Go基础进阶

反射机制

反射reflect

反射可以在运行时动态获取各种变量的信息，比如变量的类型和值等

如果是结构体还可以获取到结构体本身的各种信息，比如结构体的字段和方法

通过反射还可以修改变量的值。

为什么要用反射

需要反射的 2 个常见场景：

1. 有时你需要编写一个函数，但是并不知道传给你的参数类型是什么，可能是没约定好；也可能是传入的类型很多，这些类型并不能统一表示。这时反射就会用的上了。
2. 有时候需要根据某些条件决定调用哪个函数，比如根据用户的输入来决定。这时就需要对函数和函数的参数进行反射，在运行期间动态地执行函数。

但是对于反射，还是有几点不太建议使用反射的理由：

1. 与反射相关的代码，经常是难以阅读的。在软件工程中，代码可读性也是一个非常重要的指标。

2. Go 语言作为一门静态语言，编码过程中，编译器能提前发现一些类型错误，但是对于反射代码是无能为力的。所以包含反射相关的代码，很可能会运行很久，才会出错，这时候经常是直接 panic，可能会造成严重的后果。
3. 反射对性能影响还是比较大的，比正常代码运行速度慢一到两个数量级。所以，对于一个项目中处于运行效率关键位置的代码，尽量避免使用反射特性。

认识反射 Go 复制代码

```go
package main

import (
    "fmt"
    "reflect"
)

func main() {
    var a float64
    a = 3.14
    //通过反射可以获取一个变量的  类型和数值
    fmt.Println(reflect.TypeOf(a))
    fmt.Println(reflect.ValueOf(a))

    //根据反射的值  来获取对应的数值和类型
    v := reflect.ValueOf(a)
    //种类
    if v.Kind() == reflect.Float64 { //Kind种类 相当于 struct  Type类型相当于User类
        fmt.Println("v是一个float64类型")
    }
    fmt.Println("类型是: ", v.Type())
    fmt.Println("数值是: ", v.Float())
}
```

反射获取变量的信息

Go | 复制代码

```go
package main

import (
    "fmt"
    "reflect"
)

type User struct {
    Name string
    Age  int
    Sex  string
}

func (user User) Say(msg string) {
    fmt.Println("User说: ", msg)
}

func (user User) PrintInfo() {
    fmt.Printf("名字: %s 年龄: %d 性别%s", user.Name, user.Age, user.Sex)
}
func main() {
    user := User{
        Name: "小王",
        Age:  19,
        Sex:  "男",
    }
    reflectGetInfo(user)

}

// 反射获取结构体信息
func reflectGetInfo(input interface{}) {
    getType := reflect.TypeOf(input)
    fmt.Println("类型: ", getType.Name()) //类型:  User
    fmt.Println("种类: ", getType.Kind()) //种类:  struct  不管是什么结构，类型都是结构体
    //获取值
    getValue := reflect.ValueOf(input)
    fmt.Println(getValue)

    //获取字段和字段值
    //遍历结构体中字段的个数
    for i := 0; i < getType.NumField(); i++ {
        field := getType.Field(i) //分别得到每一个字段的名称 类型
        fmt.Println(field)
```

```
45
46          // Interface  获取对应的值
47          value := getValue.Field(i).Interface() //获取得到每一个字段的值
48          fmt.Printf("字段名：%s 字段类型： %s 字段数值： %v\n", field.Name, fiel
    d.Type, value)
49      }
50
51      //这里有问题！！！！！！！！（解决  再一个包中main方法外的方法名大写才可以被main内
    部代码得到）
52      for i := 0; i < getType.NumMethod(); i++ {
53          method := getType.Method(i)
54          fmt.Println(method)
55          fmt.Printf("方法名称：%s 方法类型：%v\n", method.Name, method.Type)
56      }
57  }
```

反射修改对应变量的值

修改对应的值 Go 复制代码

```go
package main

import (
    "fmt"
    "reflect"
)

func main() {
    var num float64 = 2.13
    fmt.Println(num)

    //newValue := reflect.ValueOf(num) // 这个方法所得到的值不能继续修改
    pointer := reflect.ValueOf(&num) //得到上面num的指针
    newValue := pointer.Elem()       // 得到指针的值

    fmt.Println(newValue.Type())
    fmt.Println(newValue.Kind())
    fmt.Println(newValue.CanSet()) //判断是否可以改变

    if newValue.CanSet() {
        newValue.SetFloat(3.14) //设置值的时候一定要设定他的类型
    }
    fmt.Println(num)
}

```

修改对象字段

修改对象字段 Go 复制代码

```go
package main

import (
   "fmt"
   "reflect"
)

type User1 struct {
   Age  int
   Name string
}

func main() {
   user1 := User1{18, "小王"}

   //得到对象的指针
   value := reflect.ValueOf(&user1)
   //如果value是一个指针的话
   if value.Kind() == reflect.Pointer {
      newValue := value.Elem() //得到指针的值
      if newValue.CanSet() {
         newValue.FieldByName("Name").SetString("小城")
      }
   }
   fmt.Println(user1)
}
```

通过反射调用方法

通过反射调用方法 Go 复制代码

```go
package main

import (
    "fmt"
    "reflect"
)

type User2 struct {
    Name string
    Age  int
    Sex  string
}

func (user User2) Say(msg string) {
    fmt.Println("User说: ", msg)
}

func (user User2) PrintInfo() {
    fmt.Printf("名字: %s 年龄: %d 性别%s\n", user.Name, user.Age, user.Sex)
}

func func3(i int, s string) string {
    fmt.Println(i, s)
    return s
}
func main() {

    user := User2{
        Name: "小王",
        Age:  18,
        Sex:  "女",
    }
    value := reflect.ValueOf(user)
    //无参方法的调用
    value.MethodByName("PrintInfo").Call(nil) //用反射得到的对象调用方法

    //有参方法的调用
    args := make([]reflect.Value, 1)//创建一个容量为1的切片
    args[0] = reflect.ValueOf("有参函数的调用来了")
    value.MethodByName("Say").Call(args)//找到对应方法的名字 有参数就在call中加入参数

    //有参有返回值的函数调用
    v3 := reflect.ValueOf(func3)
    args2 := make([]reflect.Value, 2)
```

```
    args2[0] = reflect.ValueOf(1)
    args2[1] = reflect.ValueOf("hahhaha")
    result := v3.Call(args2)            //call方法返回的切片  result接收返回值
    fmt.Println(result[0].Interface()) //通过切片下标的interface方法获取值
}
```

Go常用包

包的声明和使用

1.默认模式：导入系统包使用

2.包的别名：可以给导入的系统包前面加别名

3.简便模式：可以在导入的系统包前面加点可以直接使用该包下的方法

4.可以匿名导入：仅让该包执行init函数 在导入的包前面加“_”

init函数

init函数先于main函数执行，实现包级别的一些初始化操作。

在mian包下导入另一个包时，如果该包下有init函数（没有接着导入包）就会先执行这个包下的init函数，再执行main方法里面的函数。如果在main方法的上面有init函数的话，就会执行导入的包中的init函数，再main方法上面的函数，最后再是main方法里面程序。

在mian包下导入另一个包时，如果该包下有init函数（有接着导入包），就会先执行导入的包下的init函数，再一次类推的找下去，直到最后一个包下没有init函数，再依次向前执行程序，同上。

Go | 复制代码

```go
package main

import (
    "fmt"
    _ "gomod/lessonPackage/test"// 匿名导入  就会先看里面有没有init函数
)

// init不需要传入参数，也不需要返回值
// 与main方法相比，init没有被声明，因此也不能被调用 （意思就是不能再main方法中用init()的方式调用）
func init() {
    fmt.Println("main========1")
}

// main方法会先执行导入的包下的init文件，如果这个init方法的导的包下还有init就继续先执行后者
// 先执行最后一层导的包文件
func main() {
    //执行这段语句之前先执行导入的包中的函数
    fmt.Println("main")
}

```

strings包的常用函数

Go | 复制代码

```go
package main

import (
    "fmt"
    "strings"
)

func main() {
    //定义一个字符串
    str := "xiaowang,shishuaige"

    //Contains  会返回bool值  是精准匹配 如2  就是必须要有"ax"这个字符
    fmt.Println(strings.Contains(str, "x"))  //true
    fmt.Println(strings.Contains(str, "ax")) // false

    //ContainsAng  模糊匹配只要有一个就行  以后常用
    fmt.Println(strings.ContainsAny(str, "ax")) //true

    //Count  统计一个字符出现的次数
    fmt.Println(strings.Count(str, "a")) //3

    //HasPrefix  文件以什么开头
    file := "2023#@4.mp4"
    if strings.HasPrefix(file, "2023") {
        fmt.Println(file)
    }
    //HasSuffix  文件以什么结尾
    if strings.HasSuffix(file, ".mp4") {
        fmt.Println("文件是.mp4格式的")
    }

    //Index  寻找指定字符串 第一次 出现的位置，有就返回下标，没有就返回-1
    fmt.Println(strings.Index(str, "a"))

    //IndexAny  寻找指定字符串任意字符 第一次 出现的位置，有就返回下标，没有就返回-1
    fmt.Println(strings.IndexAny(str, "az")) //有a无z  返回的是a的最后一次出现的下标

    //LastIndex  寻找指定字符串 最后一次 出现的位置，有就返回下标，没有就返回-1
    fmt.Println(strings.LastIndex(str, "e"))

    //LastIndexAny  寻找指定字符串任意字符 最后一次 出现的位置，有就返回下标，没有就返回-1
    fmt.Println(strings.LastIndexAny(str, "ez")) //有e无z  返回的是e的最后一次出现的下标
```

```
43
44
45      //Join  将每个字符中间加入指定字符
46      str2 := []string{"a", "c", "f", "j"}
47      str3 := strings.Join(str2, "-")
48      fmt.Println(str3)
49
        //Split  按照指定符号分割字符串  如果没有找到指定的分割符号  就会直接返回原来字符
    串的格式
50      str4 := strings.Split(str3, "-")
51      fmt.Println(str4)
52
53      //Repeat  重复拼接自己
54      str5 := strings.Repeat("sx", 3)
55      fmt.Println(str5)
56      //Replace 替换 n代表替换的次数  n < 0就是所有的都替换
57      str6 := strings.Replace(str, "x", "*", -1)
58      fmt.Println(str6)
59      //ToUpper   字母转大写  ToLower  全部转化成小写
60      str7 := strings.ToUpper(str)
61      str8 := strings.ToLower(str)
62      fmt.Println(str7)
63      fmt.Println(str8)
64
65      //截取字符串  str[start, end]  所有都是左闭右开
66      str9 := str[0:5]
67      fmt.Println(str9)
68      str10 := str[5:]
69      fmt.Println(str10)
70
71  }
```

strconv常用函数

Go | 复制代码

```go
package main

import (
    "fmt"
    "strconv"
)

func main() {
    str1 := "23"
    //atoi  itoa(int转字符串)
    i, _ := strconv.Atoi(str1)
    fmt.Println(i)
    //将字符串转成bool
    str2 := "true"
    str3, _ := strconv.ParseBool(str2)
    //将bool转换成字符串
    s1 := strconv.FormatBool(str3)
    fmt.Println(s1)

    // 传入字符串  转换的进制数（就是让机器认为你给的数字2进制还是其他的格式）    输出的类型（bitSize）
    s2 := "100"
    i1, _ := strconv.ParseInt(s2, 2, 64)
    fmt.Println(i1) //4  以10进制的格式输出
    //传入一个int64的数  base转换的进制数
    i2 := 200
    str4 := strconv.FormatInt(int64(i2), 2)
    fmt.Println(str4)

    //将字符串转换成int
    i3, _ := strconv.Atoi("100")
    fmt.Printf("%T\n", i3)
    fmt.Println(i3 + 10)

    //将int转换成字符串
    str5 := strconv.Itoa(200)
    fmt.Printf("%T\n", str5)
    fmt.Println(str5)

}
```

时间与时间戳

时间的普通格式化

获取当前时间并格式化 Go 复制代码

```go
func time1() {
    now := time.Now()
    fmt.Println(now)

    year := now.Year()
    month := now.Month()
    day := now.Day()
    hour := now.Hour()
    minute := now.Minute()
    second := now.Second()
    fmt.Printf("%02d-%02d-%02d-%02d:%02d:%02d\n", year, month, day, hour, minute, second)
}

```

利用Go语言诞生时间格式化

Go 复制代码

```go
    //时间格式化    格式为Go语言诞生的时间 2006 01 02 13 14
    //口诀：2006 1234
    now1 := time.Now()
    fmt.Println(now1.Format("2006-01-02 15:04:20"))
    fmt.Println(now1.Format("2006-01-02 03:04:20 PM"))
    fmt.Println(now1.Format("2006-01-02 "))
```

解析字符串格式的时间

让字符串成为时间对象

Go | 复制代码

```go
    //解析字符串格式的时间
    now := time.Now()
    fmt.Println(now)
    //时区
    loc, _ := time.LoadLocation("Asia/Shanghai")
    //2006.01.02 15.04.05  必须写这个
    //让自己定义的时间成为一个时间对象  这样就可以对它进行一些时间的操作
    timeObj, _ := time.ParseInLocation("2006/01/02 15:04:05", "2022/01/03 14:27:27", loc)
    fmt.Println(timeObj)
    fmt.Println(timeObj.Second())
```

时间戳

时间戳是自1970年1月1日（08:00:00GMT）至当前时间的总毫秒数。它也被成为Unix时间戳。

Go | 复制代码

```go
    //将时间戳转化为时间对象    有了对象就可以取值
    time1 := time.Unix(timestamp, 0)
    fmt.Println(time1) //输出成时间的格式
    fmt.Println(time1.Year())
```

随机数

Go | 复制代码

```go
package main
import (
    "fmt"
    "math/rand"
    "time"
)
func main() {
    rand.Seed(time.Now().Unix()) // 种子中必须传入一个数值才可以生成数字
    for i := 0; i < 100; i++ {
        //rand.int 生成的数字很大  不建议使用
        num := rand.Intn(100) //返回100以内的数字[0,99]!    如果是Int 就是非常大的一个数字
        //需求 生成20-29的数字
        // num:=rand.Intn(10) + 20
        fmt.Println("num = ", num)
    }
}
```

定时器

Go | 复制代码

```go
package main

import (
    "fmt"
    "time"
)

func main() {
    //真正的定时器
    //Tick里传入的参数是Duration  指的是时间的一个间隔  返回一个时间
    t1 := time.Tick(time.Second) //间隔一秒执行一次
    for t := range t1 {
        fmt.Println(t) //时间间隔一秒就输出当前时间
    }

    //用sleep模拟的定时器
    //执行10次  每一次都休息一秒钟
    for i := 0; i < 10; i++ {
        fmt.Println(time.Now())
        time.Sleep(time.Second)
    }
}

```

时间运算

Go | 复制代码

```go
package main

import (
    "fmt"
    "time"
)

func main() {
    //时间相加
    now := time.Now()
    latter := now.Add(time.Hour)
    fmt.Println(now)
    fmt.Println(latter)

    //算两个时间的时间差值
    subtime := latter.Sub(now)
    fmt.Println(subtime) //1h

    //比较两个时间是否相等  返回bool
    fmt.Println(now.Equal(now))
    //判断一个时间是不是在他后面   返回 bool
    fmt.Println(now.After(latter))
}
```

GoIO

获取文件信息

Go | 复制代码

```go
package main

import (
    "fmt"
    "os"
)

func main() {
    //相对路劲 ./lessonGoIO/a.txt
    //返回的是文件  不能对他进行读写操作  只能查看
    fileInfo, err := os.Stat("./lessonGoIO/a.txt")
    if err != nil {
        fmt.Println(err)
        return
    }
    fmt.Println(fileInfo.Name())
    fmt.Println(fileInfo.Mode())    //文件读写属性
    fmt.Println(fileInfo.IsDir())   //是否是一个文件
    fmt.Println(fileInfo.Size())    //文件的大小
    fmt.Println(fileInfo.ModTime()) //文件修改的时间
    //反射获取文件更加详细的信息
    fmt.Println(fileInfo.Sys())
}

```

创建目录与文件

Go | 复制代码

```go
package main

import (
    "fmt"
    "os"
)

func main() {
    //os.ModePerm    给文件设置权限 777:    所有操作文件的人都可读可写可执行
    //这种方法只能先找到最后一个文件的前一个文件 如果能找到就创建   没找到就不创建   例如下面的层级目录是不能创建的
    err := os.Mkdir("/Users/dean/GoWorks/src/gomod/lessonGoIO/txtaa", os.ModePerm)
    if err != nil {
        fmt.Println(err)
        return
    }

    //创建多层目录
    err = os.MkdirAll("/Users/dean/GoWorks/src/gomod/lessonGoIO/txtbb/a/a/a", os.ModePerm)
    if err != nil {
        fmt.Println(err)
        return
    }
    fmt.Println("文件创建完毕")

    //RemoveAll  删除一个文件夹下的所有文件
    //Remove     删除一个指定的文件 文件里面不能有其他的东西
    err2 := os.Remove("/Users/dean/GoWorks/src/gomod/lessonGoIO/txtaa")
    if err2 != nil {
        fmt.Println(err)
        return
    }
    //启动该语句需要注释上面的程序内容
    //创建一个文件   如果此时已经存在该文件，并且文件内还有内容就是重新创建文件   此时文件内容为空
    file, err3 := os.Create("/Users/dean/GoWorks/src/gomod/lessonGoI O/b.txt")
    //此时返回的file是一个指针对象   可以用它来操作指针
    fmt.Println(file.Name())
    fmt.Println(err3)
}

```

IO读

Go | 复制代码

```go
package main

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

func main() {

    //与文件建立连接  返回的是指针对象 可以对文件进行读写操作  Open就是仅仅是打开
    file, err := os.Open("./lessonGoIO/a.txt")
    if err != nil {
        fmt.Println(err)
        return
    }
    fmt.Println(file.Name())
    //关闭连接
    defer file.Close()

    //对文件进行读操作
    //创建切片 大小为2 容量为1024
    s1 := make([]byte, 2, 1024)
    n, err4 := file.Read(s1) //此时n是读到的个数（这个个数就是[]byte 里面设置的大小）
    fmt.Println(n)

    //将文件中的内容用字符串并以切片的大小 2展示出来
    fmt.Println(string(s1))
    //可以一直打印  当超过文件中字符的大小时就会报错  此时读到了文件末尾 err = io.EOF
    //如果读到的n > 0 时err就为nil （意思就是读到了数据 即使没有装满整个切片）
    //没有读到内容 就返回EOF  （说明没有数据可以读了）
    fmt.Println(err4)

    //OpenFile  （文件名，打开方式，文件权限）
    //如果用上面的打开方式不能可读可写 我们就要设置打开方式
    file2, err2 := os.OpenFile("./lessonGoIO/a.txt", os.O_RDONLY|os.O_WRONLY, os.ModePerm)
    if err2 != nil {
        fmt.Println(err2)
        return
    }
    fmt.Println(file2.Name())
    defer file.Close()
```

```
}
```

IO写

```
C | 复制代码
```

```
package main

import (
    "fmt"
    "os"
)

func main() {
    //向文件中写入数据  O_RDONLY  可读  O_WRONLY  可写  O_APPEND  向文件后面直接追加写入的文件
    file2, err2 := os.OpenFile("./lessonGoIO/a.txt", os.O_RDONLY|os.O_WRONLY|os.O_APPEND, os.ModePerm)
    if err2 != nil {
        fmt.Println(err2)
        return
    }
    defer file2.Close()

    //业务代码写入数据    开发中用切片用的更多
    bs := []byte{66, 67, 68, 67}
    n, err := file2.Write(bs)
    if err != nil {
        fmt.Println(err)
        return
    }
    fmt.Println(n)

    //这里用的=赋值  因为前面已经都有n 和 err 了
    n, err = file2.WriteString("hahahha,小可爱")
    if err != nil {
        fmt.Println(err)
        return
    }
    fmt.Println(n)
}

```

文件复制

文件复制的实现 Go 复制代码

```go
package main

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

func main() {
    source := "/Users/dean/GoWorks/src/gomod/lessonGoIO/aaa/王定勇_Java后端开发.png"
    //复制过来的源文件  必须也有文件名
    destination := "/Users/dean/GoWorks/src/gomod/lessonGoIO/xk-copy3.png"
    //copy(source, destination, 1024)
    copy2(destination, source)
}

func copy3(destination, source string) {
    //不建议使用这种创建文件   因为可能一个文件太大  直接把buffer撑爆了  导致内存溢出
    buffer, _ := os.ReadFile(source)
    err := os.WriteFile(destination, buffer, 0777)
    fmt.Println(err)

}

func copy2(destination, source string) {
    //输入文件  输出的file对象
    sourceFile, err := os.Open(source)
    if err != nil {
        fmt.Println(err)
        return
    }
    defer sourceFile.Close()

    //连接输出文件   可能目标文件是不存在的所以使用O_CREATE 不存在的话就创建
    destinationFile, err := os.OpenFile(destination, os.O_WRONLY|os.O_CREATE, os.ModePerm)
    if err != nil {
        fmt.Println(err)
        return
    }
    defer destinationFile.Close()

    written, _ := io.Copy(destinationFile, sourceFile)
    fmt.Println("文件大小：", written)
```

```
}

func copy(source, destination string, bufSize int) {
    //输入文件
    sourceFile, err := os.Open(source)
    if err != nil {
        fmt.Println(err)
        return
    }
    defer sourceFile.Close()

    //连接输出文件    可能目标文件是不存在的所以使用O_CREATE 不存在的话就创建！！！
    destinationFile, err := os.OpenFile(destination, os.O_WRONLY|os.O_CREATE, os.ModePerm)
    if err != nil {
        fmt.Println(err)
        return
    }
    defer destinationFile.Close()

    //缓冲区
    buf := make([]byte, bufSize)

    for {
        //读取
        n, err := sourceFile.Read(buf)
        if err == io.EOF || n == 0 {
            fmt.Println("文件复制完毕")
            break
        }
        //写出
        _, err = destinationFile.Write(buf[:n])
        if err != nil {
            fmt.Println("写入失败", err)
        }
    }
}
```

seeker

Go | 复制代码

```go
package main

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

func main() {

    file, _ := os.OpenFile("/Users/dean/GoWorks/src/gomod/lessonGoIO/a.txt", os.O_RDWR, os.ModePerm)
    file.Seek(2, io.SeekStart)
    //括号里面的值 是用来存入新读入的值的 有几个数就在offset后面读几个
    buf := []byte{0, 2, 0, 5}
    file.Read(buf)
    fmt.Println(string(buf))

    //当前光比的末尾因该是第六个字母  需要从第七个数字开始读
    file.Seek(2, io.SeekCurrent)
    file.Read(buf)
    fmt.Println(string(buf))

    //光标在文件末尾  再写入字符  相当于 append
    file.Seek(0, io.SeekCurrent)
    file.WriteString("hahahhaha,小可爱")

}

```

断点续传

Go | 复制代码

```go
 package main

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

func main() {
    //传输的文件
    srcFile := "/Users/dean/GoWorks/src/gomod/lessonGoIO/aaa/王定勇_Java后端开发.png"
    //传输目的地
    destFile := "/Users/dean/GoWorks/src/gomod/lessonGoIO/xk-seek3.png"
    //临时记录文件
    tempFile := "/Users/dean/GoWorks/src/gomod/lessonGoIO/xk-temp.txt"
    //与文件创建连接
    file1, _ := os.Open(srcFile)
    file2, _ := os.OpenFile(destFile, os.O_CREATE|os.O_RDWR, os.ModePerm)
    file3, _ := os.OpenFile(tempFile, os.O_CREATE|os.O_RDWR, os.ModePerm)
    defer file1.Close()
    defer file2.Close()

    //1、读取临时文件记录的位置
    file3.Seek(0, io.SeekStart)     //找到光标
    buf := make([]byte, 1024, 1024) //创建一个缓冲区间
    n, _ := file3.Read(buf)         //n为读到的个数
    countStr := string(buf[:n])
    fmt.Println("countStr: ", countStr) //输出之前临时文件中存入的数据

    //ParseInt(字符串，进制数，类型)   将上面存储的string转换为int类型
    count, _ := strconv.ParseInt(countStr, 10, 64)
    //seek  找到光标位置
    file1.Seek(count, io.SeekStart)
    file2.Seek(count, io.SeekStart)

    bufData := make([]byte, 1024, 1024)
    total := int(count) //记录所有读到的count

    for {
        readnum, err := file1.Read(bufData) //readnum是读到的数据
        if err == io.EOF {
            fmt.Println("数据读取完毕")
            file3.Close()
```

```
            os.Remove(tempFile)
            break
        }
        writeNum, err := file2.Write(bufData[:readnum])
        total = total + writeNum //total是还需要写的数据

        //将文件的进度存储到临时文件当中
        file3.Seek(0, io.SeekStart)             //找到临时文件的光标位置
        file3.WriteString(strconv.Itoa(total)) //将之前存储的int类型的总字符数写出来
    }

    //模仿程序异常
    if total > 6000 {
        panic("断电")
    }

}
```

bufio

Go | 复制代码

```go
package main

import (
    "bufio"
    "fmt"
    "os"
)

func main() {
    //连接文件
    file, _ := os.OpenFile("/Users/dean/GoWorks/src/gomod/lessonGoIO/a.txt", os.O_RDWR, os.ModePerm)

    //读文件
    reader := bufio.NewReader(file)

    //缓存区
    buf := make([]byte, 1024)
    //将读的内容放放缓存区
    n, _ := reader.Read(buf)
    fmt.Println(n)                //输出的是字符的大小
    fmt.Println(string(buf[:n])) //将buf里面从0到n的所有字符输出

    /*inputReader := bufio.NewReader(os.Stdin)
    str, _ := inputReader.ReadString('\n') //   \n为最后一个字符   回车键
    fmt.Println("我输出的键盘是", str)*/

    writer := bufio.NewWriter(file) //写文件   返回一个写的对象
    //要开启文件的各种权限    写的内容需要超过buf的大小   这些数据才会写入文件   要不然就flush   强制写入
    writerNum, _ := writer.WriteString("hello")
    fmt.Println(writerNum)
    writer.Flush()
}
```

遍历目录

```go
package main

import (
    "fmt"
    "os"
)

func main() {
    listDir("/Users/dean/GoWorks/src/gomod", 0)

}
func listDir(filepath string, tabint int) {
    dir := filepath
    tab := "|--"
    for i := 0; i < tabint; i++ {
        tab = "|    " + tab  // 加入一个层级显示
    }
    dirInfor, _ := os.ReadDir(dir)
    for _, file := range dirInfor {
        fileName := dir + "/" + file.Name()//名字为总路径+文件名
        fmt.Printf("%s %s\n", tab, fileName)
        if file.IsDir() {
            //如果file还是一个文件   接着调用函数
            listDir(fileName, tabint+1)
        }

    }
}

```

Go并发编程

进程 线程 协程

进程：进程就是一个静态的程序执行后成为动态。

线程：好比于一个主程序中方法的执行。

协程：就是主程序和协程一起执行。

Goroutine

规则：

1、当新的Goroutine开始时，Goroutine调用立即返回。与函数不同，go不等待Goroutine执行结束。

2、当Goroutine被调用，并且Goroutine任何的返回值被忽略之后，go立即执行到下一行代码。（例如一个方法前面加go， 程序就不会等待这个方法结束，自己会接着向下执行其他的代码）

3、main的Goroutine应该为其他的Goroutines执行，如果main的Goroutine终止了，程序将被终止，其他的Goroutine将不会执行。

Goroutine

Go | 复制代码

```go
    package main
import (
    "fmt"
    "runtime"
)
func main() {
    fmt.Println(runtime.GOROOT()) // 获取goroot目录/usr/local/go
    fmt.Println(runtime.GOOS)     //获取操作系统信息 darwin
    fmt.Println(runtime.NumCPU())  //获取cpu的个数
    go hello() //再函数前面加go就可以实现
    for i := 0; i < 100; i++ {
        fmt.Println("========main", i)
    }
}

func hello() {
    for i := 0; i < 100; i++ {
        runtime.Gosched()   // 让出时间片     让其他的代码先执行
        fmt.Println("======hello", i)

    }
}
```

Gorountine调度与终止

runtime.Gosched()　　// 让出时间片　让其他的代码先执行

return　终止函数

runtime.Goexit()　终止当前的Goroutine

```go
package main

import (
    "fmt"
    "runtime"
    "time"
)

func main() {

    go func() {
        fmt.Println("======main")
        test()
        fmt.Println("======end")
    }()
    time.Sleep(time.Second * 2)
}
func test() {
    defer fmt.Println("test  defer")
    //return //终止函数（本代码块的后面代码不执行）
    runtime.Goexit() //终止协程  让该语句后面的程序都不执行
    fmt.Println("test====")
}
```

临界资源安全问题

同时被进程，线程，协程抢用的资源就是临界资源

Go | 复制代码

```go
package main

import (
    "fmt"
    "time"
)

func main() {
    a := 2
    go func() {
        a = 3
        fmt.Println("a:", a) //a被赋值为3
    }()
    a = 4
    fmt.Println("a:", a) //本来a=4  再主程序睡眠等待 协程（goroutine）里面的程序执行
    time.Sleep(3 * time.Second)
    fmt.Println("a:", a) //这时a资源被协程抢夺  a为3
}

```

```go
package main

import (
    "fmt"
    "time"
)

var ticket int = 10  //定义总票数
func main() {
    go sale("售票口1")
    //go sale("售票口2")
    //go sale("售票口3")
    //go sale("售票口4")

    time.Sleep(time.Second * 5)  //需要让主程序睡一下  要不然太快协程还没有执行
}
func sale(name string) {
    for {
        if ticket > 0 {
            time.Sleep(time.Millisecond * 500)
            fmt.Println(name, "剩余票数", ticket)
            ticket--
        } else {
            fmt.Println("卖光了")
            break  //结束循环
        }
    }
}

```

互斥锁 和 waitgroup等待组

锁：var mutex sync.Mutex

等待组：var wg2 sync.WaitGroup

Add() 添加协程的个数

Done() 一个协程结束add里面的个数就减一

Wait() 当Add里面的个数减的为1时 程序就自动执行完毕

使用等待组的前提是你已经知道你有几条协程！！

Go | 复制代码

```go
package main

import (
    "fmt"
    "sync"
    "time"
)

// 创建锁
var mutex sync.Mutex
var wg2 sync.WaitGroup
var ticket = 10

func main() {
    wg2.Add(4)
    //当多个协程抢夺一个资源
    go saleTickets("收票口1")
    go saleTickets("收票口2")
    go saleTickets("收票口3")
    go saleTickets("收票口4")
    wg2.Wait()
    //让住程序多睡一会 ， 要不然会出现 资源还没分配完  程序就结束的问题
    //time.Sleep(time.Second * 10)

}

// 售票函数
func saleTickets(name string) {
    defer wg2.Done()
    for {
        //打开锁 每个程序调用是会看一看 有没有人正在用这个资源  必须在for循环执行之后加锁
        mutex.Lock()
        if ticket > 0 {
            time.Sleep(time.Millisecond * 200)
            fmt.Println(name, "剩余票数：", ticket)
            ticket--
        } else {
            //操作完毕解锁
            mutex.Unlock()
            fmt.Println("票卖光了")
            break //跳出循环
        }
        //操作完毕解锁   必须在for循环结束之前解锁
        mutex.Unlock()
```

```
    }
}

```

channel通道

一个通道发送和接收数据，默认是阻塞的。当一个数据数据被发送到通道时，发送语句被阻塞，直到另一个Goroutine从通道中读取数据

相对的，当通道读取数据时，读取被阻塞，直到另一个Goroutine将数据写入通道

本身channel就是同步的，意味着同一时间，只有一条Goroutine来操作

最后：通道是Goroutine之间的连接，所以通道的发送和接收必须处在不同的Goroutine中

这些通道的特性是帮助Goroutines有效的进行通信，而无需向其他编程语言中非常常见的显示锁和条件变两个

Go | 复制代码

```go
package main

import (
    "fmt"
)

// 不要通过共享内存来通信，要通过通信的方式来共享内存
func main() {

    //定义一个布尔类型的通道
    var ch chan bool
    ch = make(chan bool)
    go func() {
        for i := 0; i < 10; i++ {
            fmt.Println("test", i)
        }
        //向通道写入数据 表示要结束了
        ch <- true
    }()
    //将通道的数据写入data
    date := <-ch
    fmt.Println(date)

}

```

关闭通道

Go | 复制代码

```go
package main

import (
    "fmt"
    "time"
)

func main() {
    //定义一个通道
    ch := make(chan int) //make（类型 放的数据类型, 容量）
    //将通道打开并传入值
    go test3(ch)
    /*for {
        time.Sleep(time.Second * 1)
        v, ok := <-ch
        //通道没关闭之前 ok为true  之后为false执行下面语句
        if !ok {
            fmt.Println("通道已经关闭", ok)
            break
        }
        //如果为true 就一直输出通道里面的数据
        fmt.Println("v:", v)

    }*/
    //更加方便的传入值
    for v := range ch {
        fmt.Println("v:", v)
    }
}

func test3(ch chan int) {
    //写数据到通道
    for i := 0; i < 10; i++ {
        time.Sleep(time.Second * 1)
        ch <- i
    }

    //当for中遍历结束 关闭通道
    close(ch)
}

```

缓冲通道

当我们遇到一次性要接收很多数据，但可以慢慢处理这些数据时，就可以使用缓冲通道。例如消息队列

Go | 复制代码

```go
package main

import (
    "fmt"
    "strconv"
    "time"
)

func main() {
    ch1 := make(chan int, 4)
    fmt.Println(len(ch1), cap(ch1))

    ch2 := make(chan int, 4)
    fmt.Println(len(ch2), cap(ch2))
    ch2 <- 2
    ch2 <- 3
    //当写入的数据 超过长度时    会报错！！
    fmt.Println(len(ch2), cap(ch2))

    ch3 := make(chan string, 5)
    //test4中是写入10个数据  我们缓存的大小是5
    go test4(ch3)
    //所以咱们先写入5个数据 再开始读出数据
    for v := range ch3 {  //使用forange遍历更加方便
        time.Sleep(time.Second)
        fmt.Println("v:", v)
    }

}

func test4(ch chan string) {
    for i := 0; i < 10; i++ {
        ch <- "test-" + strconv.Itoa(i)
        //由于上面定义的chan大小为5所以程序会一下子就写入5个  然后再慢慢读写
        fmt.Println("开始写入数据", "test-"+strconv.Itoa(i))
    }
    close(ch)  //写入数据之后必须关闭通道
}

```

定向通道

Go 复制代码

```go
package main

import (
    "fmt"
    "time"
)

//封装的思想 一个方法只做一件事
func main() {

    ch3 := make(chan int)
    go writeOnly(ch3)
    go readOnly(ch3)
    time.Sleep(time.Second * 3)

}
func writeOnly(ch chan<- int) {
    ch <- 100
}
func readOnly(ch <-chan int) {
    data := <-ch
    fmt.Println(data)
}

```

select

每个case语句都必须是一个通道操作

如果任意一个case都能执行，select会随机选择一个执行，其他的被忽略

有default就会先执行default 与Switch有区别

Go | 复制代码

```go
package main

import (
    "fmt"
    "time"
)

func main() {
    ch1 := make(chan int)
    ch2 := make(chan int)

    go func() {
        ch1 <- 100
    }()

    go func() {
        time.Sleep(time.Second)
        ch2 <- 200
    }()

    //当有多个协程同时进行  select会随机匹配执行一个
    //必须是协程才可以用select  case中也必须是chan类型
    select {
    case num1 := <-ch1:
        fmt.Println("num1", num1)
    case num2 := <-ch2:
        fmt.Println("num2", num2)

        //有default就会先执行default  与Switch有区别
    /*default:
    fmt.Println("我先执行")*/

    }

}

```

定时器

Go | 复制代码

```go
package main

import (
    "fmt"
    "time"
)

func main() {
    /*//time.NewTimer  == time.After
    timer := time.NewTimer(time.Second)
    fmt.Println(time.Now())
    timerchan2 := timer.C
    fmt.Println(<-timerchan2)

    timer2 := time.NewTimer(time.Second * 3)
    go func() {
        <-timer2.C //本来3S后停止  有stop函数 可以提前停止
        fmt.Println("子程序end。。。")
    }()
    time.Sleep(time.Second)
    stop := timer2.Stop()
    if stop { //停止或结束返回 false，否则返回true
        fmt.Println("timer2停止了")
    }*/
    //间隔多长时间后输出时间  例如：备份
    timechan := time.After(time.Second * 2)
    fmt.Println(time.Now())

    chantime := <-timechan
    fmt.Println(chantime)

    // 间隔一定时间后执行一个函数
    time.AfterFunc(time.Second*3, test8)

    time.Sleep(time.Second * 4)
}
func test8() {
    fmt.Println("time时间后执行")
}


```