Golang 函数详解

主讲教师：（大地）
合作网站：www.itying.com （IT 营）
我的专栏：https://www.itying.com/category-79-b0.html

1、函数定义.....1
2、函数的调用.....2
3、函数参数.....2
4、函数返回值.....4
5、函数变量作用域.....4
6、函数类型与变量.....6
7、高阶函数.....7
8、匿名函数和闭包.....8
9、defer 语句.....11
10、内置函数 panic/recover.....14

1、函数定义

函数是组织好的、可重复使用的、用于执行指定任务的代码块。本文介绍了 Go 语言中函数的相关内容。

Go 语言中支持：函数、匿名函数和闭包

Go 语言中定义函数使用 func 关键字，具体格式如下：

```
func 函数名(参数)(返回值){
    函数体
}
```

其中：

- **函数名**：由字母、数字、下划线组成。但函数名的第一个字母不能是数字。在同一个包内，函数名也称不能重名（包的概念详见后文）。
- **参数**：参数由参数变量和参数变量的类型组成，多个参数之间使用,分隔。

- **返回值**：返回值由返回值变量和其变量类型组成，也可以只写返回值的类型，多个返回值必须用()包裹，并用,分隔。
- **函数体**：实现指定功能的代码块。

我们先来定义一个求两个数之和的函数：

```
func intSum(x int, y int) int {
    return x + y
}
```

函数的参数和返回值都是可选的，例如我们可以实现一个既不需要参数也没有返回值的函数：

```
func sayHello() {
    fmt.Println("Hello it 营")
}
```

2、函数的调用

定义了函数之后，我们可以通过函数名()的方式调用函数。 例如我们调用上面定义的两个函数，代码如下：

```
func main() {
    sayHello()
    ret := intSum(10, 20)
    fmt.Println(ret)
}
```

注意，调用有返回值的函数时，可以不接收其返回值。

3、函数参数

类型简写

函数的参数中如果相邻变量的类型相同，则可以省略类型，例如：

```
func intSum(x, y int) int {
    return x + y
}
```

上面的代码中，intSum 函数有两个参数，这两个参数的类型均为 int，因此可以省略 x 的类型，因为 y 后面有类型说明，x 参数也是该类型。

可变参数

可变参数是指函数的参数数量不固定。Go 语言中的可变参数通过在参数名后加...来标识。

注意：可变参数通常要作为函数的最后一个参数。

举个例子：

```
func intSum2(x ...int) int {
    fmt.Println(x) //x 是一个切片
    sum := 0
    for _, v := range x {
        sum = sum + v
    }
    return sum
}
```

调用上面的函数：

```
ret1 := intSum2()
ret2 := intSum2(10)
ret3 := intSum2(10, 20)
ret4 := intSum2(10, 20, 30)
fmt.Println(ret1, ret2, ret3, ret4) //0 10 30 60
```

固定参数搭配可变参数使用时，可变参数要放在固定参数的后面，示例代码如下：

```
func intSum3(x int, y ...int) int {
    fmt.Println(x, y)
    sum := x
    for _, v := range y {
        sum = sum + v
    }
    return sum
}
```

调用上述函数：

```
ret5 := intSum3(100)
ret6 := intSum3(100, 10)
ret7 := intSum3(100, 10, 20)
ret8 := intSum3(100, 10, 20, 30)
fmt.Println(ret5, ret6, ret7, ret8) //100 110 130 160
```

本质上，函数的可变参数是通过切片来实现的。

4、函数返回值

Go 语言中通过 return 关键字向外输出返回值。

函数多返回值

Go 语言中函数支持多返回值，函数如果有多个返回值时必须用()将所有返回值包裹起来。

举个例子：

```
func calc(x, y int) (int, int) {
    sum := x + y
    sub := x - y
    return sum, sub
}
```

返回值命名

函数定义时可以给返回值命名，并在函数体中直接使用这些变量，最后通过 return 关键字返回。

例如：

```
func calc(x, y int) (sum, sub int) {
    sum = x + y
    sub = x - y
    return
}
```

5、函数变量作用域

全局变量

全局变量是定义在函数外部的变量，它在程序整个运行周期内都有效。 在函数中可以访问到全局变量。

```
package main
import "fmt"
//定义全局变量 num
var num int64 = 10
func testGlobalVar() {
    fmt.Printf("num=%d\n", num) //函数中可以访问全局变量 num
}
```

```
func main() {
    testGlobalVar() //num=10
}
```

局部变量

局部变量是函数内部定义的变量， 函数内定义的变量无法在该函数外使用

1、函数内定义的变量无法在该函数外使用

例如下面的示例代码 main 函数中无法使用 testLocalVar 函数中定义的变量 x：

```
func testLocalVar() {
    //定义一个函数局部变量x,仅在该函数内生效
    var x int64 = 100
    fmt.Printf("x=%d\n", x)
}

func main() {
    testLocalVar()
    fmt.Println(x) // 此时无法使用变量x
}
```

2、如果局部变量和全局变量重名，优先访问局部变量

```
package main
import "fmt"
//定义全局变量num
var num int64 = 10
func testNum() {
    num := 100
    fmt.Printf("num=%d\n", num) // 函数中优先使用局部变量
}
func main() {
    testNum() // num=100
}
```

接下来我们来看一下语句块定义的变量，通常我们会在 if 条件判断、for 循环、switch 语句上使用这种定义变量的方式。

```
func testLocalVar2(x, y int) {
    fmt.Println(x, y) //函数的参数也是只在本函数中生效
    if x > 0 {
        z := 100 //变量z只在if语句块生效
```

```
        fmt.Println(z)
    }
    //fmt.Println(z)//此处无法使用变量 z
}
```

还有我们之前讲过的 for 循环语句中定义的变量，也是只在 for 语句块中生效：

```
func testLocalVar3() {
    for i := 0; i < 10; i++ {
        fmt.Println(i) //变量 i 只在当前 for 语句块中生效
    }
    //fmt.Println(i) //此处无法使用变量 i
}
```

6、函数类型与变量

定义函数类型

我们可以使用 type 关键字来定义一个函数类型，具体格式如下：

```
type calculation func(int, int) int
```

上面语句定义了一个 calculation 类型，它是一种函数类型，这种函数接收两个 int 类型的参数并且返回一个 int 类型的返回值。

简单来说，凡是满足这个条件的函数都是 calculation 类型的函数，例如下面的 add 和 sub 是 calculation 类型。

```
func add(x, y int) int {
    return x + y
}

func sub(x, y int) int {
    return x - y
}
```

add 和 sub 都能赋值给 calculation 类型的变量。

```
var c calculation
c = add
```

函数类型变量

我们可以声明函数类型的变量并且为该变量赋值：

```
func main() {
    var c calculation                  // 声明一个calculation类型的变量c
    c = add                            // 把add赋值给c
    fmt.Printf("type of c:%T\n", c) // type of c:main.calculation
    fmt.Println(c(1, 2))               // 像调用add一样调用c

    f := add                           // 将函数add赋值给变量f1
    fmt.Printf("type of f:%T\n", f) // type of f:func(int, int) int
    fmt.Println(f(10, 20))             // 像调用add一样调用f
}
```

7、高阶函数

高阶函数分为函数作为参数和函数作为返回值两部分。

函数作为参数

函数可以作为参数：

```
func add(x, y int) int {
    return x + y
}
func calc(x, y int, op func(int, int) int) int {
    return op(x, y)
}
func main() {
    ret2 := calc(10, 20, add)
    fmt.Println(ret2) //30
}
```

函数作为返回值

函数也可以作为返回值：

```
package main
```

```
import (
    "fmt"
)
func add(x, y int) int {
    return x + y
}
func sub(x, y int) int {
    return x - y
}
func do(s string) func(int, int) int {
    switch s {
    case "+":
        return add
    case "-":
        return sub
    default:
        return nil
    }
}
func main() {
    var a = do("+")
    fmt.Println(a(10, 20))
}
```

8、匿名函数和闭包

匿名函数

函数当然还可以作为返回值，但是在 Go 语言中函数内部不能再像之前那样定义函数了，只能定义匿名函数。匿名函数就是没有函数名的函数，匿名函数的定义格式如下：

```
func(参数)(返回值){
    函数体
}
```

匿名函数因为没有函数名，所以没办法像普通函数那样调用，所以匿名函数需要保存到某个变量或者作为立即执行函数:

```
func main() {
    // 将匿名函数保存到变量
    add := func(x, y int) {
```

```
        fmt.Println(x + y)
    }
    add(10, 20) // 通过变量调用匿名函数

    //自执行函数：匿名函数定义完加()直接执行
    func(x, y int) {
        fmt.Println(x + y)
    }(10, 20)
}
```

匿名函数多用于实现回调函数和闭包。

闭包

闭包可以理解成“定义在一个函数内部的函数“。在本质上，闭包是将函数内部和函数外部连接起来的桥梁。或者说是函数和其引用环境的组合体。 首先我们来看一个例子：

```
func adder() func(int) int {
    var x int
    return func(y int) int {
        x += y
        return x
    }
}
func main() {
    var f = adder()
    fmt.Println(f(10)) //10
    fmt.Println(f(20)) //30
    fmt.Println(f(30)) //60

    f1 := adder()
    fmt.Println(f1(40)) //40
    fmt.Println(f1(50)) //90
}
```

变量 f 是一个函数并且它引用了其外部作用域中的 x 变量，此时 f 就是一个闭包。 在 f 的生命周期内，变量 x 也一直有效。

闭包进阶示例 1：

```
func adder2(x int) func(int) int {
    return func(y int) int {
        x += y
        return x
    }
}
func main() {
    var f = adder2(10)
    fmt.Println(f(10)) //20
    fmt.Println(f(20)) //40
    fmt.Println(f(30)) //70

    f1 := adder2(20)
    fmt.Println(f1(40)) //60
    fmt.Println(f1(50)) //110
}
```

闭包进阶示例 2：

```
func makeSuffixFunc(suffix string) func(string) string {
    return func(name string) string {
        if !strings.HasSuffix(name, suffix) {
            return name + suffix
        }
        return name
    }
}

func main() {
    jpgFunc := makeSuffixFunc(".jpg")
    txtFunc := makeSuffixFunc(".txt")
    fmt.Println(jpgFunc("test")) //test.jpg
    fmt.Println(txtFunc("test")) //test.txt
}
```

闭包进阶示例 3：

```
func calc(base int) (func(int) int, func(int) int) {
    add := func(i int) int {
        base += i
        return base
    }

    sub := func(i int) int {
        base -= i
        return base
    }
    return add, sub
}

func main() {
    f1, f2 := calc(10)
    fmt.Println(f1(1), f2(2)) //11 9
    fmt.Println(f1(3), f2(4)) //12 8
    fmt.Println(f1(5), f2(6)) //13 7
}
```

闭包其实并不复杂，只要牢记闭包=函数+引用环境。

9、defer 语句

Go 语言中的 defer 语句会将其后面跟随的语句进行延迟处理。在 defer 归属的函数即将返回时，将延迟处理的语句按 defer 定义的逆序进行执行，也就是说，先被 defer 的语句最后被执行，最后被 defer 的语句，最先被执行。

举个例子：

```
func main() {
    fmt.Println("start")
    defer fmt.Println(1)
    defer fmt.Println(2)
    defer fmt.Println(3)
    fmt.Println("end")
}
```

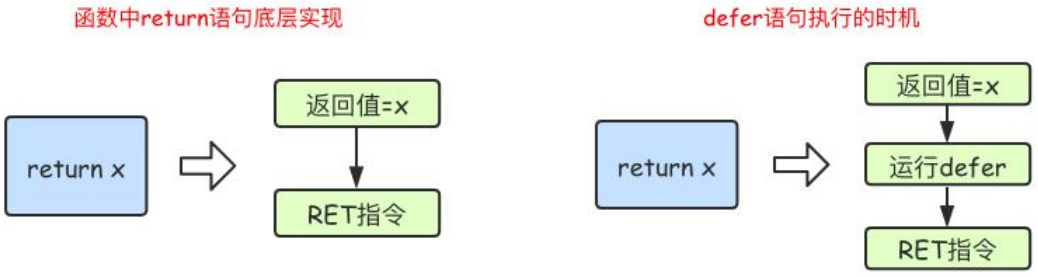
输出结果：

```
start
end
3
2
1
```

由于 defer 语句延迟调用的特性，所以 defer 语句能非常方便的处理资源释放问题。比如：资源清理、文件关闭、解锁及记录时间等。

defer 执行时机

在 Go 语言的函数中 return 语句在底层并不是原子操作，它分为给返回值赋值和 RET 指令两步。而 defer 语句执行的时机就在返回值赋值操作后，RET 指令执行前。具体如下图所示：



defer 经典案例 1

阅读下面的代码，写出最后的打印结果。

```
func f1() int {
    x := 5
    defer func() {
        x++
    }()
    return x
}

func f2() (x int) {
    defer func() {
        x++
    }()
    return 5
}

func f3() (y int) {
    x := 5
    defer func() {
        x++
    }()
    return x
}
func f4() (x int) {
    defer func(x int) {
        x++
    }(x)
    return 5
}

func main() {
    fmt.Println(f1())
    fmt.Println(f2())
    fmt.Println(f3())
    fmt.Println(f4())
}
```

defer 经典案例 2

```
func calc(index string, a, b int) int {
    ret := a + b
    fmt.Println(index, a, b, ret)
    return ret
}

func main() {
    x := 1
    y := 2
    defer calc("AA", x, calc("A", x, y))
    x = 10
    defer calc("BB", x, calc("B", x, y))
    y = 20
}
```

问，上面代码的输出结果是？（提示：defer 注册要延迟执行的函数时该函数所有的参数都需要确定其值）

10、内置函数 panic/recover

内置函数	介绍
close	主要用来关闭 channel
len	用来求长度，比如 string、array、slice、map、channel
new	用来分配内存，主要用来分配值类型，比如 int、struct。返回的是指针
make	用来分配内存，主要用来分配引用类型，比如 chan、map、slice
append	用来追加元素到数组、slice 中
panic 和 recover	用来做错误处理

Go 语言中目前（Go1.12）是没有异常机制，但是使用 panic/recover 模式来处理错误。 panic 可以在任何地方引发，但 recover 只有在 defer 调用的函数中有效。 首先来看一个例子：

1、panic/recover 的基本使用

```
func funcA() {
    fmt.Println("func A")
}

func funcB() {
    panic("panic in B")
}

func funcC() {
    fmt.Println("func C")
}
func main() {
    funcA()
    funcB()
    funcC()
}
```

输出：

```
func A
panic: panic in B

goroutine 1 [running]:
main.funcB(...)
        .../code/func/main.go:12
main.main()
        .../code/func/main.go:20 +0x98
```

程序运行期间 funcB 中引发了 panic 导致程序崩溃，异常退出了。这个时候我们就可以通过 recover 将程序恢复回来，继续往后执行。

```
func funcA() {
    fmt.Println("func A")
}

func funcB() {
    defer func() {
        err := recover()
        //如果程序出出现了 panic 错误,可以通过 recover 恢复过来
        if err != nil {
            fmt.Println("recover in B")
        }
    }()
    panic("panic in B")
}

func funcC() {
    fmt.Println("func C")
}
func main() {
    funcA()
    funcB()
    funcC()
}
```

注意：

1. recover()必须搭配 defer 使用。
2. defer 一定要在可能引发 panic 的语句之前定义。

2、defer 、recover 实现异常处理

```
func fn2() {

    defer func() {
        err := recover()
        if err != nil {
            fmt.Println("抛出异常给管理员发送邮件")
            fmt.Println(err)
        }
    }()

    num1 := 10
    num2 := 0
    res := num1 / num2
    fmt.Println("res=", res)
}
```

3、defer 、panic、recover 抛出异常

```
func readFile(fileName string) error {

    if fileName == "main.go" {

        return nil

    }

    return errors.New("读取文件错误")

}


func fn3() {

    defer func() {

        err := recover()

        if err != nil {
```

```
                fmt.Println("抛出异常给管理员发送邮件")
            }
        }()
        var err = readFile("xxx.go")
        if err != nil {
            panic(err)
        }
        fmt.Println("继续执行")
}
func main() {
        fn3()
}
```