**GO基础**

1、源文件以“go”为拓展名

2、程序的执行入口是main()函数

3、严格区分大小写，go语言的大小写含有特殊意义，不可无逻辑的使用

4、go的方法由一条条的语句构成，每个语句前后不需要用 ; 区分开，go会在语句后自动添加

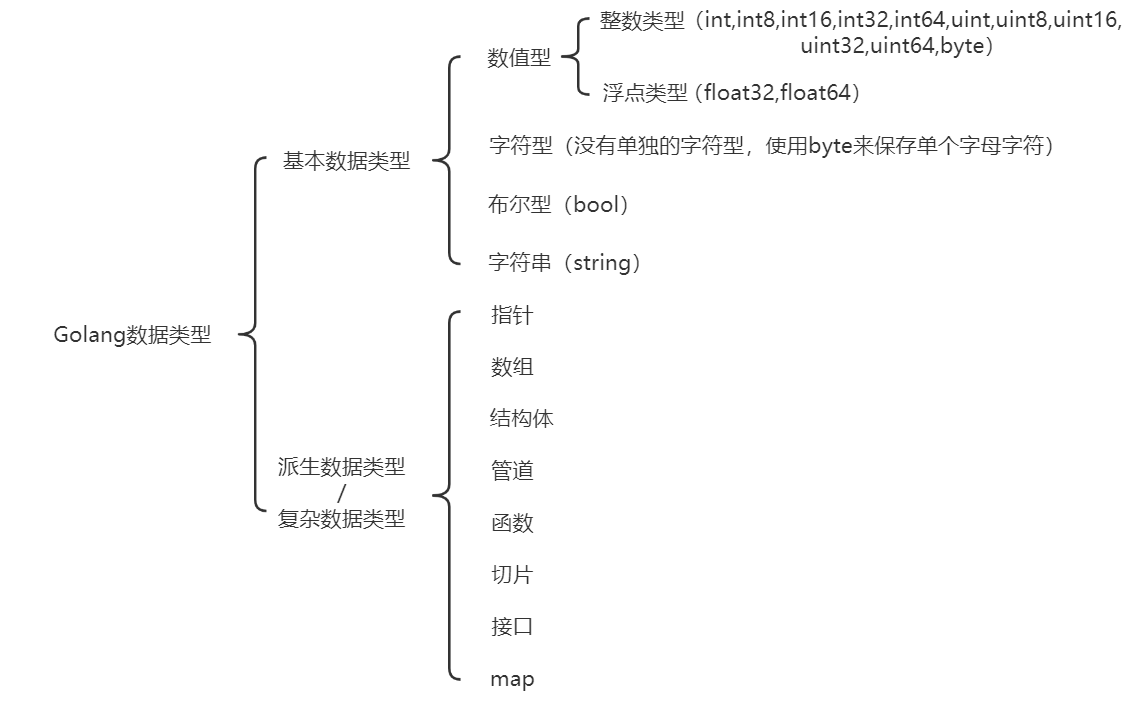
5、Go编译器是一行一行进行编译的，不同的语句必须写在不同的行，否则会语法报错。

6、代码中存在未使用的包或者变量，语法报错。（但后面会有一个 \_ 来解决import的问题）

* **变量**

|  |
| --- |
| Go  //定义在{}中的变量叫：局部变量  //第一种：变量的使用方式：指定变量的类型，并且赋值，  var num int = 18  fmt.Println(num)  //第二种：指定变量的类型，但是不赋值，使用默认值  var num2 int  fmt.Println(num2)  //第三种：如果没有写变量的类型，那么根据=后面的值进行判定变量的类型 （自动类型推断）  var num3 = "tom"  fmt.Println(num3)  //第四种：省略var，注意 := 不能写为 =  sex := "男"  fmt.Println(sex)  //声明多个变量：  var n1,n2,n3 int |

* **数据类型和数据类型的强制转换**



Go在不同类型的变量之间赋值时需要显式转换，并且只有显式转换(强制转换)。

语法：

表达式T(v)将值v转换为类型T

T : 就是数据类型

v : 就是需要转换的变量

|  |
| --- |
| Go  //进行类型转换：  var n1 int = 100  //var n2 float32 = n1 在这里自动转换不好使，比如显式转换  fmt.Println(n1)  //fmt.Println(n2)  var n2 float32 = float32(n1)  fmt.Println(n2)  //注意：n1的类型其实还是int类型，只是将n1的值100转为了float32而已，n1还是int的类型  fmt.Printf("%T",n1) //int  fmt.Println()  //将int64转为int8的时候，编译不会出错的，但是会数据的溢出  var n3 int64 = 888888  var n4 int8 = int8(n3)  fmt.Println(n4)//56  var n5 int32 = 12  var n6 int64 = int64(n5) + 30 //一定要匹配=左右的数据类型  fmt.Println(n5)  fmt.Println(n6)  var n7 int64 = 12  var n8 int8 = int8(n7) + 127 //编译通过，但是结果可能会溢出  //var n9 int8 = int8(n7) + 128 //编译不会通过  fmt.Println(n8)  //fmt.Println(n9) |

1. **基本数据类型和string的转换**

1）fmt.Sprintf("%参数",表达式) ---> 重点练习这个，推荐方式

|  |
| --- |
| Go  var n1 int = 19  var n2 float32 = 4.78  var n3 bool = false  var n4 byte = 'a'  var s1 string = fmt.Sprintf("%d",n1)  fmt.Printf("s1对应的类型是：%T ，s1 = %q \n",s1, s1)  var s2 string = fmt.Sprintf("%f",n2)  fmt.Printf("s2对应的类型是：%T ，s2 = %q \n",s2, s2)  var s3 string = fmt.Sprintf("%t",n3)  fmt.Printf("s3对应的类型是：%T ，s3 = %q \n",s3, s3)  var s4 string = fmt.Sprintf("%c",n4)  fmt.Printf("s4对应的类型是：%T ，s4 = %q \n",s4, s4) |

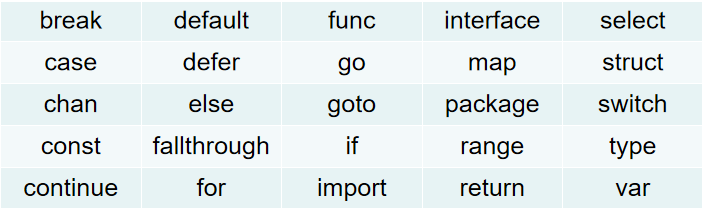
2）使用strconv包的函数

|  |
| --- |
| Go  var n1 int = 18  var s1 string = strconv.FormatInt(int64(n1),10) //参数：第一个参数必须转为int64类型 ，第二个参数指定字面值的进制形式为十进制  fmt.Printf("s1对应的类型是：%T ，s1 = %q \n",s1, s1)  var n2 float64 = 4.29  var s2 string = strconv.FormatFloat(n2,'f',9,64)  //第二个参数：'f'（-ddd.dddd） 第三个参数：9 保留小数点后面9位 第四个参数：表示这个小数是float64类型  fmt.Printf("s2对应的类型是：%T ，s2 = %q \n",s2, s2)  var n3 bool = true  var s3 string = strconv.FormatBool(n3)  fmt.Printf("s3对应的类型是：%T ，s3 = %q \n",s3, s3) |

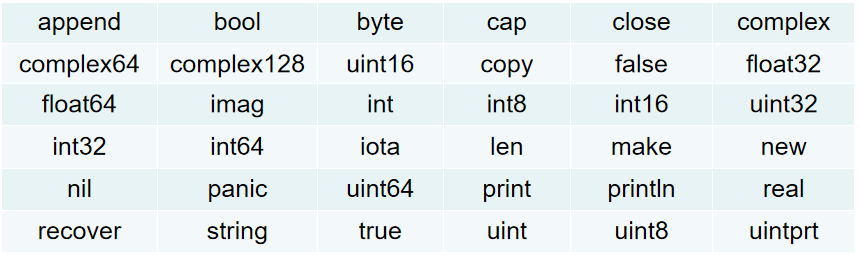
1. **string类型转基本类型**

|  |
| --- |
| Go  //string-->bool  var s1 string = "true"  var b bool  //ParseBool这个函数的返回值有两个：(value bool, err error)  //value就是我们得到的布尔类型的数据，err出现的错误  //我们只关注得到的布尔类型的数据，err可以用\_直接忽略  b , \_ = strconv.ParseBool(s1)  fmt.Printf("b的类型是：%T,b=%v \n",b,b)  //string---》int64  var s2 string = "19"  var num1 int64  num1,\_ = strconv.ParseInt(s2,10,64)  fmt.Printf("num1的类型是：%T,num1=%v \n",num1,num1)  //string-->float32/float64  var s3 string = "3.14"  var f1 float64  f1,\_ = strconv.ParseFloat(s3,64)  fmt.Printf("f1的类型是：%T,f1=%v \n",f1,f1)  //注意：string向基本数据类型转换的时候，一定要确保string类型能够转成有效的数据类型，否则最后得到的结果就是按照对应类型的默认值输出  var s4 string = "golang"  var b1 bool  b1 , \_ = strconv.ParseBool(s4)  fmt.Printf("b1的类型是：%T,b1=%v \n",b1,b1)  var s5 string = "golang"  var num2 int64  num2,\_ = strconv.ParseInt(s5,10,64)  fmt.Printf("num2的类型是：%T,num2=%v \n",num2,num2 |

* **关键字**



* **预定义标识符**



* **流程控制**

1. **if**

|  |
| --- |
| Go  /\*  语法  if 条件 {  //若上述条件为TRUE，则执行逻辑代码。否则不执行或执行其余的else分支  }  \*/    if a：=1；a==1{  fmt.Printf("age==1 is TRUE\n")  }else{  fmt.Printf("age==1 is false\n")  } |

1. **switch**

|  |
| --- |
| Go  /\*  常规语法  switch 表达式{  case1:  break;  case2:  break;  ...  default:  break;  }  \*/ |

1. **for和for range**

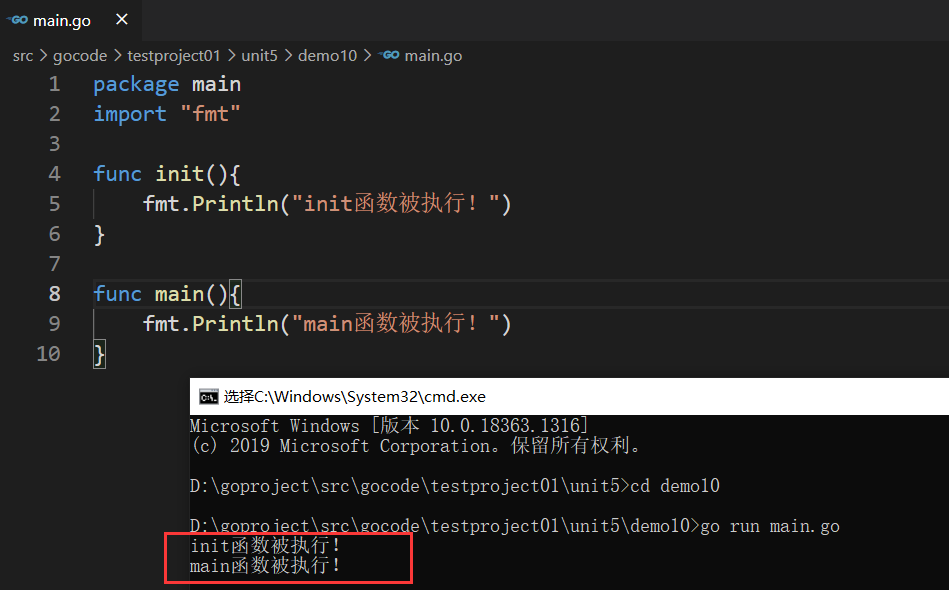
|  |
| --- |
| Go  /\*  语法  for 初始表达式;布尔表达式;迭代公式{  //逻辑代码  }  \*/    for i:=1;i<=10;i++{  i=i+1  }      /\*  语法  for 索引,变量值:=range 变量{  //逻辑代码  }  \*/  var str string = "wangxu"  for i,value := range str{  fmt.Printf("索引为：%d，具体的值为：%c\n",i,value)  }      //定义一个字符串：  var str string = "hello golang你好"  //方式1：普通for循环：按照字节进行遍历输出的 （暂时先不使用中文）  for i := 0;i < len(str);i++ {//i:理解为字符串的下标  fmt.Printf("%c \n",str[i])  }  //方式2：for range  for i , value := range str {  fmt.Printf("索引为：%d,具体的值为：%c \n",i,value)  }  //对str进行遍历，遍历的每个结果的索引值被i接收，每个结果的具体数值被value接收  //遍历对字符进行遍历的 |

* **函数**

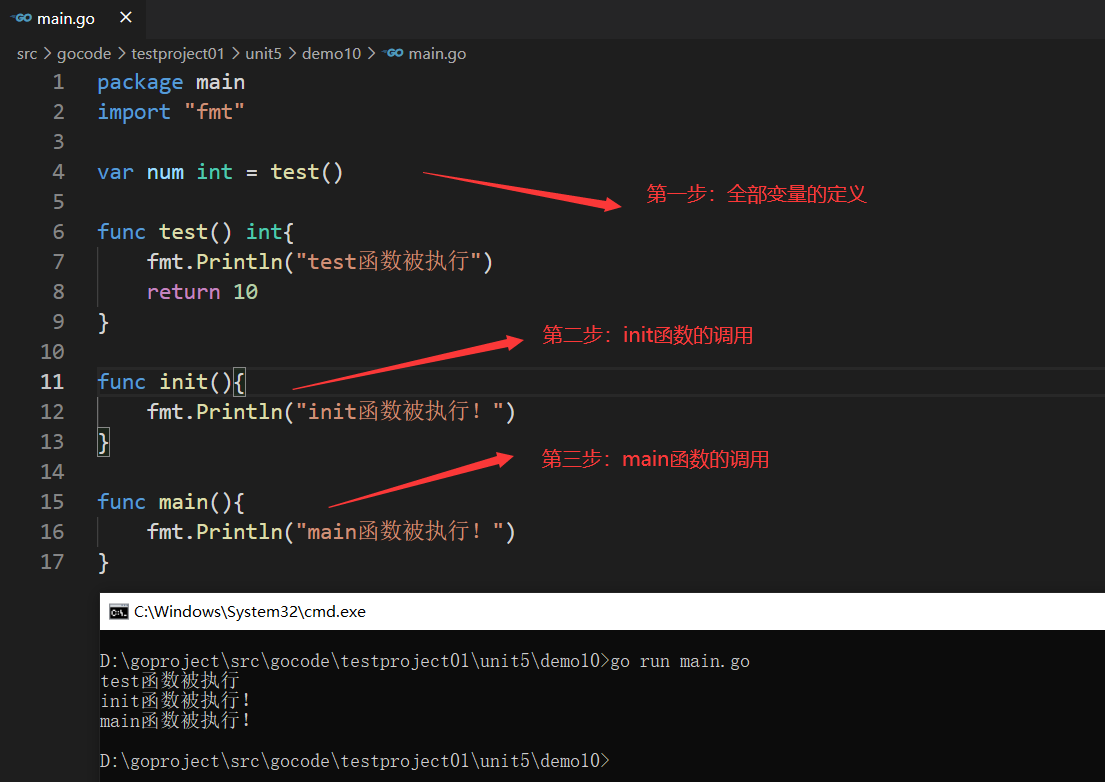
|  |
| --- |
| Go  /\*  func 函数名(形参)(返回值){  ...  //return //有返回值就需要return  }  \*/    //有形参，有返回值的函数  func GetSum(a int,b int)(sum int){  sum = a+b  return sum  }    //无形参，无返回值的函数  func main(){  }    //无形参，有返回值的函数  func test()(ret int){  ret=0  return ret  } |

1. **init函数**

1）初始化函数，可以用来进行一些初始化的操作。每一个源文件都可以包含一个init函数，该函数会在main函数执行前，被Go运行框架调用。



2）全局变量定义，init函数，main函数的执行流程



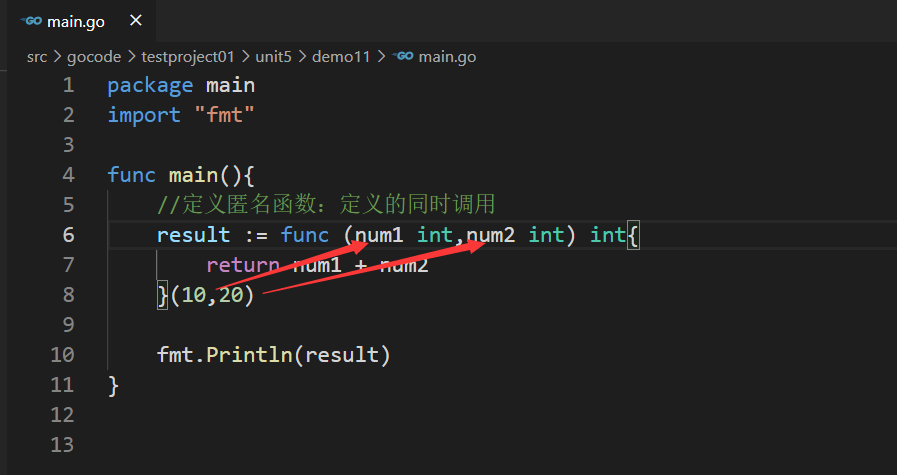
3）多个源文件都有init函数的时候，如何执行：理解成C工程中的头文件，头文件中的init函数会先执行

1. **匿名函数**

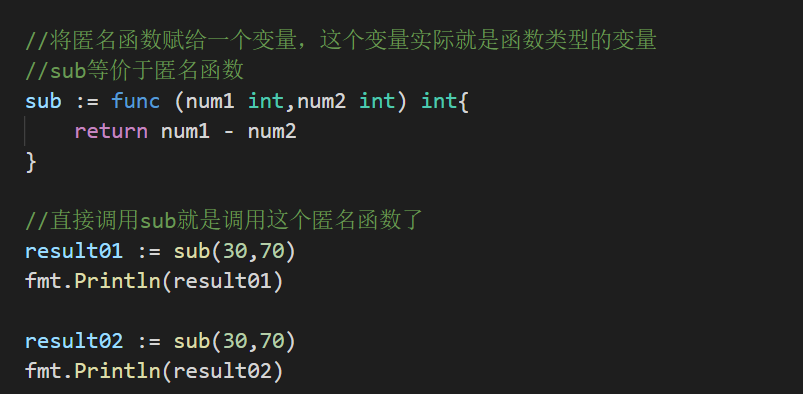
1）Go支持匿名函数，如果我们某个函数只是希望使用一次，可以考虑使用匿名函数

2）匿名函数使用方式：

在定义匿名函数时就直接调用，这种方式匿名函数只能调用一次（用的多）



将匿名函数赋给一个变量(该变量就是函数变量了)，再通过该变量来调用匿名函数（用的少）



如何让一个匿名函数，可以在整个程序中有效呢?将匿名函数给一个全局变量就可以了

|  |
| --- |
| Go  package main    import "fmt"    var Func01 = func (num1 int,num2 int) int{  return num1 \* num2  }  func main(){  //定义匿名函数：定义的同时调用  result := func (num1 int,num2 int) int{  return num1 + num2  }(10,20)  fmt.Println(result)  //将匿名函数赋给一个变量，这个变量实际就是函数类型的变量  //sub等价于匿名函数  sub := func (num1 int,num2 int) int{  return num1 - num2  }  //直接调用sub就是调用这个匿名函数了  result01 := sub(30,70)  fmt.Println(result01)    result02 := sub(30,70)  fmt.Println(result02)    result03 := Func01(3,4)  fmt.Println(result03)  } |

1. **闭包**

1）闭包就是一个函数和与其相关的引用环境组合的一个整体，匿名函数中引用的那个变量会一直保存在内存中，可以一直使用

|  |
| --- |
| Go  package main  import "fmt"  //函数功能：求和  //函数的名字：getSum 参数为空  //getSum函数返回值为一个函数【func (int) int】，这个函数的参数是一个int类型的参数，返回值也是int类型  func getSum() func (int) int {  var sum int = 0  return func (num int) int{  sum = sum + num  return sum  }  }  //闭包：返回的匿名函数+匿名函数以外的变量sum  func main(){  f := getSum()  fmt.Println(f(1))//1  fmt.Println(f(2))//3  fmt.Println(f(3))//6  fmt.Println(f(4))//10  } |

2）闭包的本质：

闭包本质依旧是一个匿名函数，只是这个函数引入外界的变量/参数

匿名函数+引用的变量/参数 = 闭包

3）闭包的特点：

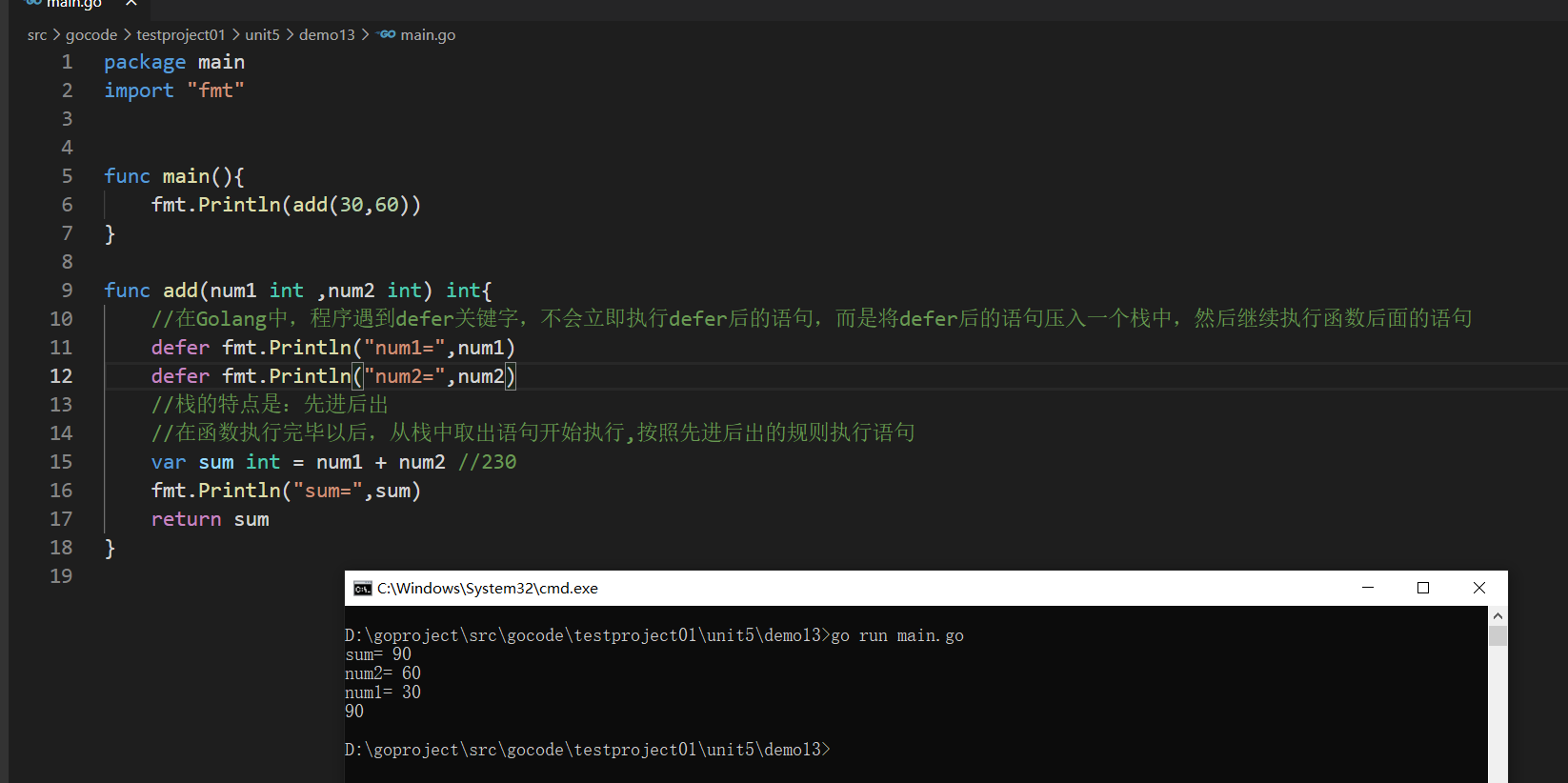
返回的是一个匿名函数，但是这个匿名函数引用到函数外的变量/参数 ,因此这个匿名函数就和变量/参数形成一个整体，构成闭包。

闭包中使用的变量/参数会一直保存在内存中，所以会一直使用---》意味着闭包不可滥用（对内存消耗大）

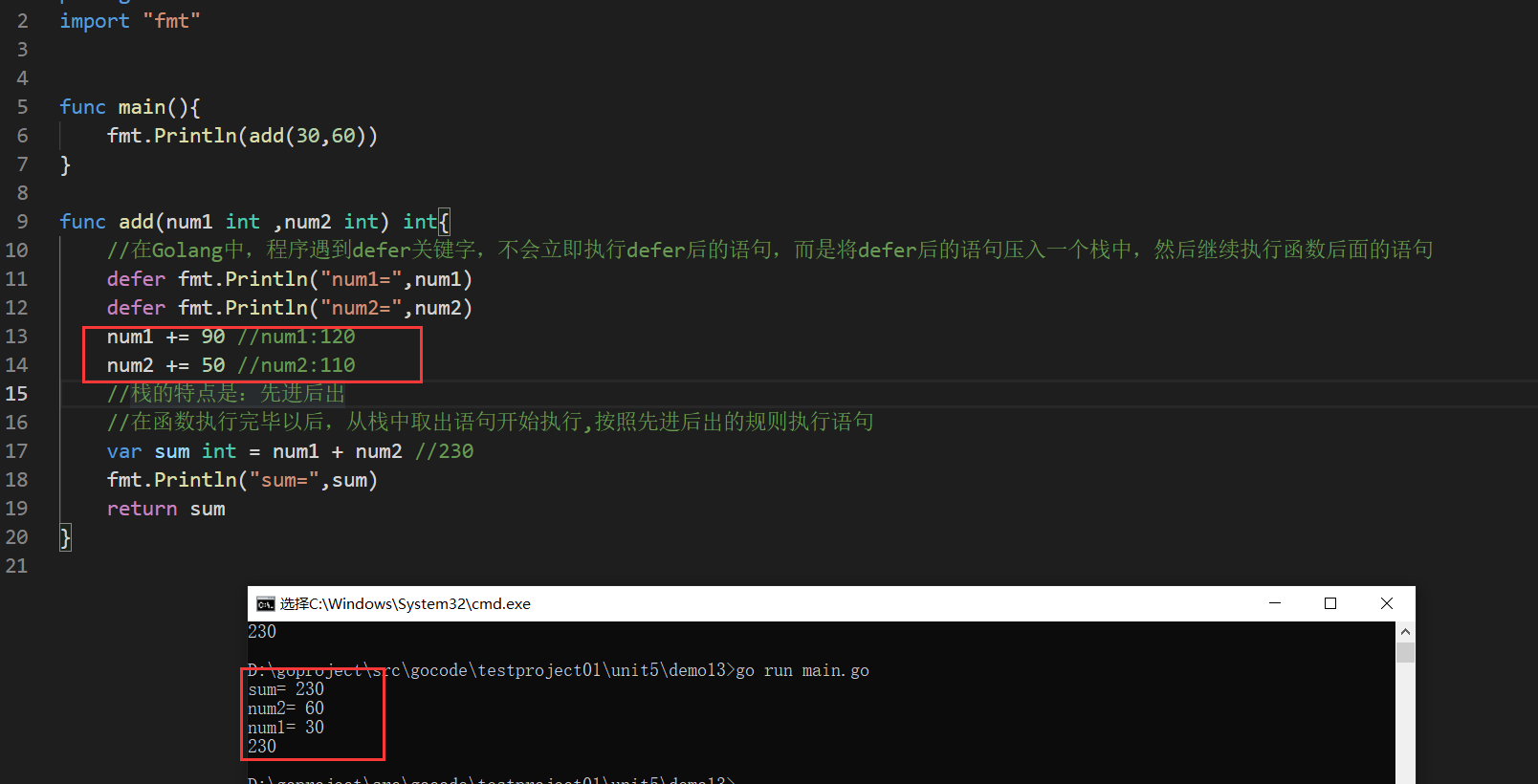
4）闭包应用场景：闭包可以保留上次引用的某个值，我们传入一次就可以反复使用了

1. **defer关键字**

在函数中，程序员经常需要创建资源，为了在函数执行完毕后，及时的释放资源，Go的设计者提供defer关键字



代码变动一下，再次看结果，发现：遇到defer关键字，会将后面的代码语句压入栈中，也会将相关的值同时拷贝入栈中，不会随着函数后面的变化而变化



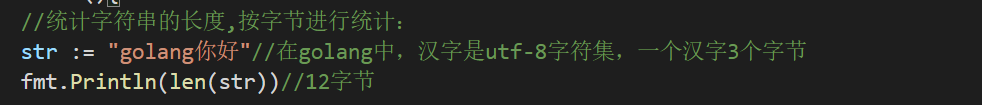
defer应用场景：

比如你想关闭某个使用的资源，在使用的时候直接随手defer，因为defer有延迟执行机制（函数执行完毕再执行defer压入栈的语句），所以你用完随手写了关闭，比较省心，省事

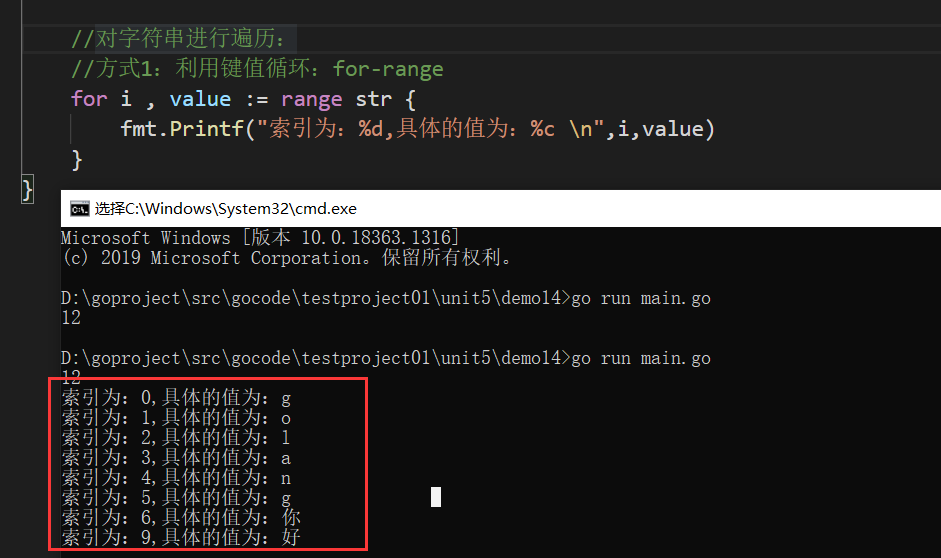
1. **字符串函数**

len(str)

统计字符串的长度,按字节进行统计



for-range



字符串转整数：

n, err := strconv.Atoi("66")

整数转字符串：

str = strconv.Itoa(6887)

查找子串是否在指定的字符串中:

strings.Contains("javaandgolang", "go")

统计一个字符串有几个指定的子串:

strings.Count("javaandgolang","a")

不区分大小写的字符串比较:

strings.EqualFold("go" , "Go")

返回子串在字符串第一次出现的索引值，如果没有返回-1 :

strings.lndex("javaandgolang" , "a")

字符串的替换：n可以指定你希望替换几个,如果n=-1表示全部替换，替换两个n就是2

strings.Replace("goandjavagogo", "go", "golang", n)

按照指定的某个字符，为分割标识，将一个学符串拆分成字符串数组:

strings.Split("go-python-java", "-")

将字符串的字母进行大小写的转换:

strings.ToLower("Go")// go

strings.ToUpper（"go")//Go

将字符串左右两边的空格去掉:

strings.TrimSpace(" go and java ")

将字符串左右两边指定的字符去掉:

strings.Trim("~golang~ ", " ~")

将字符串左边指定的字符去掉:

strings.TrimLeft("~golang~", "~")

将字符串右边指定的字符去掉:

strings.TrimRight("~golang~", "~")

判断字符串是否以指定的字符串开头:

strings.HasPrefix("http://java.sun.com/jsp/jstl/fmt", "http")

判断字符串是否以指定的字符串结束:

strings.HasSuffix("demo.png", ".png")