**Go学习笔记**

**V1.0.0**

**Author: clearuo**

Date: 2017-03-05

# 基础语法

## 基础类型和运算符

1. s:=123这种定义是int32还是int64还是其他类型？试着赋值给其给uint32或int64看是否需要强制转换

s:=123这种定义是int类型,需要强制转换才能赋值给uint32或uint64

2. 在 fmt.Printf 中使用下面的说明符来打印有关变量的相关信息：

* %+v 打印包括字段在内的实例的完整信息
* %#v 打印包括字段和限定类型名称在内的实例的完整信息
* %T 打印某个类型的完整说明

3.在 Go 语言中，&& 和 || 是具有快捷性质的运算符，当运算符左边表达式的值已经能够决定整个表达式的值的时候（&& 左边的值为 false，|| 左边的值为 true），运算符右边的表达式将不会被执行。利用这个性质，如果你有多个条件判断，应当将计算过程较为复杂的表达式放在运算符的右侧以减少不必要的运算。

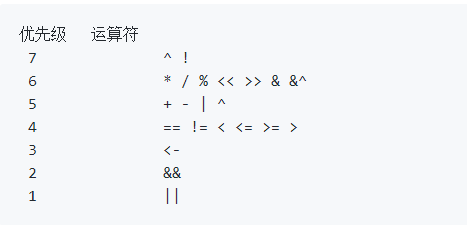
4.格式化说明

在格式化字符串里，%d 用于格式化整数（%x 和 %X 用于格式化 16 进制表示的数字），%g 用于格式化浮点型（%f 输出浮点数，%e 输出科学计数表示法），%0d 用于规定输出定长的整数，其中开头的数字 0 是必须的。

%n.mg 用于表示数字 n 并精确到小数点后 m 位，除了使用 g 之外，还可以使用 e 或者 f，例如：使用格式化字符串 %5.2e 来输出 3.4 的结果为 3.40e+00。

5.运算符与优先级

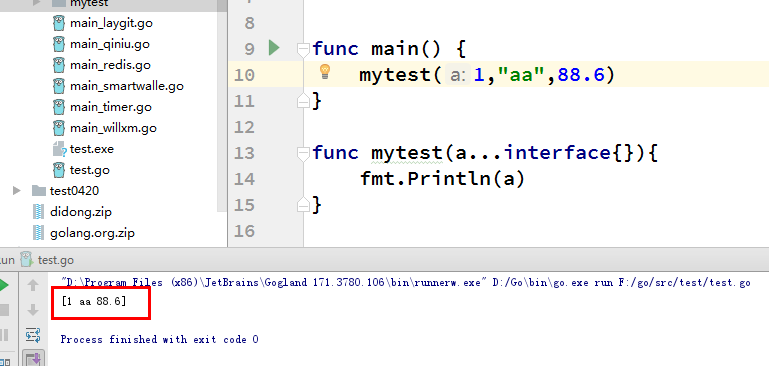
有些运算符拥有较高的优先级，二元运算符的运算方向均是从左至右。下表列出了所有运算符以及它们的优先级，由上至下代表优先级由高到低：



6.获取字符串中某个字节的地址的行为是非法的,例如:&str[i]

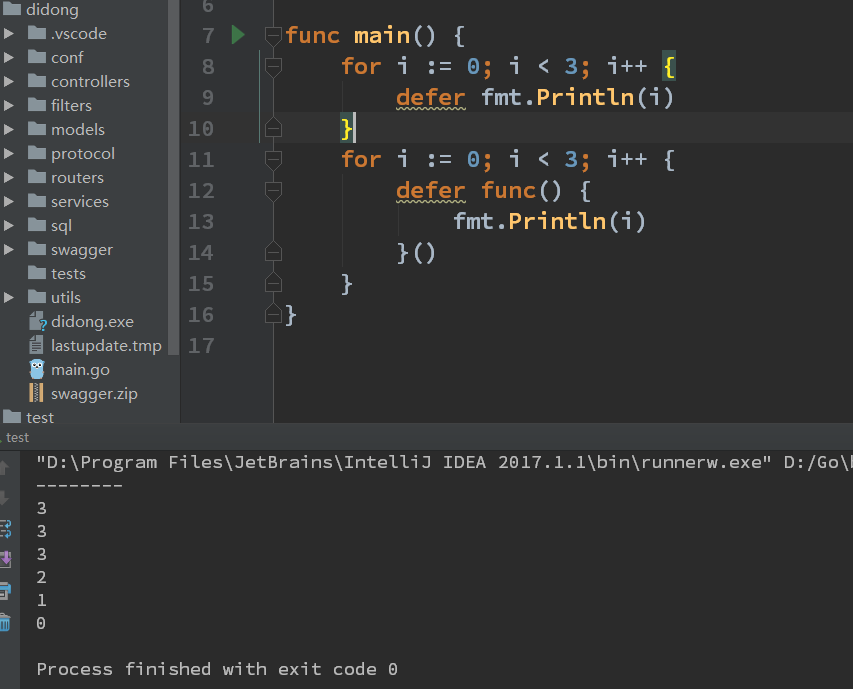
## 函数

1. func test(a…interface{})传递验证

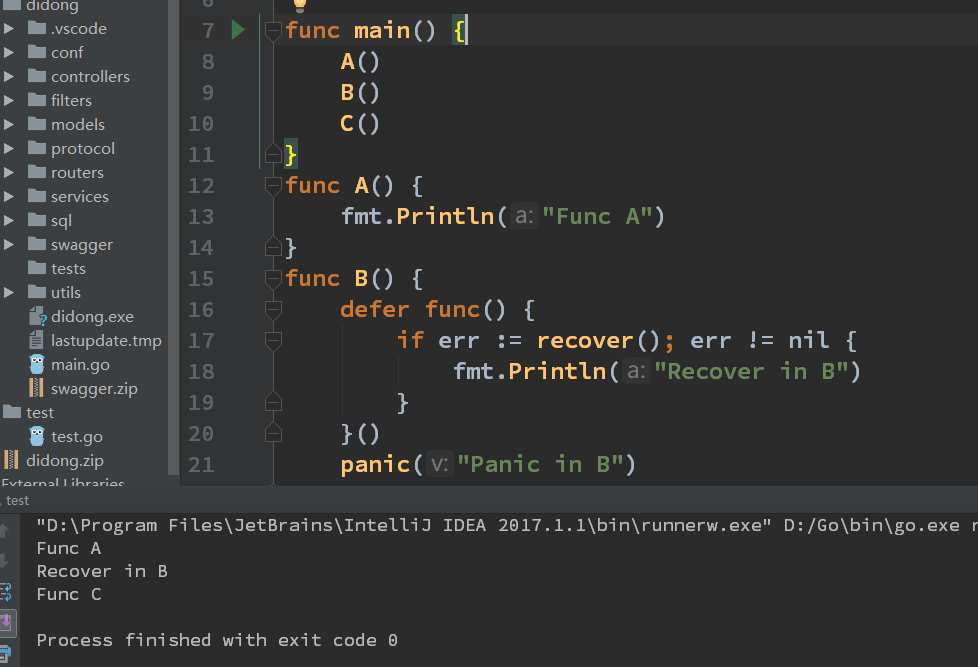


1. 闭包



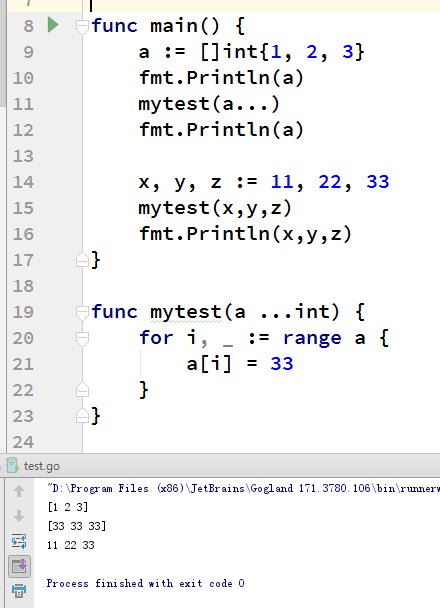


1. Panic,Recover

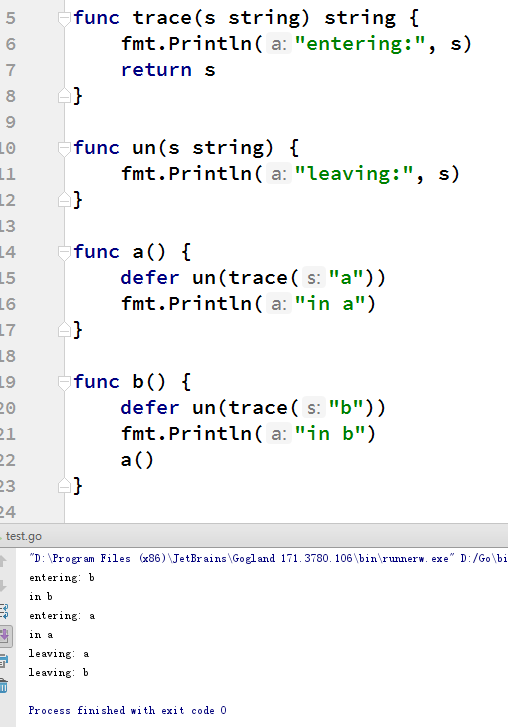




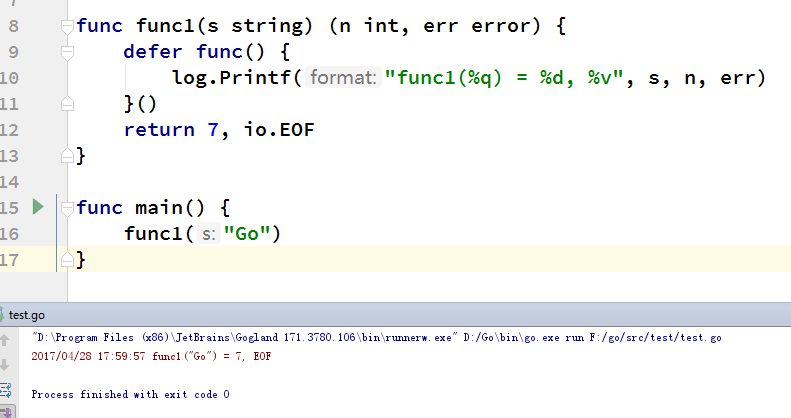
4.传递变长参数



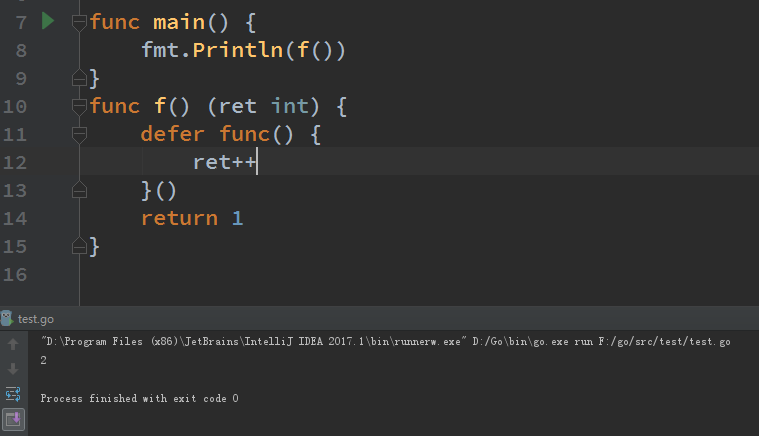
5.使用defer实现代码追踪



6.使用defer语句来记录函数的参数与返回值



1. 匿名函数同样被称之为闭包(函数时语言的术语):它们被允许调用定义在其它环境下的变量.闭包可使得某个函数扑捉到一些外部状态,例如:函数被创建时的状态.另一种表示方式为:一个闭包继承了函数所声明时的作用域.这种状态(作用域内的变量)都被共享到闭包的环境中,因此这些变量可以在闭包中被操作,直到被销毁.
2. 请学习以下示例并思考:函数f返回时,变量ret的值是什么?



变量ret的值为2,因为ret++是在执行return 1语句以后执行的,也就是说defer是在return以后才执行,这可用于在返回语句之后修改返回的error时使用

9.不使用递归但使用闭包实现斐波那契数列程序



1. 工厂函数

一个返回值为另一个函数的函数可以被称之为工厂函数，这在您需要创建一系列相似的函数的时候非常有用：书写一个工厂函数而不是针对每种情况都书写一个函数。下面的函数演示了如何动态返回追加后缀的函数：

func MakeAddSuffix(suffix string) func(string) string {

return func(name string) string {

if !strings.HasSuffix(name, suffix) {

return name + suffix

}

return name

}

}

现在，我们可以生成如下函数：

addBmp := MakeAddSuffix(“.bmp”)

addJpeg := MakeAddSuffix(“.jpeg”)

然后调用它们：

addBmp("file") // returns: file.bmp

addJpeg("file") // returns: file.jpeg

可以返回其它函数的函数和接受其它函数作为参数的函数均被称之为高阶函数，是函数式语言的特点。我们已经在第 6.7 中得知函数也是一种值，因此很显然 Go 语言具有一些函数式语言的特性。闭包在 Go 语言中非常常见，常用于 goroutine 和管道操作

## 数组与切片

### 基础概念

* 数组长度也是数组类型的一部分，所以[5]int和[10]int是属于不同类型的。数组的编译时值初始化是按照数组顺序完成的
* 元素的数目，也称为长度或者数组大小必须是固定的并且在声明该数组时就给出（编译时需要知道数组长度以便分配内存）；数组长度最大为 2Gb
* 如果我们想让数组元素类型为任意类型的话可以使用空接口作为类型当使用值时我们必须先做一个类型判断
* 由于长度也是数组类型的一部分，因此[3]int与[4]int是不同的类型，数组也就不能改变长度。数组之间的赋值是值的赋值，即当把一个数组作为参数传入函数的时候，传入的其实是该数组的副本，而不是它的指针。如果要使用指针，那么就需要用到后面介绍的slice类型了。验证函数传递

|  |
| --- |
| func main() {     arr := [...]int{1, 2, 3}     fmt.Println("all before:", arr)     test\_value(arr)     fmt.Println("value after:", arr)     test\_point(&arr)     fmt.Println("point after:", arr)     test\_slice(arr[:])     fmt.Println("slice after:", arr) } func test\_value(arr [3]int) {     arr[1] = 111     fmt.Println("test\_value:", arr) } func test\_point(arr \*[3]int) {     arr[1] = 222     fmt.Println("test\_point:", arr) } func test\_slice(arr []int) {     arr[1] = 333     fmt.Println("test\_slice:", arr) } |

* 注意 绝对不要用指针指向 slice。切片本身已经是一个引用类型，所以它本身就是一个指针!!
* []interface{}这种切片是否可以存储不同数据类型
* 可以,因为该切片存储的类型是interface{}
* 通过切片引用传递给函数,在函数中修改此切片后,调用者得到的切片一定是函数中修改后的内容吗?
* 未必,通过以此达到返回数据给调用者是不安全的,因为如果在此函数中操作切片内容大于原有切片的空间的话,将从新分配空间,此时调用者使用的还是原来数组控件,如下例子:

|  |
| --- |
| func main() {    a := []byte{0, 1, 2}    fmt.Println(a)    mytest(a)    fmt.Println(a) } func mytest(a []byte){    a[0]=00    a[1]=11    a[2]=22    a = append(a,55)    a[0]=77    fmt.Println("cap(a) = ",cap(a)) } // 运行结果 // [0 1 2]  // cap(a) =  8 // [0 11 22] |

### new和make的区别

* 看起来二者没什么区别,都在堆上分配内存,但是它们的行为不同个,适用于不同的类型;
* new(T)为每个新的类型T分配一片内存,初始化为0并且返回类型为\*T的内存地址:这种方法返回一个指向类型为T,值为0的地址的指针,它适用于值类型,如数组和结构体;它相当于&T{};
* make(T)返回一个类型为T的初始值,它只适用于3种内建的引用类型:切片,map和channel;
* 换言之,new函数分配内存,make函数初始化;

### append

* 如果你想将切片 y 追加到切片 x 后面，只要将第二个参数扩展成一个列表即可：x = append(x, y...)
* 将一个字符串追加到某一个字符数组的尾部

|  |
| --- |
| func main() {     str:="abc中国sd"     var b[]byte     b = append(b,str...)     fmt.Println(string(b)) } |

* append函数常见操作
* 将切片 b 的元素追加到切片 a 之后：a = append(a, b...)
* 复制切片 a 的元素到新的切片 b 上：b = make([]T, len(a)); copy(b, a)
* 删除位于索引 i 的元素：a = append(a[:i], a[i+1:]...)
* 切除切片 a 中从索引 i 至 j 位置的元素：a = append(a[:i], a[j:]...)
* 为切片 a 扩展 j 个元素长度：a = append(a, make([]T, j)...)
* 在索引 i 的位置插入元素 x：a = append(a[:i], append([]T{x}, a[i:]...)...)
* 在索引 i 的位置插入长度为 j 的新切片：a = append(a[:i], append(make([]T, j), a[i:]...)...)
* 在索引 i 的位置插入切片 b 的所有元素：a = append(a[:i], append(b, a[i:]...)...)
* 取出位于切片 a 最末尾的元素 x：x, a = a[len(a)-1], a[:len(a)-1]
* 将元素 x 追加到切片 a：a = append(a, x)
* 因此，您可以使用切片和 append 操作来表示任意可变长度的序列。
* 从数学的角度来看，切片相当于向量，如果需要的话可以定义一个向量作为切片的别名来进行操作。

****注意****： append 在大多数情况下很好用，但是如果你想完全掌控整个追加过程，你可以实现一个这样的 AppendByte 方法：

func AppendByte(slice []byte, data ...byte) []byte {

m := len(slice)

n := m + len(data)

if n > cap(slice) { // if necessary, reallocate

// allocate double what's needed, for future growth.

newSlice := make([]byte, (n+1)\*2)

copy(newSlice, slice)

slice = newSlice

}

slice = slice[0:n]

copy(slice[m:n], data)

return slice

}

func copy(dst, src []T) int copy 方法将类型为 T 的切片从源地址 src 拷贝到目标地址 dst，覆盖 dst 的相关元素，并且返回拷贝的元素个数。源地址和目标地址可能会有重叠。拷贝个数是 src 和 dst 的长度最小值。如果 src 是字符串那么元素类型就是 byte。如果你还想继续使用 src，在拷贝结束后执行 src = dst。

### copy

* 假设s是一个字符串(本质上是一个字节数组),那么就可以通过c:=[]byte(s)来获取一个字节的切片c,另外,您还可以通过copy函数来达到相同的目的:copy(dst []byte, src string)

### 切片和垃圾回收

* 切片的底层指向一个数组,该数组的实际容量可能要大于切片所定义的容量.只有在没有任何切片指向的时候,底层的数组内层才会被释放,这种特性有时会导致程序占用多余的内存.
* 示例函数FindDigits将一个文件加载到内存,然后搜索其中所有的数字并返回一个切片.

|  |
| --- |
| var digitRegexp = regexp.MustCompile("[0-9]+") func main() {     FindDigits("./test.txt") } func FindDigits(filename string) []byte {     b, \_ := ioutil.ReadFile(filename)     return digitRegexp.Find(b) } |

* 这段代码可以顺利运行,但返回的[]byte指向的底层是整个文件的数据.只要该返回的切片不被释放,垃圾回收器就不能释放整个文件所占用的内存.换句话说,一点点有用的数据却占用了整个文件的内存.
* 想要避免这个问题,可以通过拷贝我们需要的部分到一个新的切片中:

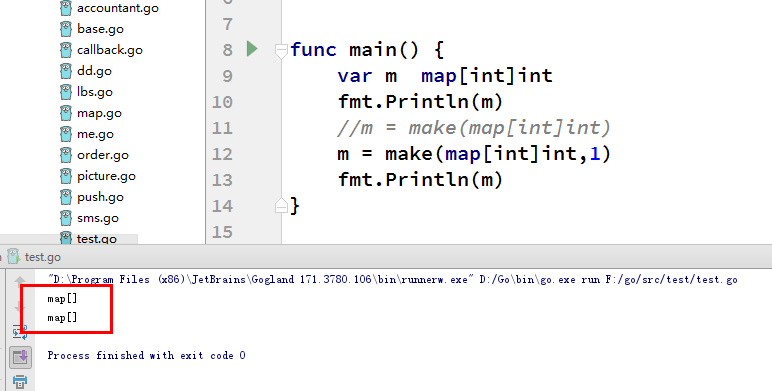
|  |
| --- |
| var digitRegexp = regexp.MustCompile("[0-9]+") func main() {     FindDigits("./test.txt") } func FindDigits(filename string) []byte {     b, \_ := ioutil.ReadFile(filename)     return digitRegexp.Find(b)     c := make([]byte, len(b))     copy(c, b)     return c } |

## Map

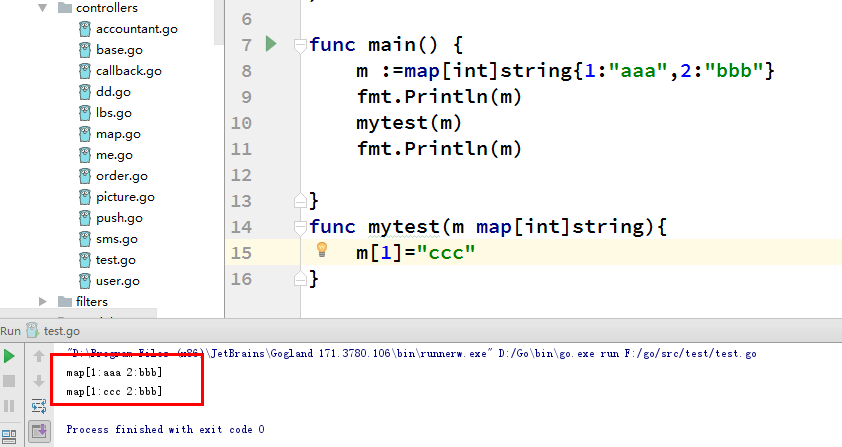
### 基础概念

* map 传递给函数的代价很小：在 32 位机器上占 4 个字节，64 位机器上占 8 个字节，无论实际上存储了多少数据。
* 通过 key 在 map 中寻找值是很快的，比线性查找快得多，但是仍然比从数组和切片的索引中直接读取要慢 100 倍；所以如果你很在乎性能的话还是建议用切片来解决问题。
* 如果 key1 是 map1 的key，那么 map1[key1] 就是对应 key1 的值，就如同数组索引符号一样（数组可以视为一种简单形式的 map，key 是从 0 开始的整数）。
* 常用的 len(map1) 方法可以获得 map 中的 pair 数目，这个数目是可以伸缩的，因为 map-pairs 在运行时可以动态添加和删除。
* 不要使用 new，永远用 make 来构造 map
* 注意 如果你错误的使用 new() 分配了一个引用对象，你会获得一个空引用的指针，相当于声明了一个未初始化的变量并且取了它的地址：
* mapCreated := new(map[string]float32)

1. 未初始化的map打印出来是怎样？map[]吗？初始化后的呢？



1. Map是值类型还是引用类型?



## 指针

1.指向数组的指针和指针数组

|  |
| --- |
| func main() {     a := [...]int{9: 1}     var p \*[10]int = &a //p是指向数组的指针     fmt.Println(p)      x, y := 1, 2     b := [...]\*int{&x, &y} //b是指针数组     fmt.Println(b) } |

2. 哪些是值传递,哪些是引用传递?(例如,struct,map,slice,数组,接口,基本类型等) 什么类型的变量不需要(\*s)这样应用可以直接s,哪些些只能\*s ?

* 基本数据,如int,float,string等是值类型,基本类型如果是指针都需要\*s
* 数组是值类型，不管是指向数组本身还是指向数组本身，都可以用a[1]这种方式，指针不需要前面加\*
* 结构体是值类型,结构体无论是值还是指针,都可以直接使用s
* 切片是引用类型,
* Map是引用类型,
* Channel是引用类型

3.函数传递

* 传指针使得多个函数能操作同一个对象。
* 传指针比较轻量级 (8bytes),只是传内存地址，我们可以用指针传递体积大的结构体。如果用参数值传递的话, 在每次copy上面就会花费相对较多的系统开销（内存和时间）。所以当你要传递大的结构体的时候，用指针是一个明智的选择。
* Go语言中channel，slice，map这三种类型的实现机制类似指针，所以可以直接传递，而不用取地址后传递指针。（注：若函数需改变slice的长度，则仍需要取地址传递指针）

## 包

1. 实验多个代码文件共用同一个包，并且有init函数?

当引入包有init函数的时候,优先执行该init函数.并且当同个项目中多次引入同个包时,只执行一次init函数.

2. golang包名必须与所在文件夹同名吗?

不必须，但是同个目录中（不含子目录）的所有文件包名必须一致，通常为了方便包定位，建议包名和目录名一致，否则你import "A"，使用起来B.xxx，看上去不统一，不能一眼看出来这个B包是哪个文件的。

1. src目录下有mytest文件夹，里面有mytest.go包，引用该包的时候为什么是import "mytest" 而不是import "mytest/mytest" ？请上机实验.

import引号后面的只是包所在路径,具体使用的时候是使用该路径下的包名(一般最好和所在文件夹名称一致)来使用. 例如fmt.Println(“hello,world”) 我们调用了fmt包里面定义的函数Println。大家可以看到，这个函数是通过<pkgName>.<funcName>的方式调用的，这一点和Python十分相似。前面提到过，包名和包所在的文件夹名可以是不同的，此处的<pkgName>即为通过package <pkgName>声明的包名，而非文件夹名。

1. GOPATH目录下三个子目录,src,pkg,bin,当我们引入第三方包时,源代码都是在src目录下,如果安装该包,那么编译后的包文件会在pkg子目录下,但是为什么使用该包的时候还是需要源代码存在?而不是只要pkg子目录下的\*.a包文件就可以了?
2. 定义在包中函数内部的大写字母开头的变量，包外可以访问吗？

不行,必须是包中函数外部的公有变量,包外部才可以访问.

6.结构体中变量的首字母大小写，包外部访问性如何？

不可以直接访问;

如果该结构体变量定义在包中,并且该结构体变量名为可访问性质,那么包外可以直接访问整个结构体,但是不能访问结构体内部的变量;

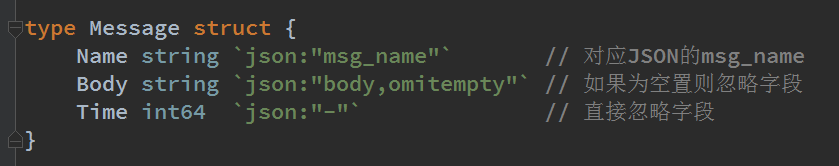
如果如果结构体变量定义在包外部,那么连这个结构体变量都不能定义

7.引入一个包的执行流程?



## 结构体

1.结构体序列化到字符串时,指定某些字段如果为空则不序列话,或者指定某字段不序列化

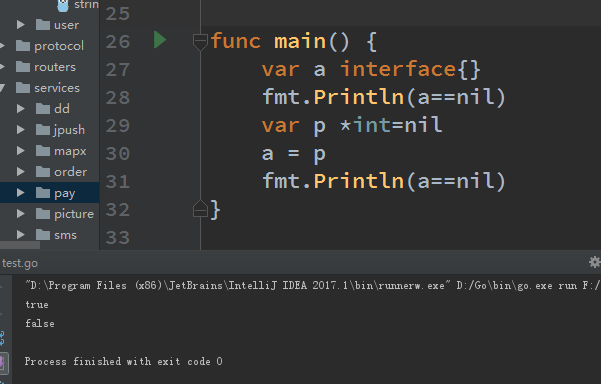


## 接口

1.接口可以匿名嵌入其它接口,或嵌入到结构体中

2.将对象赋值给接口时,会发生拷贝,而接口内部存储的是指向这个复制品的指针,既无法修改原始对象的状态,也无法获取指针

3.只有当接口存储的类型和对象都为nil时,接口才等于nil



1. 接口调用不会做receiver的自动转换

# 官方标准库

## 详情参文档:

<http://studygolang.com/pkgdoc>

## strings

### strings.Map()

func Map(mapping func(rune) rune, s string) string

将s的每一个unicode码值r都替换为mapping(r)，返回这些新码值组成的字符串拷贝。如果mapping返回一个负值，将会丢弃该码值而不会被替换。（返回值中对应位置将没有码值）

### strings.IndexFunc()

func IndexFunc(s string, f func(rune) bool) int

s中第一个满足函数f的位置i（该处的utf-8码值r满足f(r)==true），不存在则返回-1。

## unicode/utf8

### utf8.RunCountInString

|  |  |
| --- | --- |
|  | func main() {     str:="abc中d"     fmt.Println(len(str))     fmt.Println(len([]int32(str)))     // 以下方法效率更高     fmt.Println(utf8.RuneCountInString(str)) } /\* 运行结果 7 5 5 \*/ |

## bytes

类型 []byte 的切片十分常见，Go 语言有一个 bytes 包专门用来解决这种类型的操作方法。

bytes 包和字符串包十分类似（参见第 4.7 节）。而且它还包含一个十分有用的类型 Buffer:

import "bytes"

type Buffer struct {

...

}

这是一个长度可变的 bytes 的 buffer，提供 Read 和 Write 方法，因为读写长度未知的 bytes 最好使用 buffer。

Buffer 可以这样定义：var buffer bytes.Buffer。

或者使用 new 获得一个指针：var r \*bytes.Buffer = new(bytes.Buffer)。

或者通过函数：func NewBuffer(buf []byte) \*Buffer，创建一个 Buffer 对象并且用 buf 初始化好；NewBuffer 最好用在从 buf 读取的时候使用。

****通过 buffer 串联字符串****

类似于 Java 的 StringBuilder 类。

在下面的代码段中，我们创建一个 buffer，通过 buffer.WriteString(s) 方法将字符串 s 追加到后面，最后再通过buffer.String() 方法转换为 string：

var buffer bytes.Bufferfor {

if s, ok := getNextString(); ok { //method getNextString() not shown here

buffer.WriteString(s)

} else {

break

}

}

fmt.Print(buffer.String(), "\n")

这种实现方式比使用 += 要更节省内存和 CPU，尤其是要串联的字符串数目特别多的时候。

## sort

* sort.Ints(a []int)
* sort.IntsAreSorted(a []int)
* sort.Float64s(a []float64)
* sort.SearchInts(a []int, n int)
* 必须先排序再搜索