@[toc]

# Go 选择控制

Go语言支持最基本的三种程序运行结构：顺序结构、选择结构、循环结构。

* 顺序结构：程序按顺序执行，不发生跳转。
* 选择结构：依据是否满足条件，有选择的执行相应功能。
* 循环结构：依据条件是否满足，循环多次执行某段代码。

## 选择结构

### If 语句

支持一个初始化表达式, 初始化字句和条件表达式直接需要用**分号**分隔

if b := 3; b == 3 {  
 fmt.Println("b==3")  
 }

if ...else if

if a := 3; a > 3 {  
 fmt.Println("a>3")  
 } else if a < 3 {  
 fmt.Println("a<3")  
 } else if a == 3 {  
 fmt.Println("a==3")  
 } else {  
 fmt.Println("error")  
 }

### Switch 语句

Go里面switch默认相当于每个case最后带有break，匹配成功后不会自动向下执行其他case，而是跳出整个switch, 但是可以使用fallthrough强制执行后面的case代码

var score int = 90  
  
 switch score {  
 case 90:  
 fmt.Println("优秀")  
 //fallthrough  
 case 80:  
 fmt.Println("良好")  
 //fallthrough  
 case 50, 60, 70:  
 fmt.Println("一般")  
 //fallthrough  
 default:  
 fmt.Println("差")  
 }

注意： 此为 score 和 case 的值进行匹配

* 初始化条件语句

switch s1 := 90; s1 {   
 case 90:  
 fmt.Println("优秀")  
 case 80:  
 fmt.Println("良好")  
 default:  
 fmt.Println("一般")

* 同时 switch 语句可以演变为 else if 语句 细节如下

var s2 int = 90  
 switch { //这里没有写条件  
 case s2 >= 90: //这里写判断语句  
 fmt.Println("优秀")  
 case s2 >= 80:  
 fmt.Println("良好")  
 default:  
 fmt.Println("一般")  
 }

## 循环结构

### For 循环

var i, sum int  
  
 for i = 1; i <= 100; i++ {  
 sum += i  
 }  
 fmt.Println("sum = ", sum)

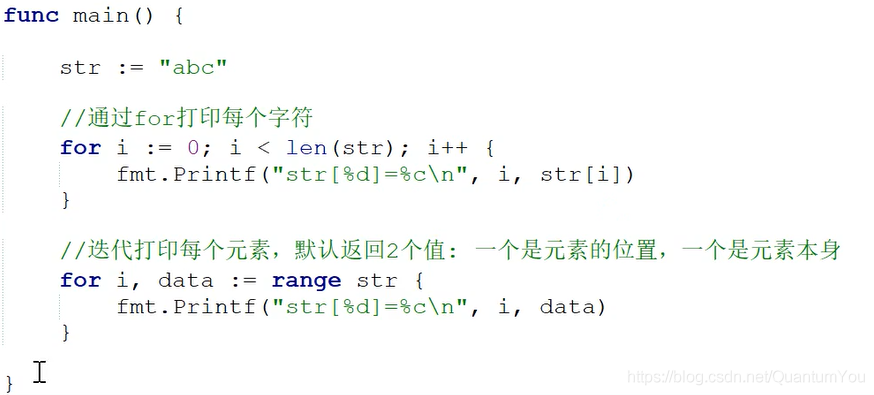
#### break 与 counite

跳出循环语句

break 操作是跳出当前循环  
 continue 跳过本次循环

### Range 迭代器

关键字 range 会返回两个值，第一个返回值是元素的数组下标，第二个返回值是元素的值：



s := "abc"  
 for i := range s { //支持 string/array/slice/map。  
 fmt.Printf("%c\n", s[i])  
 }

for \_, c := range s { // 忽略 index  
 fmt.Printf("%c\n", c)  
 }

# Go 函数

## 定义格式

func FuncName(/\*参数列表\*/) (o1 type1, o2 type2/\*返回类型\*/) {  
 //函数体  
  
 return v1, v2 //返回多个值  
}

**函数定义说明**：

* func：函数由关键字 func 开始声明
* FuncName：函数名称，根据约定，函数名首字母小写即为private，大写即为public，即分别为 公有，私有
* 参数列表：函数可以有0个或多个参数，参数格式为：变量名 类型，如果有多个参数通过逗号分隔，不支持默认参数
* 返回类型：  
  ① 上面返回值声明了两个变量名o1和o2(命名返回参数)，这个不是必须，可以只有类型没有变量名  
  ② 如果只有一个返回值且不声明返回值变量，那么你可以省略，包括返回值的括号  
  ③ 如果没有返回值，那么就直接省略最后的返回信息  
  ④ 如果有返回值， 那么必须在函数的内部添加return语句

## 无参无返回值类型

主函数 main（） 是其中的一种类型

func Test() { //无参无返回值函数定义  
 fmt.Println("this is a test func")  
}  
  
func main() {  
 Test() //无参无返回值函数调用  
}

注意 ： 函数的定义与调用

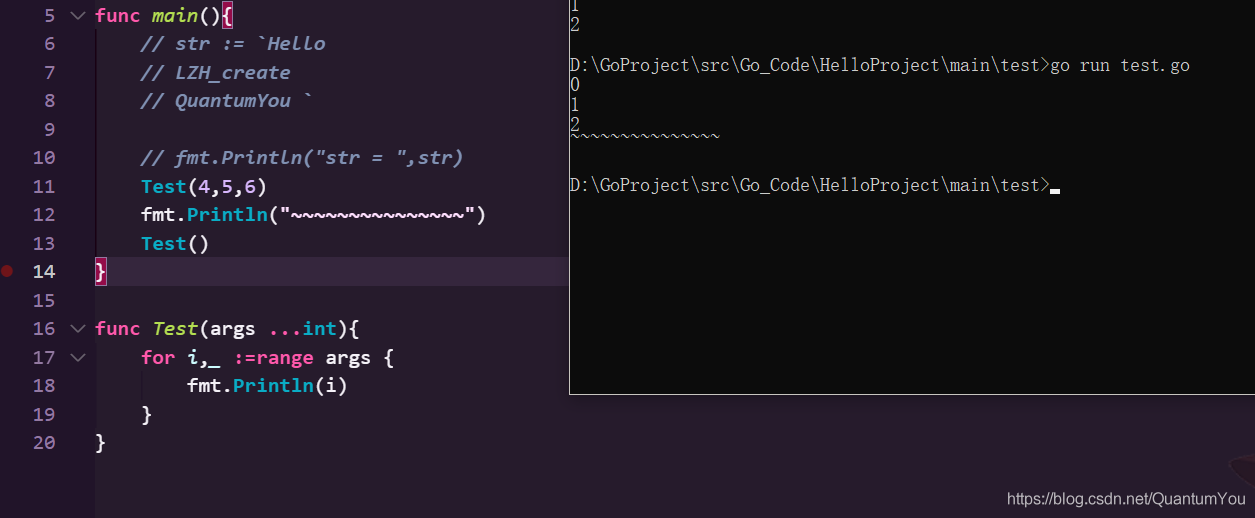
## 有参无返回值类型

func Test01(v1 int, v2 int) { //方式1  
 fmt.Printf("v1 = %d, v2 = %d\n", v1, v2)  
}  
  
func Test02(v1, v2 int) { //方式2, v1, v2都是int类型  
 fmt.Printf("v1 = %d, v2 = %d\n", v1, v2)  
}  
  
func main() {  
 Test01(10, 20) //函数调用  
 Test02(11, 22) //函数调用  
}

## 不定参数无返回值类型

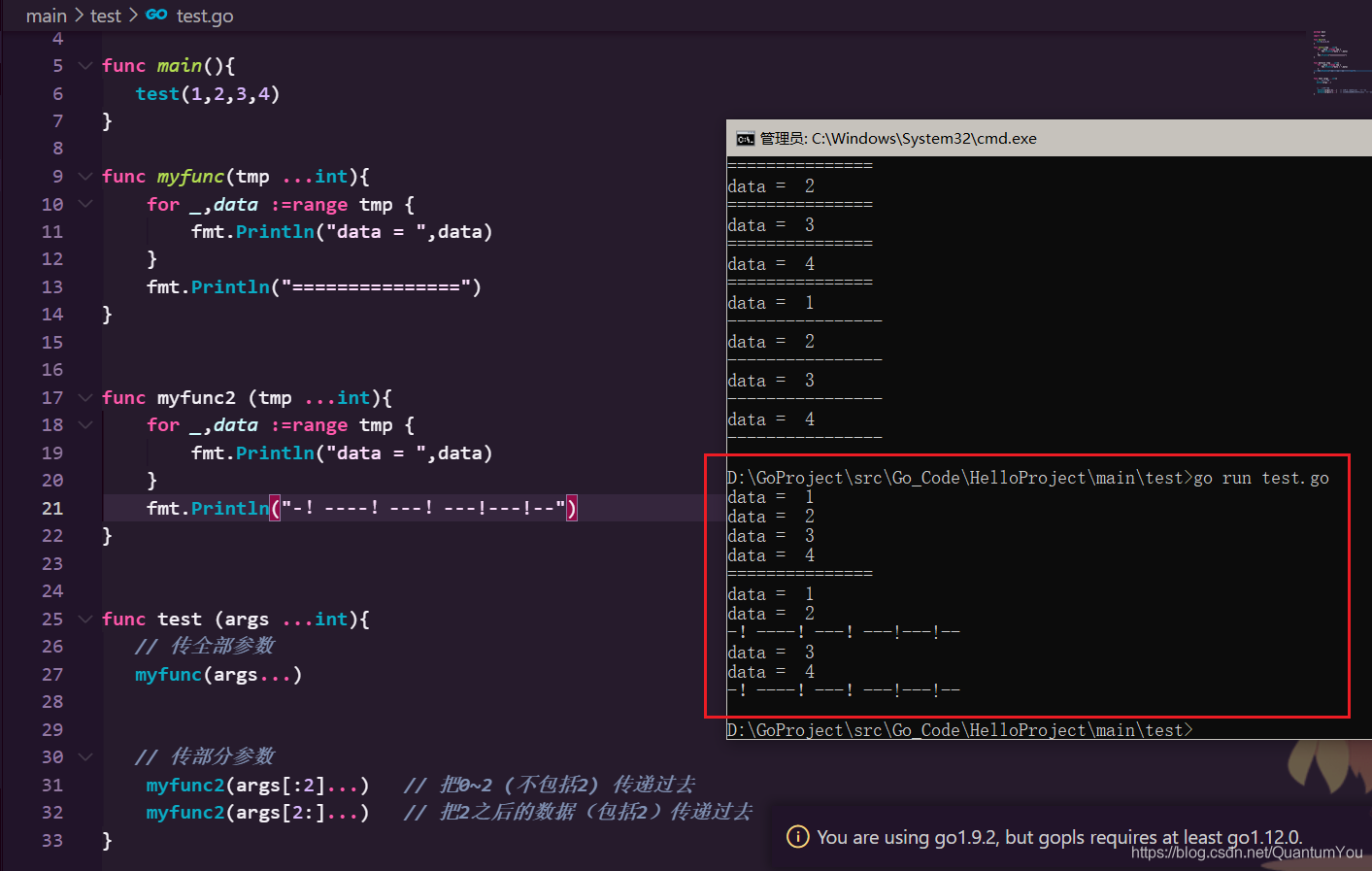
* 不定参数是指函数传入的参数个数为不定数量
* 形如...type格式的类型只能作为函数的参数类型存在，并且必须是最后一个参数.

func Test(args ...int) {  
 for \_, n := range args { //遍历参数列表  
 fmt.Println(n)  
 }  
}  
  
func main() {  
 //函数调用，可传0到多个参数  
 Test()  
 Test(1) // 0  
 Test(1, 2, 3, 4) // 0 1 2 3(换行)  
}



### 不定参数的传递

package main  
  
import "fmt"  
  
func main(){  
 test(1,2,3,4)   
}  
  
func myfunc(tmp ...int){  
 for \_,data :=range tmp {  
 fmt.Println("data = ",data)   
 }  
 fmt.Println("===============")  
}  
  
  
func myfunc2 (tmp ...int){  
 for \_,data :=range tmp {  
 fmt.Println("data = ",data)  
 }  
 fmt.Println("-！----！---！---!---!--")  
}  
  
  
func test (args ...int){  
 // 传全部参数  
 myfunc(args...)  
  
  
 // 传部分参数  
 myfunc2(args[:2]...) // 把0~2 (不包括2) 传递过去  
 myfunc2(args[2:]...) // 把2之后的数据（包括2）传递过去  
}

  
注意 ： 部分传递参数的两种用法

## 无参数有返回值类型

### 一个返回值

func Test01() int { //方式1  
 return 250  
}

官方建议：最好命名返回值，因为不命名返回值，虽然使得代码更加简洁了，但是会造成生成的文档可读性差

func Test02() (value int) { //方式2, 给返回值命名  
 value = 250  
 return value  
}  
  
func Test03() (value int) { //方式3, 给返回值命名，省区return 后值  
 value = 250  
 return  
}  
  
func main() {  
 v1 := Test01() //函数调用  
 v2 := Test02() //函数调用  
 v3 := Test03() //函数调用  
 fmt.Printf("v1 = %d, v2 = %d, v3 = %d\n", v1, v2, v3)  
}

### 两个返回值

func Test01() (int, string) { //方式1  
 return 250, "sb"  
}  
  
func Test02() (a int, str string) { //方式2, 给返回值命名  
 a = 250  
 str = "sb"  
 return  
}  
  
func main() {  
 v1, v2 := Test01() //函数调用  
 \_, v3 := Test02() //函数调用， 第一个返回值丢弃  
 v4, \_ := Test02() //函数调用， 第二个返回值丢弃  
 fmt.Printf("v1 = %d, v2 = %s, v3 = %s, v4 = %d\n", v1, v2, v3, v4)  
}

## 有参数有返回值

案例： 求两个数的最大值与最小值

func MinAndMax(num1 int, num2 int) (min int, max int) {  
 if num1 > num2 { //如果num1 大于 num2  
 min = num2  
 max = num1  
 } else {  
 max = num2  
 min = num1  
 }  
  
 return  
}  
  
func main() {  
 min, max := MinAndMax(33, 22)  
 fmt.Printf("min = %d, max = %d\n", min, max) //min = 22, max = 33

## 递归函数

递归指函数可以直接或间接的调用自身。

递归函数通常有相同的结构：一个**跳出条件**和一个**递归体**。所谓跳出条件就是根据传入的参数判断是否需要停止递归，而递归体则是函数自身所做的一些处理。

## 函数类型与回调函数

* Go 语言 中的函数类型就相当于 C 语言中的函数指针(指向函数的指针)
* 在Go语言中，函数也是一种**数据类型**，我们可以通过type来定义它，它的类型就是所有拥有相同的参数，相同的返回值的一种类型。
* 函数有一个参数是函数类型，这个函数就是**回调函数**

//声明一个函数类型, func后面没有函数名以及方法体  
type FuncType func(int, int) int   
  
//函数中有一个参数类型为函数类型：f FuncType  
func Calc(a, b int, f FuncType) (result int) {  
 result = f(a, b) //通过调用f()实现任务  
 return  
}  
  
func Add(a, b int) int {  
 return a + b  
}  
  
func Minus(a, b int) int {  
 return a - b  
}  
  
func main() {  
 //函数调用，第三个参数为函数名字，此函数的参数，返回值必须和FuncType类型一致  
 result := Calc(1, 1, Add) // 这里可以修改  
 fmt.Println(result) //2  
  
 var f FuncType = Minus  
 fmt.Println("result = ", f(10, 2)) //result = 8  
}

注意 ： 此案例类似于多态的实现

## 匿名函数与闭包

* 在Go语言里，所有的**匿名函数**(Go语言规范中称之为函数字面量)都是闭包。匿名函数是指不需要定义函数名的一种函数实现方式

### 五种形式

func main() {  
 i := 0  
 str := "mike"  
  
 //方式1  
 f1 := func() { //匿名函数，无参无返回值  
 //引用到函数外的变量  
 fmt.Printf("方式1：i = %d, str = %s\n", i, str)  
 }  
  
 f1() //函数调用  
  
 //方式1的另一种方式  
 type FuncType func() //声明函数类型, 无参无返回值  
 var f2 FuncType = f1  
 f2() //函数调用  
  
 //方式2  
 var f3 FuncType = func() {  
 fmt.Printf("方式2：i = %d, str = %s\n", i, str)  
 }  
 f3() //函数调用  
  
 //方式3  
 func() { //匿名函数，无参无返回值  
 fmt.Printf("方式3：i = %d, str = %s\n", i, str)  
 }() //别忘了后面的(), ()的作用是，此处直接调用此匿名函数  
  
 //方式4, 匿名函数，有参有返回值  
 v := func(a, b int) (result int) {  
 result = a + b  
 return  
 }(1, 1) //别忘了后面的(1, 1), (1, 1)的作用是，此处直接调用此匿名函数， 并传参  
 fmt.Println("v = ", v)  
  
}

### 闭包捕获外部变量的特点

#### 特点一

func main() {  
 i := 10  
 str := "mike"  
   
 func() {  
 i = 100  
 str = "go"  
 //内部：i = 100, str = go  
 fmt.Printf("内部：i = %d, str = %s\n", i, str)  
 }() //别忘了后面的(), ()的作用是，此处直接调用此匿名函数  
  
 //外部：i = 100, str = go  
 fmt.Printf("外部：i = %d, str = %s\n", i, str)  
}

输出发现 ：外部与内部的值相同 ，推出闭包以引用的方式捕获外部变量。

#### 特点二

闭包不关心这些捕获了的变量和常量**是否已经超出了作用域**，所以只有闭包还在使用它，这些变量就**还会存在**。

func squares() func() int {  
 var x int  
 return func() int {//匿名函数 闭包的体现  
 x++ //捕获外部变量  
 return x \* x  
 }  
}  
  
func main() {  
 f := squares()  
 fmt.Println(f()) // "1"  
 fmt.Println(f()) // "4"  
 fmt.Println(f()) // "9"  
 fmt.Println(f()) // "16"  
}

解释 ：

* squares返回一个匿名函数，func() int， 该匿名函数每次被调用时都会返回下一个数的平方。
* 变量的生命周期不由它的作用域决定：squares返回后，变量x仍然隐式的存在于f中。
* 对squares的一次调用会生成一个局部变量x并返回一个匿名函数

## defer 关键字

* 关键字 defer ⽤于延迟一个函数或者方法（或者当前所创建的匿名函数）的执行。注意，defer语句只能出现在函数或方法的内部
* defer语句经常被用于处理成对的操作，如打开、关闭、连接、断开连接、加锁、释放锁。通过defer机制，不论函数逻辑多复杂，都能保证在任何执行路径下，资源被释放。释放资源的defer应该直接跟在请求资源的语句后。
* 有点类似于 Java 处理异常机制中的 finally()

### defer 多个执行顺序

* 如果一个函数中有多个defer语句，它们会以LIFO（后进先出）的顺序执行。哪怕函数或某个延迟调用发生错误，这些调用依旧会被执⾏。

#### 模拟异常

func test(x int) {  
 fmt.Println(100 / x)//x为0时，产生异常  
}  
  
func main() {  
 defer fmt.Println("aaaaaaaa")  
 defer fmt.Println("bbbbbbbb")  
  
 defer test(0)  
  
 defer fmt.Println("cccccccc")  
}

输出结果

运行结果：  
 cccccccc  
 bbbbbbbb  
 aaaaaaaa  
 panic: runtime error: integer divide by zero

#### defer 与匿名函数

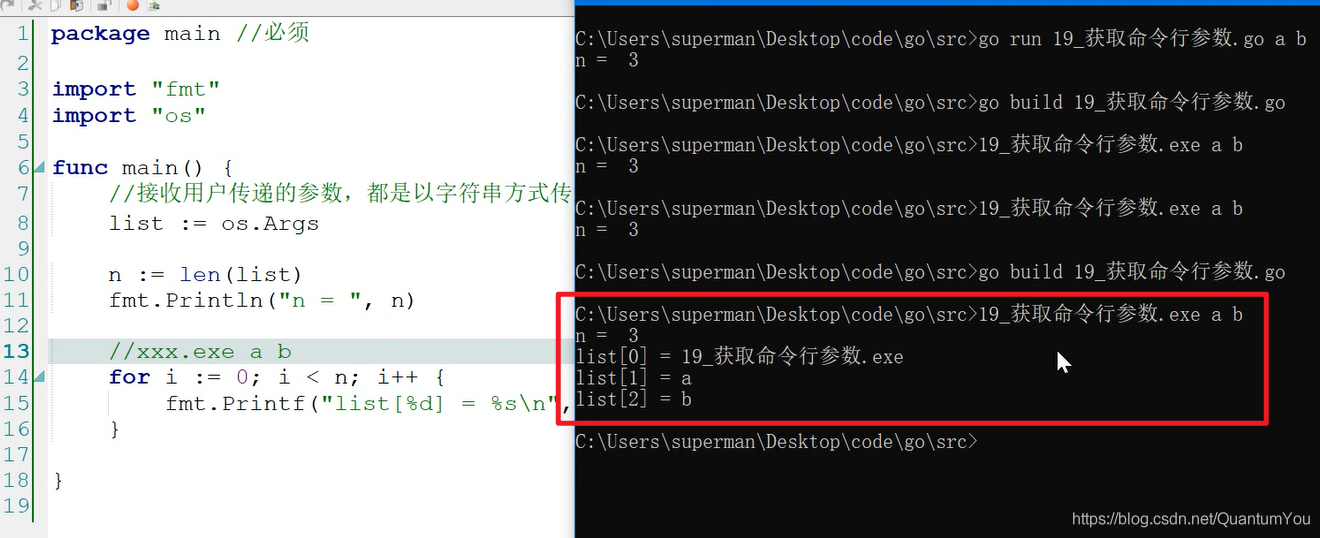
**情况一**

func main01() {  
 a := 10  
 b := 20  
  
 defer func() {  
 fmt.Printf("a = %d, b = %d\n", a, b)  
 }() //()代表调用此匿名函数  
  
 a = 111  
 b = 222  
 fmt.Printf("外部：a = %d, b = %d\n", a, b)  
}

**情况二**

func main() {  
 a := 10  
 b := 20  
  
 defer func(a, b int) {  
 fmt.Printf("a = %d, b = %d\n", a, b)  
 }(a, b)   
//()代表调用此匿名函数, 把参数传递过去，已经先传递参数，只是没有调用  
  
  
 a = 111  
 b = 222  
 fmt.Printf("外部：a = %d, b = %d\n", a, b)  
}

## 获取命令行参数



代码

func main() {  
 //接收用户传递的参数，都是以字符串方式传递  
 list := os.Args  
  
 n := len(list)  
 fmt.Println("n = ", n)  
  
 //xxx.exe a b  
 for i := 0; i < n; i++ {  
 fmt.Printf("list[%d] = %s\n", i, list[i])  
 }  
  
 for i, data := range list {  
 fmt.Printf("list[%d] = %s\n", i, data)  
 }  
}

## 作用域

定义 ： 为已声明标识符所表示的常量、类型、变量、函数或包在源代码中的作用范围。

### 局部变量

* 在函数体内声明的变量、参数和返回值变量就是局部变量，它们的作用域只在函数体内：

func test(a, b int) {  
 var c int  
 a, b, c = 1, 2, 3  
 fmt.Printf("a = %d, b = %d, c = %d\n", a, b, c)  
}  
  
func main() {  
 //a, b, c = 1, 2, 3 //err, a, b, c不属于此作用域  
 {  
 var i int  
 i = 10  
 fmt.Printf("i = %d\n", i)  
 }  
  
 //i = 20 //err, i不属于此作用域  
  
 if a := 3; a == 3 {  
 fmt.Println("a = ", a)  
 }  
 //a = 4 //err，a只能if内部使用  
}

### 全局变量

* 在函数体外声明的变量称之为全局变量，全局变量可以在整个包甚至外部包（被导出后）使用。

var a int //全局变量的声明  
  
func test() {  
 fmt.Printf("test a = %d\n", a)  
}  
  
func main() {  
 a = 10  
 fmt.Printf("main a = %d\n", a) //main a = 10  
  
 test() //test a = 10  
}

### 不同作用域同名变量

* 在不同作用域可以声明同名的变量，其访问原则为：在同一个作用域内，**就近原则**访问最近的变量，如果此作用域没有此变量声明，则访问全局变量，如果全局变量也没有，则报错。