let's Golang(四)：IO

标记

Richard
full stack, gopher, rustacean

对于任何语言而言，IO都必不可少并且非常重要的特性。比如node.js，其出色的异步IO特性为其在服务端开发占据了一定的市场。Go官方提供的IO相关的package非常完善，能够满足日常开发中的绝大多数情况。下面开始介绍。

命令行参数

首先是os.Args，返回的是一个[]string，包含所有的命令行参数：

```
func main()  {
        fmt.Println(os.Args)
}
```

执行程序：

```
$ ./io hello world
```

输出结果

```
[./io hello world]
```

我们可以看到，os.Args返回的slice中，**第一个元素为程序的名称**，后面的元素才是真正的命令行参数。

其次，flag包里面提供了非常方便的解析命令行参数的相关代码，下面的代码演示了如何使用flag来解析参数：

```
var env = flag.String("env", "development", "env argument")

func main()  {
        flag.Parse()
        fmt.Println(flag.NArg())
        fmt.Println(flag.Arg(0))
```

```
        fmt.Println(flag.Arg(1))
        fmt.Println(*env)
}
```

执行程序：

```
./io -env="production" hello world
```

输出结果：

```
2
hello
world
production
```

简单解释一下上面的代码：

1. flag.Parse()扫描参数列表，完成解析(必须要有这一过程)
2. flag.NArg()返回参数的数量，这里返回的值是2，不包括"-arg=val"这种参数的形式
3. flag.Arg(n int)返回第n+1个参数，这里如果n为0，即返回第一个参数，跟os.Args[1]是同一个值
4. flag.String，flag.Bool，http://flag.Int等方法解析"-arg=val"形式的参数
5. 上面执行程序中 “-env="production"” 跟后面的"hello world" 顺序很重要， “-arg=val” 形式的参数必须放在前面，不然会被当做普通形式的参数来解析，可以试试调换参数的顺序，看看输出什么结果

读取用户输入

在Go中，也提供了从控制台读取用户输入的方法：

```
func main()  {
        var val string
        for {
                fmt.Scanln(&val)
                if strings.Trim(val, "\n") == "exit" {
                        break
                }
                fmt.Println(val)
        }
}
```

上面这段代码读取用户输入，如果是"exit"就结束程序，否则打印用户的输入。

读取文件

在Go中，读取文件的方式由很多种，我们可以借助官方库提供的大量第三方库来读取文件：

```
func main()  {
        file, err := os.Open("./go.mod")
        if err != nil {
                log.Fatal(err)
        }
        defer file.Close()
        reader := bufio.NewReader(file)
        for {
                val, err := reader.ReadString('\n') // 或者 reader.ReadLine
                if err == io.EOF {
                        break
                }
                fmt.Println(val)
        }
}
```

我们可以看到上面的代码中：os.Open打开一个文件，然后创建*os.File类型的变量，代表打开文件的文件描述符；同时，还需要使用defer file.Close()来关闭打开的文件；在之后，我们使用bufio.newReader()来包装打开的文件，实现文件的读取；最后，在一个无限循环中，按行读取文件的内容，如果发现已经读取到文件尾，则跳出循环。

上面是按行读取文件内容，很多时候，我们希望能一次获取文件的所有内容，那么ioutil就非常有用了，但是需要注意的时，如果文件特别大，有可能占满内存，所以，尽量使用该方法来读取小文件：

```
func main() {
    buf, err := ioutil.ReadFile("./go.mod")
    if err != nil {
        fmt.Println(err)
    }
    fmt.Println(string(buf))
}
```

所以，如果我们要读取一个大文件，但是这个大文件又是二进制的，那应该怎么办呢，我们可以使用带缓冲的读取形式：

```
func main() {
    file, err := os.Open("./go.mod")
    if err != nil {
        log.Fatal(err)
    }
    defer file.Close()
    reader := bufio.NewReader(file)
    buf := make([]byte, 1024)
    _, err = reader.Read(buf)
    if err != nil {
```

```
            log.Fatal(err)
        }
        fmt.Println(string(buf))
    }
```

写文件

写文件跟读文件非常类似，我们来看个简单的例子：

```
func main()  {
        file, err := os.OpenFile("./test.txt", os.O_WRONLY | os.O_CREATE, 0666)
        if err != nil {
                log.Fatal(err)
        }
        defer file.Close()
        writer := bufio.NewWriter(file)
        writer.WriteString("hello world")
        writer.Flush()
}
```

这里跟读文件唯一的区别是：使用os.OpenFile来打开文件，os.Open是对os.OpenFile的定制，所以os.OpenFile可以指定更多的参数。可以看到，os.OpenFile函数有三个参数：文件名，一个或多个标志，使用文件的权限，通常会用到以下标志：

- os.O_RDONLY: 只读
- os.O_WRONLY: 只写
- os.O_CREATE: 创建，如果文件不存在，就创建该文件
- os.O_TRUNC: 截断，如果指定文件已存在，就将该文件的长度截断为0

在读文件的时候，文件的权限是被忽略的，所以在私用OpenFile时，传入的第三个参数可以为0；在写文件的时候，不管是Unix还是Windows，都需要使用0666；

也可以使用ioutil来写文件：

```
func main()  {
        content := "hello world"
        ioutil.WriteFile("./test.txt", []byte(content), 0666)

}
```

复制文件

我们使用io和bufio两种方式来复制文件：

```
func main()  {
        reader, _ := os.Open("./src.txt")
        writer, _ := os.OpenFile("./dist.txt", os.O_WRONLY | os.O_CREATE, 0666)
        defer reader.Close()
        defer writer.Close()
        io.Copy(writer, reader)
}
```

```
func main()  {
        readFile, _ := os.Open("./src.txt")
        writeFile, _ := os.OpenFile("./dist.txt", os.O_WRONLY | os.O_CREATE, 0666)
        defer readFile.Close()
        defer writeFile.Close()
        reader := bufio.NewReader(readFile)
        writer := bufio.NewWriter(writeFile)
        for {
                val, err := reader.ReadString('\n')
                writer.WriteString(val)
                if err == io.EOF {
                        break
                }
        }
        writer.Flush()
}
```

JSON序列化与反序列化

JSON作为重要的数据交换格式，是前后端通信的重要媒介。Go中也提供了encoding/json包来提供对JSON的操作，我们来举个完整的例子：

```
type User struct {
        Name    string `json:"name"`
        Age     int    `json:"age"`
        Address string `json:"address"`
}

func main() {
        user1 := &User{
                "jack",
                20,
                "China",
        }
        buf, err := json.Marshal(user1)
        if err != nil {
                log.Fatal(err)
        }
        fmt.Println(string(buf))
```

```
        var user2 User
        err = json.Unmarshal(buf, &user2)
        if err != nil {
                log.Fatal(err)
        }
        fmt.Println(user2)
}
```

上面是一个完整的json序列化和反序列化的例子。

- 序列化：我们创建了一个User类型的user1，然后使用json.Marshal()将user1序列化成[]byte类型的buf变量，可以进一步将该[]byte类型的变量转换成json的string
- 反序列化：借助json.Unmarshal()，将[]byte类型反序列化成User类型的变量。这里需要注意的是，json.Unmarshal()得第二个参数需要为指针，不然不会改变user的值。

上面的例子，主要是跟内存相关的json操作。很多时候，我们需要从文件读取，然后反序列化成GO中的数据结构，或者反过来，将Go中的数据结构序列化到文件中去，所以我们来看看第二个

```
func main()  {
        file, err := os.OpenFile("./user.txt", os.O_WRONLY | os.O_CREATE, 0666)
        if err != nil {
                log.Fatal(err)
        }
        defer file.Close()
        user1 := User{
                "jack",
                20,
                "China",
        }
        encoder := json.NewEncoder(file)
        encoder.Encode(user1)
}
```

```
func main()  {
        file, err := os.Open("./user.txt")
        if err != nil {
                log.Fatal(err)
        }
        defer file.Close()
        var user User
        decoder := json.NewDecoder(file)
        err = decoder.Decode(&user)
        if err != nil {
                log.Fatal(err)
        }
        fmt.Println(user)
}
```

编辑于 2019-03-16 12:48

Go 语言

编程

后端技术