Go函数

函数

数学定义

- y=f(x)，y是x的函数，x是自变量。y=f(x0, x1, ..., xn)

Go函数

- 由若干语句组成的语句块、函数名称、参数列表、返回值构成，它是组织代码的最小单元
- 完成一定的功能

函数的作用

- 结构化编程对代码的最基本的**封装**，一般按照功能组织一段代码
- 封装的目的为了**复用**，减少冗余代码
- 代码更加简洁美观、可读易懂

函数的分类

- 内建函数，如make、new、panic等
- 库函数，如math.Ceil()等
- 自定义函数，使用func关键字定义

函数定义

```
func 函数名(参数列表) [(返回值列表)]{
    函数体（代码块）
    [return 返回值]
}

这里[]表示其中的内容可有可无
```

- 函数名就是标识符，命名要求一样
- 定义中的参数列表称为**形式参数**，只是一种符号表达（标识符），简称**形参**
- 返回值列表可有可无，需要return语句配合，表示一个功能函数执行完返回的结果
- `函数名(参数列表) [(返回值列表)]`这部分称为**函数签名**
- Go语言中形参也被称为入参，返回值也被称为出参

函数调用

- 函数定义，只是声明了一个函数，它不能被执行，需要调用执行
- 调用的方式，就是**函数名后加上小括号**，如有必要在括号内填写上参数
- 调用时写的参数是**实际参数**，是实实在在传入的值，简称**实参**，这个过程称为传实参，简称传参
- 如果定义了返回值列表，就需要配合使用return来返回这些值

```go
// 函数定义
// x、y是形式参数，result是返回值
func add(x, y int) int {
    result := x + y // 函数体
    return result   // 返回值
}

func main() {
    out := add(4, 5) // 函数调用，可能有返回值，使用变量接收这个返回值
    fmt.Println(out) // 对于Println函数来说，这也是调用，传入了out实参
}
```

上面代码解释：

- 定义一个函数add，函数名是add，能接受2个整型参数
- 该函数计算的结果，通过返回值返回，需要return语句
- 调用时，通过函数名add后加2个参数，返回值可使用变量接收
- **函数名也是标识符**
- **返回值也是值**
- 一般习惯上函数定义需要在调用之前，也就是说调用时，函数已经被定义过了。请在书写代码时，也尽量这样做，便于阅读代码

函数调用原理

特别注意，函数定义只是告诉你有一个函数可以用，但这不是函数调用执行其代码。至于函数什么时候被调用，不知道。一定要分清楚定义和调用的区别。

函数调用相当于运行一次函数定义好的代码，函数本来就是为了复用，试想你可以用加法函数，我也可以用加法函数，你加你的，我加我的，应该互不干扰的使用函数。为了实现这个目标，函数调用的一般实现，都是把函数压栈（LIFO），每一个函数调用都会在栈中分配专用的栈帧，本地变量、实参、返回值等数据都保存在这里。

> 一句不准确的口诀：函数的每一次调用都是独立的，不相干的。—— wayne

上面的代码，首先调用main函数，main压栈，接着调用add(4, 5)时，add函数压栈，压在main的栈帧之上，add调用return，add栈帧消亡，回到main栈帧，将add返回值保存在main栈帧的本地变量out上。

函数类型

```go
package main

import "fmt"

func fn1()              {}
```

```
func fn2(i int) int     { return 100 }
func fn3(j int) (r int) { return 200 }

func main() {
    fmt.Printf("%T\n", fn1)
    fmt.Printf("%T\n", fn2)
    fmt.Printf("%T\n", fn3)
}

输出如下
func()
func(int) int
func(int) int
```

可以看出同一种签名的函数是同一种类型

返回值

参考 https://go.dev/ref/spec#Return_statements

- 返回值变量是**局部变量**

1、无返回值函数

在Go语言中仅仅一个return并不一定表示无返回值，只能说在一个无返回值的函数中，return表示无返回值函数返回。

```
// 无返回值函数，可以不使用return，或在必要时使用return
func fn1()  {
    fmt.Println("无返回值函数")
    return // return可有可无，如有需要，在必要的时候使用return来返回
}

t := fn1()          // 错误，无返回值函数无返回值可用
fmt.Println(fn1()) // 错误，无返回值函数无返回值可打印
```

2、返回一个值

在函数体中，必须显式执行return

```
// 返回一个值，没有变量名只有类型
func fn2() int {
    a := 100
    return a + 1 // return后面只要类型匹配就行
}

fmt.Println(fn2()) // 返回101
t := fn2()         // 返回101
```

```
// 返回一个值，有变量名和类型
func fn3() (r int) {
    r = 200
    return r - 1 // 类型匹配
```

```
}

fmt.Println(fn3())

//上面的函数还可以写成下面的形式
func fn3() (r int) {
    r = 200
    return // 如果返回的标识符就是返回值列表中的标识符，可以省略
}

fmt.Println(fn3())
```

3、返回多值

Go语言是运行函数返回多个值

```
// 返回多个值
func fn4() (int, bool) {
    a, b := 100, true
    return a, b
}

fmt.Println(fn4())
x, y := fn4() // 需要两个变量接收返回值
```

```
// 返回多个值
func fn4() (i int, b bool) {
    i, b = 100, true
    return // 如果和返回值列表定义的标识符名称和顺序一样，可省略
}

fmt.Println(fn4())
x, y := fn4() // 需要两个变量接收返回值
```

```
// 下面写法对吗？
func fn4() (i int, b bool) {
    return
}
```

这种写法对的，因为返回值i、b也是函数的局部变量，进入函数是，也会被初始化为零值。

```
// 注意下面写法的错误
func fn5() (i int, err error) {
    if _, err := os.Open("o:/t"); err != nil {
        return // 错误，因为err被重新定义，只能在if中使用，返回值的err就被覆盖了，就是上一行:=的问题
        // return -1, err // 正确
    }
    return
}
```

返回值

- 可以返回0个或**多个**值
- 可以在函数定义中写好返回值参数列表
    - 可以没有标识符，只写类型。但是有时候不便于代码阅读，不知道返回参数的含义
    - 可以和形参一样，写标识符和类型来命名返回值变量，相邻类型相同可以合并写
    - 如果返回值参数列表中只有一个返回参数类型，小括号可以省略
    - 以上2种方式不能混用，也就是返回值参数要么都命名，要么都不要命名
- return
    - return之后的语句不会执行，函数将结束执行
    - 如果函数无返回值，函数体内根据实际情况使用return
    - return后如果写值，必须写和返回值参数类型和个数一致的数据
    - return后什么都不写那么就使用返回值参数列表中的返回参数的值，如果返回值参数没有赋过值，就用零值

形式参数

- 可以无形参，也可以多个形参
- 不支持形式参数的默认值
- 形参是**局部变量**

```go
func fn1()                    {} // 无形参
func fn2(int)                 {} // 有一个int形参，但是没法用它，不推荐
func fn3(x int)               {} // 单参函数
func fn4(x int, y int)        {} // 多参函数
func fn5(x, y int, z string) {} // 相邻形参类型相同，可以写到一起

fn1()
fn2(5)
fn3(10)
fn4(4, 5)
fn5(7, 8, "ok")
```

可变参数

可变参数variadic。其他语言也有类似的被称为剩余参数，但Go语言有所不同。

```go
func fn6(nums ...int) { // 可变形参
    fmt.Printf("%T %[1]v, %d, %d\n", nums, len(nums), cap(nums))
}

fn6(1)       // []int, [1]
fn6(3, 5)    // []int, [3 5]
fn6(7, 8, 9) // []int, [7 8 9]
```

- 可变参数收集实参到一个切片中
- 如果有可变参数，那它必须位于参数列表中最后。`func fn7(x, y int, nums ...int, z string){}`这是错误的

```go
func fn7(x, y int, nums ...int) {
    fmt.Printf("%d %d; %T %[3]v, %d, %d\n", x, y, nums, len(nums), cap(nums))
}

fn7(1, 2)       // 1 2; []int [], 0, 0
fn7(1, 2, 3)    // 1 2; []int [3], 1, 1
fn7(1, 2, 3, 4) // 1 2; []int [3 4], 2, 2
```

可以看出有剩下的实参才留给剩余参数。

切片分解

也可以使用切片分解传递给可变参数，这个功能和其他语言的参数解构很像，但是不一样。

```go
func fn4(x int, y int)   {} // 多参函数

p := []int{4, 5}
fn4(p...) // 错误，这在Go中不行，不能用在非可变参数non-variadic上
```

```go
func fn7(x, y int, nums ...int) {
    fmt.Printf("%d %d; %T %[3]v, %d, %d\n", x, y, nums, len(nums), cap(nums))
}

p := []int{4, 5, 6}
fn7(p...) // 这在Go中不行，报奇怪的错，原因还是不能用在非可变参数上，就用4、5用在x、y上了
// 这个例子，本以为p被分解，4和5分别对应x和y，6被可变参数nums收集，但是这在Go语言中是错误的
```

> If the final argument is assignable to a slice type `[]T` and is followed by `...`, it is passed unchanged as the value for a `...T` parameter.
>
> 如果最终的参数是某类型的切片且其后跟着...，它将无变化的传递给...T的可变参数。注意，这个过程无新的切片创建。

帮助文档这一句话，原来指的是，`切片...`只能为可变参数传参。

```go
func fn7(x, y int, nums ...int) {
    fmt.Printf("%d %d; %T %[3]v, %d, %d\n", x, y, nums, len(nums), cap(nums))
}

p := []int{4, 5}
fn7(p...)           // 错误，不能用在普通参数上
fn7(1, p...)        // 错误，不能用在普通参数上
fn7(1, 2, 3, p...) // 错误，不能用2种方式为可变参数传参，不能混用
// fn7(1, 2, p..., 9, 10) // 语法错误
// fn7(1, 2, []int{4, 5}..., []int{6, 7}...) // 语法错误，不能连续使用p...，只能一次

// 正确的如下
fn7(1, 2, []int{4, 5}...)
fn7(1, 2, p...)
fn7(1, 2, 3, 4, 5)
```

可以看出，可变参数限制较多

- 直接提供对应实参
- 可以使用使用切片分解的方式`切片...`，但是这种方式只能单独为可变形参提供实参

这和Python、JavaScript中的参数解构不一样，也确实没有它们灵活方便。

作用域

函数会开辟一个局部作用域，其中定义的标识符仅能在函数之中使用，也称为标识符在函数中的可见范围。

这种对标识符约束的可见范围，称为作用域。

1、语句块作用域

if、for、switch等语句中使用短格式定义的变量，可以认为就是该语句块的变量，作用域仅在该语句块中。

```go
s := []int{1, 3, 5}
for i, v := range s {
    fmt.Println(i, v) // i和v在for块中可见
}
fmt.Println(i, v) // 错误，在for外不可见
```

```go
if f, err := os.Open("o:/t.txt"); err != nil {
    fmt.Println(f, err) // 可见
}
fmt.Println(f, err) // 错误，不可见
```

swith、select语句中的每个子句都被视为一个隐式的代码块。

2、显式的块作用域

在任何一个大括号中定义的标识符，其作用域只能在这对大括号中。

```go
{
    // 块作用域
    const a = 100
    var b = 200
    c := 300
    fmt.Println(a, b, c) // 可见
}
fmt.Println(a, b, c) // 错误，不可见
```

3、universe块

宇宙块，意思就是全局块，不过是语言内建的。预定义的标识符就在这个全局环境中，因此什么bool、int、nil、true、false、iota、append等标识符全局可见，随处可用。

4、包块

每一个package包含该包所有源文件，形成的作用域。有时在包中顶层代码定义标识符，也称为全局标识符。

所有包内定义全局标识符，包内可见。包外需要大写首字母导出，使用时也要加上包名。

5、函数块

函数声明的时候使用了花括号，所以整个函数体就是一个显式代码块。这个函数就是一个块作用域。

下面就用一个例子理解标识符作用域

```go
package main

import "fmt"

// 包级常量、变量定义，只能使用const、var定义
const a = 100

var b = 200

// c := 300 // 不可以使用短格式
var d = 400

func showB() int {
    return b
}

func main() {
    fmt.Println(1, a)
    // fmt.Println(1.1, &a) // 注意常量不可寻址，这是对常量的保护
    var a = 500            // 重新定义a为变量可以吗？
    fmt.Println(2, a, &a) // 可以访问吗？

    // 以下对b的操作，思考一下，b变了吗？
    fmt.Println(3, b, &b)
    b = 600
    fmt.Println(3.1, b, &b)
    b := 601
    fmt.Println(3.2, b, &b)
    fmt.Println(3.3, showB()) // 这里显示多少？

    {
        const j = 'A'
        var k = "magedu"
        t := true
        a = 700
        b := 800
        fmt.Println(4, j, k, t, a, b)
        {
            x := 900
            fmt.Println(4.1, a, b, d, j, k, t, x)
        }
```

```
        fmt.Println(4.2, x) // 是多少
    }
    fmt.Println(4.3, j, k, t) // 是多少？
    fmt.Println(4.4, a, b) // 是多少

    for i, v := range []int{1, 3, 5} {
        fmt.Println(i, v)
    }
    fmt.Println(i, v) // i、v不可见，它们在for的作用域中
}
```

标识符作用域

- 标识符对外不可见，在标识符所在作用域外是看不到标识符的
- 使用标识符，自己这一层定义的标识符优先，如果没有，就向外层找同名标识符——自己优先，由近及远
- 标识符对内可见，在内部的局部作用域中，可以使用外部定义的标识符——向内穿透
- 包级标识符
    - 在所在包内，都可见
    - 跨包访问，包级变量必须大写开头，才可以在包外使用`xx包名.VarName`方式访问