

# 定义格式

函数构成代码执行的逻辑结构。在Go语言中，函数的基本组成为：关键字func、函数名、参数列表、返回值、函数体和返回语句。

Go 语言函数定义格式如下：

```
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

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

## 自定义函数

### 无参无返回值

```
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
}
```

### 有参无返回值

#### 普通参数列表

```
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  
}
```

## 不定参数列表

### 不定参数类型

不定参数是指函数传入的参数个数为不定数量。为了做到这点，首先需要将函数定义为接受不定参数类型：

```
//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)  
    Test(1, 2, 3, 4)  
}
```

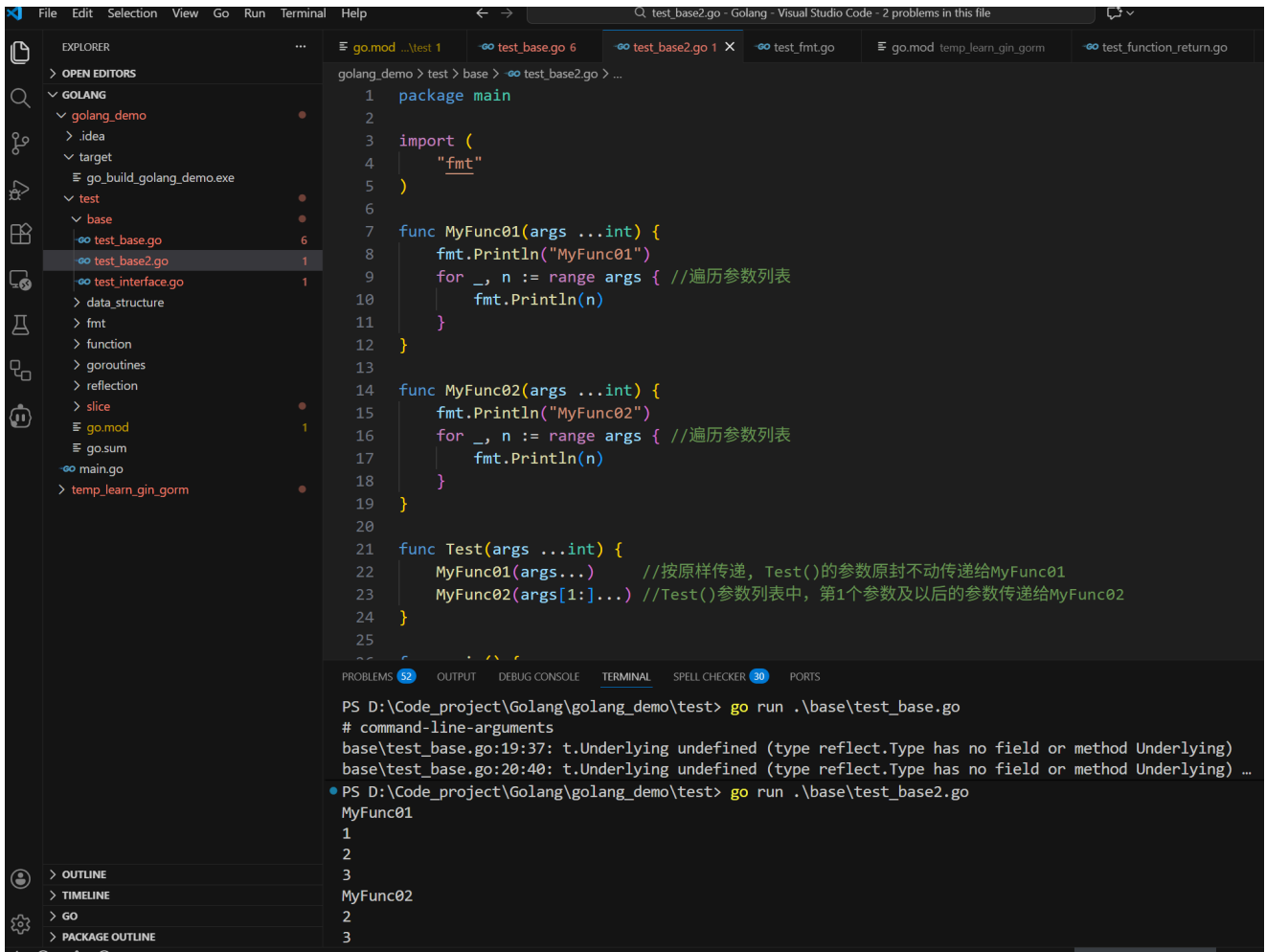
### 不定参数的传递

```
func MyFunc01(args ...int) {  
    fmt.Println("MyFunc01")  
    for _, n := range args { //遍历参数列表  
        fmt.Println(n)  
    }  
}
```

```
func MyFunc02(args ...int) {  
    fmt.Println("MyFunc02")  
    for _, n := range args { //遍历参数列表  
        fmt.Println(n)  
    }  
}
```

```
func Test(args ...int) {  
    MyFunc01(args...) //按原样传递, Test()的参数  
    //原封不动传递给MyFunc01  
    MyFunc02(args[1:]...) //Test()参数列表中, 第1个参  
    //数及以后的参数传递给MyFunc02  
}
```

```
func main() {  
    Test(1, 2, 3) //函数调用  
}
```



## 无参有返回值

有返回值的函数，必须有明确的终止语句，否则会引发编译错误。

## 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
    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)
}

```

## 多个返回值

```

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)
}

```

## 有参有返回值

```

//求2个数的最小值和最大值
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。

```

//通过循环实现1+2+3.....+100
func Test01() int {
    i := 1
    sum := 0
    for i = 1; i <= 100; i++ {
        sum += i
    }

    return sum
}

```

```

//通过递归实现1+2+3.....+100
func Test02(num int) int {
    if num == 1 {
        return 1
    }
}

```

```

    return num + Test02(num-1) //函数调用本身
}

//通过递归实现1+2+3.....+100
func Test03(num int) int {
    if num == 100 {
        return 100
    }

    return num + Test03(num+1) //函数调用本身
}

func main() {

    fmt.Println(Test01())      //5050
    fmt.Println(Test02(100))  //5050
    fmt.Println(Test03(1))    //5050
}

```

## 函数类型

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。

```

type FuncType func(int, int) int //声明一个函数类型，
func后面没有函数名

```

```
//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)
    fmt.Println(result) //2

    var f FuncType = Minus
    fmt.Println("result = ", f(10, 2)) //result =
8
}
```

## 匿名函数与闭包

所谓闭包就是一个函数“捕获”了和它在同一作用域的有关常量和变量。这就意味着当闭包被调用的时候，不管在程序什么地方调用，闭包能够使用这些常量或者变量。它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。

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，它并不是一个新概念，最早可以回溯到1958年的Lisp语言。

```
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)
}
```

闭包捕获外部变量特点：

```
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)
}
```

函数返回值为匿名函数:

```
// 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。
func squares() func() int {
    var x int
    return func() int { //匿名函数
        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"  
}
```

函数squares返回另一个类型为 func() int 的函数。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。每次调用时匿名函数时，该函数都会先使x的值加1，再返回x的平方。第二次调用squares时，会生成第二个x变量，并返回一个新的匿名函数。新匿名函数操作的是第二个x变量。

通过这个例子，我们看到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。

```

7 // squares返回一个匿名函数, func() int
8 // 该匿名函数每次被调用时都会返回下一个数的平方。
9 func squares() func() int {
10     var x int
11     return func() int { //匿名函数
12         x++ //捕获外部变量
13         return x * x
14     }
15 }
16
17 func main() {
18     // f := squares()
19     // fmt.Println(f()) // "1"
20     // fmt.Println(f()) // "4"
21     // fmt.Println(f()) // "9"
22     // fmt.Println(f()) // "16"
23     fmt.Println(squares()) // "1"
24     fmt.Println(squares()) // "1"
25     fmt.Println(squares()) // "1"
26     fmt.Println(squares()) // "1"
27 }
28

```

```

16
• 0x7ff6e5adefef0
PS D:\Code_project\Golang\golang_demo\test> go run .\base\test_closure.go\
main module (test) does not contain package test/base/test_closure.go
PS D:\Code_project\Golang\golang_demo\test> go run .\base\test_closure.go\
• main module (test) does not contain package test/base/test_closure.go
• PS D:\Code_project\Golang\golang_demo\test> go run .\base\test_closure.go
• 1
1
1
1
PS D:\Code_project\Golang\golang_demo\test>

```

## 延迟调用defer

### defer的作用

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。

```

func main() {
    fmt.Println("this is a test")
    defer fmt.Println("this is a defer") //main结束
前调用

```

```

    /*
        运行结果:
        this is a test
        this is a defer
    */
}

```

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。

## 多个defer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。

```

func test(x int) {
    fmt.Println(100 / x)//x为0时，产生异常
}

func main() {
    defer fmt.Println("aaaaaaaaa")
    defer fmt.Println("bbbbbbbbbb")

    defer test(0)
}

```

```
defer fmt.Println("cccccccc")
/*
运行结果:
cccccccc
bbbbbbbb
aaaaaaaa
panic: runtime error: integer divide by zero
*/
}
```

## defer和匿名函数结合使用

```
func main() {
    a, b := 10, 20
    defer func(x int) { // a以值传递方式传给x
        fmt.Println("defer:", x, b) // b 闭包引用
    }(a)

    a += 10
    b += 100

    fmt.Printf("a = %d, b = %d\n", a, b)

    /*
    运行结果:
    a = 20, b = 120
    defer: 10 120
    */
}
```

```
*/  
}
```

## 获取命令行参数

```
package main  
  
import (  
    "fmt"  
    "os"    //os.Args所需的包  
)  
  
func main() {  
    args := os.Args //获取用户输入的所有参数  
  
    //如果用户没有输入,或参数个数不够,则调用该函数提示用户  
    if args == nil || len(args) < 2 {  
        fmt.Println("err: xxx ip port")  
        return  
    }  
    ip := args[1]    //获取输入的第一个参数  
    port := args[2]  //获取输入的第二个参数  
    fmt.Printf("ip = %s, port = %s\n", ip, port)  
}
```

运行结果如下：

```
C:\Users\superman\Desktop\code\go\src>dir
驱动器 C 中的卷没有标签。
卷的序列号是 4040-7EEB

C:\Users\superman\Desktop\code\go\src 的目录
2017/12/29  22:51    <DIR>          .
2017/12/29  22:51    <DIR>          ..
2017/12/29  22:51                448 main.go
2017/12/29  22:51           1,950,720 src.exe
2017/12/17  17:03    <DIR>          test
                2 个文件      1,951,168 字节
                3 个目录 78,069,923,840 可用字节

C:\Users\superman\Desktop\code\go\src>src 127.0.0.1 8888
ip = 127.0.0.1, port = 8888

C:\Users\superman\Desktop\code\go\src>
```

## 作用域

作用域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。

```

var a int //全局变量的声明

func test01(a float32) {
    fmt.Printf("a type = %T\n", a) //a type =
float32
}

func main() {
    fmt.Printf("a type = %T\n", a) //a type = int,
说明使用全局变量的a

    var a uint8 //局部变量声明

    {
        var a float64 //局部变量声
明

        fmt.Printf("a type = %T\n", a) //a type =

```

```
float64
```

```
}
```

```
    fmt.Printf("a type = %T\n", a) //a type =
uint8
```

```
test01(3.14)
```

```
test02()
```

```
}
```

```
func test02() {
```

```
    fmt.Printf("a type = %T\n", a) //a type = int
}
```

The screenshot shows the Visual Studio Code interface with a Go project named 'golang\_demo'. The Explorer panel on the left shows the project structure, including files like 'test\_base.go', 'test\_base2.go', 'test\_closure.go', 'test\_interface.go', 'test\_variable.go', 'data\_structure', 'fmt', 'function', 'test\_function\_named.go', 'test\_function\_return.go', 'goroutines', 'reflection', 'slice', 'go.mod', 'go.sum', 'main.go', and 'temp\_learn\_gin\_gorm'. The main editor displays the 'test\_variable.go' file, which contains the following code:

```
6
7 var a int //全局变量的声明
8
9 func test01(a float32) {
10     fmt.Printf("a type = %T\n", a) //a type = float32
11 }
12
13 func main() {
14     fmt.Printf("a type = %T\n", a) //a type = int, 说明使用全局变量的a
15
16     var a uint8 //局部变量声明
17
18     {
19         var a float64 //局部变量声明
20         fmt.Printf("a type = %T\n", a) //a type = float64
21     }
22
23     fmt.Printf("a type = %T\n", a) //a type = uint8
24
25     test01(3.14)
26     test02()
27 }
28
29 func test02() {
30     fmt.Printf("a type = %T\n", a) //a type = int
```

The terminal window at the bottom shows the following output:

```
PS D:\Code_project\Golang\golang_demo\test> go run .\slice\test_slice3.go
[100 2 3 0 0]
PS D:\Code_project\Golang\golang_demo\test> go run .\slice\test_slice3.go
[100 2 3 0 200]
PS D:\Code_project\Golang\golang_demo\test> go run .\base\test_variable.go
# command-line-arguments
base\test_variable.go:16:6: a redeclared in this block
    base\test_variable.go:15:6: other declaration of a
PS D:\Code_project\Golang\golang_demo\test> go run .\base\test_variable.go
# command-line-arguments
base\test_variable.go:25:9: cannot use a (variable of type uint8) as float32 value in argument to test01
PS D:\Code_project\Golang\golang_demo\test>
```

