Golang学习笔记

# 位运算细节

* **计算机中所有的运算都是按照【补码】来计算的**；
* 正数的原码，补码，反码就是正数本身；
* 负数的原码是其本身，反码是符号位不变，其他位取反，补码是反码加1

通过异或操作来说明：

func main() {  
 fmt.Println("a&3 = ", 2&3) //3  
 fmt.Println("a&3 = ", 2|3) //3  
 fmt.Println("a&3 = ", 2^3) //异或 1  
 fmt.Println("a^-3 = ", 2^-3) //-1  
}

上述的异或操作中，有负数的存在，着重说一下：

1. 2的补码就是正数2的本身 0000 0010
2. -3的补码是需要经过反码转换，而负数的反码需要从原码转换

* -3的原码：1000 0011
* -3的反码：1111 1100
* -3的补码：1111 1101 （负数的补码是经过负数的反码加1）

1. 因为计算是按照补码来的，所以2^-3异或操作时，有2的补码和-3的补码做异或：

* 2 0000 0010
* -3 1111 1101
* 2^-3 1111 1111

1. 上述计算结果为1111 1111是补码计算的结果，可以看书符号位为1，所以是一个负数，需要恢复到其原码，负数恢复到原码，需要经过符号位不变，补码减1，然后取反的过程：

1111 1111

减1 -> 1111 1110

取反-> 1000 0001 //-1

结果： -1

1. 所以 2^-3的结果就是-1

# 算数取余数的细节

## Golang的取余数奇数一个公式即可：

***a%b = a – (a/b) \* b***

就是先取得a/b得商，然后得到的这个商再乘以b, 然后用a – 商\*b

func main() {  
 //注意取余数后，余数的符号  
 fmt.Println("10%4 = ", 10%4) //2  
 fmt.Println("-10%4 = ", -10%4) // -2  
 fmt.Println("-10%-4 = ", -10%-4) // -2  
 fmt.Println("10%-4 = ", 10%-4) // 2  
}

特别对有负数的时候取余数的情况说一下：

-10%4 => -10-(-10/4)\*4 => -10 – (-2)\*4 => -10-(-8) => -10+8 => -2

-10%(-4)=> -10-(-10/-4)\*(-4）=> -10 – 2\*(-4) => -10 – (-8) => -10 + 8 => -2

10%(-4) => 10-(10/-4)\*(-4) => 10-（-2）\*(-4) => 10-8 = 2

# 类型转换细节

## 避免把变量从一个大范围的类型向小范围的类型转换

var num1 int64 = 99999  
var num2 int8 = int8(num1)  
//大范围的数转小范围的数，会有溢出，编译没问题，但是结果不可预料  
fmt.Println("num2:", num2) //num2的值是-97,不符合预期

输出如下：

num2: -97

因为num1是int64位的，num2是int8的，num1转num2是从大范围转到小范围了，会有溢出，虽然程序编译没有问题不会报错，但是转换结果不可预料。

**所以应当避免从一个大范围类型的数，向小范围的类型转换；**

## Go中不带类型自动转换，需要手动转换类型

var d1 int32 = 12  
var d2 int64  
var d3 int8  
  
//d2 = d1 + 20 //编译报错，无法将 'd1 + 20' (类型 int32) 用作类型 int64  
d2 = int64(d1) + 20  
//d3 = d1 + 20 //编译报错，无法将 'd1 + 20' (类型 int32) 用作类型 int8  
d3 = int8(d1) + 20 //不推荐大类型int32转小类型int8  
fmt.Println("d2:", d2, "d3:", d3)

结果为：

d2: 32 d3: 32

* go中没有类型自动转换，再不不加类型转换的时候，直接把d1+20的结果赋给int64的d3会有编译错误
* 在d1和常量20运算的时候加了int类型转换；
* d1和d2的类型不一样，一个int32一个int64，在对它们做计算时，需要手动转换类型。
* 转换时记得避免把大的类型向小的转换，要把小的类型向大的转换，所以转换int32的d1转为int64 再和20做运算；
* 常量10在int8的范围内,d1的值为12也在int8的范围内，所以在这里把int32转int8，然后计算也可以成功，但是正如前一条所说，避免把大的类型向小的转换，所以虽然结果正确，但是不推荐这么做；

## 计算的结果值的类型和数据的大小范围需要一致

//127在int8的取值范围类，但是计算结果会有溢出，编译不会报错，但是结果错误  
d3 = int8(d1) + 127  
fmt.Println("d3:", d3) //结果为-117 不符合预期  
  
//d3 = int8(d1) + 128 //编译直接报错因为128超过了int8的最大值

结果为：

d3: -117

接上一部分，d3为int8类型，但是，127已经达到int8的最大值，和d1相加的结果超出了int8的最大值范围，溢出了。结果不可预料。

# 把不同类型的数据转换成字符串——fmt.Sprintf

## 数字转字符串(%d)

var num1 int = 99

str = fmt.Sprintf("**%d**", num1)  
fmt.Printf("str的类型是 **%T**, str=**%q\n**", str, str) //使用%q可以给字符串使用”包裹起来

输出：

str的类型是 string, str="99"

使用%q的格式关键字，可以把字符串的结果用“”包起来显示；

## 浮点型转字符串（%f）

var num2 float32 = 23.223

str = fmt.Sprintf("**%f**", num2)  
fmt.Printf("str的类型是 **%T**, str=**%q\n**", str, str)

输出：

str的类型是 string, str="23.223000"

## 布尔型转字符串(%t)

var b bool = true

str = fmt.Sprintf("**%t**", b)  
fmt.Printf("str的类型是 **%T**, str=**%q\n**", str, str)

结果为：

str的类型是 string, str="true"

## byte转字符串（%c）

var mychar byte = 'c'

str = fmt.Sprintf("**%c**", mychar)  
fmt.Printf("str的类型是 **%T**, str=**%q\n**", str, str)

## strconv使用FormatInt转int为字符串——strconv.FormatInt

var num1 int = 99

str = strconv.FormatInt(int64(num1), 10) //第二个参数是进制，10就是转成10机制，2就是转成2进制  
fmt.Printf("str的类型是 **%T**, str=**%q\n**", str, str)

输出结果：

str的类型是 string, str="99"

## strconv使用Itoa转换int为字符串——strvconv.Itoa

var num2 float64 = 23.456

str = strconv.Itoa(int(num2)) //另外一种int转string方式，参数只能int类型  
fmt.Printf("str的类型是 **%T**, str=**%q\n**", str, str)

输出结果：

str的类型是 string, str="23"

使用strconv.Itoa的时候需要注意的是，**Itoa的入参只能为int类型，**所以如果需要转换的参数不是int类型那么需要转换为int类型。

## strconv转浮点型为字符串——strconv.FormatFloat

var num2 float64 = 23.456

str = strconv.FormatFloat(num2, 'f', 10, 64) //’f'表示格式，10表示精度，也就是保留多少位的小数，64表示数字为float64类型  
//除了'f'之外，其他格式如下：  
//'f'（-ddd.dddd）、'b'（-ddddp±ddd，指数为二进制）、'e'（-d.dddde±dd，十进制指数）、'E'（-d.ddddE±dd，十进制指数）、'g'（指数很大时用'e'格式，否则'f'格式）、'G'（指数很大时用'E'格式，否则'f'格式）。  
fmt.Printf("str的类型是 **%T**, str=**%q\n**", str, str)

结果：

str的类型是 string, str="23.4560000000"

* FormatFloat转换时，入参可以设置转化后的字符串以什么格式表示；
* 可以设置精度，也就是保留的小数后的位数；

# Byte和string的互相转换

Golang中，byte是另外一种整型，其实就是uint8

type **byte** = uint8

byte和string的转换实例如下：

var s string = "helo中国"   
fmt.Println("[]byte(s):", []byte(s)) //按照字节值输出数字  
  
var bs = []byte{97, 98, 99} // abc  
fmt.Println("string：", string(bs))

打印结果为：

[]byte(s): [104 101 108 111 228 184 173 229 155 189]

string： abc

* 字符串s转换为byte类型的字符串直接使用：[]byte(s)初始化就行，转换是把字符转换成了其ASCII码（按照字符串的形式）；
* byte类型的数组[]byte转换为string直接使用：string(bs)初始化即可，转换会把原来内容的ASCII码转换成字符串；

# 空接口对const类型的断言会失败

如下定义了一个const类型X,对其做类型断言:

const ***X*** = 7.0  
var x interface{} = ***X***//if y, ok := x.(int); ok {  
if y, ok := x.(float32); ok {  
 //const类型断言会失败，值y为0  
 fmt.Println("if")  
 fmt.Println(y)  
} else {  
 fmt.Println("else")  
 fmt.Println(int(y))  
}

打印结果：

else

0  
也就是断言结果没有走if成功的分支，走了断言失败的分支 else。

**结论：对于const类型的变量直接做类型断言会断言失败，断言结果y返回0。**

# 类型断言和实际的类型不匹配会panic

var a interface{}  
var f float32 = 1.1  
a = f //空接口可以接受任意的数据类型  
  
y := a.(float32)  
fmt.Printf("y的类型是:**%T\n**", y) //float32  
  
y2 := a.(float64)  
//panic: interface conversion: interface {} is float32, not float64  
//因a代表的变量f本身是float32,所以断言转换为其他类型是会panic的  
fmt.Printf("y2的类型是：**%T\n**", y2)

打印输出：

y的类型是:float32

panic: interface conversion: interface {} is float32, not float64

goroutine 1 [running]:

main.main()

D:/script/code/go/GO/练习/类型转换的细节.go:13 +0x7f

exit status 2

空接口类型的变量a可以接受任何类型的变量，所以它可以被float32类型的变量f赋值。

但是如果把a断言为非float32类型的其他类型如float64则会导致panic。

# const类型可以做强制转换

虽然不可以使用空接口对空接口做断言，但是可以对const变量做强制类型转换

const ***NUM*** = 100  
//对于const类型，强制转换可以  
fmt.Println("const trans", int(***NUM***))  
fmt.Printf("NUM的类型：**%T\n**", ***NUM***)  
rnum := float32(***NUM***)  
fmt.Printf("const trans:**%f\n**", rnum)  
fmt.Printf("NUM的类型：**%T\n**", ***NUM***) //强制转换不影响源数据的类型

打印结果：

const trans 100

NUM的类型：int

const trans:100.000000

NUM的类型：int

可见const强制类型转换没有问题。

但是有一个细节，**强制类型转换只是返回了一个新的类型结果数据，但是源数据类型和值并不受影响**；

# Go中len方法按字节计算字符串的长度

var s string = "hello中国" //utf8中，每个汉字3个字节 5 + 3 + 3  
fmt.Println("len(s):", len(s)) //11  
  
var strs = []rune(s)  
fmt.Println("len(strs):", len(strs)) // 7

打印结果：

len(s): 11

len(strs): 7

len按照字节极计算字符串长度，单个汉字节占据3个长度。

# Go中全局变量不能在函数外部单独的语句中赋值

var Age int = 10 //全局变量在申明时候赋值是可以的  
  
//var nn int  
//nn = 11 //在函数外不允许在单独的行给全局变量赋值  
  
//Name := "aaa" //报错 这句相当于两条语句 var Name sring; Name = "aaa"  
//全局变量不能在外部直接赋值，需要在声明的时候就赋值  
var Name string = "aaa"  
  
func main() {  
 fmt.Println("Name:", Name)  
}

打印结果：

Name: aaa

全局变量在函数外部定义的时候可以不赋值，但是如果要赋值，需在声明的同时赋值，而不能单独在起一个语句对其赋值；

但是不能使用:=操作再声明的时候同时赋值，使用:= 相当于两条语句,如：

Name := “aaa”

相当于：

var name Name

Name = “aaa”

而的做法在函数外是不能单独起一个语句给变量赋值的；

# 结构体的方法绑定

## 按照值传递的方式绑定方法

type **Circle** struct {  
 radius float64  
}  
  
func (c Circle) getArea() float64 {  
 fmt.Printf("c的地址：**%p\n**", &c) //绑定结构体的方法使用的是值传递方式，会拷贝一份Circle结构体  
 return 3.14 \* c.radius \* 2  
}  
  
func main() {  
 var c Circle = Circle{2.0}  
 fmt.Printf("面积：**%v\n**", c.getArea())  
 fmt.Printf("主函数中,c的地址:**%p\n**", &c) //和getArea中获取的结构体的地址不一样，因为是两份单独的数据  
  
 //结构体绑定方法时如果不采用指针，会在方法函数中单独拷贝一份结构体数据，这在一定程度上会消耗内存  
 // 所以一般建议结构体绑定方法时，采用结构体的指针  
  
 fmt.Printf("面积2：**%v\n**", (&c).getArea()) //即使是取得结构体实例的地址调用，在调用函数中也无法获取到结构体的源地址  
}

打印结果：

c的地址：0xc00001a0b0

面积：12.56

主函数中,c的地址:0xc00001a098

c的地址：0xc00001a0e0

面积2：12.56

定义了一个计算源周长的函数getArea，传入半径计算周长。

可以看到绑定结构体Circle的时候，getArea方法使用的是按照值传递的方式绑定的，这样会在绑定的函数中拷贝一份结构体数据实体，如果在调用的时候时候是使用的结构体实例直接调用的方法，再getArea中打印的结构体数据的地址和main函数中打印的地址不是同一份。

# 在for…range过程中修改序号——不影响遍历值

nums := []int{1, 2, 3, 4, 5}  
sum := 0  
for i, n := range nums {  
 i = 6  
 fmt.Println(i, n)  
 sum += n  
}  
fmt.Println(sum)

打印结果：

6 1

6 2

6 3

6 4

6 5

15

i只是每次range迭代返回的序号，即使在循环中修改了他的值，但是range返回的值n丝毫不受影响。

# 空map赋值

m := make(map[int]int, 3)  
x := len(m)  
m[1] = m[2]  
y := len(m)  
fmt.Println(x, y)  
  
for k, v := range m {  
 fmt.Println("k,v==>", k, v)  
}

打印结果：

0 1

k,v==> 1 0

* 只有在map中明确指定了key的，才会有key-value键值对；
* 对不存在的键值对取value,得到的值类型的默认值；

此处定义了一个map变量m,但是没有任何键值对，直接m[1]赋值为取m的不存在的key为2的键值对的值，得到的是类型int的默认值0，所有m[1] = m[2] => m[1]=0,也就是给m设置了一个key-value为1:0 的键值对。

而从打印结果上看，只出现了一对键值对，所有没有指定key的情况下，没有键值对存在。

# 使用Unmarshal转字符串为结构体

### 转换时Unmarshal对结构体实实体的重用问题

type **AutoGenerated** struct {  
 Age int `json:"age"`  
 Name string `json:"name"`  
 Child []int `json:"child"`  
}  
  
func main() {  
 jsonStr1 := `{"age":14,"name":"potter","child":[1,2,3]}`  
 a := AutoGenerated{}  
 json.Unmarshal([]byte(jsonStr1), &a)  
 aa := a.Child  
 fmt.Println(aa) //打印[1,2,3]  
 jsonStr2 := `{"age":12,"name":"potter","child":[3,4,5,6,7,8]}`  
 json.Unmarshal([]byte(jsonStr2), &a)  
 fmt.Println(aa) // 打印[3,4,5]  
  
 a2 := AutoGenerated{}  
 json.Unmarshal([]byte(jsonStr2), &a2)  
 fmt.Println(a2.Child) //打印[3 4 5 6 7 8]  
  
 // 对于json从字符串解析到结构体中，当第一次解析生成的结构体实例长度确定后，就不在变化了  
 // 如果第二次再重复利用之前的结构体，一定要注意各个字段的长度变化，因为超过初始生成的结构体的字段长度，会被截断  
 // 要么每次解析的时候读重新实例化一份结构体避免截断问题  
}

打印结果：

[1 2 3]

[3 4 5]

[3 4 5 6 7 8]

我们先把jaonstr1转换为结构体，第一次转换使用的是实例化的结构体a,在实例化的时候我们并没有给a进行任何的数据空间设置，Unmarshal在转转字符串到结构体的时候自动拓展了数据的空间大小。所以我们看到转换后，结构体中切片类型的成员child为3个元素；

而在转换jsonstr2的时候，我们依然使用的是结构体a,他在已经作为第一次转换后的结构体实例，空间大小在第一次Unmarshal的时候被分配好了，第二次Unmarshal相当于使用的旧的大小，所以在第二次Unmarshal后，超出空间大小的数据被截断了，结构体切片成员变量child只有前3个元素[3,4,5]，少了jsonstr2的child的后3个元素[6,7,8]。

为避免数据的截断问题，每次进行Unmarsha的时候，都重新实例化一份结构体数据，这样每次Unmarshal拿到的都是崭新的数据区，转化就会是完成整的数据；

# 在结构体中嵌套接口作为结构体的成员

通常结构体中的成员属性都是基本的数据类型int,float,或者是切片，map等，以及嵌套的结构体。

而接口interface本身也是一种类型，只要是类型，那么就可以在结构体中放置类型或者该类型的变量。

type **worker** interface {  
 work()  
}  
  
type **person** struct {  
 name string  
 worker  
}  
  
func main() {  
 var w worker = person{name: "张三"}  
 fmt.Println(w) //{张三 <nil>}

fmt.Printf("worker类型:**%T\n**", w) //worker类型:main.person

//w.work() //panic,结构体person没有实现worker方法  
}

打印输出：

{张三 <nil>}

worker类型:main.person

在这个例子中，定义了一个worker的接口，然后再定义的结构体person中把worker接口作为一个成员添加到了person结构体中。

然后再main函数中，可以把person结构体的实例化数据体赋值给worker类型的变量w.

从打印可以看出，类型没有问题，name属性可以正常访问，只是作为成员的worker接口不能通过w.worker访问。

而w的类型本身是worker接口类型，在被赋值为person后，类型转换为person，这也是接口类型的特点，在被赋值后，就转换成为了被赋值的数据的类型。

**这种在结构体中嵌套interface的做法，被视为一种特殊的结构体对interfacce接口的实现**。也就是说在结构体添加了接口interface之后，这个结构体被视为实现了这个接口，从而这个结构体的数据实例可以赋值给这个接口的类型变量。而且这样添加后，结构体可以不实现接口中集合的方法，结构体只要不调用接口的没实现的方法就不会有panic。

**这种做法间接的提供了一种不用实现接口方法就能实现接口的途径**。

# Go时间格字符串格式化——time模块

## time模块获取年-月-日 时-分-秒并格式化

now := time.Now() //获取当前时间  
fmt.Printf("now = **%v**,type : **%T**", now, now)  
  
//获取年月日，时分秒  
fmt.Println("年：", now.Year())  
fmt.Println("月：", now.Month())  
fmt.Println("月(数字)", int(now.Month()))  
fmt.Println("日：", now.Day())  
fmt.Println("时：", now.Hour())  
fmt.Println("分：", now.Minute())  
fmt.Println("秒：", now.Second())

打印输出：

now = 2023-06-04 12:49:42.8733841 +0800 CST m=+0.001627401,type : time.Time

年： 2023

月： June

月(数字) 6

日： 4

时： 12

分： 49

秒： 42

* time.Now 返回一个time.Time类型，获取当前时间；
* time.Time类型的方法Year,Month，Day,Hour，Minute,Second 分别返回当前的时间的年，月，日，时，分，秒数据；

得到年月日，时分秒后，可以把时间格式化为字符串，并使用fmt.Sprintf转为一个字符串保存在字符串变量中：

//时间格式化  
fmt.Printf("当前年月日 **%d**-**%d**-**%d %d**:**%d**:**%d\n**", now.Year(), now.Month(), now.Day(),  
 now.Hour(), now.Minute(), now.Second())

//格式化时间字符串保存到变量中  
dataStr := fmt.Sprintf("当前年月日 **%d**-**%d**-**%d %d**:**%d**:**%d\n**", now.Year(), now.Month(), now.Day(), now.Hour(), now.Minute(), now.Second())  
fmt.Printf("dataStr:**%v\n**", dataStr)

打印输出：

当前年月日 2023-6-4 13:0:26

dataStr:当前年月日 2023-6-4 13:0:26

## 使用固定数字格式化字符串——20060102 15:04:05

使用固定数字 20060102 15::04:05这中固定的数字，这是go中约定俗成的固定格式：

//格式化日期时间的第二种方式 ,这种方式的时间格式是一组固定的数字  
fmt.Printf(now.Format("2006-01-02 15:04:05"))  
fmt.Println()  
fmt.Printf(now.Format("2006-01-02"))  
fmt.Println()  
fmt.Printf(now.Format("15:04:05"))  
fmt.Println()  
fmt.Printf(now.Format("2006"))  
fmt.Println("年")  
fmt.Printf(now.Format("01"))  
fmt.Println("月")  
fmt.Printf(now.Format("02"))  
fmt.Println("日")

打印输出：

2023-06-04 13:00:26

2023-06-04

13:00:26

2023年

06月

04日

这串数字中：

* 其中2006代表年份；
* 01代表月份；
* 02代表天，也就是日；
* 15代表小时
* 04代表分钟数；
* 05代表秒数；

# 获取时间戳和时间

## 获取当前时间戳

now: = time. Now()

//时间戳  
fmt.Println(now.Unix())  
//事件戳(纳秒)  
fmt.Println("----------------纳秒时间戳----------------------")  
fmt.Println(now.UnixNano())

在这个例子中，先使用time.Now获取到当前时间now,然后使用now.Unix获取到事件戳，这个时间戳是得到的从1970年1月1日到现在的秒数；

在调用now,UnixNano获取到的是1970年1月1日距离现在的纳秒数；

## 根据时间戳得到对应的日期

//根据时间戳得到相应的时间  
//形参：秒，纳秒  
ret := time.Unix(1651671768,0)  
//fmt.Println(ret.Year())  
fmt.Printf("**%v**年**%v**月**%v**号**\n**",ret.Year(),ret.Month(),ret.Day())

输出：

2022年May月4号

使用time.Unix传入秒数和纳秒数，得到对应时间戳的日期；

## 根据时间格式字符串获取对应的时间戳

//解析时间格式字符串，得到相应时间的时间戳  
tstr := "2022-05-04"  
timeobj,err := time.Parse("2006-01-02",tstr)  
if err != nil {  
 fmt.Println("解析事件错误：",err)  
 return  
}  
fmt.Printf("**%s**对应的事件戳：**%d\n**",tstr,timeobj.Unix())  
//fmt.Println(timeobj.Unix())

输出：

2022-05-04对应的事件戳：1651622400

调用time.Parse传入时间格式字符串，第一个参数的数字是固定格式，年对应2006，月对应01，日对应01（具体可参考[使用固定数字格式化字符串——20060102 15:04:05](#_使用固定数字格式化字符串——20060102_15:04:05)），第二个参数是时间格式字符串，结果解析后得到年月日，如果想要得到更加精确的事件，可以在第一个入参中添加更多的数字格式。

## 睡眠指定的时间

//sleep  
//go 中sleep需要指明时间单位  
n := 5  
time.Sleep(5\*time.***Second***)  
time.Sleep(time.Duration(n)\*time.***Second***)

调用time.sleep需要传入 数量\*时间单位，或者是使用time.Duration\*时间单位

## 时间计算

1. 获取当前时区LoadLocation

//按照字符串加载时区  
location,err := time.LoadLocation("Asia/Shanghai")  
if err != nil {  
 fmt.Println("解析location出错")  
 return  
}

使用time.LoadLocation传入时区字符串，来获取当前所在的时区。

1. 获取UTC时间Parse

//按照字符串获取时间对应的时间戳(把传入的事件字符串视为是UTC对应时间,没有当地时区)  
timestr := "2022-05-04 20:00:00"  
timeobj, err := time.Parse("2006-01-02 15:04:05", timestr)  
fmt.Println(timestr, "对应的时间戳为：", timeobj)

输出为：

2022-05-04 20:00:00 对应的时间戳为： 2022-05-04 20:00:00 +0000 UTC

1. 按照时区解析时间ParseInLocation

//按照时区解析时间

timestr := "2022-05-04 20:00:00"  
timeobj2, err := time.ParseInLocation("2006-01-02 15:04:05", timestr, location)  
if err != nil {  
 fmt.Println("解析时区时间戳出错!")  
 return  
}  
fmt.Println("按照本地时区解析,", timestr, "对应的时间戳为:", timeobj2)

输出为：

按照本地时区解析, 2022-05-04 20:00:00 对应的时间戳为: 2022-05-04 20:00:00 +0800 CST

在获取到了当前的时区之后，调用time.ParseInLocatrion方法，传入固定的时间格式字符串2006-01-02 15:04:05，这里的时间格式字符串中，分隔符可以随意自行指定，但是数字是固定的。然后第二个参数是要格式的时间字符串，其格式和第一个时间格式字符串保持一致，第三个参数传入当前的时区location.

1. 时间相减Sub

//时间对象相减

now := time.Now()  
timeduration := timeobj2.Sub(now)  
fmt.Println(timeduration)

输出为：

-9664h28m46.7855091s

用一个过去的时间减当前的时间，得到相差的时，分，秒;

# 空接口和数据类型的默认类型

Go中对于常量，会自动认定其默认类型。

如下:

* 常量0的默认类型就是int
* 常量0.0的默认类型是float64
* 单个的字符’a’默认类型是int32
* 字符串’abc’默认类型是string

var (  
 a int = 0  
 b int64 = 0  
 c interface{} = int(0)  
 d interface{} = int64(0)  
)  
  
println(c == 0)  
println(c == a)  
println(c == b)  
println(d == b)  
println(d == 0)  
println(c == 0)  
  
fmt.Printf("0的默认类型是:**%T\n**", 0)  
fmt.Printf("0.0的默认类型是:**%T\n**", 0.0)  
fmt.Printf("’a’的默认类型是:**%T\n**", 'a')  
fmt.Printf("’abc’的默认类型是:**%T\n**", "abc")

打印输出：

true

true

false

true

false

true

0的默认类型是:int

0.0的默认类型是:float64

'a'的默认类型是:int32

'abc'的默认类型是:string

**判断两个变量是否相等，在同类型，值也相同的时候，才会被认为是相等，结为true;**

在这个例子中，a是int，b是int64, c和d分别是两个空接口类型的变量；

因为空接口的类型可以被任意类型赋值，所以把空接口类型c赋值为int(0),把空接口类型d赋值为int64(0)

空接口类型在赋值后就转为赋的值的类型，所以，接口c类型为int,接口d类型为int64;

* 常量0的默认类型是int，是和空接口c被赋值后的类型是一样的；且值都为0，类型相等值也相等，所以c==0为true;
* 变量a的类型是int，和接口c类型int是一样的，值的大小都是0，所以a==c为true;
* 变量b的类型为int64,可接口c的类型int不一样，所以，c==b为false
* 接口d的类型在赋值为int64(0）后，d的类型也为int64,而b也是int64,且值都是0.类型相同，值也相同，所以d==b为true
* d的类型是int64，常量0的类型是int,所以类型不同，d==0为false
* c的类型是int,和常量0的类型int是一样的，而且值相同，所以c==0为true

# 结构体元素中有切片类型时候的复制问题

切片的本质是一个引用，它执行了一片数组区域，**当复制切片的时候，实际上是复制的引用，但是引用指向的实体数据并没有被复制一份**。切片相当于存放了一个个房间的钥匙的盒子，复制切片是，复制的只是钥匙本身，但是钥匙指向的数据“房间”并没有被重新复制；

import (  
 "container/list"  
 "fmt"  
)

type **node** struct {  
 chars []rune  
 //结构体元素是切片类型，那么如果是根据某个节点不同的复制了很多的副本，这些副本只是在”外层”有了各自的结构体的副本  
 //但是这些副本的切片元素里的值，仍然还是同一份，这是基于切片的特性，切片是内存中一块数组区域的映射，复制节点的时候只是复制了元素本身  
 //如果元素是一种带有指向意味的类型，那么复制不了它指向的值  
 //避免办法就是不用切片，用数组就不会存在这种问题  
 count int  
}  
  
type **node2** struct {  
 nums [2]int  
 count int  
}  
  
func main() {  
 n := node{[]rune{'B', 'A', 'A', 'A'}, 0}  
 nn := node2{[2]int{1, 2}, 0}  
 q := list.List{}  
 qq := list.List{}  
 for i := 0; i < 10; i++ {  
 n.count, nn.count = i, i  
 index1 := i % len(n.chars)  
 n.chars[index1] += rune(i)  
  
 index2 := i % len(nn.nums)  
 nn.nums[index2] += i  
 q.PushBack(n)  
 qq.PushBack(nn)  
 }  
  
 for e := qq.Front(); e != nil; e = e.Next() {  
 fmt.Println(e.Value)  
 v := e.Value.(node2)  
 fmt.Printf("address:**%p\n**", &v.nums[0])  
 }  
  
 fmt.Println("-------------------------")  
 for e := q.Front(); e != nil; e = e.Next() {  
 //fmt.Println(e.Value)  
 v := e.Value.(node)  
 fmt.Printf("**%s**,chars address:**%p**,chars pointer to address:**%p\n**", string(v.chars), &v.chars, &v.chars[0])  
 }  
}

打印结果：

{[1 2] 0}

address:0xc000016240

{[1 3] 1}

address:0xc000016270

{[3 3] 2}

address:0xc0000162a0

{[3 6] 3}

address:0xc0000162d0

{[7 6] 4}

address:0xc000016300

{[7 11] 5}

address:0xc000016330

{[13 11] 6}

address:0xc000016360

{[13 18] 7}

address:0xc000016390

{[21 18] 8}

address:0xc0000163c0

{[21 27] 9}

address:0xc0000163f0

-------------------------

{[78 80 73 75] 0}

NPIK,chars address:0xc000050520,chars pointer to address:0xc00001a0b0

{[78 80 73 75] 1}

NPIK,chars address:0xc000050540,chars pointer to address:0xc00001a0b0

{[78 80 73 75] 2}

NPIK,chars address:0xc000050560,chars pointer to address:0xc00001a0b0

NPIK,chars address:0xc0000505a0,chars pointer to address:0xc00001a0b0

{[78 80 73 75] 5}

NPIK,chars address:0xc0000505c0,chars pointer to address:0xc00001a0b0

{[78 80 73 75] 6}

NPIK,chars address:0xc0000505e0,chars pointer to address:0xc00001a0b0

{[78 80 73 75] 7}

NPIK,chars address:0xc000050600,chars pointer to address:0xc00001a0b0

{[78 80 73 75] 8}

NPIK,chars address:0xc000050620,chars pointer to address:0xc00001a0b0

{[78 80 73 75] 9}

NPIK,chars address:0xc000050640,chars pointer to address:0xc00001a0b0

在上述的例子中，node1和node2分别是两个结构体；

node1的结构体中有一个chars切片，其元素为rune类型，再有一个count变量用于计数；

在node2中有一个nums类型的数组，同时也有一个count变量用于计数；

在main函数中，n和nn分别是初始化了一份node和node2的结构体的实例化数据；

而q和qq分别是使用了go内置的list模块定义的列表数据结构，用于存放node和niode1两种结构体的实例化后数据；

在for循环中，分别对node和node1的实例化的数据n和nn赋值，这里使用了循环的序号i对chars的长度取余数的做法，应为chars的长度在初始化后就已经固定了，所以每次i增长1时，取余数就比上一次大1，这样只在char的长度数len的范围类循环跳转；

把取余数的值index作为list的序号，然后分别给chars切片以及数组index对应的元素chars[index]赋值；

可以看到打印结果中，对于是数组类型的结构体node2,它追加到list变量qq中，每一个追加，其chars类型都是单独的，因为在node2结构体中chars是一个数组，他在append的时候，会复制完整的数据实体比并添加到qq中；

但是对于node结构体，它的chars是一个切片类型，它在被赋值的时候，实际上是赋值给了切片引用所指向的一片数据中的元素，**切片可以复制多份，但是它指向的数组只有一份，也就是引用可以有多份，但是引用指向的数组只有一份**，所以可以通过遍历打印切片的数据的地址看出来，数据在append到list中以后，每一个q中的结构体元素，也就是node2，其chars因为是数组类型，所以都是单独的复制了一份数组，**但是对于chars是切片类型的结构体node而言，就不一样了，它只是把引用复制了多份，但是chars切片映射的数据区域还是同一片区域，所以他最终的值，在list中每个元素的值都是最后一次赋值的值**（切片的值都是一样的，因为应用指向的是同一片数组）。

# 结构体即使没有任何的字段属性也可以绑定方法

type **MethodUtils** struct {  
 //空字段  
}  
  
func (m MethodUtils) Print() {  
 for i := 0; i < 8; i++ {  
 for j := 0; j < 10; j++ {  
 fmt.Print("\*")  
 }  
 fmt.Println()  
 }  
}  
  
func main(){  
 var m MethodUtils  
 m.Print()  
}

打印输出：

\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*

定义了一个结构体**MethodUtils，**但是没有任何的属性字段，绑定方法也是没有任何问题的。

# 结构体赋值时候的拷贝问题

结构体在赋值的时候就会拷贝一份数据实体，

type **Person** struct {  
 Name string  
 Age int  
}  
  
func main() {  
 var p1 Person  
 p1.Age = 30  
 p1.Name = "小明"  
  
 //结构体用另外一个结构体赋值的时候，就拷贝了一份  
 var p2 = p1   
 fmt.Println(p2.Name, p2.Age)  
  
 //因为是拷贝，赋值互不干扰  
 p2.Name = "小李"  
 p2.Age = 25  
 fmt.Println("p1",p1) //p1 {小明 30}  
 fmt.Println("p2",p2) //p2 {小李 25}

}

打印输出：

小明 30

p1 {小明 30}

p2 {小李 25}

从打印结果看，我们先初始化了一个结构体Person的变量p1,然后给结构体p1的Name和Age都赋了值，然后使用了另外一个结构体变量p2，用已经赋值的p1在赋值给p2，然后修改p2里面的Name和Age。

在p2重新赋值后，分别打印p1和p2可以看到，p1和p2的属性字段值互不影响。

# 结构体绑定String方法

自定义的结构体绑定了String方法后，在使用fmt.Println的时候，就会自动调用其绑定的String方法。

// 给结构体绑定的方法中如果有实现了String方法，那么当使用fmt.Println在打印这个方法时就会默认调用它  
  
type **Stu** struct{  
 Name string  
 Age int  
}  
  
func (s \*Stu) String() string{  
 str := fmt.Sprintf("Name=[**%s**]**\t** Age=[**%d**]**\n**",s.Name,s.Age)  
 return str  
}  
  
func main(){  
 s := Stu{"小明",24}  
  
 //直接传递结构体实例，调用的是系统的默认打印方式  
 fmt.Println(s) //{小明 24}  
 // 传递结构体指针时调用的是绑定的String方法，因为String绑定使用的是指针  
 fmt.Println(&s) //Name=[小明] Age=[24]  
}

打印结果：

{小明 24}

Name=[小明] Age=[24]

可以看到，结构体Stu绑定了自己的String方法，在String方法中实现了自己的打印格式，使用fmt.Println的时候就会使用String方法中的格式。

有一点值得注意的是，绑定方法是使用的是指针类型 s \*Stu,所以再调用的时候，只有使用指针类型的时候，才会调用自己绑定的String方法去打印。使用普通的调用打印方式还是会使用系统默认格式。

所以如果要实现自定义的打印，需要在发起调用的时候，保持传值方式形同，如果绑定的函数是指针方式绑定的，那么调用的时候就用指针类型调用，如果是普通的按照值的方式绑定的，调用的时候是按照值传递调用。

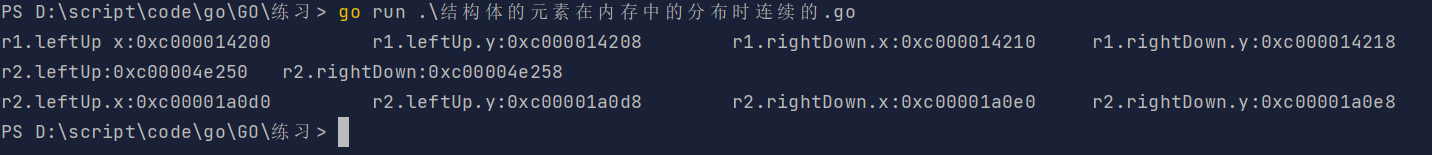
# 结构体元素在内存中的分布

如果结构体的属性字段是普通的类型int,,float等，那么他们在内存分布上是连续的。

如果结构体的的属性字段是引用类型，如切片，指针等。那么他应用本省的内存分布也是连续的，但是应用指向的内存区域不是连续的。

type **Point** struct {  
 x int  
 y int  
}  
  
type **Rect** struct {  
 leftUp, rightDown Point  
}  
  
type **Rect2** struct{  
 leftUp, rightDown \*Point  
}  
  
func main() {  
 r1 := Rect{Point{1, 2}, Point{7, 8}}  
  
 //r1有4个int值，在内存中是连续分布的  
 fmt.Printf("r1.leftUp x:**%p\t** r1.leftUp.y:**%p\t** r1.rightDown.x:**%p\t** r1.rightDown.y:**%p\n**",  
 &r1.leftUp.x,&r1.leftUp.y,&r1.rightDown.x,&r1.rightDown.y)  
 // r1.leftUp x:0xc0000141e0 r1.leftUp.y:0xc0000141e8 r1.rightDown.x:0xc0000141f0 r1.rightDown.y:0xc0000141f8  
  
 r2 := Rect2{&Point{1,1},&Point{3,3}}  
 // r1有两个Point元素是指针类型，这两个元素的地址是连续的  
 fmt.Printf("r2.leftUp:**%p\t** r2.rightDown:**%p\n**",&r2.leftUp,&r2.rightDown)  
 //元素指向的内存空间不一定是连续的  
 fmt.Printf("r2.leftUp.x:**%p\t** r2.leftUp.y:**%p\t** r2.rightDown.x:**%p\t** r2.rightDown.y:**%p\n**",  
 &r2.leftUp.x,&r2.leftUp.y,&r2.rightDown.x,&r2.rightDown.y)  
 // r2.leftUp.x:0xc0000160c0 r2.leftUp.y:0xc0000160c8 r2.rightDown.x:0xc0000160d0 r2.rightDown.y:0xc0000160d8  
 // leftUp的x和rightDown的x并不连续  
}

打印输出：



在这个例子中，定义了一个基本的坐标类型结构体Point,以及定义了两种矩形类型的结构体Rect和Rect2;

Rect和Rect2不同之处在于，他们存放坐标数据的方式，前者是存放点坐标的数据实体，后者是存放的是坐标类型的数据的指针。

第一行打印的是Rect类型的数据r1,其LeftUp的x,y以及，rightDown的x,y这几个元素在内存分布上是连续的.

第二行打印的是Rect2类型的数据，其LeftUp和RightDown是两个指针，可以看出指针本身在内存分布上也是连续的，但是指针指向的内存区域，不一定连续（虽然在本例子中指针指向的区域的LeftUp的x,y和RightDown的x,y看起来是连续的，但是这是在比较理想情况下呈现的结果，实际中不及一定连续）。

# 结构体继承时的就近访问原则

在go中，结构体的继承是通过匿名嵌套结构体的方式。

当结构体继承的时候，嵌套的匿名结构体互相之间有相同的字段或者方法时，编译器或采用就近访问的原则，入如果指定访问某个匿名成员结构体成的某个特定的方法或者字段，需要逐层指明匿名结构体名字来访问。

匿名结构体的访问方式吗，如果本结构体中没有，需要访问的字段，那么会自动到继承的结构体中找。

非匿名的结构体嵌套方式，严格来说不能成为继承，而是称其为“组合”，这种方式嵌套的结构体，如果要访问“父“结构体中的字段，需要逐层指明结构体的名称，也就是要写全访问路径。

例如下面的结构体的继承和嵌套方式：

type **A** struct {  
 Name string  
 age int  
}  
  
func (a \*A) SayOk() {  
 fmt.Println("A sayOk", a.Name)  
}  
  
func (a \*A) hello() {  
 fmt.Println("A hello", a.Name)  
}  
  
type **B** struct {  
 A  
 Name string  
}  
  
func (b \*B) sayOk() {  
 fmt.Println("B sayOK", b.Name)  
}  
  
func (b \*B) hello() {  
 fmt.Println("B hello", b.Name)  
}  
  
type **C** struct {  
 A  
 B  
}  
  
type **D** struct {  
 a A  
}

1. 在这个例子中，先定义了结构体A,并用指针方式绑定了SayOK和hello方法；
2. 然后定义了结构体B匿名继承了A,以及添加了一个自己的属性Name,，然后B也以指针的方式绑定了方法SayOk和hello.
3. 然后由定义了结构体C，匿名继承了A和B.
4. 以及最后D，有名嵌套了A.

## 匿名继承时，访问本结构体绑定的方法

var b B  
b.Name = "小明" //给B中的Name赋值，但是不会给A中的Name赋值  
b.hello() //B hello 小明  
b.SayOk() //B sayOK 小明

打印输出：

B hello 小明

B sayOK 小明

定义一个B结构体类型的变量b, 就近原则，调用SayOK和hello都是调用的B本身绑定的方法；

## 访问其父结构体A的方法

// 如果要访问结构体中嵌套的匿名结构体，需要写应用的全路径  
b.A.SayOk() //A sayOk  
b.A.hello() //A sayOk  
// 这里没有获取到Name。因为Name只在B中被赋值了，没有给A赋值

打印输出：

A sayOk

A hello

在这里因为B的父结构体中也绑定了同样的方法SayOk和hello,所以如果想要访问父类的同名方法，需要逐层次写明访问路径。

## 修改父结构体A中的属性字段值

b.A.Name = "小红" //这里给A中的Name赋值了，和B中的Name互不影响，b中的Name仍然是原来的值  
b.hello() //B hello 小明  
b.SayOk() //B sayOK 小明

打印输出：

B hello 小明

B sayOK 小明

从打印结果看，修改了父结构体A中的字段属性Name,但是在调用b本身的方法时，b本身的Name属性并没有被影响，因为B自己本身有子级单独定义的Name

## 多重继承时，各个结构体中有相同的属性名——编译报错

当使用匿名嵌套继承时，如果被继承的各个结构体中有相同的字段属性，或报错

//var c C  
// 编译报错  
// 当结构体中的子成员有相同的字段时，而结构体没有该成员，那么此时必须要明确结构体名来区分  
//c.Name = "小强"

这里C继承A和B, 而结构体A中有成员Name, 结构体B中也有成员Name;

结构体C在同时继承A和B时，因为A和By有相同的成员Name,导致二义性，编译不通过。

## 有名嵌套接结构体访问嵌套结构体的属性和方法

var d D  
//fmt.Println(d.Name) //d.Name undefined 因为D结构体继承了A不是按照匿名结构体的方式继承的，  
// D不是按照匿名方式继承机构体A的，那么如果D中没有Name，不会自动去A中寻找，会直接报错  
fmt.Println(d.a.Name) //通过有名方式继承，如果要访问继承的类中的方法，需要写全路径

打印输出：

A sayOk 小红

A hello 小红

结构体D有名嵌套了结构体A,那么这种方式就不能直接访问结构体A中的属性字段，需要逐层指明结构体的路径。

# Go实现类似闭包

* 闭包返回一个函数引用；
* 闭包函数中的局部不变量有保持作用，因为其保持的作用，可以实现类似全局变量的作用；

func called() func() int {  
 var num int  
 return func() int {  
 num += 1  
 return num  
 }  
  
}  
  
func main() {  
 f := called()  
  
 for i := 0; i < 10; i++ {  
 fmt.Println(f())  
 }  
}

打印输出：

1

2

3

4

5

6

7

8

9

10

在这个例子中，定义了一个闭包函数called.，这个函数返回一个类型为func(）int类型，也即，返回一个“返回值为int类型的函数”；

在闭包函数called内部，变量num，在main中每次调用完后并不会释放消失，而是在下次调用后，沿用之前的值进行累加。通过打印结果看，在主函数main中循环调用了闭包函数10次，num每次加1，最终累加到了10。

# 文件操作

## 打开文件os.Open

**方法os.Open打开文件是按照只读方式打开文件的**。

//打开文件  
fileobj, err := os.Open("./file.txt")  
if err != nil {  
 fmt.Printf("打开文件失败!**\n**")  
 return  
}  
  
//延迟执行,在全部返回之前再关闭文件  
defer fileobj.Close()

打开文件使用的是os模块的os.Open函数，入参为文件的路径。如果打开成功，返回的两个值中第一个就是文件对象，第二个错误标识为nil，如果错误标识不是nil，那么标识文件打开失败。

在打开文件后，**使用defer fileobk.close()函数来延迟关闭打开的文件对象**——这是一个好习惯。

## 读文件

### 文件对象读取文件

//读文件  
for {  
 var tmp [128]byte  
 n, err := fileobj.Read(tmp[:])  
 if err != nil {  
 if err == io.EOF {  
 fmt.Println("文件读取完毕!")  
 return  
 } else {  
 fmt.Println("读文件失败,err:", err)  
 return  
 }  
 }  
 //到文件尾部，即使读取不到tmp满长度，err也不会报错，只是n返回的读取长度小于len(tmp)就是了  
 fmt.Printf("读取**%d**字节**\n**", n)  
 fmt.Println(string(tmp[:]))  
}

在打开文件获取到文件对象fileobj后可以直接使用fileobj.Read来读取文件，入参为一个byte类型的切片缓存区，这个切片长度决定了每次最多从文件中读取多少个字节。

两个返回值，第一个返回的是成功读取了多少个字节，第二个返回值是如果有错误的话返回错误。

当读取到最后一次时，如果剩下的内容不足切片缓存区的长度，也不会有错误err,只是返回的成功读取的字节长度小于缓存区的切片长度。

当读取到文件尾部的时候，才会返回错误err为io.EOF,，因此可以把err和io.EOF做判断每次读取的返回错误值，如果错误等于io.EOF，则说明已经读取到文件尾部，就可以终止循环读取文件了。

使得注意的是因为使用的是byte类型的作为存放读取的文件数据的地方看，在读取中文字符串的时候，容易遇到汉字被不完整截断读取的问题，打印出的读取内容会有乱码的问题

### 带缓冲的读取文件bufio.NewReader——推荐使用

打开文件仍然使用os.Open打开文件。

使用bufio读取文件可以避免字符截断的问题，bufio自带缓冲区，可以指定读取文件的时候，遇到哪个字符就停止读取，如遇到换行符’\n’ 就停止读取内容，这样就实现了按行读取。

Bufio可以适用于不论是大文件还是小文件的读取。

//bufio可按照行读取  
getfuncinfo.PrintFuncName()  
fileobj, err := os.Open("./file.txt")  
if err != nil {  
 fmt.Println("打开文件失败!")  
 return  
}  
defer fileobj.Close()  
  
reader := bufio.NewReader(fileobj)  
for {  
 //设定遇到\n就停止读取，也就是按照行读取，在line里会读取并包含换行符  
 line, err := reader.ReadString('\n')  
 if err != nil {  
 if err == io.EOF {  
 fmt.Println("到达文件末尾!")  
 return  
 } else {  
 fmt.Printf("read line 错误，err:**%v\n**", err)  
 return  
 }  
 }  
 fmt.Print(line) //每次读取打印一行  
}

使用bufio读取文件时，在文件已经os.Open成功打开之后：

1. 用bufio.NewReader创建一个读取器reader并返回;
2. 使用返回的reader读取器的方法，如ReadString读取文件内容，可以指定遇到某个字符串就停止读取;
3. Reader实际上是一个结构体，该结构体绑定了很多方法，如ReadString, ReadRune, Readbytes等等，还自带一个ReadLine方法（实际就是ReadString遇到\r就停止读取的版本）
4. 读取器调用读取方法如ReadString读取内容的返回值有两个，第一个为读取到的内容，第二个是返回的错误err，读取到文件尾部的时候返回错误io.EOF，可以使用err == io.EOF判断读取到we年尾部，来终止循环读取；

### 一次读取整个文件内容——ioutil模块

使用ioutil读取文件的时候，会一次性读取文件的所有内容。

读取方法为ioutil.ReadFile,该方法不使用os.Open打开文件，它会自行打开文件；

//ioutil 一次读取整个文件  
getfuncinfo.PrintFuncName()  
ret, err := ioutil.ReadFile("./file.txt")  
if err != nil {  
 fmt.Printf("read err:**%v\n**", err)  
 return  
}  
fmt.Println(string(ret))

Ioutil.ReadFIle入参传入文件的路径；

返回值有两个，第一个是读取的文件内容，是一个byte数组

第二个为err,如果读取出错则不为nil;

使用ioutil.ReadFile适用于读取文件内容较短小的文件，不适用于大文件的读写；

## 写文件

### 文件对象写文件

不同于os.Open使用只读方式打开文件的方式，要写入文件时，需要指定文件的打开方式，使用的是os.OpenFile方法，该方法有3个入参，分别为：

* 文件路径；
* 文件的打开(创建)方式；
* 设置的文件权限（该参数只在linux环境下有效）；

该方法有两个返回值，分别是：

* 文件对象；
* 错误err;

文件的写入使用文件对象的方法fileobj.Write或者fileobj.WriteString，方法传入的入参就是要写入的内容。

//文件名,创建方式,文件权限(权限控制只在linux下生效)  
//fileobj,err := os.OpenFile("./xxx.txt",os.O\_WRONLY|os.O\_CREATE|os.O\_APPEND,0644)  
fileobj, err := os.OpenFile("./xxx.txt", os.***O\_WRONLY***|os.***O\_CREATE***|os.***O\_TRUNC***, 0644)  
//O\_TRUNC 每次打开都重新写入  
if err != nil {  
 fmt.Printf("打开文件失败!")  
 return  
}  
defer fileobj.Close()  
  
fileobj.Write([]byte("测试文件文件写入byte。。。**\n**"))  
fileobj.WriteString("测试直接写入string。。。**\n**")

文件的打开方式可以由多个方式进行或操作来组合起来。打开方式的枚举实际上是一个定义在os模块中的静态变量，基础类型为int。

const (  
 // Exactly one of O\_RDONLY, O\_WRONLY, or O\_RDWR must be specified.  
 ***O\_RDONLY*** int = syscall.***O\_RDONLY*** // open the file read-only.  
 ***O\_WRONLY*** int = syscall.***O\_WRONLY*** // open the file write-only.  
 ***O\_RDWR*** int = syscall.***O\_RDWR*** // open the file read-write.  
 // The remaining values may be or'ed in to control behavior.  
 ***O\_APPEND*** int = syscall.***O\_APPEND*** // append data to the file when writing.  
 ***O\_CREATE*** int = syscall.***O\_CREAT*** // create a new file if none exists.  
 ***O\_EXCL*** int = syscall.***O\_EXCL*** // used with O\_CREATE, file must not exist.  
 ***O\_SYNC*** int = syscall.***O\_SYNC*** // open for synchronous I/O.  
 ***O\_TRUNC*** int = syscall.***O\_TRUNC*** // truncate regular writable file when opened.  
)

### 使用bufio写文件

Bufio写文件的方式如下：

1. 使用os.FileOpen打开文件；
2. 使用bufio.NewWriter创建一个文件写入器wr；
3. 使用创建的文件写入器的方法如wr.WriteString写入内容；
4. 使用文件写入器wr的Flush把写入的内容刷线到文件中；

**因为bufio自带缓冲，所以再写入文件内容后，是暂时保留在缓冲区中，需要调用Flush把缓存中的内容刷到文件中。**

fileobj, err := os.OpenFile("./xx.txt", os.***O\_WRONLY***|os.***O\_CREATE***|os.***O\_TRUNC***, 0644)  
if err != nil {  
 fmt.Printf("打开文件失败!")  
 return  
}  
defer fileobj.Close()  
wr := bufio.NewWriter(fileobj) //bufio.Writer是自带缓冲的，所以Flush操作时必要的  
wr.WriteString("测试写入string,use bufio。。。**\n**") //把内容写到bufio缓存中  
wr.Flush() //把缓存内容写入文件中

### 使用ioutil写入文件

使用ioutil.WriteFile可以直接写入文件内容，也无需调用os.OpenFile打开文件，ioutil会自行打开文件写入。

入参有3个：

* 文件路径；
* 写入内容；
* 文件权限；

str := "直接使用ioutil写入byte。。。**\n**"  
err := ioutil.WriteFile("./xxxxx.txt", []byte(str), 0666)  
if err != nil {  
 fmt.Println("ioutil写文件失败")  
 return  
}

## 文件拷贝

### 使用io.Copy复制文件

使用io.Copy拷贝文件，其自带缓存，也可以适用于拷贝大文件。

使用os.Copy步骤如下：

1. 使用os.Open打开文件，获取源文件对象srcfile；
2. 使用bufio.NewReader创建文件读取器reader;
3. 使用os.OpenFile打开新文件并获取待写入的文件对象dstfile；
4. 使用bufio.NewWriter创建文件写入器writer;
5. 调用io.Copy,一次传入文件文件写入器writer和读取器reader;

func FileCopy(dstFilename, srcFilename string) (n int64, err error) {  
 srcfile, err := os.Open(srcFilename)  
 if err != nil {  
 fmt.Println("打开源文件失败!")  
 return  
 }  
 defer srcfile.Close() //defer关闭打开的文件  
  
 reader := bufio.NewReader(srcfile)  
  
 dstfile, err := os.OpenFile(dstFilename, os.***O\_WRONLY***|os.***O\_CREATE***, 0666)  
 if err != nil {  
 fmt.Println("打开目标文件失败!")  
 return  
 }  
 defer dstfile.Close() //defer关闭打开的文件  
  
 writer := bufio.NewWriter(dstfile)  
  
 return io.Copy(writer, reader)  
}

调用文件拷贝函数：

srcfile := "./a.jpg"  
dstfile := "./b.jpg"  
if \_, err := FileCopy(dstfile, srcfile); err != nil {  
 fmt.Println("拷贝文件失败! err=", err)  
} else {  
 fmt.Println("拷贝完成!")  
}

**总结：使用io.Copy复制文件袋的时候最重要的是获取源文件的读取器reader和目标文件的写入器writer，在打开了源文件和目标文件后记得使用defer 延时关闭打开的文件。**

# 获取终端输入

## 模块fmt获取输入

### 使用fmt.Scanln

var age int  
fmt.Println("输入年龄:")  
fmt.Scanln(&age)

var s string  
fmt.Print("请输入：")  
fmt.Scanln(&s)  
fmt.Printf("输入的是:**%s\n**",s)  
//Scanln 无法处理输入的时候有空格的情况，它遇到空格就停止读取了

使用fmt,Scanln无法处理输入有空格的情况，遇到空格就停止读取输入了，适用于单个变量的输入获取；方便的一点是使用Scanln不用顾及到变量的类型，它会根据传入的变量的类型自动转换输入为对应的类型。

### 使用fmt.Scanf获取输入

var age2 int  
var name2 string  
var score2 float32  
var isVIP2 bool  
  
fmt.Println("一次输入姓名，年龄，成绩，是否VIP，空格分割:")  
fmt.Scanf("**%d %s %f %t**",&age2,&name2,&score2,&isVIP2)  
fmt.Printf("姓名:**%v**,年龄:**%v**,成绩:**%v**,是否VIP:**%v\n**",name2,age2,score2,isVIP2)

使用Scanf获取输入的时候需要指定对应变量的类型格式，空格分割输入；

### 使用fmt.Scan连续获取输入

**空格**或者**换行**来分割输入，

Scan可以连续获取输入，也可以根据变量自动转换输入为对应的类型，但是一旦遇到输入的类型和定义的变量类型不相同，就会立即引发输入错误，退出输入的读取；

var name string  
var age int8  
var sex int8  
fmt.Println("请输入姓名, 年龄, 姓别")  
fmt.Scan(&name, &age, &sex)  
fmt.Printf("姓名: **%v** 年龄: **%v** 性别: **%v**", name, age, sex )

* 获取输入如下：

请输入姓名, 年龄, 姓别

zhw

23

1

姓名: zhw 年龄: 23 性别: 1

在这个输入演示中，每个变量分别单独换行输入，也可以正常读取并打印出来；

* 获取输入时如下：

请输入姓名, 年龄, 姓别

zf

45 4

姓名: zf 年龄: 45 性别: 4

在这个输入演示中，第一个输入姓名为一个string类型，然后换行连续输入了年龄45和性别4，因为后两个类型定义的是int所以输入数字后能够正确获取到输入并打印出来；

* 获取输入如下：

请输入姓名, 年龄, 姓别

zgw 34 1

姓名: zgw 年龄: 34 性别: 1

在这个输入例子中，把输入按照一行输入，使用空格分割，可以看到变量也能够正常获取到输入。

* 获取输入如下：

请输入姓名, 年龄, 姓别

zg

er

姓名: zg 年龄: 0 性别: 0

在这个输入的例子中，因为输入时年龄是一个int类型，但是输入的是一个字符串，在换行时就直接提前退出了输入的获取。

### Scan输入的类型错误时的注意点

值得注意的是，Scan虽然能够识别换行符，但是在输入的类型错误的时候它不会吸收换行符，换行符会被留在标准输入中，所以如果有连续多个输入语句时，如果前面的输入有类型不匹配的情况，常常会导致后面的输入获取不到正确的输入，如下

var name string  
var age int8  
var sex int8  
fmt.Println("请输入姓名, 年龄, 姓别")  
fmt.Scan(&name, &age, &sex)  
fmt.Printf("姓名: **%v** 年龄: **%v** 性别: **%v**", name, age, sex)  
  
var en int  
fmt.Print("请输入一个数字：")  
fmt.Scan(&en)  
fmt.Println("输入的数字是:", en)  
fmt.Scan(&en)

输入时，如果在第一次输入中输入了错误的类型：

请输入姓名, 年龄, 姓别

zg er 4

姓名: zg 年龄: 0 性别: 0请输入一个数字：输入的数字是: 0

可以看到，首个获取输入的时候，年龄应该数如int类型，但是输入了字符串，导致后面的scan直接没有获取到输入。

# 一个统计文件中字符类型的例子

通过这个例子，将会进一步的说明switch case的用法以及文件的读取：

type **cntStruct** struct {  
 letter int  
 digit int  
 other int  
}  
  
func CharactersCount() {  
 file, err := os.Open("./a.txt")  
 if err != nil {  
 fmt.Println("文件打开失败，err=", err)  
 return  
 }  
 defer file.Close()  
  
 var cnt cntStruct  
  
 reader := bufio.NewReader(file)  
 for {  
 s, err := reader.ReadString('\n')  
 if err == io.EOF {  
 break  
 }  
 for \_, c := range s {  
 //fmt.Println(c)  
 /\*  
 switch c 会报错  
 此处swotch后不能添加条件c，因为case 是按照值byte码大小来比较的，计算结果是bool类型，而c是byte类型，二者不匹配，编译不通过；  
 此处直接用switch相当于转为 if ... else if ...的用法  
 \*/  
 switch {  
 case c >= 'a' && c <= 'z':  
 //cnt.letter++  
 fallthrough  
 case c >= 'A' && c <= 'Z':  
 cnt.letter++  
 case c >= '0' && c <= '9':  
 cnt.digit++  
 default:  
 cnt.other++  
 }  
 }  
 }  
 fmt.Printf("字母个数:**%v** 数字个数:**%v** 其他字符个数:**%v\n**", cnt.letter, cnt.digit, cnt.other)  
}

函数CharactersCount 来统计文件a.txt中出现的字母，数字，以及其他的字符的数量。

* 结构体cntStruct用来存放统计的额字母，数字，以及其他这3中字符类型的个数；
* 使用os.Open打开文件获取到文件对象file；
* 使用bufio.NewRreader创建文件读取器reader；
* 使用读取器reader.ReadString读取文件内容，并设置遇到换行符就停止一次阅读，也就是按行读取；
* 循环读取是判断如果返回err为io.EOF则说明读取到文件尾，停止循环读取；
* 每次读取一行后，使用for循环统计每一行中出现的字母，数字，和其他的字符类型个数；
* 统计的时候使用switch匹配字母，数字类型，不是这两种类型的则统计到其他字符中；
* 使用switch时不加switch后的变量条件，因为在case匹配条件上使用的是ASCII的大小范围匹配，这个大小范围匹配返回的结果是bool类型，如果对每一行的每一个字符c加载switch后面作为匹配条件，而c是字符类型byte,这会导致和case后的条件表达式返回类型bool类型不匹配，所以switch后不加变量，直接在case中进行条件匹配即可；
* 这里使用了fallthrough穿透，在匹配到小写的字符a-z后，会直接转到下一个紧挨着的case A-Z，并执行这个case下的语句(穿透的时候不在匹配下一个case的条件，直接执行case下的语句)；也就是在输入为a-z时会直接转到A-Z的case语句累加字母类型的个数，在输入为A-Z时匹配到该case下累加字母类型个数；

# 关于switch…case

## Switch计算的的表达式最终的值

func test(b byte) byte {  
 return b + 1  
}

var key byte  
fmt.Println("请输入一个字符 [a,b,c,d,e,f,g]")  
\_, err := fmt.Scanf("**%c**", &key)  
if err != nil {  
 fmt.Println(err)  
}  
  
switch test(key) + 1 { //switch计算的是表达式的最终的值  
//case结束不用加break  
case 'a':  
 fmt.Println("捕获到a") //输入单个字符的时候不会被执行到，及时输入最小的a,表达式计算结果都会是'a' + 2 = 'c'  
case 'b':  
 fmt.Println("捕获到b")  
case 'c':  
 fmt.Println("捕获到c")  
case 'd':  
 fmt.Println("捕获到d")  
default:  
 fmt.Println("其他...")  
}

输入输出如下：

请输入一个字符 [a,b,c,d,e,f,g]

a

捕获到c

在这个例子中，case ‘a’ 永远不会被匹配到，因为switch计算的是表达式test(key)+1的最终结果值，只要输入是一个字母，即使是输入的是最小的字母a,计算后结果也是经过了test函数的加1,以及表达式的再次加1，key值比输入的值大2，也即，即使输入的是a，switch表达式的结果也是c，不会匹配到a,b;

## Switch（计算结果）的类型和case的类型需要是一致的

var n1 int32 = 20  
var n2 int32 = 20  
//var n2 int64 = 20  
switch n1 {  
//case n2: // case后的检查值的类型需要和switch的一致  
case n2, 10, 5: //case后可以添加多个条件  
 fmt.Println("ok1")  
default:  
 fmt.Println("其他...")  
} //输出ok1

输出：

ok1

如果放开注释的一行//var n2 int64 = 10那么这个代码片段就会报错类型不匹配，因为n1是int32类型的，而n2是int64类型的；

## Switch后面不加表达式，相当于if…else…的用法

var age int = 10  
switch {  
case age == 10:  
 fmt.Println("age == 10")  
case age == 20:  
 fmt.Println("age == 20")  
default:  
 fmt.Println("其他...")  
}

输出：

age == 10

匹配age的值大小，age的值为10所以，case age == 10条件成立，执行该case下的语句；

## Case做条件判断匹配时，是从上到下的的顺序

var score int  
fmt.Printf("输入分数：")  
\_, err = fmt.Scan(&score)  
if err != nil {  
 fmt.Println(err)  
}  
switch {  
case score > 90:  
 fmt.Println("成绩优秀")  
case score > 80:  
 fmt.Println("成绩良好")  
case score > 70:  
 fmt.Println("再接再厉")  
case score > 60:  
 fmt.Println("多多努力")  
}

输入1：

输入分数：70

多多努力

输入2：

输入分数：80

再接再厉

输入3：

输入分数：91

成绩优秀

可以看到，3次输入，当输入70的时候，不满足>90,>80,>70三个条件，直接走到了>60的case中；

当输入80的时候，不满足>90,>80的条件，走到了>70的条件中；

输入91时，满足>90的条件，走到了case的>90条件；

**范围判断时，从上到下的顺序判断，一旦满足条件执行其匹配的case语句，然后返回；**

## Switch 穿透——fallthrough

//var num int = 10  
var num int = 20  
switch num {  
case 10:  
 fmt.Println("ok1")  
case 20:  
 fmt.Println("ok2")  
 fallthrough //默认只能穿透1层  
case 30:  
 fmt.Println("ok3") //上一个case有fallthrough，当上一个条件满足的时候，这个case也会执行  
 fallthrough //如果要穿透多层那么加多个fallthrough  
case 40:  
 fmt.Println("ok4")  
default:  
 fmt.Println("其他")  
}

输出为：

ok2

ok3

ok4

这个例子中，匹配num大小为20，匹配到了case 30，但是在其分支下使用了fallthrough穿透，这会导致继续执行下一个敬爱着的case 20，同时case30也有fallthrough穿透，所以又会直接执行下一个case 40下的语句，直到遇到default语句，没有fallthrough穿透则停止。

fallthrough穿透不会再次判断case是否成立，会直接执行穿透的case中的语句，同时单个fallthrough语句只能穿透一层,如果要连续穿透，就需要连续在多个case下添加fallthrough，穿透会在没有遇到fallthrough语句的一层case停止穿透。

## Switch判断数据类型

var x interface{}  
var y = 10.0  
x = y  
switch i := x.(type) {  
case nil:  
 fmt.Printf("x的类型 **%T\n**", i)  
case int:  
 fmt.Printf("x 是 int类型")  
case float64:  
 fmt.Println("x 是float64类型")  
default:  
 fmt.Println("其他")  
}

输出：

x 是float64类型

在这个例子中，x变量是一个空接口类型，它可以被任何类型的数据赋值，在switch中使用x.(type) 获取x的类型，和case的各个条件匹配。

# 关于goroutine

## 显式发起go reoutine函数

func hello(i int) {  
 fmt.Printf("hello!--**%d\n**", i)  
}  
  
func Test1() {  
 for i := 0; i < 1000; i++ {  
 go hello(i)  
 }  
}

在这个例子中，hello函数用于打印hello!---i语句，其中i是传入的参数值；

通过调用Test1，输出如下：

hello!--0

hello!--28

hello!--3

hello!--51

hello!--11

hello!--114

hello!—7

…

因为是值传递，在使用go语句运行routine的时候传入变量，没有访问外部的公公公变量，没有资源竞争的问题。

## 使用匿名函数goroutine

for i := 0; i < 50; i++ {  
 go func() { //匿名函数，闭包 ——使用goroutine 并发  
 fmt.Println("hello,test2", i)  
 // 会出现很多的重复输出，因为打印取的是匿名函数外部的变量；  
 // 在输出时可能外循环已经不是刚启动goroutine的时候的外部变量的值了  
 // 并行启动打印时候，有可能是几乎同一时间访问外部循环变量，所以会有很多的形同打印  
 }()  
}

输出如下：

hello,test2 2

hello,test2 12

hello,test2 17

hello,test2 17

hello,test2 17

hello,test2 18

hello,test2 18

…

这个例子中，在for循环中，使用go关键字直接触发匿名函数调用，因为访问匿名函数外部变量i的关系，在多个routine并行时，导致获取的值不一致。

## 匿名函数传入参数

for i := 0; i < 50; i++ {  
 go func(j int) { //匿名函数，闭包 ——使用goroutine 并发  
 fmt.Println("hello,test2", j)  
 }(i) //实时传值进入匿名函数就不会输出重复了，这样就不会访问外部变量  
}

输出：

hello,test2 2

hello,test2 24

hello,test2 7

hello,test2 5

hello,test2 6

hello,test2 8

hello,test2 1

hello,test2 3

hello,test2 9

……

在这个例子中，使用go语句发起匿名函数routine的时候同时传入了参数，所以在匿名函数内部没有访问外部变量，输出没有混乱的情况。

## 多routine使用sync.WaitGroup

**注意：sync.WaitGroup是一个值类型**

func worker(i int) {  
 defer wg.Done() //计数减一  
 time.Sleep(time.***Second*** \* time.Duration(rand.Intn(5))) //随机睡眠1~5 秒  
 fmt.Println(i)  
}  
  
var wg sync.WaitGroup  
  
func routineWait() {  
 for i := 0; i < 10; i++ {  
 wg.Add(1) //计数加1  
 go worker(i)  
 }

func main() {  
 routineWait()  
 wg.Wait() //等待计数为0再退出main  
}

输出：

worker2:0

worker2:1

worker2:2

worker2:3

worker2:4

worker2:5

worker1:6

worker1:7

worker1:8

worker1:9

在这个例子中，使用了sync模块的WaitGroup，它相当于一个信号量计数器，每发起一个routine就及计数加1；

* 定义工作函数worker，该函数中使用defer在完成工作后给wg减1——wg.Done()
* 定义WaitGroup的全局变量wg;
* routineWait函数负责发起一个个的routine，并在发起的时候给wg奇数加1——wg.Add(1)；
* 在主函数中调用routineWait函数后，需要使用wg.Wait(),等待wg的计数减小到0，wg减小到0也就意味着所有的worker都执行完毕；
* 如果在主main函数中不加wg.Wait()，那么主routine会提前运行完成，没有等待其他routine的动作，导致整个程序提前终止；

## 设置routine并行线程数runtime.GOMAXPROCS

使用runtime.GOMAXPROCS(n int)函数设置routine并行时，能够最多在多少个CPU线程上运行；

var sWg sync.WaitGroup  
  
func worker1() {  
 defer sWg.Done()  
 for i := 0; i < 10; i++ {  
 fmt.Printf("worker1:**%d\n**", i)  
 }  
}  
  
func worker2() {  
 defer sWg.Done()  
 for i := 0; i < 10; i++ {  
 fmt.Printf("worker2:**%d\n**", i)  
 }  
}  
  
func Start() {  
 runtime.GOMAXPROCS(2) //不设置则默认跑满所有CPU核心  
 sWg.Add(2)  
 go worker1()  
 go worker2()  
 sWg.Wait()  
}

在这个例子中，Start函数是起点；

定义了一个全局的WaitGroup变量sWg,然后再Start函数中，先给sWg添加了奇数2,也就意味着可以等待两个routine。

在Start函数一开始就使用了runtime.GOMAXPROCS函数设置最多程序可以占用两个计算机的线程(核数);如果不设置runtime. GOMAXPROCS则程序默认会占据所有的CPU线程数(核数);

## 使用上下文context.Context

**注意：conetxt是接口类型**

Context的使用场景是，对于有多级并发的时候，如在程序的主routine中发起了一个协程A,然后又在协程A重中发起了另外一个协程B, B又发起了C…

在这种场景下，无法在使用WaitGroup来控制协程的退出，就需要使用到context。

再使用context的时候，一般是不直接使用context本身，而是使用其衍生的方法，如conetxt.WithCancel

### 使用context.WithCancel

该方法返回一个cancel函数，这个函数可以作为一个通知context终止的信号被调用，当调用cancel()方法时，在其他的goutine协程中会感知到该cancel信号，从而控制协程退出。

func work2(wg \*sync.WaitGroup, ctx \*context.Context) {  
 defer (\*wg).Done()  
 (\*wg).Add(1)  
 go childwork2(wg, ctx) //在gorotine中发起子goroutine  
FORLOOP:  
 for {  
 fmt.Println("work2 running...")  
 time.Sleep(500 \* time.***Millisecond***)  
 select {  
 case x := <-(\*ctx).Done(): //如果context.Done()有返回则退出work  
 fmt.Printf("work2收到结束消息,**%#v\n**", x)  
 break FORLOOP  
 default:  
 }  
 }

fmt.Println("work2 end!")  
}  
  
func childwork2(wg \*sync.WaitGroup, ctx \*context.Context) {  
 defer (\*wg).Done()  
FORLOOP:  
 for {  
 fmt.Println("childwork2 running...")  
 time.Sleep(200 \* time.***Millisecond***)  
 select {  
 case x := <-(\*ctx).Done(): //如果context.Done()有返回则退出work  
 fmt.Printf("childwork2收到结束消息,**%#v\n**", x)  
 break FORLOOP  
 default:  
 }  
 }  
}

func main() {  
 var wg sync.WaitGroup  
 ctx, cancel := context.WithCancel(context.Background())  
 wg.Add(1)  
 go work2(&wg, &ctx)  
 time.Sleep(5 \* time.***Second***) //主gorouine睡眠5s  
 cancel() //主gorouine睡眠5s后使用cancel()通知子goroutine逐级退出  
 wg.Wait()  
}

输出为：

work2 running...

childwork2 running...

childwork2 running...

childwork2 running...

work2 running...

childwork2 running...

childwork2 running...

work2 running...

childwork2 running...

childwork2 running...

childwork2 running...

work2 running...

childwork2 running...

childwork2 running...

work2 running...

childwork2 running...

childwork2 running...

childwork2 running...

work2 running...

childwork2 running...

childwork2 running...

work2 running...

childwork2 running...

childwork2 running...

work2 running...

childwork2 running...

childwork2 running...

childwork2 running...

work2 running...

childwork2 running...

childwork2 running...

work2 running...

childwork2 running...

childwork2 running...

work2收到结束消息,struct {}{}

work2 end!

childwork2收到结束消息,struct {}{}

在这个例子中，main函数中定义了一个WaitGroup,用于计数线程的数量，在main中放置wg,Wait()是为了保证主routine不会提前退出。

使用context时，一个context可以作为其他的context的父级context,所以在这里可以使用conetxt.BackGround()获取到根context(关于context的父子包含关系后面在讲述);

在有了wg和context之后，使用go语句发起协程work2函数，并把在main中的wg和context作为参数传入工作函数work2;

然后主routine睡眠5秒钟；

主线程睡眠5秒钟后，调用cancal()通知各个routine中的context终止协程的运行，也就是在主routine中，通过自主调用cancal()来控制其余的routine的退出；

在工作函数work2中使用defer wg.Done()是为了延迟给wg的计数减一，这对应的是在mian函数中发起work函数时的wg.Add(1)。

然后在work2函数中，for死循环每500毫秒使用一次select检测一下是否收到了context的信号，收到信号会使得select匹配到 <-(\*ctx).Done()的返回，进而直接break跳出FORLOOP标签所属的代码块，关于FOOLOOP标识只是一个标签，可以换成任意的其他的名称，在go中添加标签则意味着标签后的代码块均是属于该标签下；可以看到在输出中，当work2收到结束消息的时候，连着跳出了select和for两层循环，也就是跳出了FORLOOP标签对应的代码块，然后输出了work2 end!

在工作协程函数work2中又发起了一个子协程，运行函数childwork2，在发起协程之前设置wg.Add(1)，并也把wg和context传入。注意这里wg和context始终是在nain函数中创建的那些。

在工作函数childwork2中也和work2类似，添加延迟动作defer wg.Down()，这是对应于在work2函数中发起workchild2时的wg.Add(1)动作。以及for循环中使用select检测是否收到context的信号，一旦检测到<-(\*ctx).Done()有返回，那么break跳出代码块FORLOOP；注意虽然work2和workchild2都有相同的标签FORLOOP，但是标签的作用范围只在其所在的函数范围内，所以work2中跳转FORLOOP和workchild2的FORLOOP不会互相影响；

从打印结果可以看出，context发出的结束信号，是一个空的结构体。而context在发起cancel()信号后，信号的传输方式是先work2收到，然后workchild2收到，也就是先有父级的routine收到，在向下传递给子routine收到。