, int64等，而tag中带有",string"选项，那么这个字段在输出到JSON的时候会把该字

段对应的值转换成JSON字符串

示例代码：

```
type IT struct {
    //Company不会导出到JSON中
    Company string `json:"-"`

    // Subjects 的值会进行二次JSON编码
    Subjects []string `json:"subjects"`

    //转换为字符串，再输出
    IsOk bool `json:",string"`

    // 如果 Price 为空，则不输出到JSON串中
    Price float64 `json:"price, omitempty"`
}

func main() {
    t1 := IT{Company: "itcast", Subjects:
[]string{"Go", "C++", "Python", "Test"}, IsOk:
true}

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

```
    //输出结果: {"subjects":
["Go","C++","Python","Test"],"IsOk":"true","price"
:0}
}
```



通过map生成JSON

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

```
    t1["price"] = 666.666

    b, err := json.Marshal(t1)
    //json.MarshalIndent(t1, "", "    ")
    if err != nil {
        fmt.Println("json err:", err)
    }
    fmt.Println(string(b))
    //输出结果:
{"company":"itcast","isok":true,"price":666.666,"s
ubjects ":["Go","C++","Python","Test"]}
```

解码JSON

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
```

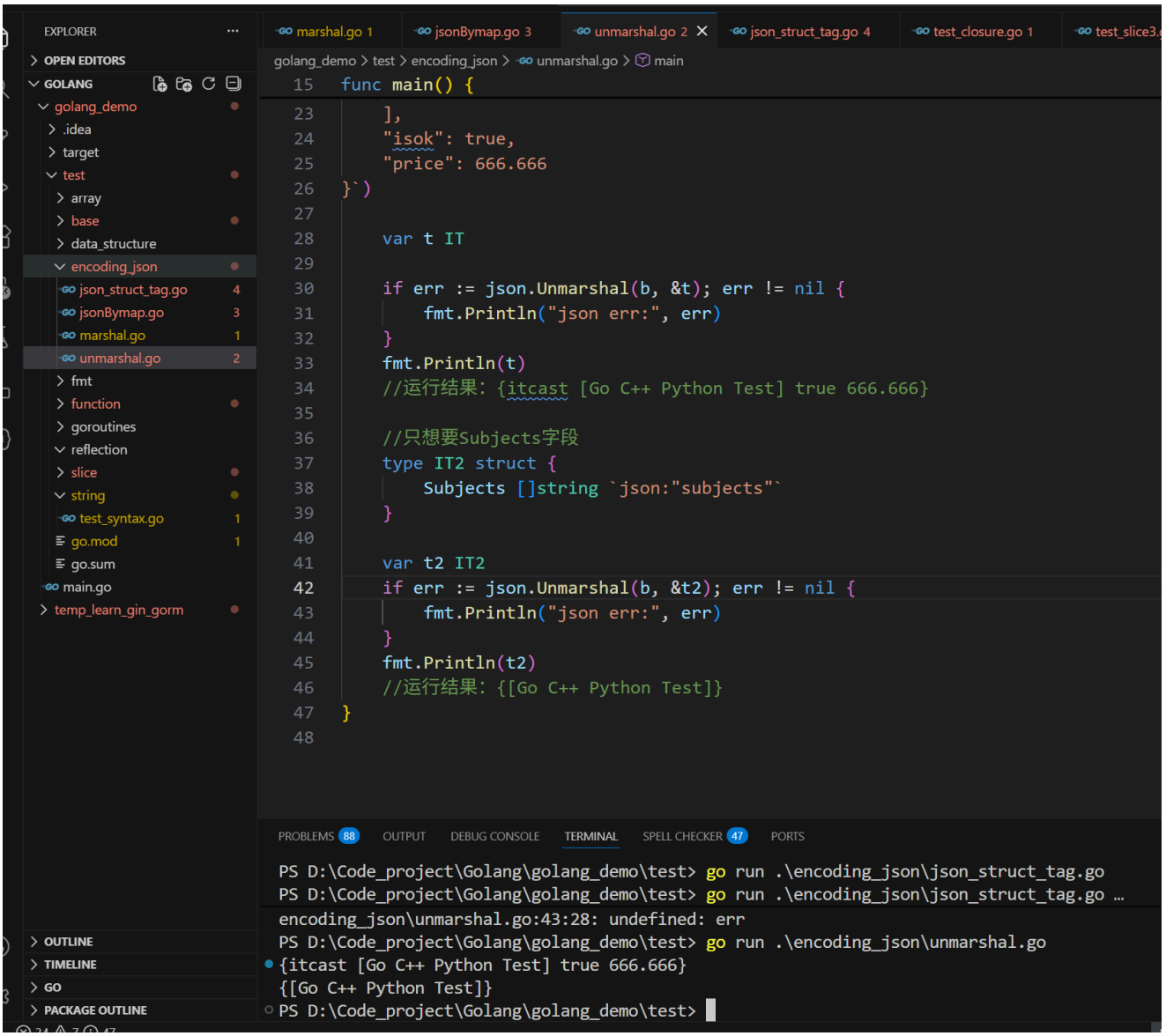
该函数的第一个参数是输入，即JSON格式的文本（比特序列），第二个参数表示目标输出容器，用于存放解码后的值。

解析到结构体

```
type IT struct {
    Company  string   `json:"company"`
    Subjects []string `json:"subjects"`
```

```
    IsOk     bool     `json:"isok"`
    Price    float64  `json:"price"`
}

func main() {
    b := []byte(`{
    "company": "itcast",
    "subjects": [
        "Go",
        "C++",
        "Python",
        "Test"
    ],
    "isok": true,
    "price": 666.666
}`)

    var t IT
    err := json.Unmarshal(b, &t)
    if err != nil {
        fmt.Println("json err:", err)
    }
    fmt.Println(t)
    //运行结果：{itcast [Go C++ Python Test] true
666.666}

    //只想要Subjects字段
    type IT2 struct {
```

```
        Subjects []string `json:"subjects"`
    }

    var t2 IT2
    err = json.Unmarshal(b, &t2)
    if err != nil {
        fmt.Println("json err:", err)
    }
    fmt.Println(t2)
    //运行结果：{[Go C++ Python Test]}
}
```



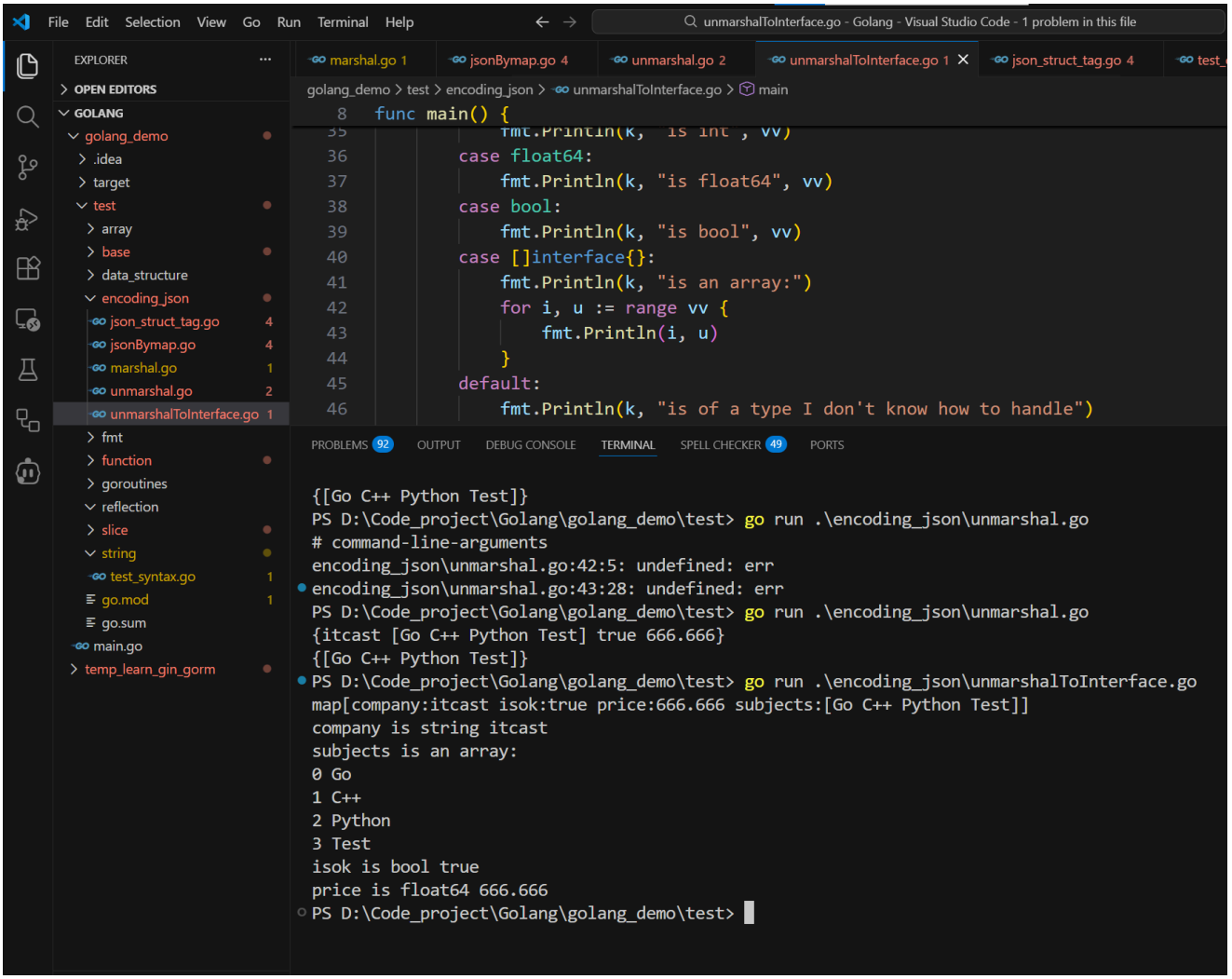
解析到interface

示例代码：

```
func main() {
    b := []byte(`{
    "company": "itcast",
    "subjects": [
        "Go",
        "C++",
        "Python",
        "Test"
```

```
    ],
    "isok": true,
    "price": 666.666
}`)

    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 vv := v.(type) {
        case string:
            fmt.Println(k, "is string", vv)
        case int:
            fmt.Println(k, "is int", vv)
        case float64:
            fmt.Println(k, "is float64", vv)
        case bool:
            fmt.Println(k, "is bool", vv)
        case []interface{}:
            fmt.Println(k, "is an array:")
            for i, u := range vv {
                fmt.Println(i, u)
```

```
            }
        default:
            fmt.Println(k, "is of a type I don't know how to handle")
        }
    }
}
```



文件操作

相关API介绍

建立与打开文件

新建文件可以通过如下两个方法

```
func Create(name string) (file *File, err Error)
```

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 NewFile(fd uintptr, name string) *File
```

根据文件描述符创建相应的文件，返回一个文件对象

通过如下两个方法来打开文件：

```
func Open(name string) (file *File, err Error)
```

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 OpenFile(name string, flag int, perm uint32)
(file *File, err Error)
```

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

写文件

```
func (file *File) Write(b []byte) (n int, err
Error)
```

写入byte类型的信息到文件

```
func (file *File) WriteAt(b []byte, off int64) (n
int, err Error)
```

在指定位置开始写入byte类型的信息

```
func (file *File) WriteString(s string) (ret int, err Error)
```

写入string信息到文件

读文件

```
func (file *File) Read(b []byte) (n int, err Error)
```

读取数据到b中

```
func (file *File) ReadAt(b []byte, off int64) (n int, err Error)
```

从off开始读取数据到b中

删除文件

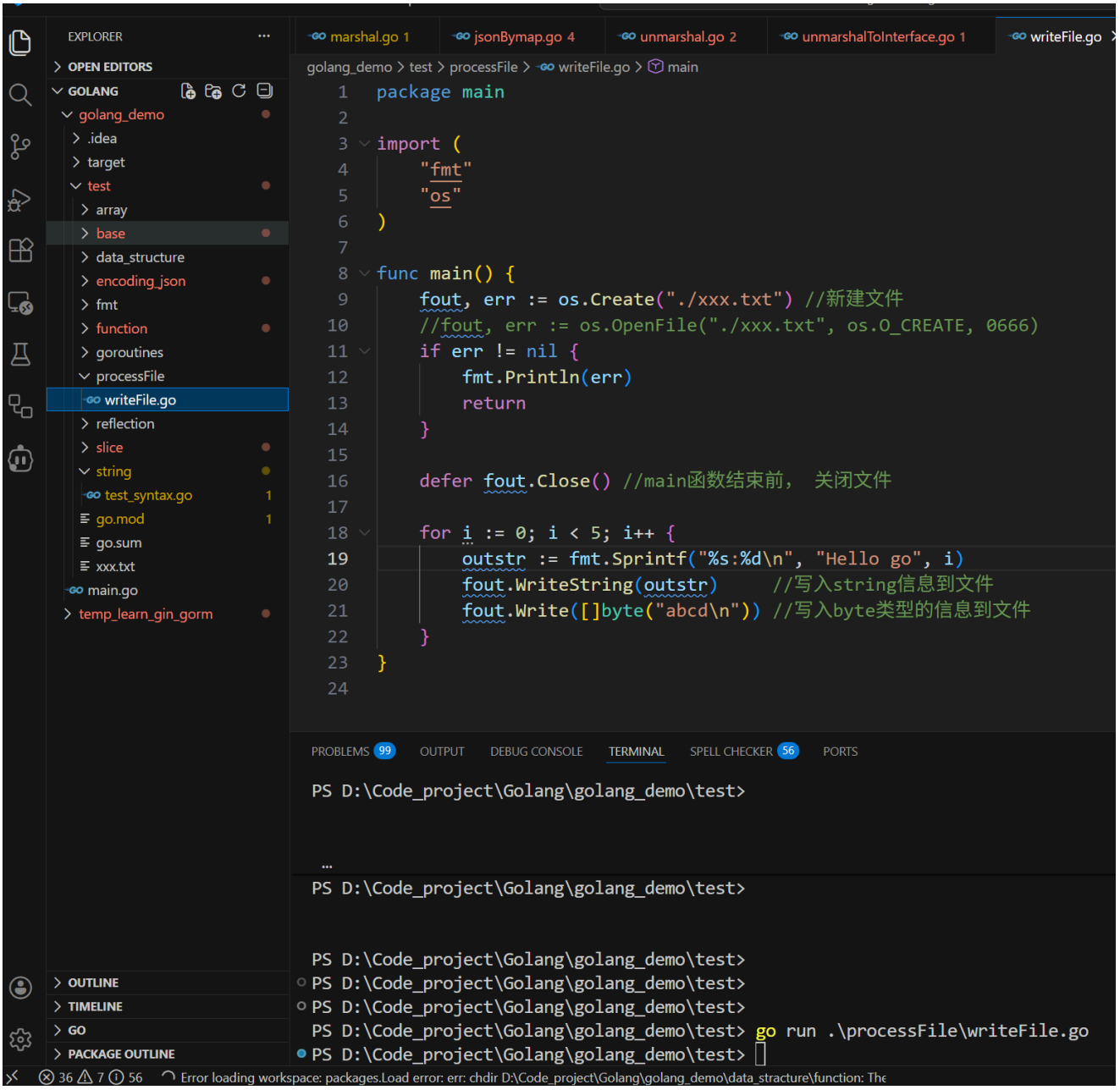
```
func Remove(name string) Error
```

调用该函数就可以删除文件名为name的文件

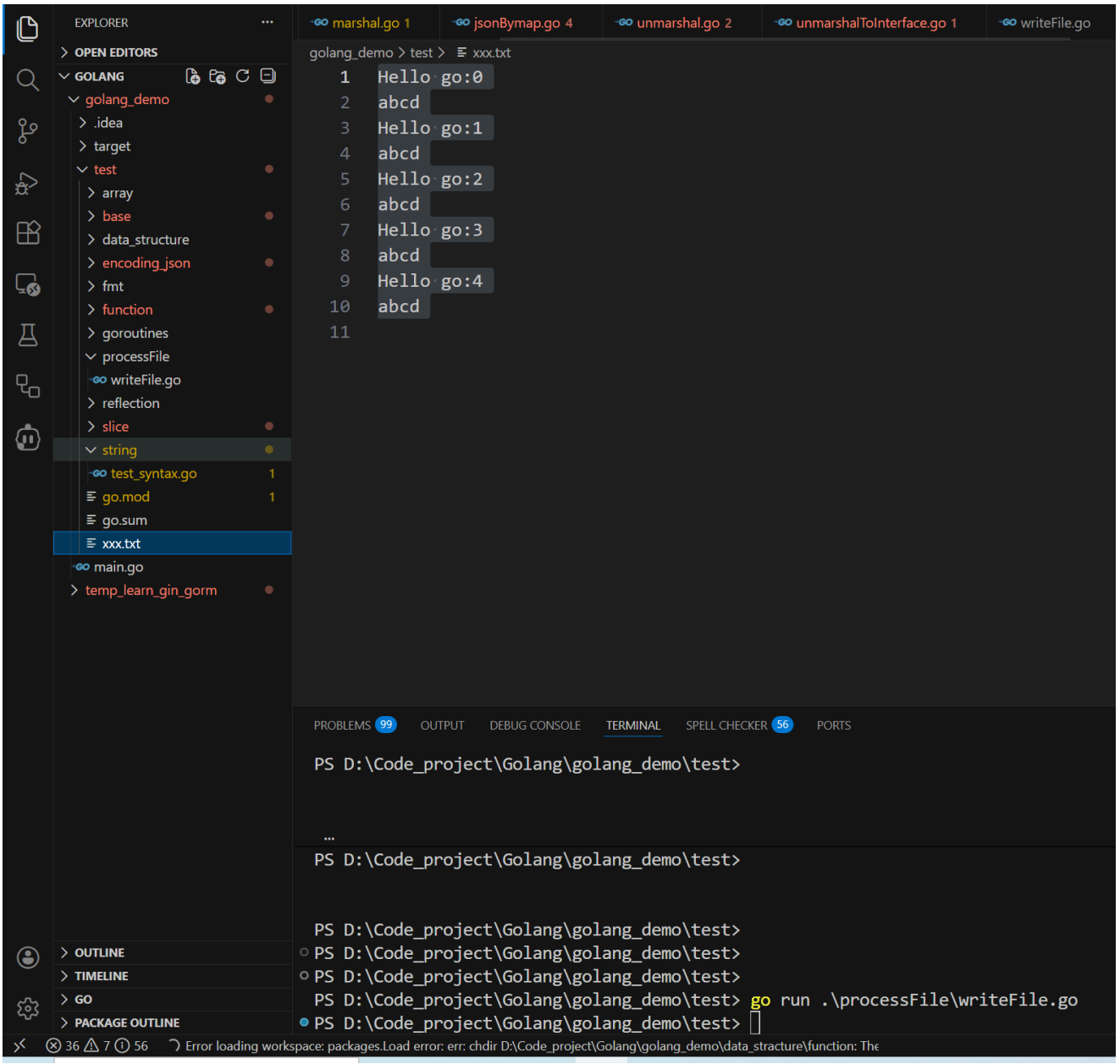
示例

写文件

```go
package main

import (
    "fmt"
    "os"
)

func main() {
    fout, err := os.Create("./xxx.txt") //新建文件
    //fout, err := os.OpenFile("./xxx.txt", os.O_CREATE, 0666)
    if err != nil {
        fmt.Println(err)
        return
    }

    defer fout.Close() //main函数结束前， 关闭文件

    for i := 0; i < 5; i++ {
        outstr := fmt.Sprintf("%s:%d\n", "Hello go", i)
        fout.WriteString(outstr)     //写入string信息到文件
        fout.Write([]byte("abcd\n")) //写入byte类型的信息到文件
    }
}
```


EXPLORER
OPEN EDITORS
GOLANG
golang_demo
.idea
target
test
array
base
data_structure
encoding_json
fmt
function
goroutines
processFile
writeFile.go
reflection
slice
string
test_syntax.go
go.mod
go.sum
xxx.txt
main.go
temp_learn_gin_gorm
marshal.go 1
jsonBymap.go 4
unmarshal.go 2
unmarshalToInterface.go 1
writeFile.go
golang_demo > test > processFile > writeFile.go > main
1 package main
2
3 import (
4 "fmt"
5 "os"
6)
7
8 func main() {
9 fout, err := os.Create("./xxx.txt") //新建文件
10 //fout, err := os.OpenFile("./xxx.txt", os.O_CREATE, 0666)
11 if err != nil {
12 fmt.Println(err)
13 return
14 }
15
16 defer fout.Close() //main函数结束前， 关闭文件
17
18 for i := 0; i < 5; i++ {
19 outstr := fmt.Sprintf("%s:%d\n", "Hello go", i)
20 fout.WriteString(outstr) //写入string信息到文件
21 fout.Write([]byte("abcd\n")) //写入byte类型的信息到文件
22 }
23 }
24
PROBLEMS 99
OUTPUT
DEBUG CONSOLE
TERMINAL
SPELL CHECKER 56
PORTS
PS D:\Code_project\Golang\golang_demo\test>
PS D:\Code_project\Golang\golang_demo\test>
PS D:\Code_project\Golang\golang_demo\test>
PS D:\Code_project\Golang\golang_demo\test>
PS D:\Code_project\Golang\golang_demo\test>
PS D:\Code_project\Golang\golang_demo\test> go run .\processFile\writeFile.go
PS D:\Code_project\Golang\golang_demo\test>
OUTLINE
TIMELINE
GO
PACKAGE OUTLINE
36 7 56
Error loading workspace: packages.Load error: err: chdir D:\Code_project\Golang\golang_demo\data_structure\function: The

xxx.txt内容如下：



读文件

```go
func main() {
    fin, err := os.Open("./xxx.txt") //打开文件
    if err != nil {
        fmt.Println(err)
    }
    defer fin.Close()

    buf := make([]byte, 1024) //开辟1024个字节的
```

```
slice作为缓冲
    for {
        n, _ := fin.Read(buf) //读文件
        if n == 0 {             //0表示已经到文件结束
            break
        }

        fmt.Println(string(buf)) //输出读取的内容
    }
}
```

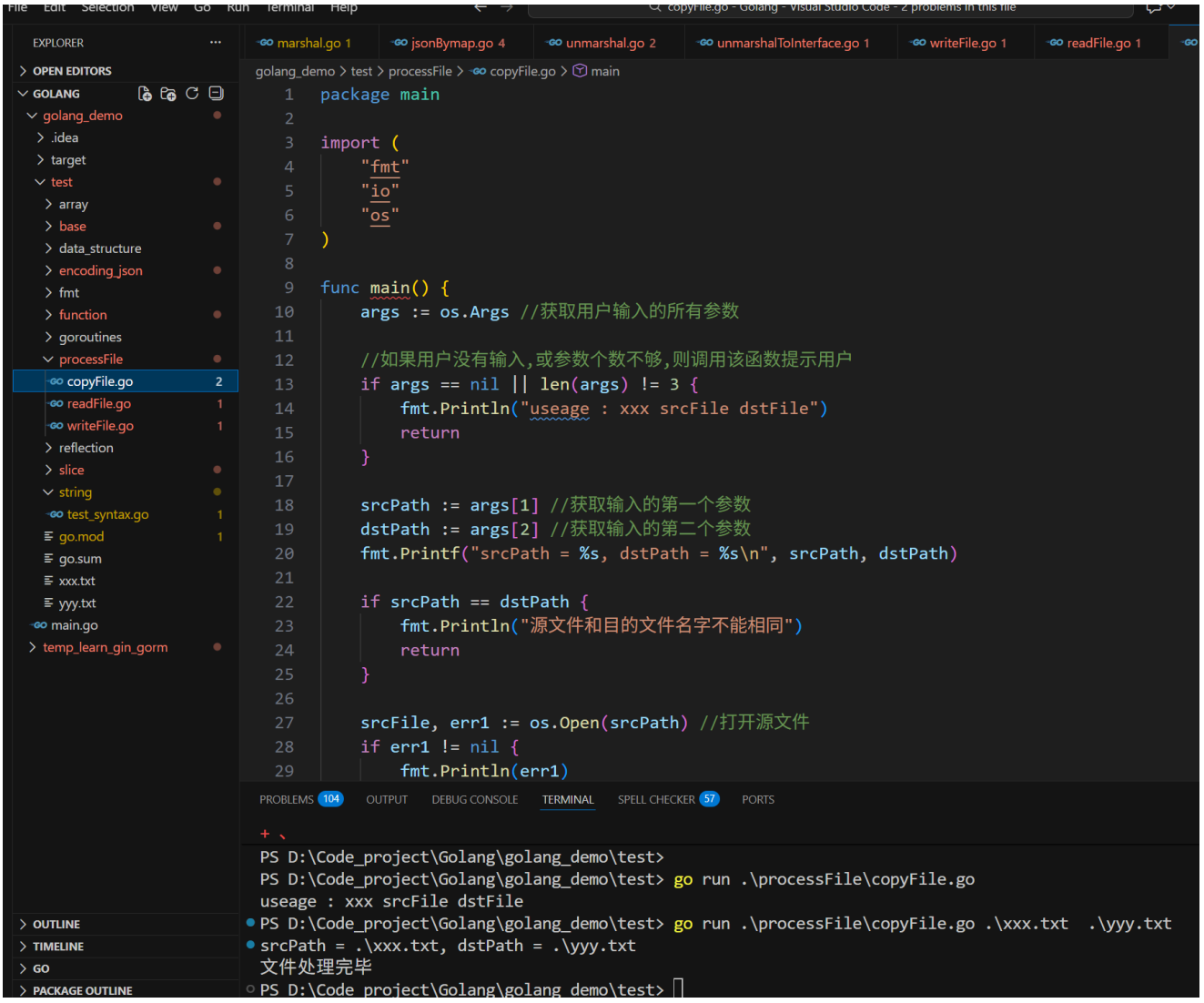
拷贝文件

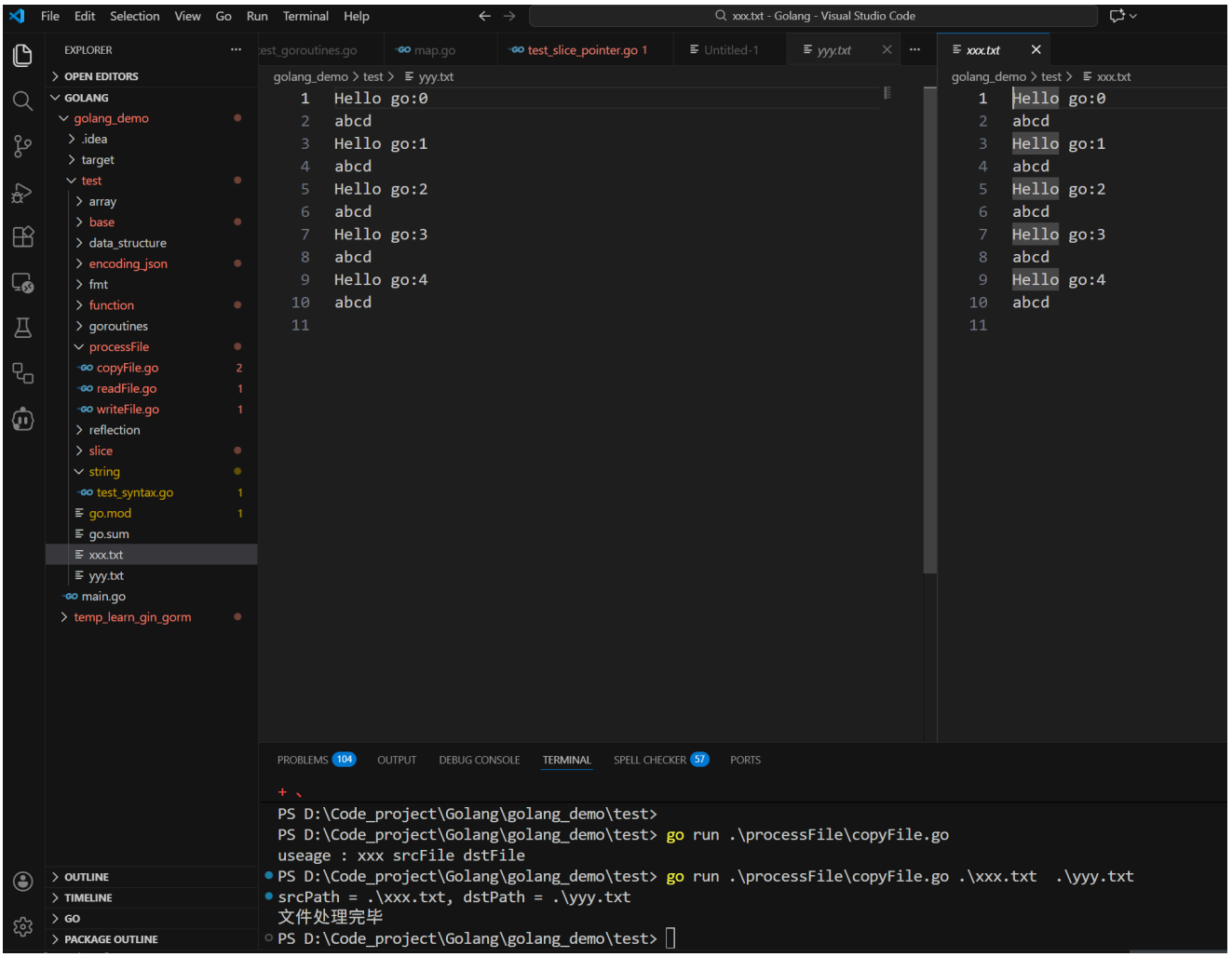
示例代码:

```
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()
}
```


EXPLORER
OPEN EDITORS
GOLANG
golang_demo
.idea
target
test
array
base
data_structure
encoding_json
fmt
function
goroutines
processFile
copyFile.go 2
readFile.go 1
writeFile.go 1
reflection
slice
string
test_syntax.go 1
go.mod 1
go.sum
xxx.txt
yyy.txt
main.go
temp_learn_gin_gorm
OUTLINE
TIMELINE
GO
PACKAGE OUTLINE
marshal.go 1
jsonBymap.go 4
unmarshal.go 2
unmarshalToInterface.go 1
writeFile.go 1
readFile.go 1
golang_demo > test > processFile > copyFile.go > main
1 package main
2
3 import (
4 "fmt"
5 "io"
6 "os"
7)
8
9 func main() {
10 args := os.Args //获取用户输入的所有参数
11
12 //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
13 if args == nil || len(args) != 3 {
14 fmt.Println("useage : xxx srcFile dstFile")
15 return
16 }
17
18 srcPath := args[1] //获取输入的第一个参数
19 dstPath := args[2] //获取输入的第二个参数
20 fmt.Printf("srcPath = %s, dstPath = %s\n", srcPath, dstPath)
21
22 if srcPath == dstPath {
23 fmt.Println("源文件和目的文件名字不能相同")
24 return
25 }
26
27 srcFile, err1 := os.Open(srcPath) //打开源文件
28 if err1 != nil {
29 fmt.Println(err1)
PROBLEMS 104
OUTPUT
DEBUG CONSOLE
TERMINAL
SPELL CHECKER 57
PORTS
PS D:\Code_project\Golang\golang_demo\test>
PS D:\Code_project\Golang\golang_demo\test> go run .\processFile\copyFile.go
useage : xxx srcFile dstFile
PS D:\Code_project\Golang\golang_demo\test> go run .\processFile\copyFile.go .\xxx.txt .\yyy.txt
srcPath = .\xxx.txt, dstPath = .\yyy.txt
文件处理完毕
PS D:\Code_project\Golang\golang_demo\test>


File Edit Selection View Go Run Terminal Help
xxx.txt - Golang - Visual Studio Code
EXPLORER
OPEN EDITORS
GOLANG
golang_demo
.idea
target
test
array
base
data_structure
encoding_json
fmt
function
goroutines
processFile
copyFile.go
readFile.go
writeFile.go
reflection
slice
string
test_syntax.go
go.mod
go.sum
xxx.txt
yyy.txt
main.go
temp_learn_gin_gorm
map.go
test_slice_pointer.go 1
Untitled-1
yyy.txt
xxx.txt
golang_demo > test > yyy.txt
golang_demo > test > xxx.txt
Hello go:0
abcd
Hello go:1
abcd
Hello go:2
abcd
Hello go:3
abcd
Hello go:4
abcd
PROBLEMS 104
OUTPUT
DEBUG CONSOLE
TERMINAL
SPELL CHECKER 57
PORTS
PS D:\Code_project\Golang\golang_demo\test>
PS D:\Code_project\Golang\golang_demo\test> go run .\processFile\copyFile.go
useage : xxx srcFile dstFile
PS D:\Code_project\Golang\golang_demo\test> go run .\processFile\copyFile.go .\xxx.txt .\yyy.txt
srcPath = .\xxx.txt, dstPath = .\yyy.txt
文件处理完毕
PS D:\Code_project\Golang\golang_demo\test>
OUTLINE
TIMELINE
GO
PACKAGE OUTLINE