# 函数

1. 函数可以带有0或多个参数。下面的示例中, `add` 函数包含两个 `int` 类型的参数。

```
package main

import "fmt"

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

func main() {
    fmt.Println(add(40, 60))
}
```

注意**类型在变量的后面**。
可以参考Go的作者之一Rob Pike写的文章《Go的声明语法》解释为什么使用这种类型声明在后，字段名称在前的方式。
通过阅读，其实最大的好处是在函数式编程中，go语法更便于阅读，将类型放后，函数名中间，func声明在前。

1. 当两个或更多连续的函数参数是同一种类型时，可以省略类型除了最后一个参数。
   例如: 可以将 `x int, y int` 简写为 `x, y int`

`1.` 中的 `add` 函数参数可以简写为以下:

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

# 多个返回值

Go函数可以有多个返回值。下面的 `swap` 函数返回两个字符串

```
package main

import "fmt" // gogland(IDE) 键入词imp 当使用包组织方式而只有一个包路径时，删除后是可以自动补全包路径的，很智能
```

```
func swap(x, y string) (string, string) {
    return y, x
}

func main() {
    a, b := swap("hello", "world")
    fmt.Println(a, b)
}
```

参数是传递变量副本。不论是值的副本还是指针地址，都变量副本。值的拷贝。当然在指针地址的拷贝传递后盖板会影响到原变量，因为指针指向是原变量。后面会更深刻的阐述。

```
a, b := "hello", "world"
m, n := swap(a, b)
fmt.Println(a, b) //a,b的值并未交换
fmt.Println(m, n) //返回值变量是a,b变量副本的交换后的值
```

## 命名返回值(naked return)

Go的返回值是可以命名的，它们将被当做变量声明在方法的签名上。

返回值的名称通常被用在标记返回值的含义上。

没有带参数的`return`语句块返回被命名的返回值。这被称作是"赤裸裸的"回报("naked" return)...即"裸"返回或者直接返回语句。

直接返回语句仅应该用在短函数中，例如下面的示例。但注意:命名返回值用在内容较长的函数中影响可读性。

```
package main

import "fmt"

func add(x int, y int) (result int) {
    result = x + y
    return
}
func main() {
    fmt.Println(add(2,1))
}
```

```
func split(sum int) (x, y int) {
```

```
    x = sum * 4 / 9
    y = sum - x
    return //naked return
}

func main() {
    x, y := split(99)
    fmt.Println(x, y)
}
```

尽量避免使用命名返回值
```go
package main

import "fmt"

func main() {
fmt.Println(greet("Jane ", "Doe"))
}

func greet(fname string, lname string) (s string) {
s = fmt.Sprint(fname, lname)
return
}

/*
IMPORTANT
Avoid using named returns.

Occasionally(偶尔) named returns are useful. Read this article for more information:
https://www.goinggo.net/2013/10/functions-and-naked-returns-in-go.html
*/
```

```
## 函数值

```go
// squares返回一个匿名函数。
// 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 {
        x++
        return x * x
```
```

```
    }
}
func main() {
    f := squares()
    fmt.Println(f()) // "1"
    fmt.Println(f()) // "4"
    fmt.Println(f()) // "9"
    fmt.Println(f()) // "16"
}
```

squares的例子证明，**函数值不仅仅是一串代码，还记录了状态**。在squares中定义的匿名内部函数可以访问和更新squares宿主函数中的局部变量，这意味着**匿名函数和squares宿主函数，存在变量引用**。这就是**函数值属于引用类型**的原因，并且也是函数值不可比较的原因(引用类型不可比较)。**Go使用闭包（closures）技术实现函数值，Go程序员也把函数值叫做闭包。**

变量的生命周期不由它的作用域决定(闭包):square闭包函数返回后,变量x仍然隐式的存在于f中。

```
func main() {
    fmt.Println(square()())
    fmt.Println(square()())
    fmt.Println(square()())
    fmt.Println(square()())
    fmt.Println("-------")
    f:= square()
    fmt.Println(f())
    fmt.Println(f())
    fmt.Println(f())
    fmt.Println(f())
}

func square() func() int {
    var x int
    return func() int {
        x++
        return x * x
    }
}
```

需要注意一定要获取到函数值的引用，否则只是相同的值而已。因为只有获得闭包返回变量后才能获得匿名函数的隐式变量。

# 回调

## 案例1 打印

```
package main

import "fmt"

func main() {
    nums := []int{2, 1, 3, 4}
    visit2(nums, myPrint)
}

func visit2(nums []int, callback func(int)) {
    for _, v := range nums {
        callback(v)
    }
}
func myPrint(num int) {
    fmt.Println(num)
}
```

## 案例2 过滤

```
package main

import "fmt"

func main() {
    var myArr = []int{1, 2, 3, 4, 5}
    fmt.Println(filter3(myArr, myFilter))
}
func myFilter(i int) bool {
    if i%2 == 0 {
        return true
    }
    return false
}

func filter3(arr []int, callback func(int) bool) []int {
    var filter_arr []int
    for _, n := range arr {
        if callback(n) {
            filter_arr = append(filter_arr, n)
        }
    }
```

```
    return filter_arr
}
```

# 函数进阶 函数类型与函数值(闭包)

go语言本质都是**值**类型，函数是**函数值**类型。因此我们可以简单的声明一个函数类型 `var f func(x int, y int) int` 其零值为 `nil`

```
package main

import "fmt"

type myFunc func(x int, y int) int

func main() {
    var f1 func(a string) string //匿名函数类型（而f2则是myFunc类型）
    fmt.Println(f1)              // nil

    var f2 myFunc = func(x int, y int) int{return x+y} //其中myFunc也可以省略
    fmt.Printf("%T\n",f2) // myFunc
    fmt.Println(f2(3, 4))

    var f3 = func(x int, y int) int { //函数值（一个函数的实现需要被使用，这里就是一种函数的值）
        return x + y
    }
    fmt.Println(f3(1, 2)) // 3
}

//<nil>
//main.myFunc
//7
//3
```

接下来更加进一步的将函数表达式(函数变量)进行调用, 本质是匿名函数的调用

- **函数调用表达式**直接返回结果值
- 匿名函数调用返回值

```
var result = func(x,y int) int {
        return x+y
    }(11,2)
```

```
fmt.Println(result)
```

**函数值之间是不可以比较的，因此也不能作为map的key**
**函数值**不仅仅使我们可以使用**数据参数化函数**，也可以通过**行为**。(函数可以**值参数化**和**行为参数化**)

```
package main

import "fmt"

func main() {
    var a = add //var a int = add
    fmt.Println(a(3, 5))

    var b func(x,y int) int
    b = add
    fmt.Println(b(3,1))
}

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

- **函数行为参数化**

```
package main

import (
    "fmt"
    "strings"
)

func main() {
    fmt.Println(strings.Map(add1, "abcd"))
    fmt.Println(strings.Map(add1, "HAL-9000"))

}

func add1(r rune) rune {
    return r + 1
}
//bcde
//IBM.:111
```

函数行为参数化思想是很有意思的一种编程思维，在函数式编程中和接口编程中有很大使用价值。要掌握。

```
package main

import "fmt"

func visit(numbers []int, callback func(int)) {
    for _, n := range numbers {
        callback(n)
    }
}

func main() {
    visit([]int{1, 2, 3, 4}, func(n int) {
        fmt.Println(n)
    })
}

// callback: passing a func as an argument
```

## 匿名函数作为函数值在使用的时候定义

```
package main

import (
    "fmt"
    "strings"
)

func main() {
    fmt.Println(strings.Map(func(r rune) rune {
        return r + 1
    }, "HAL-9000"))
}
```

匿名函数字面值表达式在使用时在定义，这种定义方式使得**匿名函数可以访问其所在函数的完整词法环境**。这意味着在函数中定义的**内部函数可以引用该函数的变量**。

go使用闭包技术实现函数值，Go也把函数值称之为闭包。

```
package main
```

```go
import (
    "fmt"
)

/*func square() int {
    var x int
    x++
    return func(x int) int {
        return  x * x
    }(x)
}*/


func square() func() int{
    var x int

    return func() int{
        x++
        fmt.Printf("%v\t%p\n",x,&x)
        return x*x
    }
}
func main() {
    f := square()
    fmt.Println(f())
    fmt.Println(f())
    fmt.Println(f())
    fmt.Println(f())

    /*
    1   0xc420070188
    1
    2   0xc420070188
    4
    3   0xc420070188
    9
    4   0xc420070188
    16
    */

    //fmt.Println(square()())
    //fmt.Println(square()())
    //fmt.Println(square()())
    //fmt.Println(square()())
    //1 0xc42000e238
    //1
    //1 0xc42000e298
```

```
    //1
    //1 0xc42000e2c0
    //1
    //1 0xc42000e2e8
    //1
}
```

> squares的例子证明,函数值不仅仅是一串代码,还记录了状态。在`squares`中定义的匿名内部函数可以访问和更新`squares`中的局部变量,这意味着匿名函数和squares中,存在变量引用。这就是函数值属于引用类型和函数值不可比较的原因。Go使用闭包(`closures`)技术实现函数值,Go程序员也把函数值叫做`闭包`。

这里再次证明了变量的**作用域和生命周期是两个不同的概念，一个是静态的，一个是动态的。**

> closure helps us limit the scope of variables used by multiple functions
> without closure, for two or more funcs to have access to the same variable,
> that variable would need to be package scope

## 函数值的重要特性-记录迭代变量的内存地址

- 首先我们先来看下普通变量在迭代中如何记录值的. golang中使用`:=`赋值初始化新变量，如果多次则会**复用地址**。这样在循环中，可以节省很多空间浪费。

```
package main

import "fmt"

func main() {
    var strs = []string{"hello","world","china"}

    for _, v := range strs{
        fmt.Printf("%v\t%p\n", v,&v)
    }
    fmt.Printf("%v\n", "-----------------")
    for _, v := range strs {
        v = v
        fmt.Printf("%v\t%p\n", v, &v)
    }

    fmt.Printf("%v\n", "-----------------")
    for _, v := range strs {
        v := v
        fmt.Printf("%v\t%p\n", v, &v)
```

```
    }
}

// hello    0xc4200701b0
// world    0xc4200701b0
// china    0xc4200701b0
// ----------------
// hello    0xc420070200
// world    0xc420070200
// china    0xc420070200
// ----------------
// hello    0xc420070250
// world    0xc420070270
// china    0xc420070290
```

但是我们要说的重点是在于`for range`和闭包的结合使用，会产生的坑效果。

```
    strs := []string{"hello", "world", "china"}
    for _, v := range strs {
        go func() {
            fmt.Println(v) //闭包 记录迭代变量的内存地址，而不是迭代某一刻的值！！
        }()
    }
    select {}

    //china
    //china
    //china
    //fatal error: all goroutines are asleep - deadlock!
    //
    //  goroutine 1 [select (no cases)]:
    //  main.main()
    //  /Users/fqc/work/src/myecho/main.go:14 +0xfc
    //  exit status 2
```

我们看到程序并未如我们所料，而是打印出了三次china，说明**v的地址是指向了最后的china，内部是函数闭包方式，产生了延迟调用(而迭代是已经执行完了)【这里的延迟调用是最后发生调用()，在v被迭代赋值完毕】，而v的地址在函数值被调用的时候已经是指向了最后的china。**
原来**闭包函数值中记录迭代变量的内存地址，而不是迭代变量某一刻的值。单次执行看不出什么，但是在迭代中则会等待迭代执行完，才去执行函数值(闭包)。** (迭代变量<->循环变量好多了)

在`go语句`和`defer语句`中也是如此。
所以需要注意**在循环体中将循环变量赋值给新的局部变量非常重要，否则由于延迟调用每次获取**

**的都是最后一次迭代值。** [另外需要注意，闭包延迟调用外层循环是立即执行，并且参数拷贝也是立即执行的，正利用这一点，所以将循环变量赋值给新的局部变量非常重要]。

所谓闭包是指内层函数引用了外层函数中的变量或称为引用了自由变量的函数，其返回值也是一个函数，了解过的语言中有闭包概念的像 js，python，golang 都类似这样。

```
strs := []string{"hello", "world", "china"}
    for _, v := range strs {
        go func(v string) {
            fmt.Println(v) //闭包函数值记录迭代变量的内存地址，而不是迭代某一刻的值!
!
        }(v)
    }
    select {}
```

我们看到结果将会是strs数组的每个值。可以先忽略select{}，目前只是为了防止主routine先退出而看不到结果。

## 函数列表和匿名函数的使用不当

```
package main

import "fmt"

func main() {
    for _, f := range test() {
        f()
    }
}

func test() []func() {
    var s []func()
    for i := 0; i < 3; i++ {
        s = append(s, func() {
            fmt.Printf("%v,%p\n", i, &i)
        })
    }
    return s
}

//3,0xc42007a050
//3,0xc42007a050
//3,0xc42007a050
```

解决方案

```
func test() []func() {
    var s []func()
    for i := 0; i < 3; i++ {
        x := i
        /*  s = append(s, func(x int) {
                fmt.Printf("%v,%p\n", x, &x)
            }(x)) //这里就不应该调用啊
            */
        s = append(s, func() {
            fmt.Printf("%v,%p\n", x, &x)
        })
    }
    return s
}
```

## 使用闭包修改全局变量

```
package main

import (
    "fmt"
)

var x int = 1

func main() {
    y := func() int {
        x += 1
        return x
    }()
    fmt.Println("main:", x, y)

}

// 结果:     main: 2 2
```

## 延迟调用

defer 调用会在当前函数执行结束前才被执行，这些调用被称为延迟调用，
defer 中使用匿名函数依然是一个闭包。

```
package main

import "fmt"

func main() {
    x, y := 1, 2

    defer func(a int) {
        fmt.Printf("x:%d,y:%d\n", a, y)  // y 为闭包引用
    }(x)      // 复制 x 的值，并且是立即复制了，a的值肯定是1，而y呢则是闭包引用，直到defer执行时，才能最终确定y的值

    x += 100
    y += 100
    fmt.Println(x, y)
}
```

输出结果：

101 102
x:1,y:102

> 从形式上看，匿名函数都是闭包。闭包的使用非常灵活，上面仅是几个比较简单的示例，不当的使用容易产生难以发现的 **bug，当出现意外情况时，首先检查函数的参数，声明可以接收参数的匿名函数，这些类型的闭包问题也就引刃而解了。**

当然具体的情况还是看实际的需求。

# defer

- Go独有的defer机制让繁琐的重复代码问题变得简单。
- 只需要调用的普通函数之前增加defer关键字，就完成了defer语法， > 当defer语句执行时，跟在defer后面的函数会被延迟执行。直到包含该defer语句的函数(外层函数)执行完毕时，defer后的函数才会被执行，不论包含defer语句的函数是通过return正常结束还是由于panic导致的异常结束。也就是说return后或panic后都会执行defer语句。
- defer语句推迟到外层函数返回之前才执行。
- 即使发生严重错误，defer也会执行
- 常用语资源清理，文件关闭，解锁以及记录时间等操作。
- defer语句解决类似多处资源关闭的问题，可以精简代码处理。
- defer经常被用于处理成对的操作，比如打开，关闭，连接，断开，加锁，释放锁。

- **释放资源的defer语句应该直接跟在请求资源的语句后。** 所以一个函数内的多处资源释放使用defer只需要一行代码就可以替代。
- 调试复杂程序时，defer的机制也常被用于记录何时进入和退出函数
- (解释上个特性)defer语句调用参数`会立刻求值`，但是函数直到外层函数返回之前是不会调用的。
- **defer语句中的函数会在return更新返回值变量之后再执行，又因为在函数中定义的匿名函数可以访问该函数的包括返回值在内的所有变量，因此对匿名函数采用defer机制，可以使其观察函数的返回值。**

```
package main

import (
    "fmt"
)

func main() {
    defer fmt.Println(3 + 2)
    defer fmt.Println("Hello")
    fmt.Println("World")
}

// world
// Hello
// 5
```

```
fmt.Println("a")
defer fmt.Println("b")
defer fmt.Println("c")
// a
// c
// b
```

简单的defer实现代码的切面编程
实现追踪记录方法的开始、结束、持续时间

```
package main

import (
    "log"
    "time"
)
```

```
func bigSlowOperation() {
    defer trace("bigSlowOperation()")()
    //
    time.Sleep(2 * time.Second)
    //
}

func trace(msg string) func() {
    start := time.Now()
    log.Printf("enter %s\t%s\n", msg, start)
    return func() {
        log.Printf("exist %s (%s)", msg, time.Since(start))
    }
}

func main() {
    for i := 0; i < 5; i++ {
        bigSlowOperation()
    }
}
```

- 匿名函数版本

```
package main

import (
    "log"
    "time"
)

func bigSlowOperation() {
    defer func() func(){
        start := time.Now()
        log.Printf("enter bigSlowOperation() %s",start)
        return func() {
            log.Printf("exist bigSlowOperation() %s",time.Since(start))
        }
    }()() // notice
    time.Sleep(2 * time.Second)
}

func main() {
    for i := 0; i < 5; i++ {
        bigSlowOperation()
    }
}
```

下面的版本直接使用匿名函数调用的方式，最为简便。

```
package main

import (
    "time"
    "fmt"
)

func trace() {
    now := time.Now()
    fmt.Printf("start at: %v\n", now)

    defer func() {
        fmt.Println("stop at:",time.Now())
        fmt.Printf("duaration: %v\n", time.Since(now))
    }()

    time.Sleep(2 * time.Second)

}

func main() {
    trace()
}
```

defer语句的函数会在return更新返回值之后执行，又因为在函数中定义的匿名函数可以访问该函数的包括返回值在内的所有变量，因此对匿名函数采用defer机制，可以使其观察函数的返回值。

- 1. 记录观察函数的返回值

```
package main

import "fmt"

func main() {
    double(10)
}

func double(x int) (result int) {
    defer func() { fmt.Printf("double(%d) = %d", x, result) }()
    return x + x
}
```

- 2.

```
func triple(x int) (result int) {
    defer func(){result+=x}
    return double(x)
}
triple(8)
```

- 被延迟执行的匿名函数可以修改函数返回给调用者的返回值

```
package main

import (
    "fmt"
)

func main() {
    fmt.Println(triple(5))
}

func double(x int) (result int) {
    defer func() { fmt.Printf("double(%d) = %d\n", x, result) }()
    return x + x
}

func triple(x int) (result int) {
    defer func(){result+=x}()
    return double(x)
}
```

## defer栈

推迟的函数调用被压入栈中。当一个函数返回，它将以last-in-first-out(先进后出或后进先出)的顺序调用。阅读这篇文章可以学习更多defer相关内容。

```
package main
import (
    "fmt"
)

func main() {
    fmt.Println("counting...")
    for i := 0; i< 10; i++ {
        defer fmt.Println(i)
```

```
    }
    fmt.Println("done.")
}
/**结果如下
counting...
done.
9
8
7
6
5
4
3
2
1
0
*/
```

## 函数体中的defer函数体执行完毕才能执行

在循环体中的defer语句需要特别注意
需要在函数执行完毕之后，这些被延迟的函数才会执行
注意：这样做是比较危险的操作，可能会导致资源消耗过快，导致内存溢出。

```
package main

import (
    "fmt"
)

func main() {
    for i := 0; i < 5; i++ {
        defer fmt.Println("defer", i)//possible resource leak,"defer" is cal
led in a for loop
        fmt.Println("函数体", i)
    }
}
```

```
for _,file := range fileNames {
    f,err := os.Open(file)
    if err != nil {
        return err
    }
```

```
    defer f.Close() //Note: risky! could run out of the file 用完或耗尽资源

}
```

解决方案. 将循环体内的defer语句移到另一个函数中，每次循环时调用该函数。

```
for _,file := range fileNames {
    if f,err := doFile(file);err!=nil{
        return err
    }
}

func doFile(file string) error {
    if f,err := os.Open(file); err != nil {
        return err
    }
    defer f.Close()
}
```

# defer使用时要注意的坑

1. 注意命名返回参数与defer的使用

```
package main

import "fmt"

func f() (result int) {
    defer func() {
        result++
    }()
    return 0
}

func main() {
    fmt.Printf("result: %v", f())
}
```

案例二

```
package main
```

```go
import "fmt"

func f() (r int) {
    t := 5
    defer func() {
        t = t + 5
        //r = t + 5
    }()
    //return r
    return t
}

func main() {
    fmt.Printf("result: %v", f())
}
```

案例三

```go
package main

import "fmt"

func f() (r int) {
    defer func() {
        r = r + 5
    }()
    return 1
}

func main() {
    fmt.Printf("result: %v", f())
}
```

```go
package main

import "fmt"

func f() (r int) {
    defer func(r int) {
        r = r + 5
    }(r)
    return 1
}
```

```
func main() {
    fmt.Printf("result: %v", f())
}
```

return语句并非一条原子指令。函数返回的过程是这样的:先给返回值赋值,然后调用defer表达式,最后才是返回到调用函数中。因此，我们就看到在回到调用函数之前，defer表达式可能修改了返回值，与我们的预估判断截然不同。

解决方案将语句分解

```
returnValue = xxx //1. 先给返回值赋值
defer(){}() // 2. 调用defer
return //空的return
```

总结:defer与命名返回参数共用时，强烈建议使用空return。

1. 闭包和引用

```
package main

import "fmt"

func main() {
    for i := 0; i < 3; i++ {
        defer fmt.Println(i)// 获得了i的地址拷贝
    }

    for i := 0; i < 3; i++ {
        defer func() {
            fmt.Println(i) // 定义defer时就获得了i=3时的拷贝
        }()
    }
}

// 3
// 3
// 3
// 2
// 1
// 0
```

如果函数体内某个变量作为defer时匿名函数的参数时，则在定义defer时已经获得了拷贝，否则

则是引用某个变量的地址。

## os.Create函数打开的文件 不使用defer的原因

```
// Fetch downloads the URL and returns the
// name and length of the local file.
func fetch(url string) (filename string, n int64, err error) {
    resp, err := http.Get(url)
    if err != nil {
        return "", 0, err
    }

    defer resp.Body.Close()
    local := path.Base(resp.Request.URL.Path)
    if local == "/" {
        local = "index.html"
    }

    f, err := os.Create(local)
    if err != nil {
        return "", 0, err
    }

    n, err = io.Copy(f, resp.Body)
    // Close file, but prefer error from Copy, if any.
    if closeErr := f.Close(); err == nil {
        err = closeErr
    }
    return local, n, err
}
```

通过os.Create打开文 件进行写入,在关闭文件时,我们没有对f.close采用defer机制,因为这会产生一些微妙的错 误。许多文件系统,尤其是NFS,写入文件时发生的错误会被延迟到文件关闭时反馈。如果 没有检查文件关闭时的反馈信息,可能会导致数据丢失,而我们还误以为写入操作成功。如 果io.Copy和f.close都失败了,我们倾向于将io.Copy的错误信息反馈给调用者,因为它先于f.close发生,更有可能接近问题的本质。

## defer与闭包

```
package main

import "fmt"
```

```go
func main() {
    fs := [4]func(){}
    fmt.Println(fs)

    for i := 0; i < 4; i++ {
        defer fmt.Println("defer i = ",i) // i 为int类型的参数 遵循值拷贝的规则
        defer func() {fmt.Println("defer closure i = ",i) }() // 闭包引用外部变量的地址
        fs[i] = func() { fmt.Println("closure i = ",i) } // 闭包引用外部变量的地址，到下面的迭代中执行
    }

    for _, f := range fs {
        f()
    }
}

// [<nil> <nil> <nil> <nil>]
// closure i =  4
// closure i =  4
// closure i =  4
// closure i =  4
// defer closure i =  4
// defer i =  3
// defer closure i =  4
// defer i =  2
// defer closure i =  4
// defer i =  1
// defer closure i =  4
// defer i =  0
```

## defer与naked-return

```go
package main

import "fmt"

func ReturnId() (id int, err error) {
   //id =10
    defer func(id int) {
        if id == 10 {
            err = fmt.Errorf("Invalid Id\n")
        }
    }(id)

    id = 10
```

```
    return
}
func main() {
    id, err := ReturnId()
    fmt.Println(id,err)
}
```

两种情况完全不同
naked-return

# 传值还是传引用?

```
func add(x,y,_ int) int {
    return x+y
}
func main() {
    fmt.Printf("%T",add) //func(int, int, int) int
}
```

函数类型 `func(int, int, int) int` 也被称作是函数标识符，与函数的参数和返回值 `类型` 有关，与名称无关。
`_` 空白标识符(blank identifier)，标识该参数未使用。

**实参是通过值的方式传递，因此函数的形参是实参的拷贝。对形参的修改不会影响到实参。但是如果实参包括引用类型如 slice，map，function，channel等类型，实参可能会由于函数的间接引用而改变**(引用类型的形参实质也是值传递，只不过值是实参的引用地址值，修改会间接的通过引用修改指针指向的地址)。
go没有默认参数值。

没有函数体的函数声明说明不是go的实现
这样的声明定义其实就是函数标识符
```go
package math

func sin(x float64) float // implemented in assembly language
```