# Go语言函数（Go语言func）

< [Go语言分布式id生成器](http://c.biancheng.net/view/vip_7314.html)[Go语言函数声明](http://c.biancheng.net/view/52.html) >

函数是组织好的、可重复使用的、用来实现单一或相关联功能的代码段，其可以提高应用的模块性和代码的重复利用率。  
  
Go 语言支持普通函数、匿名函数和闭包，从设计上对函数进行了优化和改进，让函数使用起来更加方便。  
  
Go 语言的函数属于“一等公民”（first-class），也就是说：

* 函数本身可以作为值进行传递。
* 支持匿名函数和闭包（closure）。
* 函数可以满足接口。

# Go语言函数声明（函数定义）

< [Go语言函数](http://c.biancheng.net/golang/func/)[示例：将秒转换为具体的时间](http://c.biancheng.net/view/vip_7315.html) >

[C语言中文网推出辅导班啦，包括「C语言辅导班、C++辅导班、算法/数据结构辅导班」，全部都是一对一教学：一对一辅导 + 一对一答疑 + 布置作业 + 项目实践 + 永久学习。QQ在线，随时响应！](http://fudao.biancheng.net/" \t "http://c.biancheng.net/view/_blank)

函数构成了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，每一个程序都包含很多的函数，函数是基本的代码块。  
  
因为Go语言是编译型语言，所以函数编写的顺序是无关紧要的，鉴于可读性的需求，最好把 main() 函数写在文件的前面，其他函数按照一定逻辑顺序进行编写（例如函数被调用的顺序）。  
  
编写多个函数的主要目的是将一个需要很多行代码的复杂问题分解为一系列简单的任务来解决，而且，同一个任务（函数）可以被多次调用，有助于代码重用（事实上，好的程序是非常注意 DRY 原则的，即不要重复你自己（Don't Repeat Yourself），意思是执行特定任务的代码只能在程序里面出现一次）。  
  
当函数执行到代码块最后一行}之前或者 return 语句的时候会退出，其中 return 语句可以带有零个或多个参数，这些参数将作为返回值供调用者使用，简单的 return 语句也可以用来结束 for 的死循环，或者结束一个协程（goroutine）。  
  
Go语言里面拥三种类型的函数：

* 普通的带有名字的函数
* 匿名函数或者 lambda 函数
* 方法

## 普通函数声明（定义）

函数声明包括函数名、形式参数列表、返回值列表（可省略）以及函数体。

func 函数名(形式参数列表)(返回值列表){  
    函数体  
}

形式参数列表描述了函数的参数名以及参数类型，这些参数作为局部变量，其值由参数调用者提供，返回值列表描述了函数返回值的变量名以及类型，如果函数返回一个无名变量或者没有返回值，返回值列表的括号是可以省略的。  
  
如果一个函数声明不包括返回值列表，那么函数体执行完毕后，不会返回任何值，在下面的 hypot 函数中：

1. func hypot(x, y float64) float64 {
2. return math.Sqrt(x\*x + y\*y)
3. }
4. fmt.Println(hypot(3,4)) // "5"

x 和 y 是形参名，3 和 4 是调用时的传入的实数，函数返回了一个 float64 类型的值，返回值也可以像形式参数一样被命名，在这种情况下，每个返回值被声明成一个局部变量，并根据该返回值的类型，将其初始化为 0。  
  
如果一个函数在声明时，包含返回值列表，那么该函数必须以 return 语句结尾，除非函数明显无法运行到结尾处，例如函数在结尾时调用了 panic 异常或函数中存在无限循环。  
  
正如 hypot 函数一样，如果一组形参或返回值有相同的类型，我们不必为每个形参都写出参数类型，下面 2 个声明是等价的：

1. func f(i, j, k int, s, t string) { /\* ... \*/ }
2. func f(i int, j int, k int, s string, t string) { /\* ... \*/ }

下面，我们给出 4 种方法声明拥有 2 个 int 型参数和 1 个 int 型返回值的函数，空白标识符\_可以强调某个参数未被使用。

1. func add(x int, y int) int {return x + y}
2. func sub(x, y int) (z int) { z = x - y; return}
3. func first(x int, \_ int) int { return x }
4. func zero(int, int) int { return 0 }
5. fmt.Printf("%T\n", add) // "func(int, int) int"
6. fmt.Printf("%T\n", sub) // "func(int, int) int"
7. fmt.Printf("%T\n", first) // "func(int, int) int"
8. fmt.Printf("%T\n", zero) // "func(int, int) int"

函数的类型被称为函数的标识符，如果两个函数形式参数列表和返回值列表中的变量类型一一对应，那么这两个函数被认为有相同的类型和标识符，形参和返回值的变量名不影响函数标识符也不影响它们是否可以以省略参数类型的形式表示。  
  
每一次函数在调用时都必须按照声明顺序为所有参数提供实参（参数值），在函数调用时，Go语言没有默认参数值，也没有任何方法可以通过参数名指定形参，因此形参和返回值的变量名对于函数调用者而言没有意义。  
  
在函数中，实参通过值传递的方式进行传递，因此函数的形参是实参的拷贝，对形参进行修改不会影响实参，但是，如果实参包括引用类型，如指针、slice(切片)、map、function、channel 等类型，实参可能会由于函数的间接引用被修改。

## 函数的返回值

Go语言支持多返回值，多返回值能方便地获得函数执行后的多个返回参数，Go语言经常使用多返回值中的最后一个返回参数返回函数执行中可能发生的错误，示例代码如下：

1. conn, err := connectToNetwork()

在这段代码中，connectToNetwork 返回两个参数，conn 表示连接对象，err 返回错误信息。

#### 其它编程语言中函数的返回值

* C/[C++](http://c.biancheng.net/cplus/" \t "http://c.biancheng.net/view/_blank) 语言中只支持一个返回值，在需要返回多个数值时，则需要使用结构体返回结果，或者在参数中使用指针变量，然后在函数内部修改外部传入的变量值，实现返回计算结果，C++ 语言中为了安全性，建议在参数返回数据时使用“引用”替代指针。
* [C#](http://c.biancheng.net/csharp/" \t "http://c.biancheng.net/view/_blank) 语言也没有多返回值特性，C# 语言后期加入的 ref 和 out 关键字能够通过函数的调用参数获得函数体中修改的数据。
* lua 语言没有指针，但支持多返回值，在大块数据使用时方便很多。

Go语言既支持安全指针，也支持多返回值，因此在使用函数进行逻辑编写时更为方便。

#### 1) 同一种类型返回值

如果返回值是同一种类型，则用括号将多个返回值类型括起来，用逗号分隔每个返回值的类型。  
  
使用 return 语句返回时，值列表的顺序需要与函数声明的返回值类型一致，示例代码如下：

1. func typedTwoValues() (int, int) {
2. return 1, 2
3. }
4. func main() {
5. a, b := typedTwoValues()
6. fmt.Println(a, b)
7. }

代码输出结果：

1 2

纯类型的返回值对于代码可读性不是很友好，特别是在同类型的返回值出现时，无法区分每个返回参数的意义。

#### 2) 带有变量名的返回值

Go语言支持对返回值进行命名，这样返回值就和参数一样拥有参数变量名和类型。  
  
命名的返回值变量的默认值为类型的默认值，即数值为 0，字符串为空字符串，布尔为 false、指针为 nil 等。  
  
下面代码中的函数拥有两个整型返回值，函数声明时将返回值命名为 a 和 b，因此可以在函数体中直接对函数返回值进行赋值，在命名的返回值方式的函数体中，在函数结束前需要显式地使用 return 语句进行返回，代码如下：

1. func namedRetValues() (a, b int) {
2. a = 1
3. b = 2
4. return
5. }

代码说明如下：

* 第 1 行，对两个整型返回值进行命名，分别为 a 和 b。
* 第 3 行和第 4 行，命名返回值的变量与这个函数的布局变量的效果一致，可以对返回值进行赋值和值获取。
* 第 6 行，当函数使用命名返回值时，可以在 return 中不填写返回值列表，如果填写也是可行的，下面代码的执行效果和上面代码的效果一样。
  1. func namedRetValues() (a, b int) {
  2. a = 1
  3. return a, 2
  4. }

#### 提示

同一种类型返回值和命名返回值两种形式只能二选一，混用时将会发生编译错误，例如下面的代码：

1. func namedRetValues() (a, b int, int)

编译报错提示：

mixed named and unnamed function parameters

意思是：在函数参数中混合使用了命名和非命名参数。

## 调用函数

函数在定义后，可以通过调用的方式，让当前代码跳转到被调用的函数中进行执行，调用前的函数局部变量都会被保存起来不会丢失，被调用的函数运行结束后，恢复到调用函数的下一行继续执行代码，之前的局部变量也能继续访问。  
  
函数内的局部变量只能在函数体中使用，函数调用结束后，这些局部变量都会被释放并且失效。  
  
Go语言的函数调用格式如下：

返回值变量列表 = 函数名(参数列表)

下面是对各个部分的说明：

* 函数名：需要调用的函数名。
* 参数列表：参数变量以逗号分隔，尾部无须以分号结尾。
* 返回值变量列表：多个返回值使用逗号分隔。

例如，加法函数调用样式如下：

result := add(1,1)

# Go语言函数声明（函数定义）

< [Go语言函数](http://c.biancheng.net/golang/func/)[示例：将秒转换为具体的时间](http://c.biancheng.net/view/vip_7315.html) >

[C语言中文网推出辅导班啦，包括「C语言辅导班、C++辅导班、算法/数据结构辅导班」，全部都是一对一教学：一对一辅导 + 一对一答疑 + 布置作业 + 项目实践 + 永久学习。QQ在线，随时响应！](http://fudao.biancheng.net/" \t "http://c.biancheng.net/view/_blank)

函数构成了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，每一个程序都包含很多的函数，函数是基本的代码块。  
  
因为Go语言是编译型语言，所以函数编写的顺序是无关紧要的，鉴于可读性的需求，最好把 main() 函数写在文件的前面，其他函数按照一定逻辑顺序进行编写（例如函数被调用的顺序）。  
  
编写多个函数的主要目的是将一个需要很多行代码的复杂问题分解为一系列简单的任务来解决，而且，同一个任务（函数）可以被多次调用，有助于代码重用（事实上，好的程序是非常注意 DRY 原则的，即不要重复你自己（Don't Repeat Yourself），意思是执行特定任务的代码只能在程序里面出现一次）。  
  
当函数执行到代码块最后一行}之前或者 return 语句的时候会退出，其中 return 语句可以带有零个或多个参数，这些参数将作为返回值供调用者使用，简单的 return 语句也可以用来结束 for 的死循环，或者结束一个协程（goroutine）。  
  
Go语言里面拥三种类型的函数：

* 普通的带有名字的函数
* 匿名函数或者 lambda 函数
* 方法

## 普通函数声明（定义）

函数声明包括函数名、形式参数列表、返回值列表（可省略）以及函数体。

func 函数名(形式参数列表)(返回值列表){  
    函数体  
}

形式参数列表描述了函数的参数名以及参数类型，这些参数作为局部变量，其值由参数调用者提供，返回值列表描述了函数返回值的变量名以及类型，如果函数返回一个无名变量或者没有返回值，返回值列表的括号是可以省略的。  
  
如果一个函数声明不包括返回值列表，那么函数体执行完毕后，不会返回任何值，在下面的 hypot 函数中：

1. func hypot(x, y float64) float64 {
2. return math.Sqrt(x\*x + y\*y)
3. }
4. fmt.Println(hypot(3,4)) // "5"

x 和 y 是形参名，3 和 4 是调用时的传入的实数，函数返回了一个 float64 类型的值，返回值也可以像形式参数一样被命名，在这种情况下，每个返回值被声明成一个局部变量，并根据该返回值的类型，将其初始化为 0。  
  
如果一个函数在声明时，包含返回值列表，那么该函数必须以 return 语句结尾，除非函数明显无法运行到结尾处，例如函数在结尾时调用了 panic 异常或函数中存在无限循环。  
  
正如 hypot 函数一样，如果一组形参或返回值有相同的类型，我们不必为每个形参都写出参数类型，下面 2 个声明是等价的：

1. func f(i, j, k int, s, t string) { /\* ... \*/ }
2. func f(i int, j int, k int, s string, t string) { /\* ... \*/ }

下面，我们给出 4 种方法声明拥有 2 个 int 型参数和 1 个 int 型返回值的函数，空白标识符\_可以强调某个参数未被使用。

1. func add(x int, y int) int {return x + y}
2. func sub(x, y int) (z int) { z = x - y; return}
3. func first(x int, \_ int) int { return x }
4. func zero(int, int) int { return 0 }
5. fmt.Printf("%T\n", add) // "func(int, int) int"
6. fmt.Printf("%T\n", sub) // "func(int, int) int"
7. fmt.Printf("%T\n", first) // "func(int, int) int"
8. fmt.Printf("%T\n", zero) // "func(int, int) int"

函数的类型被称为函数的标识符，如果两个函数形式参数列表和返回值列表中的变量类型一一对应，那么这两个函数被认为有相同的类型和标识符，形参和返回值的变量名不影响函数标识符也不影响它们是否可以以省略参数类型的形式表示。  
  
每一次函数在调用时都必须按照声明顺序为所有参数提供实参（参数值），在函数调用时，Go语言没有默认参数值，也没有任何方法可以通过参数名指定形参，因此形参和返回值的变量名对于函数调用者而言没有意义。  
  
在函数中，实参通过值传递的方式进行传递，因此函数的形参是实参的拷贝，对形参进行修改不会影响实参，但是，如果实参包括引用类型，如指针、slice(切片)、map、function、channel 等类型，实参可能会由于函数的间接引用被修改。

## 函数的返回值

Go语言支持多返回值，多返回值能方便地获得函数执行后的多个返回参数，Go语言经常使用多返回值中的最后一个返回参数返回函数执行中可能发生的错误，示例代码如下：

1. conn, err := connectToNetwork()

在这段代码中，connectToNetwork 返回两个参数，conn 表示连接对象，err 返回错误信息。

#### 其它编程语言中函数的返回值

* C/[C++](http://c.biancheng.net/cplus/" \t "http://c.biancheng.net/view/_blank) 语言中只支持一个返回值，在需要返回多个数值时，则需要使用结构体返回结果，或者在参数中使用指针变量，然后在函数内部修改外部传入的变量值，实现返回计算结果，C++ 语言中为了安全性，建议在参数返回数据时使用“引用”替代指针。
* [C#](http://c.biancheng.net/csharp/" \t "http://c.biancheng.net/view/_blank) 语言也没有多返回值特性，C# 语言后期加入的 ref 和 out 关键字能够通过函数的调用参数获得函数体中修改的数据。
* lua 语言没有指针，但支持多返回值，在大块数据使用时方便很多。

Go语言既支持安全指针，也支持多返回值，因此在使用函数进行逻辑编写时更为方便。

#### 1) 同一种类型返回值

如果返回值是同一种类型，则用括号将多个返回值类型括起来，用逗号分隔每个返回值的类型。  
  
使用 return 语句返回时，值列表的顺序需要与函数声明的返回值类型一致，示例代码如下：

1. func typedTwoValues() (int, int) {
2. return 1, 2
3. }
4. func main() {
5. a, b := typedTwoValues()
6. fmt.Println(a, b)
7. }

代码输出结果：

1 2

纯类型的返回值对于代码可读性不是很友好，特别是在同类型的返回值出现时，无法区分每个返回参数的意义。

#### 2) 带有变量名的返回值

Go语言支持对返回值进行命名，这样返回值就和参数一样拥有参数变量名和类型。  
  
命名的返回值变量的默认值为类型的默认值，即数值为 0，字符串为空字符串，布尔为 false、指针为 nil 等。  
  
下面代码中的函数拥有两个整型返回值，函数声明时将返回值命名为 a 和 b，因此可以在函数体中直接对函数返回值进行赋值，在命名的返回值方式的函数体中，在函数结束前需要显式地使用 return 语句进行返回，代码如下：

1. func namedRetValues() (a, b int) {
2. a = 1
3. b = 2
4. return
5. }

代码说明如下：

* 第 1 行，对两个整型返回值进行命名，分别为 a 和 b。
* 第 3 行和第 4 行，命名返回值的变量与这个函数的布局变量的效果一致，可以对返回值进行赋值和值获取。
* 第 6 行，当函数使用命名返回值时，可以在 return 中不填写返回值列表，如果填写也是可行的，下面代码的执行效果和上面代码的效果一样。
  1. func namedRetValues() (a, b int) {
  2. a = 1
  3. return a, 2
  4. }

#### 提示

同一种类型返回值和命名返回值两种形式只能二选一，混用时将会发生编译错误，例如下面的代码：

1. func namedRetValues() (a, b int, int)

编译报错提示：

mixed named and unnamed function parameters

意思是：在函数参数中混合使用了命名和非命名参数。

## 调用函数

函数在定义后，可以通过调用的方式，让当前代码跳转到被调用的函数中进行执行，调用前的函数局部变量都会被保存起来不会丢失，被调用的函数运行结束后，恢复到调用函数的下一行继续执行代码，之前的局部变量也能继续访问。  
  
函数内的局部变量只能在函数体中使用，函数调用结束后，这些局部变量都会被释放并且失效。  
  
Go语言的函数调用格式如下：

返回值变量列表 = 函数名(参数列表)

下面是对各个部分的说明：

* 函数名：需要调用的函数名。
* 参数列表：参数变量以逗号分隔，尾部无须以分号结尾。
* 返回值变量列表：多个返回值使用逗号分隔。

例如，加法函数调用样式如下：

result := add(1,1)

# Go语言函数中的参数传递效果测试

< [示例：将秒转换为具体的时间](http://c.biancheng.net/view/vip_7315.html)[Go语言函数变量](http://c.biancheng.net/view/55.html) >

[C语言中文网推出辅导班啦，包括「C语言辅导班、C++辅导班、算法/数据结构辅导班」，全部都是一对一教学：一对一辅导 + 一对一答疑 + 布置作业 + 项目实践 + 永久学习。QQ在线，随时响应！](http://fudao.biancheng.net/" \t "http://c.biancheng.net/view/_blank)

Go语言中传入与返回参数在调用和返回时都使用值传递，这里需要注意的是指针、切片和 map 等引用型对象在参数传递中不会发生复制，而是将指针进行复制，类似于创建一次引用。  
  
下面通过一个例子来详细了解Go语言的参数值传递，完整的示例代码如下所示：

1. package main
2. import "fmt"
3. // 用于测试值传递效果的结构体
4. type Data struct {
5. complax []int // 测试切片在参数传递中的效果
6. instance InnerData // 实例分配的innerData
7. ptr \*InnerData // 将ptr声明为InnerData的指针类型
8. }
9. // 代表各种结构体字段
10. type InnerData struct {
11. a int
12. }
13. // 值传递测试函数
14. func passByValue(inFunc Data) Data {
15. // 输出参数的成员情况
16. fmt.Printf("inFunc value: %+v\n", inFunc)
17. // 打印inFunc的指针
18. fmt.Printf("inFunc ptr: %p\n", &inFunc)
19. return inFunc
20. }
21. func main() {
22. // 准备传入函数的结构
23. in := Data{
24. complax: []int{1, 2, 3},
25. instance: InnerData{
26. 5,
27. },
28. ptr: &InnerData{1},
29. }
30. // 输入结构的成员情况
31. fmt.Printf("in value: %+v\n", in)
32. // 输入结构的指针地址
33. fmt.Printf("in ptr: %p\n", &in)
34. // 传入结构体，返回同类型的结构体
35. out := passByValue(in)
36. // 输出结构的成员情况
37. fmt.Printf("out value: %+v\n", out)
38. // 输出结构的指针地址
39. fmt.Printf("out ptr: %p\n", &out)
40. }

## 1) 测试数据类型

为了测试结构体、切片、指针及结构体中嵌套的结构体在值传递中会发生的情况，需要定义一些结构，代码如下：

1. // 用于测试值传递效果的结构体
2. type Data struct {
3. complax []int // 测试切片在参数传递中的效果
4. instance InnerData // 实例分配的innerData
5. ptr \*InnerData // 将ptr声明为InnerData的指针类型
6. }
7. // 代表各种结构体字段
8. type InnerData struct {
9. a int
10. }

代码说明如下：

* 第 2 行，将 Data 声明为结构体类型，结构体是拥有多个字段的复杂结构。
* 第 3 行，complax 为整型切片类型，切片是一种动态类型，内部以指针存在。
* 第 5 行，instance 成员以 InnerData 类型作为 Data 的成员。
* 第 7 行，将 ptr 声明为 InnerData 的指针类型。
* 第 11 行，声明一个内嵌的结构 InnerData。

## 2) 值传递的测试函数

示例代码中定义的 passByValue() 函数用于值传递的测试，该函数的参数和返回值都是 Data 类型，在调用过程中，Data 的内存会被复制后传入函数，当函数返回时，又会将返回值复制一次，赋给函数返回值的接收变量，代码如下：

1. // 值传递测试函数
2. func passByValue(inFunc Data) Data {
3. // 输出参数的成员情况
4. fmt.Printf("inFunc value: %+v\n", inFunc)
5. // 打印inFunc的指针
6. fmt.Printf("inFunc ptr: %p\n", &inFunc)
7. return inFunc
8. }

代码说明如下：

* 第 5 行，使用格式化的%+v动词输出 inFunc 变量的详细结构，以便观察 Data 结构在传递前后内部数值的变化情况。
* 第 8 行，打印传入参数 inFunc 的指针地址，在计算机中，拥有相同地址且类型相同的变量，表示的是同一块内存区域。
* 第 10 行，将传入的变量作为返回值返回，返回的过程将发生值复制。

## 3) 测试流程

测试流程会准备一个 Data 格式的数据结构并填充所有成员，这些成员类型包括切片、结构体成员及指针，通过调用测试函数，传入 Data 结构数据，并获得返回值，对比输入和输出后的 Data 结构数值变化，特别是指针变化情况以及输入和输出整块数据是否被复制，代码如下：

1. // 准备传入函数的结构
2. in := Data{
3. complax: []int{1, 2, 3},
4. instance: InnerData{
5. 5,
6. },
7. ptr: &InnerData{1},
8. }
9. // 输入结构的成员情况
10. fmt.Printf("in value: %+v\n", in)
11. // 输入结构的指针地址
12. fmt.Printf("in ptr: %p\n", &in)
13. // 传入结构体, 返回同类型的结构体
14. out := passByValue(in)
15. // 输出结构的成员情况
16. fmt.Printf("out value: %+v\n", out)
17. // 输出结构的指针地址
18. fmt.Printf("out ptr: %p\n", &out)

代码说明如下：

* 第 2 行，创建一个 Data 结构的实例 in。
* 第 3 行，将切片数据赋值到 in 的 complax 成员。
* 第 4 行，为 in 的 instance 成员赋值 InnerData 结构的数据。
* 第 8 行，为 in 的 ptr 成员赋值 InnerData 的指针类型数据。
* 第 12 行，打印输入结构的成员情况。
* 第 15 行，打印输入结构的指针地址。
* 第 18 行，传入 in 结构，调用 passByvalue() 测试函数获得 out 返回，此时，passByValue() 函数会打印 in 传入后的数据成员情况。
* 第 21 行，打印返回值变量 out 的成员情况。
* 第 24 行，打印输出结构的地址。

运行代码，输出结果为：

in value: {complax:[1 2 3] instance:{a:5} ptr:0xc042008100}  
in ptr: 0xc042066060  
inFunc value: {complax:[1 2 3] instance:{a:5} ptr:0xc042008100}  
inFunc ptr: 0xc0420660f0  
out value: {complax:[1 2 3] instance:{a:5} ptr:0xc042008100}  
out ptr: 0xc0420660c0

从运行结果中发现：

* 所有的 Data 结构的指针地址都发生了变化，意味着所有的结构都是一块新的内存，无论是将 Data 结构传入函数内部，还是通过函数返回值传回 Data 都会发生复制行为。
* 所有的 Data 结构中的成员值都没有发生变化，原样传递，意味着所有参数都是值传递。
* Data 结构的 ptr 成员在传递过程中保持一致，表示指针在函数参数值传递中传递的只是指针值，不会复制指针指向的部分。

# Go语言函数变量——把函数作为值保存到变量中

< [示例：函数中的参数传递效果测试](http://c.biancheng.net/view/vip_7316.html)[Go语言字符串的链式处理](http://c.biancheng.net/view/vip_7317.html) >

[C语言中文网推出辅导班啦，包括「C语言辅导班、C++辅导班、算法/数据结构辅导班」，全部都是一对一教学：一对一辅导 + 一对一答疑 + 布置作业 + 项目实践 + 永久学习。QQ在线，随时响应！](http://fudao.biancheng.net/" \t "http://c.biancheng.net/view/_blank)

在Go语言中，函数也是一种类型，可以和其他类型一样保存在变量中，下面的代码定义了一个函数变量 f，并将一个函数名为 fire() 的函数赋给函数变量 f，这样调用函数变量 f 时，实际调用的就是 fire() 函数，代码如下：

1. package main
2. import (
3. "fmt"
4. )
5. func fire() {
6. fmt.Println("fire")
7. }
8. func main() {
9. var f func()
10. f = fire
11. f()
12. }

代码输出结果：

fire

代码说明：

* 第 7 行，定义了一个 fire() 函数。
* 第 13 行，将变量 f 声明为 func() 类型，此时 f 就被俗称为“回调函数”，此时 f 的值为 nil。
* 第 15 行，将 fire() 函数作为值，赋给函数变量 f，此时 f 的值为 fire() 函数。
* 第 17 行，使用函数变量 f 进行函数调用，实际调用的是 fire() 函数。

# Go语言字符串的链式处理——操作与数据分离的设计技巧

< [Go语言函数变量](http://c.biancheng.net/view/55.html)[Go语言匿名函数](http://c.biancheng.net/view/57.html) >

[C语言中文网推出辅导班啦，包括「C语言辅导班、C++辅导班、算法/数据结构辅导班」，全部都是一对一教学：一对一辅导 + 一对一答疑 + 布置作业 + 项目实践 + 永久学习。QQ在线，随时响应！](http://fudao.biancheng.net/" \t "http://c.biancheng.net/view/_blank)

使用 SQL 语言从数据库中获取数据时，可以对原始数据进行排序（sort by）、分组（group by）和去重（distinct）等操作，SQL 将数据的操作与遍历过程作为两个部分进行隔离，这样操作和遍历过程就可以各自独立地进行设计，这就是常见的数据与操作分离的设计。  
  
对数据的操作进行多步骤的处理被称为链式处理，本例中使用多个字符串作为数据集合，然后对每个字符串进行一系列的处理，用户可以通过系统函数或者自定义函数对链式处理中的每个环节进行自定义。  
  
首先给出本节完整代码：

1. package main
2. import (
3. "fmt"
4. "strings"
5. )
6. // 字符串处理函数，传入字符串切片和处理链
7. func StringProccess(list []string, chain []func(string) string) {
8. // 遍历每一个字符串
9. for index, str := range list {
10. // 第一个需要处理的字符串
11. result := str
12. // 遍历每一个处理链
13. for \_, proc := range chain {
14. // 输入一个字符串进行处理，返回数据作为下一个处理链的输入。
15. result = proc(result)
16. }
17. // 将结果放回切片
18. list[index] = result
19. }
20. }
21. // 自定义的移除前缀的处理函数
22. func removePrefix(str string) string {
23. return strings.TrimPrefix(str, "go")
24. }
25. func main() {
26. // 待处理的字符串列表
27. list := []string{
28. "go scanner",
29. "go parser",
30. "go compiler",
31. "go printer",
32. "go formater",
33. }
34. // 处理函数链
35. chain := []func(string) string{
36. removePrefix,
37. strings.TrimSpace,
38. strings.ToUpper,
39. }
40. // 处理字符串
41. StringProccess(list, chain)
42. // 输出处理好的字符串
43. for \_, str := range list {
44. fmt.Println(str)
45. }
46. }

## 1) 字符串处理函数

字符串处理函数（StringProccess）需要外部提供数据源，一个字符串切片（list[]string），另外还要提供一个链式处理函数的切片（chain[]func(string)string），链式处理切片中的一个处理函数的定义如下：

func(string)string

这种处理函数能够接受一个字符串输入，处理后输出。  
  
strings 包中将字符串变为小写就是一种处理函数的形式，strings.ToLower() 函数能够将传入的字符串的每一个字符变为小写，strings.ToLower 定义如下：

func ToLower(s string) string

字符串处理函数（StringProccess）内部遍历每一个数据源提供的字符串，每个字符串都需要经过一系列链式处理函数处理后被重新放回切片，参见下面代码。  
  
字符串的链式处理：

1. // 字符串处理函数, 传入字符串切片和处理链
2. func StringProccess(list []string, chain []func(string) string) {
3. // 遍历每一个字符串
4. for index, str := range list {
5. // 第一个需要处理的字符串
6. result := str
7. // 遍历每一个处理链
8. for \_, proc := range chain {
9. // 输入一个字符串进行处理, 返回数据作为下一个处理链的输入
10. result = proc(result)
11. }
12. // 将结果放回切片
13. list[index] = result
14. }
15. }

代码说明如下：

* 第 2 行，传入字符串切片 list 作为数据源，一系列的处理函数作为 chain 处理链。
* 第 5 行，遍历字符串切片的每个字符串，依次对每个字符串进行处理。
* 第 8 行，将当前字符串保存到 result 变量中，作为第一个处理函数的参数。
* 第 11 行，遍历每一个处理函数，将字符串按顺序经过这些处理函数处理。
* 第 14 行，result 变量即是每个处理函数的输入变量，处理后的变量又会重新保存到 result 变量中。
* 第 18 行，将处理完的字符串保存回切片中。

## 2) 自定义的处理函数

处理函数可以是系统提供的处理函数，如将字符串变大写或小写，也可以使用自定义函数，本例中的字符串处理的逻辑是使用一个自定义的函数实现移除指定 go 前缀的过程，参见下面代码：

1. // 自定义的移除前缀的处理函数
2. func removePrefix(str string) string {
3. return strings.TrimPrefix(str, "go")
4. }

此函数使用了 strings.TrimPrefix() 函数实现移除字符串的指定前缀，处理后，移除前缀的字符串结果将通过 removePrefix() 函数的返回值返回。

## 3) 字符串处理主流程

字符串处理的主流程包含以下几个步骤：

1. 准备要处理的字符串列表。
2. 准备字符串处理链。
3. 处理字符串列表。
4. 打印输出后的字符串列表。

详细流程参考下面的代码：

1. func main() {
2. // 待处理的字符串列表
3. list := []string{
4. "go scanner",
5. "go parser",
6. "go compiler",
7. "go printer",
8. "go formater",
9. }
10. // 处理函数链
11. chain := []func(string) string{
12. removePrefix,
13. strings.TrimSpace,
14. strings.ToUpper,
15. }
16. // 处理字符串
17. StringProccess(list, chain)
18. // 输出处理好的字符串
19. for \_, str := range list {
20. fmt.Println(str)
21. }
22. }

代码说明如下：

* 第 4 行，定义字符串切片，字符串包含 go 前缀及空格。
* 第 13 行，准备处理每个字符串的处理链，处理的顺序与函数在切片中的位置一致，removePrefix() 为自定义的函数，功能是移除 go 前缀，移除前缀的字符串左边有一个空格，使用 strings.TrimSpace 移除，这个函数的定义刚好符合处理函数的格式 func(string)string，strings.ToUpper 用于将字符串转为大写。
* 第 20 行，传入字符串切片和字符串处理链，通过 StringProcess() 函数对字符串进行处理。
* 第 23 行，遍历字符串切片的每一个字符串，打印处理好的字符串结果。

#### 提示

链式处理器是一种常见的编程设计，Netty 是使用 Java 语言编写的一款异步事件驱动的网络应用程序框架，支持快速开发可维护的高性能的面向协议的服务器和客户端，Netty 中就有类似的链式处理器的设计。  
  
Netty 可以使用类似的处理链对封包进行收发编码及处理，Netty 的开发者可以分为 3 种：第一种是 Netty 底层开发者；第二种是每个处理环节的开发者；第三种是业务实现者。在实际开发环节中，后两种开发者往往是同一批开发者，链式处理的开发思想将数据和操作拆分、解耦，让开发者可以根据自己的技术优势和需求，进行系统开发，同时将自己的开发成果共享给其他的开发者。

# Go语言匿名函数——没有函数名字的函数

< [Go语言字符串的链式处理](http://c.biancheng.net/view/vip_7317.html)[Go语言函数类型实现接口](http://c.biancheng.net/view/58.html) >

[C语言中文网推出辅导班啦，包括「C语言辅导班、C++辅导班、算法/数据结构辅导班」，全部都是一对一教学：一对一辅导 + 一对一答疑 + 布置作业 + 项目实践 + 永久学习。QQ在线，随时响应！](http://fudao.biancheng.net/" \t "http://c.biancheng.net/view/_blank)

Go语言支持匿名函数，即在需要使用函数时再定义函数，匿名函数没有函数名只有函数体，函数可以作为一种类型被赋值给函数类型的变量，匿名函数也往往以变量方式传递，这与C语言的回调函数比较类似，不同的是，Go语言支持随时在代码里定义匿名函数。  
  
匿名函数是指不需要定义函数名的一种函数实现方式，由一个不带函数名的函数声明和函数体组成，下面来具体介绍一下匿名函数的定义及使用。

## 定义一个匿名函数

匿名函数的定义格式如下：

func(参数列表)(返回参数列表){  
    函数体  
}

匿名函数的定义就是没有名字的普通函数定义。

#### 1) 在定义时调用匿名函数

匿名函数可以在声明后调用，例如：

1. func(data int) {
2. fmt.Println("hello", data)
3. }(100)

注意第3行}后的(100)，表示对匿名函数进行调用，传递参数为 100。

#### 2) 将匿名函数赋值给变量

匿名函数可以被赋值，例如：

1. // 将匿名函数体保存到f()中
2. f := func(data int) {
3. fmt.Println("hello", data)
4. }
5. // 使用f()调用
6. f(100)

匿名函数的用途非常广泛，它本身就是一种值，可以方便地保存在各种容器中实现回调函数和操作封装。

## 匿名函数用作回调函数

下面的代码实现对切片的遍历操作，遍历中访问每个元素的操作使用匿名函数来实现，用户传入不同的匿名函数体可以实现对元素不同的遍历操作，代码如下：

1. package main
2. import (
3. "fmt"
4. )
5. // 遍历切片的每个元素, 通过给定函数进行元素访问
6. func visit(list []int, f func(int)) {
7. for \_, v := range list {
8. f(v)
9. }
10. }
11. func main() {
12. // 使用匿名函数打印切片内容
13. visit([]int{1, 2, 3, 4}, func(v int) {
14. fmt.Println(v)
15. })
16. }

代码说明如下：

* 第 8 行，使用 visit() 函数将整个遍历过程进行封装，当要获取遍历期间的切片值时，只需要给 visit() 传入一个回调参数即可。
* 第 18 行，准备一个整型切片 []int{1,2,3,4} 传入 visit() 函数作为遍历的数据。
* 第 19～20 行，定义了一个匿名函数，作用是将遍历的每个值打印出来。

匿名函数作为回调函数的设计在Go语言的系统包中也比较常见，strings 包中就有类似的设计，代码如下：

1. func TrimFunc(s string, f func(rune) bool) string {
2. return TrimRightFunc(TrimLeftFunc(s, f), f)
3. }

## 使用匿名函数实现操作封装

下面这段代码将匿名函数作为 map 的键值，通过命令行参数动态调用匿名函数，代码如下：

1. package main
2. import (
3. "flag"
4. "fmt"
5. )
6. var skillParam = flag.String("skill", "", "skill to perform")
7. func main() {
8. flag.Parse()
9. var skill = map[string]func(){
10. "fire": func() {
11. fmt.Println("chicken fire")
12. },
13. "run": func() {
14. fmt.Println("soldier run")
15. },
16. "fly": func() {
17. fmt.Println("angel fly")
18. },
19. }
20. if f, ok := skill[\*skillParam]; ok {
21. f()
22. } else {
23. fmt.Println("skill not found")
24. }
25. }

代码说明如下：

* 第 8 行，定义命令行参数 skill，从命令行输入 --skill 可以将=后的字符串传入 skillParam 指针变量。
* 第 12 行，解析命令行参数，解析完成后，skillParam 指针变量将指向命令行传入的值。
* 第 14 行，定义一个从字符串映射到 func() 的 map，然后填充这个 map。
* 第 15～23 行，初始化 map 的键值对，值为匿名函数。
* 第 26 行，skillParam 是一个 \*string 类型的指针变量，使用 \*skillParam 获取到命令行传过来的值，并在 map 中查找对应命令行参数指定的字符串的函数。
* 第 29 行，如果在 map 定义中存在这个参数就调用，否则打印“技能没有找到”。

运行代码，结果如下：

PS D:\code> go run main.go --skill=fly

angel fly

PS D:\code> go run main.go --skill=run

soldier run

# Go语言函数类型实现接口——把函数作为接口来调用

< [Go语言匿名函数](http://c.biancheng.net/view/57.html)[Go语言闭包（Closure）](http://c.biancheng.net/view/59.html) >

[C语言中文网推出辅导班啦，包括「C语言辅导班、C++辅导班、算法/数据结构辅导班」，全部都是一对一教学：一对一辅导 + 一对一答疑 + 布置作业 + 项目实践 + 永久学习。QQ在线，随时响应！](http://fudao.biancheng.net/" \t "http://c.biancheng.net/view/_blank)

函数和其他类型一样都属于“一等公民”，其他类型能够实现接口，函数也可以，本节将对结构体与函数实现接口的过程进行对比。  
  
首先给出本节完整的代码：

1. package main
2. import (
3. "fmt"
4. )
5. // 调用器接口
6. type Invoker interface {
7. // 需要实现一个Call方法
8. Call(interface{})
9. }
10. // 结构体类型
11. type Struct struct {
12. }
13. // 实现Invoker的Call
14. func (s \*Struct) Call(p interface{}) {
15. fmt.Println("from struct", p)
16. }
17. // 函数定义为类型
18. type FuncCaller func(interface{})
19. // 实现Invoker的Call
20. func (f FuncCaller) Call(p interface{}) {
21. // 调用f函数本体
22. f(p)
23. }
24. func main() {
25. // 声明接口变量
26. var invoker Invoker
27. // 实例化结构体
28. s := new(Struct)
29. // 将实例化的结构体赋值到接口
30. invoker = s
31. // 使用接口调用实例化结构体的方法Struct.Call
32. invoker.Call("hello")
33. // 将匿名函数转为FuncCaller类型，再赋值给接口
34. invoker = FuncCaller(func(v interface{}) {
35. fmt.Println("from function", v)
36. })
37. // 使用接口调用FuncCaller.Call，内部会调用函数本体
38. invoker.Call("hello")
39. }

有如下一个接口：

1. // 调用器接口
2. type Invoker interface {
3. // 需要实现一个Call()方法
4. Call(interface{})
5. }

这个接口需要实现 Call() 方法，调用时会传入一个 interface{} 类型的变量，这种类型的变量表示任意类型的值。  
  
接下来，使用结构体进行接口实现。

## 结构体实现接口

结构体实现 Invoker 接口的代码如下：

1. // 结构体类型
2. type Struct struct {
3. }
4. // 实现Invoker的Call
5. func (s \*Struct) Call(p interface{}) {
6. fmt.Println("from struct", p)
7. }

代码说明如下：

* 第 2 行，定义结构体，该例子中的结构体无须任何成员，主要展示实现 Invoker 的方法。
* 第 6 行，Call() 为结构体的方法，该方法的功能是打印 from struct 和传入的 interface{} 类型的值。

将定义的 Struct 类型实例化，并传入接口中进行调用，代码如下：

1. // 声明接口变量
2. var invoker Invoker
3. // 实例化结构体
4. s := new(Struct)
5. // 将实例化的结构体赋值到接口
6. invoker = s
7. // 使用接口调用实例化结构体的方法Struct.Call
8. invoker.Call("hello")

代码说明如下：

* 第 2 行，声明 Invoker 类型的变量。
* 第 5 行，使用 new 将结构体实例化，此行也可以写为 s:=&Struct。
* 第 8 行，s 类型为 \*Struct，已经实现了 Invoker 接口类型，因此赋值给 invoker 时是成功的。
* 第 11 行，通过接口的 Call() 方法，传入 hello，此时将调用 Struct 结构体的 Call() 方法。

接下来，对比下函数实现结构体的差异。  
  
代码输出如下：

from struct hello

## 函数体实现接口

函数的声明不能直接实现接口，需要将函数定义为类型后，使用类型实现结构体，当类型方法被调用时，还需要调用函数本体。

1. // 函数定义为类型
2. type FuncCaller func(interface{})
3. // 实现Invoker的Call
4. func (f FuncCaller) Call(p interface{}) {
5. // 调用f()函数本体
6. f(p)
7. }

代码说明如下：

* 第 2 行，将 func(interface{}) 定义为 FuncCaller 类型。
* 第 5 行，FuncCaller 的 Call() 方法将实现 Invoker 的 Call() 方法。
* 第 8 行，FuncCaller 的 Call() 方法被调用与 func(interface{}) 无关，还需要手动调用函数本体。

上面代码只是定义了函数类型，需要函数本身进行逻辑处理，FuncCaller 无须被实例化，只需要将函数转换为 FuncCaller 类型即可，函数来源可以是命名函数、匿名函数或闭包，参见下面代码：

1. // 声明接口变量
2. var invoker Invoker
3. // 将匿名函数转为FuncCaller类型, 再赋值给接口
4. invoker = FuncCaller(func(v interface{}) {
5. fmt.Println("from function", v)
6. })
7. // 使用接口调用FuncCaller.Call, 内部会调用函数本体
8. invoker.Call("hello")

代码说明如下：

* 第 2 行，声明接口变量。
* 第 5 行，将 func(v interface{}){} 匿名函数转换为 FuncCaller 类型（函数签名才能转换），此时 FuncCaller 类型实现了 Invoker 的 Call() 方法，赋值给 invoker 接口是成功的。
* 第 10 行，使用接口方法调用。

代码输出如下：

from function hello

## HTTP包中的例子

HTTP 包中包含有 Handler 接口定义，代码如下：

1. type Handler interface {
2. ServeHTTP(ResponseWriter, \*Request)
3. }

Handler 用于定义每个 HTTP 的请求和响应的处理过程。  
  
同时，也可以使用处理函数实现接口，定义如下：

1. type HandlerFunc func(ResponseWriter, \*Request)
2. func (f HandlerFunc) ServeHTTP(w ResponseWriter, r \*Request) {
3. f(w, r)
4. }

要使用闭包实现默认的 HTTP 请求处理，可以使用 http.HandleFunc() 函数，函数定义如下：

1. func HandleFunc(pattern string, handler func(ResponseWriter, \*Request)) {
2. DefaultServeMux.HandleFunc(pattern, handler)
3. }

而 DefaultServeMux 是 ServeMux 结构，拥有 HandleFunc() 方法，定义如下：

1. func (mux \*ServeMux) HandleFunc(pattern string, handler func
2. (ResponseWriter, \*Request)) {
3. mux.Handle(pattern, HandlerFunc(handler))
4. }

上面代码将外部传入的函数 handler() 转为 HandlerFunc 类型，HandlerFunc 类型实现了 Handler 的 ServeHTTP 方法，底层可以同时使用各种类型来实现 Handler 接口进行处理。

# Go语言闭包（Closure）——引用了外部变量的匿名函数

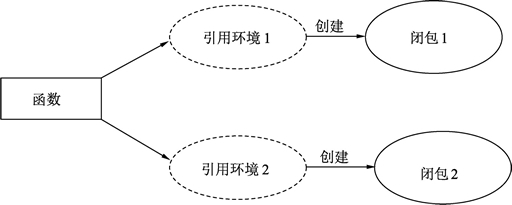
< [Go语言函数类型实现接口](http://c.biancheng.net/view/58.html)[Go语言可变参数](http://c.biancheng.net/view/60.html) >

[C语言中文网推出辅导班啦，包括「C语言辅导班、C++辅导班、算法/数据结构辅导班」，全部都是一对一教学：一对一辅导 + 一对一答疑 + 布置作业 + 项目实践 + 永久学习。QQ在线，随时响应！](http://fudao.biancheng.net/" \t "http://c.biancheng.net/view/_blank)

Go语言中闭包是引用了自由变量的函数，被引用的自由变量和函数一同存在，即使已经离开了自由变量的环境也不会被释放或者删除，在闭包中可以继续使用这个自由变量，因此，简单的说：

函数 + 引用环境 = 闭包

同一个函数与不同引用环境组合，可以形成不同的实例，如下图所示。

  
图：闭包与函数引用

一个函数类型就像结构体一样，可以被实例化，函数本身不存储任何信息，只有与引用环境结合后形成的闭包才具有“记忆性”，函数是编译期静态的概念，而闭包是运行期动态的概念。

#### 其它编程语言中的闭包

闭包（Closure）在某些编程语言中也被称为 Lambda 表达式。  
  
闭包对环境中变量的引用过程也可以被称为“捕获”，在 [C++](http://c.biancheng.net/cplus/" \t "http://c.biancheng.net/view/_blank)11 标准中，捕获有两种类型，分别是引用和复制，可以改变引用的原值叫做“引用捕获”，捕获的过程值被复制到闭包中使用叫做“复制捕获”。  
  
在 Lua 语言中，将被捕获的变量起了一个名字叫做 Upvalue，因为捕获过程总是对闭包上方定义过的自由变量进行引用。  
  
闭包在各种语言中的实现也是不尽相同的，在 Lua 语言中，无论闭包还是函数都属于 Prototype 概念，被捕获的变量以 Upvalue 的形式引用到闭包中。  
  
C++ 与 [C#](http://c.biancheng.net/csharp/" \t "http://c.biancheng.net/view/_blank) 中为闭包创建了一个类，而被捕获的变量在编译时放到类中的成员中，闭包在访问被捕获的变量时，实际上访问的是闭包隐藏类的成员。

## 在闭包内部修改引用的变量

闭包对它作用域上部的变量可以进行修改，修改引用的变量会对变量进行实际修改，通过下面的例子来理解：

1. // 准备一个字符串
2. str := "hello world"
3. // 创建一个匿名函数
4. foo := func() {
6. // 匿名函数中访问str
7. str = "hello dude"
8. }
9. // 调用匿名函数
10. foo()

代码说明如下：

* 第 2 行，准备一个字符串用于修改。
* 第 5 行，创建一个匿名函数。
* 第 8 行，在匿名函数中并没有定义 str，str 的定义在匿名函数之前，此时，str 就被引用到了匿名函数中形成了闭包。
* 第 12 行，执行闭包，此时 str 发生修改，变为 hello dude。

代码输出：

hello dude

## 示例：闭包的记忆效应

被捕获到闭包中的变量让闭包本身拥有了记忆效应，闭包中的逻辑可以修改闭包捕获的变量，变量会跟随闭包生命期一直存在，闭包本身就如同变量一样拥有了记忆效应。  
  
累加器的实现：

1. package main
2. import (
3. "fmt"
4. )
5. // 提供一个值, 每次调用函数会指定对值进行累加
6. func Accumulate(value int) func() int {
7. // 返回一个闭包
8. return func() int {
9. // 累加
10. value++
11. // 返回一个累加值
12. return value
13. }
14. }
15. func main() {
16. // 创建一个累加器, 初始值为1
17. accumulator := Accumulate(1)
18. // 累加1并打印
19. fmt.Println(accumulator())
20. fmt.Println(accumulator())
21. // 打印累加器的函数地址
22. fmt.Printf("%p\n", &accumulator)
23. // 创建一个累加器, 初始值为1
24. accumulator2 := Accumulate(10)
25. // 累加1并打印
26. fmt.Println(accumulator2())
27. // 打印累加器的函数地址
28. fmt.Printf("%p\n", &accumulator2)
29. }

代码说明如下：

* 第 8 行，累加器生成函数，这个函数输出一个初始值，调用时返回一个为初始值创建的闭包函数。
* 第 11 行，返回一个闭包函数，每次返回会创建一个新的函数实例。
* 第 14 行，对引用的 Accumulate 参数变量进行累加，注意 value 不是第 11 行匿名函数定义的，但是被这个匿名函数引用，所以形成闭包。
* 第 17 行，将修改后的值通过闭包的返回值返回。
* 第 24 行，创建一个累加器，初始值为 1，返回的 accumulator 是类型为 func()int 的函数变量。
* 第 27 行，调用 accumulator() 时，代码从 11 行开始执行匿名函数逻辑，直到第 17 行返回。
* 第 32 行，打印累加器的函数地址。

对比输出的日志发现 accumulator 与 accumulator2 输出的函数地址不同，因此它们是两个不同的闭包实例。  
  
每调用一次 accumulator 都会自动对引用的变量进行累加。

## 示例：闭包实现生成器

闭包的记忆效应被用于实现类似于[设计模式](http://c.biancheng.net/design_pattern/" \t "http://c.biancheng.net/view/_blank)中工厂模式的生成器，下面的例子展示了创建一个玩家生成器的过程。  
  
玩家生成器的实现：

1. package main
2. import (
3. "fmt"
4. )
5. // 创建一个玩家生成器, 输入名称, 输出生成器
6. func playerGen(name string) func() (string, int) {
7. // 血量一直为150
8. hp := 150
9. // 返回创建的闭包
10. return func() (string, int) {
11. // 将变量引用到闭包中
12. return name, hp
13. }
14. }
15. func main() {
16. // 创建一个玩家生成器
17. generator := playerGen("high noon")
18. // 返回玩家的名字和血量
19. name, hp := generator()
20. // 打印值
21. fmt.Println(name, hp)
22. }

代码输出如下：

high noon 150

代码说明如下：

* 第 8 行，playerGen() 需要提供一个名字来创建一个玩家的生成函数。
* 第 11 行，声明并设定 hp 变量为 150。
* 第 14～18 行，将 hp 和 name 变量引用到匿名函数中形成闭包。
* 第 24 行中，通过 playerGen 传入参数调用后获得玩家生成器。
* 第 27 行，调用这个玩家生成器函数，可以获得玩家的名称和血量。

闭包还具有一定的封装性，第 11 行的变量是 playerGen 的局部变量，playerGen 的外部无法直接访问及修改这个变量，这种特性也与面向对象中强调的封装性类似。

# Go语言可变参数（变参函数）

< [Go语言闭包（Closure）](http://c.biancheng.net/view/59.html)[Go语言defer（延迟执行语句）](http://c.biancheng.net/view/61.html) >

[C语言中文网推出辅导班啦，包括「C语言辅导班、C++辅导班、算法/数据结构辅导班」，全部都是一对一教学：一对一辅导 + 一对一答疑 + 布置作业 + 项目实践 + 永久学习。QQ在线，随时响应！](http://fudao.biancheng.net/" \t "http://c.biancheng.net/view/_blank)

在C语言时代大家一般都用过 printf() 函数，从那个时候开始其实已经在感受可变参数的魅力和价值，如同C语言中的 printf() 函数，Go语言标准库中的 fmt.Println() 等函数的实现也依赖于语言的可变参数功能。  
  
本节我们将介绍可变参数的用法。合适地使用可变参数，可以让代码简单易用，尤其是输入输出类函数，比如日志函数等。

## 可变参数类型

可变参数是指函数传入的参数个数是可变的，为了做到这点，首先需要将函数定义为可以接受可变参数的类型：

1. func myfunc(args ...int) {
2. for \_, arg := range args {
3. fmt.Println(arg)
4. }
5. }

上面这段代码的意思是，函数 myfunc() 接受不定数量的参数，这些参数的类型全部是 int，所以它可以用如下方式调用：

myfunc(2, 3, 4)  
myfunc(1, 3, 7, 13)

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，它是一个语法糖（syntactic sugar），即这种语法对语言的功能并没有影响，但是更方便程序员使用，通常来说，使用语法糖能够增加程序的可读性，从而减少程序出错的可能。  
  
从内部实现机理上来说，类型...type本质上是一个数组切片，也就是[]type，这也是为什么上面的参数 args 可以用 for 循环来获得每个传入的参数。  
  
假如没有...type这样的语法糖，开发者将不得不这么写：

1. func myfunc2(args []int) {
2. for \_, arg := range args {
3. fmt.Println(arg)
4. }
5. }

从函数的实现角度来看，这没有任何影响，该怎么写就怎么写，但从调用方来说，情形则完全不同：

myfunc2([]int{1, 3, 7, 13})

大家会发现，我们不得不加上[]int{}来构造一个数组切片实例，但是有了...type这个语法糖，我们就不用自己来处理了。

## 任意类型的可变参数

之前的例子中将可变参数类型约束为 int，如果你希望传任意类型，可以指定类型为 interface{}，下面是Go语言标准库中 fmt.Printf() 的函数原型：

func Printf(format string, args ...interface{}) {  
    // ...  
}

用 interface{} 传递任意类型数据是Go语言的惯例用法，使用 interface{} 仍然是类型安全的，这和 C/[C++](http://c.biancheng.net/cplus/" \t "http://c.biancheng.net/view/_blank) 不太一样，下面通过示例来了解一下如何分配传入 interface{} 类型的数据。

1. package main
2. import "fmt"
3. func MyPrintf(args ...interface{}) {
4. for \_, arg := range args {
5. switch arg.(type) {
6. case int:
7. fmt.Println(arg, "is an int value.")
8. case string:
9. fmt.Println(arg, "is a string value.")
10. case int64:
11. fmt.Println(arg, "is an int64 value.")
12. default:
13. fmt.Println(arg, "is an unknown type.")
14. }
15. }
16. }
17. func main() {
18. var v1 int = 1
19. var v2 int64 = 234
20. var v3 string = "hello"
21. var v4 float32 = 1.234
22. MyPrintf(v1, v2, v3, v4)
23. }

该程序的输出结果为：

1 is an int value.  
234 is an int64 value.  
hello is a string value.  
1.234 is an unknown type.

## 遍历可变参数列表——获取每一个参数的值

可变参数列表的数量不固定，传入的参数是一个切片，如果需要获得每一个参数的具体值时，可以对可变参数变量进行遍历，参见下面代码：

1. package main
2. import (
3. "bytes"
4. "fmt"
5. )
6. // 定义一个函数, 参数数量为0~n, 类型约束为字符串
7. func joinStrings(slist ...string) string {
8. // 定义一个字节缓冲, 快速地连接字符串
9. var b bytes.Buffer
10. // 遍历可变参数列表slist, 类型为[]string
11. for \_, s := range slist {
12. // 将遍历出的字符串连续写入字节数组
13. b.WriteString(s)
14. }
15. // 将连接好的字节数组转换为字符串并输出
16. return b.String()
17. }
18. func main() {
19. // 输入3个字符串, 将它们连成一个字符串
20. fmt.Println(joinStrings("pig ", "and", " rat"))
21. fmt.Println(joinStrings("hammer", " mom", " and", " hawk"))
22. }

代码输出如下：

pig and rat  
hammer mom and hawk

代码说明如下：

* 第 8 行，定义了一个可变参数的函数，slist 的类型为 []string，每一个参数的类型都是 string，也就是说，该函数只接受字符串类型作为参数。
* 第 11 行，bytes.Buffer 在这个例子中的作用类似于 StringBuilder，可以高效地进行字符串连接操作。
* 第 13 行，遍历 slist 可变参数，s 为每个参数的值，类型为 string。
* 第 15 行，将每一个传入参数放到 bytes.Buffer 中。
* 第 19 行，将 bytes.Buffer 中的数据转换为字符串作为函数返回值返回。
* 第 24 行，输入 3 个字符串，使用 joinStrings() 函数将参数连接为字符串输出。
* 第 25 行，输入 4 个字符串，连接后输出。

如果要获取可变参数的数量，可以使用 len() 函数对可变参数变量对应的切片进行求长度操作，以获得可变参数数量。

## 获得可变参数类型——获得每一个参数的类型

当可变参数为 interface{} 类型时，可以传入任何类型的值，此时，如果需要获得变量的类型，可以通过 switch 获得变量的类型，下面的代码演示将一系列不同类型的值传入 printTypeValue() 函数，该函数将分别为不同的参数打印它们的值和类型的详细描述。  
  
打印类型及值：

1. package main
2. import (
3. "bytes"
4. "fmt"
5. )
6. func printTypeValue(slist ...interface{}) string {
7. // 字节缓冲作为快速字符串连接
8. var b bytes.Buffer
9. // 遍历参数
10. for \_, s := range slist {
11. // 将interface{}类型格式化为字符串
12. str := fmt.Sprintf("%v", s)
13. // 类型的字符串描述
14. var typeString string
15. // 对s进行类型断言
16. switch s.(type) {
17. case bool: // 当s为布尔类型时
18. typeString = "bool"
19. case string: // 当s为字符串类型时
20. typeString = "string"
21. case int: // 当s为整型类型时
22. typeString = "int"
23. }
24. // 写字符串前缀
25. b.WriteString("value: ")
26. // 写入值
27. b.WriteString(str)
28. // 写类型前缀
29. b.WriteString(" type: ")
30. // 写类型字符串
31. b.WriteString(typeString)
32. // 写入换行符
33. b.WriteString("\n")
34. }
35. return b.String()
36. }
37. func main() {
38. // 将不同类型的变量通过printTypeValue()打印出来
39. fmt.Println(printTypeValue(100, "str", true))
40. }

代码输出如下：

value: 100 type: int  
value: str type: string  
value: true type: bool

代码说明如下：

* 第 8 行，printTypeValue() 输入不同类型的值并输出类型和值描述。
* 第 11 行，bytes.Buffer 字节缓冲作为快速字符串连接。
* 第 14 行，遍历 slist 的每一个元素，类型为 interface{}。
* 第 17 行，使用 fmt.Sprintf 配合%v动词，可以将 interface{} 格式的任意值转为字符串。
* 第 20 行，声明一个字符串，作为变量的类型名。
* 第 23 行，switch s.(type) 可以对 interface{} 类型进行类型断言，也就是判断变量的实际类型。
* 第 24～29 行为 s 变量可能的类型，将每种类型的对应类型字符串赋值到 typeString 中。
* 第 33～42 行为写输出格式的过程。

## 在多个可变参数函数中传递参数

可变参数变量是一个包含所有参数的切片，如果要将这个含有可变参数的变量传递给下一个可变参数函数，可以在传递时给可变参数变量后面添加...，这样就可以将切片中的元素进行传递，而不是传递可变参数变量本身。  
  
下面的例子模拟 print() 函数及实际调用的 rawPrint() 函数，两个函数都拥有可变参数，需要将参数从 print 传递到 rawPrint 中。  
  
可变参数传递：

1. package main
2. import "fmt"
3. // 实际打印的函数
4. func rawPrint(rawList ...interface{}) {
5. // 遍历可变参数切片
6. for \_, a := range rawList {
7. // 打印参数
8. fmt.Println(a)
9. }
10. }
11. // 打印函数封装
12. func print(slist ...interface{}) {
13. // 将slist可变参数切片完整传递给下一个函数
14. rawPrint(slist...)
15. }
16. func main() {
17. print(1, 2, 3)
18. }

代码输出如下：

1  
2  
3

对代码的说明：

* 第 9～13 行，遍历 rawPrint() 的参数列表 rawList 并打印。
* 第 20 行，将变量在 print 的可变参数列表中添加...后传递给 rawPrint()。
* 第 25 行，传入 1、2、3 这 3 个整型值并进行打印。

如果尝试将第 20 行修改为：

1. rawPrint("fmt", slist)

再次执行代码，将输出：

[1 2 3]

此时，slist（类型为 []interface{}）将被作为一个整体传入 rawPrint()，rawPrint() 函数中遍历的变量也就是 slist 的切片值。  
  
可变参数使用...进行传递与切片间使用 append 连接是同一个特性。

# Go语言defer（延迟执行语句）

< [Go语言可变参数](http://c.biancheng.net/view/60.html)[Go语言递归函数](http://c.biancheng.net/view/4193.html) >

[C语言中文网推出辅导班啦，包括「C语言辅导班、C++辅导班、算法/数据结构辅导班」，全部都是一对一教学：一对一辅导 + 一对一答疑 + 布置作业 + 项目实践 + 永久学习。QQ在线，随时响应！](http://fudao.biancheng.net/" \t "http://c.biancheng.net/view/_blank)

Go语言的 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 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。  
  
关键字 defer 的用法类似于面向对象编程语言 [Java](http://c.biancheng.net/java/" \t "http://c.biancheng.net/view/_blank) 和 [C#](http://c.biancheng.net/csharp/" \t "http://c.biancheng.net/view/_blank) 的 finally 语句块，它一般用于释放某些已分配的资源，典型的例子就是对一个互斥解锁，或者关闭一个文件。

## 多个延迟执行语句的处理顺序

当有多个 defer 行为被注册时，它们会以逆序执行（类似栈，即后进先出），下面的代码是将一系列的数值打印语句按顺序延迟处理，如下所示：

1. package main
2. import (
3. "fmt"
4. )
5. func main() {
6. fmt.Println("defer begin")
7. // 将defer放入延迟调用栈
8. defer fmt.Println(1)
9. defer fmt.Println(2)
10. // 最后一个放入, 位于栈顶, 最先调用
11. defer fmt.Println(3)
12. fmt.Println("defer end")
13. }

代码输出如下：

defer begin  
defer end  
3  
2  
1

结果分析如下：

* 代码的延迟顺序与最终的执行顺序是反向的。
* 延迟调用是在 defer 所在函数结束时进行，函数结束可以是正常返回时，也可以是发生宕机时。

## 使用延迟执行语句在函数退出时释放资源

处理业务或逻辑中涉及成对的操作是一件比较烦琐的事情，比如打开和关闭文件、接收请求和回复请求、加锁和解锁等。在这些操作中，最容易忽略的就是在每个函数退出处正确地释放和关闭资源。  
  
defer 语句正好是在函数退出时执行的语句，所以使用 defer 能非常方便地处理资源释放问题。

#### 1) 使用延迟并发解锁

在下面的例子中会在函数中并发使用 map，为防止竞态问题，使用 sync.Mutex 进行加锁，参见下面代码：

1. var (
2. // 一个演示用的映射
3. valueByKey = make(map[string]int)
4. // 保证使用映射时的并发安全的互斥锁
5. valueByKeyGuard sync.Mutex
6. )
7. // 根据键读取值
8. func readValue(key string) int {
9. // 对共享资源加锁
10. valueByKeyGuard.Lock()
11. // 取值
12. v := valueByKey[key]
13. // 对共享资源解锁
14. valueByKeyGuard.Unlock()
15. // 返回值
16. return v
17. }

代码说明如下：

* 第 3 行，实例化一个 map，键是 string 类型，值为 int。
* 第 5 行，map 默认不是并发安全的，准备一个 sync.Mutex 互斥量保护 map 的访问。
* 第 9 行，readValue() 函数给定一个键，从 map 中获得值后返回，该函数会在并发环境中使用，需要保证并发安全。
* 第 11 行，使用互斥量加锁。
* 第 13 行，从 map 中获取值。
* 第 15 行，使用互斥量解锁。
* 第 17 行，返回获取到的 map 值。

使用 defer 语句对上面的语句进行简化，参考下面的代码。

1. func readValue(key string) int {
2. valueByKeyGuard.Lock()
4. // defer后面的语句不会马上调用, 而是延迟到函数结束时调用
5. defer valueByKeyGuard.Unlock()
6. return valueByKey[key]
7. }

上面的代码中第 6~8 行是对前面代码的修改和添加的代码，代码说明如下：

* 第 6 行在互斥量加锁后，使用 defer 语句添加解锁，该语句不会马上执行，而是等 readValue() 函数返回时才会被执行。
* 第 8 行，从 map 查询值并返回的过程中，与不使用互斥量的写法一样，对比上面的代码，这种写法更简单。

#### 2) 使用延迟释放文件句柄

文件的操作需要经过打开文件、获取和操作文件资源、关闭资源几个过程，如果在操作完毕后不关闭文件资源，进程将一直无法释放文件资源，在下面的例子中将实现根据文件名获取文件大小的函数，函数中需要打开文件、获取文件大小和关闭文件等操作，由于每一步系统操作都需要进行错误处理，而每一步处理都会造成一次可能的退出，因此就需要在退出时释放资源，而我们需要密切关注在函数退出处正确地释放文件资源，参考下面的代码：

1. // 根据文件名查询其大小
2. func fileSize(filename string) int64 {
3. // 根据文件名打开文件, 返回文件句柄和错误
4. f, err := os.Open(filename)
5. // 如果打开时发生错误, 返回文件大小为0
6. if err != nil {
7. return 0
8. }
9. // 取文件状态信息
10. info, err := f.Stat()
12. // 如果获取信息时发生错误, 关闭文件并返回文件大小为0
13. if err != nil {
14. f.Close()
15. return 0
16. }
17. // 取文件大小
18. size := info.Size()
19. // 关闭文件
20. f.Close()
22. // 返回文件大小
23. return size
24. }

代码说明如下：

* 第 2 行，定义获取文件大小的函数，返回值是 64 位的文件大小值。
* 第 5 行，使用 os 包提供的函数 Open()，根据给定的文件名打开一个文件，并返回操作文件用的句柄和操作错误。
* 第 8 行，如果打开的过程中发生错误，如文件没找到、文件被占用等，将返回文件大小为 0。
* 第 13 行，此时文件句柄 f 可以正常使用，使用 f 的方法 Stat() 来获取文件的信息，获取信息时，可能也会发生错误。
* 第 16～19 行对错误进行处理，此时文件是正常打开的，为了释放资源，必须要调用 f 的 Close() 方法来关闭文件，否则会发生资源泄露。
* 第 22 行，获取文件大小。
* 第 25 行，关闭文件、释放资源。
* 第 28 行，返回获取到的文件大小。

在上面的例子中，第 25 行是对文件的关闭操作，下面使用 defer 对代码进行简化，代码如下：

1. func fileSize(filename string) int64 {
2. f, err := os.Open(filename)
3. if err != nil {
4. return 0
5. }
6. // 延迟调用Close, 此时Close不会被调用
7. defer f.Close()
8. info, err := f.Stat()
9. if err != nil {
10. // defer机制触发, 调用Close关闭文件
11. return 0
12. }
13. size := info.Size()
14. // defer机制触发, 调用Close关闭文件
15. return size
16. }

代码中加粗部分为对比前面代码而修改的部分，代码说明如下：

* 第 10 行，在文件正常打开后，使用 defer，将 f.Close() 延迟调用，注意，不能将这一句代码放在第 4 行空行处，一旦文件打开错误，f 将为空，在延迟语句触发时，将触发宕机错误。
* 第 16 行和第 22 行，defer 后的语句（f.Close()）将会在函数返回前被调用，自动释放资源。

# Go语言递归函数

< [Go语言defer（延迟执行语句）](http://c.biancheng.net/view/61.html)[Go语言处理运行时错误](http://c.biancheng.net/view/62.html) >

[C语言中文网推出辅导班啦，包括「C语言辅导班、C++辅导班、算法/数据结构辅导班」，全部都是一对一教学：一对一辅导 + 一对一答疑 + 布置作业 + 项目实践 + 永久学习。QQ在线，随时响应！](http://fudao.biancheng.net/" \t "http://c.biancheng.net/view/_blank)

很对编程语言都支持递归函数，Go语言也不例外，所谓递归函数指的是在函数内部调用函数自身的函数，从数学解题思路来说，递归就是把一个大问题拆分成多个小问题，再各个击破，在实际开发过程中，递归函数可以解决许多数学问题，如计算给定数字阶乘、产生斐波系列等。  
  
构成递归需要具备以下条件：

* 一个问题可以被拆分成多个子问题；
* 拆分前的原问题与拆分后的子问题除了数据规模不同，但处理问题的思路是一样的；
* 不能无限制的调用本身，子问题需要有退出递归状态的条件。

注意：编写递归函数时，一定要有终止条件，否则就会无限调用下去，直到内存溢出。

下面通过几个示例来演示一下递归函数的使用。

## 斐波那契数列

下面我们就以递归函数的经典示例 —— 斐波那契数列为例，演示如何通过Go语言编写的递归函数来打印斐波那契数列。  
  
数列的形式如下所示：

1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, 4181, 6765, 10946, …

使用Go语言递归函数实现斐波那契数列的具体代码如下所示：

1. package main
2. import "fmt"
3. func main() {
4. result := 0
5. for i := 1; i <= 10; i++ {
6. result = fibonacci(i)
7. fmt.Printf("fibonacci(%d) is: %d\n", i, result)
8. }
9. }
10. func fibonacci(n int) (res int) {
11. if n <= 2 {
12. res = 1
13. } else {
14. res = fibonacci(n-1) + fibonacci(n-2)
15. }
16. return
17. }

输出结果为：

fibonacci(1) is: 1  
fibonacci(2) is: 1  
fibonacci(3) is: 2  
fibonacci(4) is: 3  
fibonacci(5) is: 5  
fibonacci(6) is: 8  
fibonacci(7) is: 13  
fibonacci(8) is: 21  
fibonacci(9) is: 34  
fibonacci(10) is: 55

## 数字阶乘

一个正整数的阶乘（factorial）是所有小于及等于该数的正整数的积，并且 0 的阶乘为 1，自然数 n 的阶乘写作n!，“基斯顿·卡曼”在 1808 年发明了n!这个运算符号。  
  
例如，n!=1×2×3×…×n，阶乘亦可以递归方式定义：0!=1，n!=(n-1)!×n。  
  
使用递归函数计算给定数的阶乘，示例代码如下所示：

1. package main
2. import "fmt"
3. func Factorial(n uint64) (result uint64) {
4. if n > 0 {
5. result = n \* Factorial(n-1)
6. return result
7. }
8. return 1
9. }
10. func main() {
11. var i int = 10
12. fmt.Printf("%d 的阶乘是 %d\n", i, Factorial(uint64(i)))
13. }

输出结果为：

10 的阶乘是 3628800

## 多个函数组成递归

Go语言中也可以使用相互调用的递归函数，多个函数之间相互调用形成闭环，因为Go语言编译器的特殊性，这些函数的声明顺序可以是任意的，下面这个简单的例子展示了函数 odd 和 even 之间的相互调用：

1. package main
2. import (
3. "fmt"
4. )
5. func main() {
6. fmt.Printf("%d is even: is %t\n", 16, even(16)) // 16 is even: is true
7. fmt.Printf("%d is odd: is %t\n", 17, odd(17))
8. // 17 is odd: is true
9. fmt.Printf("%d is odd: is %t\n", 18, odd(18))
10. // 18 is odd: is false
11. }
12. func even(nr int) bool {
13. if nr == 0 {
14. return true
15. }
16. return odd(RevSign(nr) - 1)
17. }
18. func odd(nr int) bool {
19. if nr == 0 {
20. return false
21. }
22. return even(RevSign(nr) - 1)
23. }
24. func RevSign(nr int) int {
25. if nr < 0 {
26. return -nr
27. }
28. return nr
29. }

运行效果如下所示：

16 is even: is true  
17 is odd: is true  
18 is odd: is false

# Go语言处理运行时错误

< [Go语言递归函数](http://c.biancheng.net/view/4193.html)[Go语言宕机（panic）](http://c.biancheng.net/view/63.html) >

[C语言中文网推出辅导班啦，包括「C语言辅导班、C++辅导班、算法/数据结构辅导班」，全部都是一对一教学：一对一辅导 + 一对一答疑 + 布置作业 + 项目实践 + 永久学习。QQ在线，随时响应！](http://fudao.biancheng.net/" \t "http://c.biancheng.net/view/_blank)

Go语言的错误处理思想及设计包含以下特征：

* 一个可能造成错误的函数，需要返回值中返回一个错误接口（error），如果调用是成功的，错误接口将返回 nil，否则返回错误。
* 在函数调用后需要检查错误，如果发生错误，则进行必要的错误处理。

Go语言没有类似 [Java](http://c.biancheng.net/java/" \t "http://c.biancheng.net/view/_blank) 或 .NET 中的异常处理机制，虽然可以使用 defer、panic、recover 模拟，但官方并不主张这样做，Go语言的设计者认为其他语言的异常机制已被过度使用，上层逻辑需要为函数发生的异常付出太多的资源，同时，如果函数使用者觉得错误处理很麻烦而忽略错误，那么程序将在不可预知的时刻崩溃。  
  
Go语言希望开发者将错误处理视为正常开发必须实现的环节，正确地处理每一个可能发生错误的函数，同时，Go语言使用返回值返回错误的机制，也能大幅降低编译器、运行时处理错误的复杂度，让开发者真正地掌握错误的处理。

## net 包中的例子

net.Dial() 是Go语言系统包 net 即中的一个函数，一般用于创建一个 Socket 连接。  
  
net.Dial 拥有两个返回值，即 Conn 和 error，这个函数是阻塞的，因此在 Socket 操作后，会返回 Conn 连接对象和 error，如果发生错误，error 会告知错误的类型，Conn 会返回空。  
  
根据Go语言的错误处理机制，Conn 是其重要的返回值，因此，为这个函数增加一个错误返回，类似为 error，参见下面的代码：

1. func Dial(network, address string) (Conn, error) {
2. var d Dialer
3. return d.Dial(network, address)
4. }

在 io 包中的 Writer 接口也拥有错误返回，代码如下：

1. type Writer interface {
2. Write(p []byte) (n int, err error)
3. }

io 包中还有 Closer 接口，只有一个错误返回，代码如下：

1. type Closer interface {
2. Close() error
3. }

## 错误接口的定义格式

error 是 Go 系统声明的接口类型，代码如下：

1. type error interface {
2. Error() string
3. }

所有符合 Error()string 格式的方法，都能实现错误接口，Error() 方法返回错误的具体描述，使用者可以通过这个字符串知道发生了什么错误。

## 自定义一个错误

返回错误前，需要定义会产生哪些可能的错误，在Go语言中，使用 errors 包进行错误的定义，格式如下：

1. var err = errors.New("this is an error")

错误字符串由于相对固定，一般在包作用域声明，应尽量减少在使用时直接使用 errors.New 返回。

#### 1) errors 包

Go语言的 errors 中对 New 的定义非常简单，代码如下：

1. // 创建错误对象
2. func New(text string) error {
3. return &errorString{text}
4. }
5. // 错误字符串
6. type errorString struct {
7. s string
8. }
9. // 返回发生何种错误
10. func (e \*errorString) Error() string {
11. return e.s
12. }

代码说明如下：

* 第 2 行，将 errorString 结构体实例化，并赋值错误描述的成员。
* 第 7 行，声明 errorString 结构体，拥有一个成员，描述错误内容。
* 第 12 行，实现 error 接口的 Error() 方法，该方法返回成员中的错误描述。

#### 2) 在代码中使用错误定义

下面的代码会定义一个除法函数，当除数为 0 时，返回一个预定义的除数为 0 的错误。

1. package main
2. import (
3. "errors"
4. "fmt"
5. )
6. // 定义除数为0的错误
7. var errDivisionByZero = errors.New("division by zero")
8. func div(dividend, divisor int) (int, error) {
9. // 判断除数为0的情况并返回
10. if divisor == 0 {
11. return 0, errDivisionByZero
12. }
13. // 正常计算，返回空错误
14. return dividend / divisor, nil
15. }
16. func main() {
17. fmt.Println(div(1, 0))
18. }

代码输出如下：

0 division by zero

代码说明：

* 第 9 行，预定义除数为 0 的错误。
* 第 11 行，声明除法函数，输入被除数和除数，返回商和错误。
* 第 14 行，在除法计算中，如果除数为 0，计算结果为无穷大，为了避免这种情况，对除数进行判断，并返回商为 0 和除数为 0 的错误对象。
* 第 19 行，进行正常的除法计算，没有发生错误时，错误对象返回 nil。

## 示例：在解析中使用自定义错误

使用 errors.New 定义的错误字符串的错误类型是无法提供丰富的错误信息的，那么，如果需要携带错误信息返回，就需要借助自定义结构体实现错误接口。  
  
下面代码将实现一个解析错误（ParseError），这种错误包含两个内容，分别是文件名和行号，解析错误的结构还实现了 error 接口的 Error() 方法，返回错误描述时，就需要将文件名和行号返回。

1. package main
2. import (
3. "fmt"
4. )
5. // 声明一个解析错误
6. type ParseError struct {
7. Filename string // 文件名
8. Line int // 行号
9. }
10. // 实现error接口，返回错误描述
11. func (e \*ParseError) Error() string {
12. return fmt.Sprintf("%s:%d", e.Filename, e.Line)
13. }
14. // 创建一些解析错误
15. func newParseError(filename string, line int) error {
16. return &ParseError{filename, line}
17. }
18. func main() {
19. var e error
20. // 创建一个错误实例，包含文件名和行号
21. e = newParseError("main.go", 1)
22. // 通过error接口查看错误描述
23. fmt.Println(e.Error())
24. // 根据错误接口具体的类型，获取详细错误信息
25. switch detail := e.(type) {
26. case \*ParseError: // 这是一个解析错误
27. fmt.Printf("Filename: %s Line: %d\n", detail.Filename, detail.Line)
28. default: // 其他类型的错误
29. fmt.Println("other error")
30. }
31. }

代码输出如下：

main.go:1  
Filename: main.go Line: 1

代码说明如下：

* 第 8 行，声明了一个解析错误的结构体，解析错误包含有 2 个成员，分别是文件名和行号。
* 第 14 行，实现了错误接口，将成员的文件名和行号格式化为字符串返回。
* 第 19 行，根据给定的文件名和行号创建一个错误实例。
* 第 25 行，声明一个错误接口类型。
* 第 27 行，创建一个实例，这个错误接口内部是 \*ParserError 类型，携带有文件名 main.go 和行号 1。
* 第 30 行，调用 Error() 方法，通过第 15 行返回错误的详细信息。
* 第 33 行，通过错误断言，取出发生错误的详细类型。
* 第 34 行，通过分析这个错误的类型，得知错误类型为 \*ParserError，此时可以获取到详细的错误信息。
* 第 36 行，如果不是我们能够处理的错误类型，会打印出其他错误做出其他的处理。

错误对象都要实现 error 接口的 Error() 方法，这样，所有的错误都可以获得字符串的描述，如果想进一步知道错误的详细信息，可以通过类型断言，将错误对象转为具体的错误类型进行错误详细信息的获取。

# Go语言宕机（panic）——程序终止运行

< [Go语言处理运行时错误](http://c.biancheng.net/view/62.html)[Go语言宕机恢复（recover）](http://c.biancheng.net/view/64.html) >

[C语言中文网推出辅导班啦，包括「C语言辅导班、C++辅导班、算法/数据结构辅导班」，全部都是一对一教学：一对一辅导 + 一对一答疑 + 布置作业 + 项目实践 + 永久学习。QQ在线，随时响应！](http://fudao.biancheng.net/" \t "http://c.biancheng.net/view/_blank)

Go语言的类型系统会在编译时捕获很多错误，但有些错误只能在运行时检查，如数组访问越界、空指针引用等，这些运行时错误会引起宕机。  
  
宕机不是一件很好的事情，可能造成体验停止、服务中断，就像没有人希望在取钱时遇到 ATM 机蓝屏一样，但是，如果在损失发生时，程序没有因为宕机而停止，那么用户将会付出更大的代价，这种代价可以是金钱、时间甚至生命，因此，宕机有时也是一种合理的止损方法。  
  
一般而言，当宕机发生时，程序会中断运行，并立即执行在该 goroutine（可以先理解成线程）中被延迟的函数（defer 机制），随后，程序崩溃并输出日志信息，日志信息包括 panic value 和函数调用的堆栈跟踪信息，panic value 通常是某种错误信息。  
  
对于每个 goroutine，日志信息中都会有与之相对的，发生 panic 时的函数调用堆栈跟踪信息，通常，我们不需要再次运行程序去定位问题，日志信息已经提供了足够的诊断依据，因此，在我们填写问题报告时，一般会将宕机和日志信息一并记录。  
  
虽然Go语言的 panic 机制类似于其他语言的异常，但 panic 的适用场景有一些不同，由于 panic 会引起程序的崩溃，因此 panic 一般用于严重错误，如程序内部的逻辑不一致。任何崩溃都表明了我们的代码中可能存在漏洞，所以对于大部分漏洞，我们应该使用Go语言提供的错误机制，而不是 panic。

## 手动触发宕机

Go语言可以在程序中手动触发宕机，让程序崩溃，这样开发者可以及时地发现错误，同时减少可能的损失。  
  
Go语言程序在宕机时，会将堆栈和 goroutine 信息输出到控制台，所以宕机也可以方便地知晓发生错误的位置，那么我们要如何触发宕机呢，示例代码如下所示：

1. package main
2. func main() {
3. panic("crash")
4. }

代码运行崩溃并输出如下：

panic: crash  
  
goroutine 1 [running]:  
main.main()  
    D:/code/main.go:4 +0x40  
exit status 2

以上代码中只用了一个内建的函数 panic() 就可以造成崩溃，panic() 的声明如下：

1. func panic(v interface{}) //panic() 的参数可以是任意类型的。

## 在运行依赖的必备资源缺失时主动触发宕机

regexp 是Go语言的正则表达式包，正则表达式需要编译后才能使用，而且编译必须是成功的，表示正则表达式可用。  
  
编译正则表达式函数有两种，具体如下：

#### 1) func Compile(expr string) (\*Regexp, error)

编译正则表达式，发生错误时返回编译错误同时返回 Regexp 为 nil，该函数适用于在编译错误时获得编译错误进行处理，同时继续后续执行的环境。

#### 2) func MustCompile(str string) \*Regexp

当编译正则表达式发生错误时，使用 panic 触发宕机，该函数适用于直接使用正则表达式而无须处理正则表达式错误的情况。  
  
MustCompile 的代码如下：

1. func MustCompile(str string) \*Regexp {
2. regexp, error := Compile(str)
3. if error != nil {
4. panic(`regexp: Compile(` + quote(str) + `): ` + error.Error())
5. }
6. return regexp
7. }

代码说明如下：

* 第 1 行，编译正则表达式函数入口，输入包含正则表达式的字符串，返回正则表达式对象。
* 第 2 行，Compile() 是编译正则表达式的入口函数，该函数返回编译好的正则表达式对象和错误。
* 第 3 和第 4 行判断如果有错，则使用 panic() 触发宕机。
* 第 6 行，没有错误时返回正则表达式对象。

手动宕机进行报错的方式不是一种偷懒的方式，反而能迅速报错，终止程序继续运行，防止更大的错误产生，不过，如果任何错误都使用宕机处理，也不是一种良好的设计习惯，因此应根据需要来决定是否使用宕机进行报错。

## 在宕机时触发延迟执行语句

当 panic() 触发的宕机发生时，panic() 后面的代码将不会被运行，但是在 panic() 函数前面已经运行过的 defer 语句依然会在宕机发生时发生作用，参考下面代码：

1. package main
2. import "fmt"
3. func main() {
4. defer fmt.Println("宕机后要做的事情1")
5. defer fmt.Println("宕机后要做的事情2")
6. panic("宕机")
7. }

代码输出如下：

宕机后要做的事情2  
宕机后要做的事情1  
panic: 宕机  
  
goroutine 1 [running]:  
main.main()  
    D:/code/main.go:8 +0xf8  
exit status 2

对代码的说明：

* 第 6 行和第 7 行使用 defer 语句延迟了 2 个语句。
* 第 8 行发生宕机。

宕机前，defer 语句会被优先执行，由于第 7 行的 defer 后执行，因此会在宕机前，这个 defer 会优先处理，随后才是第 6 行的 defer 对应的语句，这个特性可以用来在宕机发生前进行宕机信息处理。

# Go语言宕机恢复（recover）——防止程序崩溃

< [Go语言宕机（panic）](http://c.biancheng.net/view/63.html)[Go语言计算函数执行时间](http://c.biancheng.net/view/4194.html) >

Recover 是一个Go语言的内建函数，可以让进入宕机流程中的 goroutine 恢复过来，recover 仅在延迟函数 defer 中有效，在正常的执行过程中，调用 recover 会返回 nil 并且没有其他任何效果，如果当前的 goroutine 陷入恐慌，调用 recover 可以捕获到 panic 的输入值，并且恢复正常的执行。  
  
通常来说，不应该对进入 panic 宕机的程序做任何处理，但有时，需要我们可以从宕机中恢复，至少我们可以在程序崩溃前，做一些操作，举个例子，当 web 服务器遇到不可预料的严重问题时，在崩溃前应该将所有的连接关闭，如果不做任何处理，会使得客户端一直处于等待状态，如果 web 服务器还在开发阶段，服务器甚至可以将异常信息反馈到客户端，帮助调试。

#### 提示

在其他语言里，宕机往往以异常的形式存在，底层抛出异常，上层逻辑通过 try/catch 机制捕获异常，没有被捕获的严重异常会导致宕机，捕获的异常可以被忽略，让代码继续运行。  
  
Go语言没有异常系统，其使用 panic 触发宕机类似于其他语言的抛出异常，recover 的宕机恢复机制就对应其他语言中的 try/catch 机制。

## 让程序在崩溃时继续执行

下面的代码实现了 ProtectRun() 函数，该函数传入一个匿名函数或闭包后的执行函数，当传入函数以任何形式发生 panic 崩溃后，可以将崩溃发生的错误打印出来，同时允许后面的代码继续运行，不会造成整个进程的崩溃。  
  
保护运行函数：

package main

import (

"fmt"

"runtime"

)

// 崩溃时需要传递的上下文信息

type panicContext struct {

function string // 所在函数

}

// 保护方式允许一个函数

func ProtectRun(entry func()) {

// 延迟处理的函数

defer func() {

// 发生宕机时，获取panic传递的上下文并打印

err := recover()

switch err.(type) {

case runtime.Error: // 运行时错误

fmt.Println("runtime error:", err)

default: // 非运行时错误

fmt.Println("error:", err)

}

}()

entry()

}

func main() {

fmt.Println("运行前")

// 允许一段手动触发的错误

ProtectRun(func() {

fmt.Println("手动宕机前")

// 使用panic传递上下文

panic(&panicContext{

"手动触发panic",

})

fmt.Println("手动宕机后")

})

// 故意造成空指针访问错误

ProtectRun(func() {

fmt.Println("赋值宕机前")

var a \*int

\*a = 1

fmt.Println("赋值宕机后")

})

fmt.Println("运行后")

}

代码输出结果：

运行前  
手动宕机前  
error: &{手动触发panic}  
赋值宕机前  
runtime error: runtime error: invalid memory address or nil pointer dereference  
运行后

对代码的说明：

* 第 9 行声明描述错误的结构体，保存执行错误的函数。
* 第 17 行使用 defer 将闭包延迟执行，当 panic 触发崩溃时，ProtectRun() 函数将结束运行，此时 defer 后的闭包将会发生调用。
* 第 20 行，recover() 获取到 panic 传入的参数。
* 第 22 行，使用 switch 对 err 变量进行类型断言。
* 第 23 行，如果错误是有 Runtime 层抛出的运行时错误，如空指针访问、除数为 0 等情况，打印运行时错误。
* 第 25 行，其他错误，打印传递过来的错误数据。
* 第 44 行，使用 panic 手动触发一个错误，并将一个结构体附带信息传递过去，此时，recover 就会获取到这个结构体信息，并打印出来。
* 第 57 行，模拟代码中空指针赋值造成的错误，此时会由 Runtime 层抛出错误，被 ProtectRun() 函数的 recover() 函数捕获到。

## panic 和 recover 的关系

panic 和 recover 的组合有如下特性：

* 有 panic 没 recover，程序宕机。
* 有 panic 也有 recover，程序不会宕机，执行完对应的 defer 后，从宕机点退出当前函数后继续执行。

#### 提示

虽然 panic/recover 能模拟其他语言的异常机制，但并不建议在编写普通函数时也经常性使用这种特性。  
  
在 panic 触发的 defer 函数内，可以继续调用 panic，进一步将错误外抛，直到程序整体崩溃。  
  
如果想在捕获错误时设置当前函数的返回值，可以对返回值使用命名返回值方式直接进行设置。

# Go语言计算函数执行时间

< [Go语言宕机恢复（recover）](http://c.biancheng.net/view/64.html)[示例：通过内存缓存来提升性能](http://c.biancheng.net/view/vip_7318.html) >

[C语言中文网推出辅导班啦，包括「C语言辅导班、C++辅导班、算法/数据结构辅导班」，全部都是一对一教学：一对一辅导 + 一对一答疑 + 布置作业 + 项目实践 + 永久学习。QQ在线，随时响应！](http://fudao.biancheng.net/" \t "http://c.biancheng.net/view/_blank)

函数的运行时间的长短是衡量这个函数性能的重要指标，特别是在对比和基准测试中，要得到函数的运行时间，最简单的办法就是在函数执行之前设置一个起始时间，并在函数运行结束时获取从起始时间到现在的时间间隔，这个时间间隔就是函数的运行时间。  
  
在Go语言中我们可以使用 time 包中的 Since() 函数来获取函数的运行时间，Go语言官方文档中对 Since() 函数的介绍是这样的。

func Since(t Time) Duration

Since() 函数返回从 t 到现在经过的时间，等价于time.Now().Sub(t)。  
  
【示例】使用 Since() 函数获取函数的运行时间。

1. package main
2. import (
3. "fmt"
4. "time"
5. )
6. func test() {
7. start := time.Now() // 获取当前时间
8. sum := 0
9. for i := 0; i < 100000000; i++ {
10. sum++
11. }
12. elapsed := time.Since(start)
13. fmt.Println("该函数执行完成耗时：", elapsed)
14. }
15. func main() {
16. test()
17. }

运行结果如下所示：

该函数执行完成耗时： 39.8933ms

上面我们提到了 time.Now().Sub() 的功能类似于 Since() 函数，想要使用 time.Now().Sub() 获取函数的运行时间只需要把我们上面代码的第 14 行简单修改一下就行。  
  
【示例 2】使用 time.Now().Sub() 获取函数的运行时间。

1. package main
2. import (
3. "fmt"
4. "time"
5. )
6. func test() {
7. start := time.Now() // 获取当前时间
8. sum := 0
9. for i := 0; i < 100000000; i++ {
10. sum++
11. }
12. elapsed := time.Now().Sub(start)
13. fmt.Println("该函数执行完成耗时：", elapsed)
14. }
15. func main() {
16. test()
17. }

运行结果如下所示：

该函数执行完成耗时： 36.8769ms

由于计算机 CPU 及一些其他因素的影响，在获取函数运行时间时每次的结果都有些许不同，属于正常现象。

# Go语言通过内存缓存来提升性能

< [Go语言计算函数执行时间](http://c.biancheng.net/view/4194.html)[Go语言哈希函数](http://c.biancheng.net/view/vip_7319.html) >

[C语言中文网推出辅导班啦，包括「C语言辅导班、C++辅导班、算法/数据结构辅导班」，全部都是一对一教学：一对一辅导 + 一对一答疑 + 布置作业 + 项目实践 + 永久学习。QQ在线，随时响应！](http://fudao.biancheng.net/" \t "http://c.biancheng.net/view/_blank)

前面我们介绍了[递归函数](http://c.biancheng.net/view/4193.html" \t "http://c.biancheng.net/view/_blank)，递归函数的缺点就是比较消耗内存，而且效率比较低，那么我们要怎样提高程序的执行效率呢？  
  
当在进行大量计算的时候，提升性能最直接有效的一种方式是避免重复计算，通过在内存中缓存并重复利用缓存从而避免重复执行相同计算的方式称为内存缓存。  
  
下面我们以经典的递归求斐波那契数列为例，来对比一下普通实现方法和加入内存缓存后程序的执行情况。

#### 普通的实现方法

普通方法的实现思路是，要计算数列中第 n 个数字，需要先得到它前面的两个数，以此类推。这么做的弊端是会产生大量的重复计算，代码如下所示：

1. package main
2. import (
3. "fmt"
4. "time"
5. )
6. func main() {
7. result := 0
8. start := time.Now()
9. for i := 1; i <= 40; i++ {
10. result = fibonacci(i)
11. fmt.Printf("数列第 %d 位: %d\n", i, result)
12. }
13. end := time.Now()
14. delta := end.Sub(start)
15. fmt.Printf("程序的执行时间为: %s\n", delta)
16. }
17. func fibonacci(n int) (res int) {
18. if n <= 2 {
19. res = 1
20. } else {
21. res = fibonacci(n-1) + fibonacci(n-2)
22. }
23. return
24. }

运行结果如下所示：

数列第 1 位: 1  
数列第 2 位: 1  
数列第 3 位: 2  
数列第 4 位: 3  
...  
数列第 39 位:  63245986  
数列第 40 位: 102334155  
程序的执行时间为: 2.2848865s

通过运行结果可以看出，获取第 40 位的数字所需要的时间是 2.2848865 秒（这个时间可能根据计算机性能的差异，略有不同）。

#### 内存缓存的实现方法

内存缓存的实现思路是在计算得到第 n 个数的同时，将它的值保存到数组中索引为 n 的位置上，在后续的计算中先在数组中查找所需要的值是否计算过，如果找到了，则直接从数组中获取，如果没找到，则再进行计算，代码如下所示：

1. package main
2. import (
3. "fmt"
4. "time"
5. )
6. const LIM = 41
7. var fibs [LIM]uint64
8. func main() {
9. var result uint64 = 0
10. start := time.Now()
11. for i := 1; i < LIM; i++ {
12. result = fibonacci(i)
13. fmt.Printf("数列第 %d 位: %d\n", i, result)
14. }
15. end := time.Now()
16. delta := end.Sub(start)
17. fmt.Printf("程序的执行时间为: %s\n", delta)
18. }
19. func fibonacci(n int) (res uint64) {
20. // 记忆化：检查数组中是否已知斐波那契（n）
21. if fibs[n] != 0 {
22. res = fibs[n]
23. return
24. }
25. if n <= 2 {
26. res = 1
27. } else {
28. res = fibonacci(n-1) + fibonacci(n-2)
29. }
30. fibs[n] = res
31. return
32. }

运行结果如下所示：

数列第 1 位: 1  
数列第 2 位: 1  
数列第 3 位: 2  
数列第 4 位: 3  
...  
数列第 39 位: 63245986  
数列第 40 位: 102334155  
程序的执行时间为: 0.0149603s

通过运行结果可以看出，同样获取数列第 40 位的数字，使用内存缓存后所用的时间为 0.0149603 秒，对比之前未使用内存缓存时的执行效率，可见内存缓存的优势还是相当明显的。

# Go语言哈希函数

< [示例：通过内存缓存来提升性能](http://c.biancheng.net/view/vip_7318.html)[Go语言函数的底层实现](http://c.biancheng.net/view/vip_7321.html) >

[C语言中文网推出辅导班啦，包括「C语言辅导班、C++辅导班、算法/数据结构辅导班」，全部都是一对一教学：一对一辅导 + 一对一答疑 + 布置作业 + 项目实践 + 永久学习。QQ在线，随时响应！](http://fudao.biancheng.net/" \t "http://c.biancheng.net/view/_blank)

Go语言中提供了 MD5、SHA-1 等几种哈希函数，下面我们用例子做一个介绍，代码如下所示。

1. package main
2. import (
3. "crypto/md5"
4. "crypto/sha1"
5. "fmt"
6. )
7. func main() {
8. TestString := "http://c.biancheng.net/golang/"
9. Md5Inst := md5.New()
10. Md5Inst.Write([]byte(TestString))
11. Result := Md5Inst.Sum([]byte(""))
12. fmt.Printf("%x\n\n", Result)
13. Sha1Inst := sha1.New()
14. Sha1Inst.Write([]byte(TestString))
15. Result = Sha1Inst.Sum([]byte(""))
16. fmt.Printf("%x\n\n", Result)
17. }

这个程序的执行结果为：

go run main.go  
6dc42d81095839903edf352ef1ec0a6a  
32313d69e3f0e4bbf6738858274e7e2c9a46d293

再举一个例子，对文件内容计算 SHA1，具体代码如下所示。

1. package main
2. import (
3. "crypto/md5"
4. "crypto/sha1"
5. "fmt"
6. "io"
7. "os"
8. )
9. func main() {
10. TestFile := "123.txt"
11. infile, inerr := os.Open(TestFile)
12. if inerr == nil {
13. md5h := md5.New()
14. io.Copy(md5h, infile)
15. fmt.Printf("%x %s\n", md5h.Sum([]byte("")), TestFile)
16. sha1h := sha1.New()
17. io.Copy(sha1h, infile)
18. fmt.Printf("%x %s\n", sha1h.Sum([]byte("")), TestFile)
19. } else {
20. fmt.Println(inerr)
21. os.Exit(1)
22. }
23. }

若要运行上面的代码，当前目录下需要包含一个 123.txt 文件，运行结果如下：

go run main.go  
6dc42d81095839903edf352ef1ec0a6a 123.txt  
da39a3ee5e6b4b0d3255bfef95601890afd80709 123.txt

# Go语言函数的底层实现

< [Go语言哈希函数](http://c.biancheng.net/view/vip_7319.html)[Go语言Test功能测试函数](http://c.biancheng.net/view/5409.html) >

[C语言中文网推出辅导班啦，包括「C语言辅导班、C++辅导班、算法/数据结构辅导班」，全部都是一对一教学：一对一辅导 + 一对一答疑 + 布置作业 + 项目实践 + 永久学习。QQ在线，随时响应！](http://fudao.biancheng.net/" \t "http://c.biancheng.net/view/_blank)

基于堆栈式的程序执行模型决定了函数是语言的一个核心元素，分析Go语言函数的底层实现，对理解整个程序的执行过程有很大的帮助，研究底层实现有两种办法，一种是看语言编译器源码，分析其对函数的各个特性的处理逻辑，另一种是反汇编，将可执行程序反汇编出来。  
  
本节使用反汇编这种短、平、快的方法，首先介绍Go语言的函数调用规约，接着介绍Go语言使用汇编语言的基本概念，然后通过反汇编技术来剖析Go语言函数某些特性的底层实现。

提示：阅读本节需要有一定的汇编基础，想学习汇编的同学，我们这里准备了一套《[汇编语言入门教程](http://c.biancheng.net/asm/" \t "http://c.biancheng.net/view/_blank)》供大家学习。

## 函数调用规约

Go语言函数使用的是 caller-save 的模式，即由调用者负责保存寄存器，所以在函数的头尾不会出现push ebp; mov esp ebp这样的代码，相反其是在主调函数调用被调函数的前后有一个保存现场和恢复现场的动作。  
  
主调函数保存和恢复现场的通用逻辑如下：

1. //开辟栈空间，压栈 BP 保存现场
2. SUBQ $x, SP //为函数开辟裁空间
3. MOVQ BP, y(SP) //保存当前函数 BP 到 y(SP）位直， y 为相对 SP 的偏移量
4. LEAQ y(SP), BP //重直 BP，使其指向刚刚保存 BP 旧值的位置，这里主要
5. //是方便后续 BP 的恢复
6. //弹出栈，恢复 BP
7. MOVQ y(SP), BP //恢复 BP 的值为调用前的值
8. ADDQ $x, SP //恢复 SP 的值为函数开始时的位

## 汇编基础

Go 编译器产生的汇编代码是一种中间抽象态，它不是对机器码的映射，而是和平台无关的一个中间态汇编描述，所以汇编代码中有些寄存器是真实的，有些是抽象的，几个抽象的寄存器如下：

* SB (Static base pointer)：静态基址寄存器，它和全局符号一起表示全局变量的地址。
* FP (Frame pointer)：栈帧寄存器，该寄存器指向当前函数调用栈帧的栈底位置。
* PC (Program counter)：程序计数器，存放下一条指令的执行地址，很少直接操作该寄存器，一般是 CALL、RET 等指令隐式的操作。
* SP (Stack pointer)：栈顶寄存器，一般在函数调用前由主调函数设置 SP 的值对栈空间进行分配或回收。

#### Go 汇编简介

1) Go 汇编器采用 AT&T 风格的汇编，早期的实现来自 plan9 汇编器，源操作数在前，目的操作数在后。  
  
2) Go 内嵌汇编和反汇编产生的代码并不是一一对应的，汇编编译器对内嵌汇编程序自动做了调整，主要差别就是增加了保护现场，以及函数调用前的保持 PC 、SP 偏移地址重定位等逻辑，反汇编代码更能反映程序的真实执行逻辑。  
  
3) Go 的汇编代码并不是和具体硬件体系结构的机器码一一对应的，而是一种半抽象的描述，寄存器可能是抽象的，也可能是具体的。  
  
下面代码的分析基于 AMD64 位架构下的 Linux 环境。

## 多值返回分析

多值返回函数 swap 的源码如下：

1. package main
2. func swap (a, b int) (x int, y int) {
3. x = b
4. y = a
5. return
6. }
7. func main() {
8. swap(10, 20)
9. }

#### 编译生成汇编如下

//- S 产生汇编的代码  
//- N 禁用优化  
//- 1 禁用内联  
  
GOOS=linux GOARCH=amd64 go tool compile -1 -N -S swap.go >swap.s 2>&1

#### 汇编代码分析

1) swap 函数和 main 函数汇编代码分析。例如：

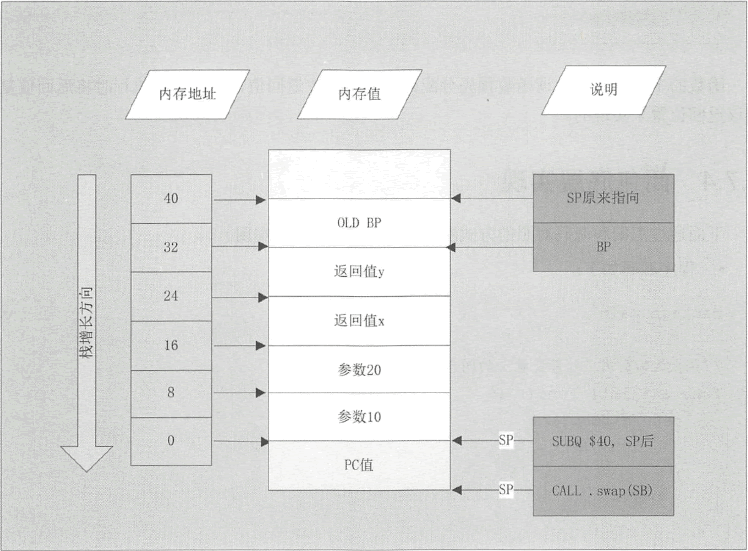
1. "".swap STEXT nosplit size=39 args=0x20 locals=0x0
2. 0x0000 00000 (swap.go:4) TEXT "".swap(SB), NOSPLIT, $0 - 32
3. 0x0000 00000 (swap.go:4) FUNCDATA $0, gclocals.ff19ed39bdde8a01a800918ac3ef0ec7(SB)
4. 0x0000 00000 (swap.go:4) FUNCDATA $1, gclocals.33cdeccccebe80329flfdbee7f5874cb(SB)
5. 0x0000 00000 (swap.go:4) MOVQ $0, "".x+24(SP)
6. 0x0009 00009 (swap.go:4) MOVQ $0, "".y+32(SP)
7. 0x0012 00018 (swap.go:5) MOVQ "".b+16(SP), AX
8. 0x0017 00023 (swap.go:5) MOVQ AX, "".x+24(SP)
9. 0xOO1c 00028 (swap.go:6) MOVQ "".a+8(SP), AX
10. 0x0021 00033 (swap.go:6) MOVQ AX, "".y+32(SP)
11. 0x0026 00038 (swap.go:7) RET
12. "".main STEXT size=68 args=0x0 locals=0x28
13. 0x0000 00000 (swap.go:10) TEXT "".main(SB), $40 - 0
14. 0x0000 00000 (swap.go:10) MOVQ (TLS), CX
15. 0x0009 00009 (swap.go:10) CMPQ SP, 16(CX)
16. 0x000d 00013 (swap.go:10) JLS 61
17. 0x000f 00015 (swap.go:10) SUBQ $40, SP
18. 0x0013 00019 (swap.go:10) MOVQ BP, 32 (SP)
19. 0x0018 00024 (swap.go:10) LEAQ 32(SP), BP
20. 0x001d 00029 (swap.go:10) FUNCDATA $0, gclocals ·33cdeccccebe80329flfdbee7f5874cb(SB)
21. 0x001d 00029 (swap.go:10) FUNCDATA $1, gclocals ·33cdeccccebe80329flfdbee7f5874cb(SB)
22. 0x001d 00029 (swap.go:11) MOVQ $10, (SP)
23. 0x0025 00037 (swap.go:11) MOVQ $20 , 8 (SP)
24. 0x002e 00046 (swap.go:11) PCDATA $0 , $0
25. 0x002e 00046 (swap.go:11) CALL "". swap(SB)
26. 0x0033 00051 (swap.go:12) MOVQ 32(SP), BP
27. 0x0038 00056 (swap.go:12) ADDQ $40, SP
28. 0x003c 00060 (swap.go:12) RET
29. 0x003d 00061 (swap.go:12) NOP
30. 0x003d 00061 (swap.go:10) PCDATA $0, $ - 1

* 第 5 行初始化返回值 x 为 0。
* 第 6 行初始化返回值 y 为 0。
* 第 7～8 行取第 2 个参数赋值给返回值 x。
* 第 9～10 行取第 1 个参数赋值给返回值 y。
* 第 11 行函数返回，同时进行栈回收，FUNCDATA 和垃圾收集可以忽略。
* 第 15～24 行 main 函数堆栈初始化：开辟栈空间，保存 BP 寄存器。
* 第 25 行初始化 add 函数的调用参数 1 的值为 10。
* 第 26 行初始化 add 函数的调用参数 2 的值为 20。
* 第 28 行调用 swap 函数，注意 call 隐含一个将 swap 下一条指令地址压栈的动作，即 sp=sp+8。
* 所以可以看到在 swap 里面的所有变量的相对位置都发生了变化，都在原来的地址上 ＋8。
* 第 29～30 行恢复措空间。

从汇编的代码得知：

* 函数的调用者负责环境准备，包括为参数和返回值开辟栈空间。
* 寄存器的保存和恢复也由调用方负责。
* 函数调用后回收栈空间，恢复 BP 也由主调函数负责。

函数的多值返回实质上是在栈上开辟多个地址分别存放返回值，这个并没有什么特别的地方，如果返回值是存放到堆上的，则多了一个复制的动作。  
  
main 调用 swap 函数栈的结构如下图所示。

  
图：Go函数栈

函数调用前己经为返回值和参数分配了栈空间，分配顺序是从右向左的，先是返回值，然后是参数，通用的栈模型如下：

＋----------＋  
 | 返回值 y    |  
 |------------|  
 | 返回值 x    |  
 |------------|  
 |  参数 b      |  
 |------------|  
 |  参数 a      |  
＋----------＋

函数的多返回值是主调函数预先分配好空间来存放返回值，被调函数执行时将返回值复制到该返回位置来实现的。

## 闭包底层实现

下面通过汇编和源码对照的方式看一下 Go 闭包的内部实现。  
  
程序源码如下：

1. package main
2. //函数返回引用了外部变量 i 的闭包
3. func a(i int) func () {
4. return func() {
5. print(i)
6. }
7. }
8. func main() {
9. f := a (1)
10. f ()
11. }

编译汇编如下：

GOOS=linux GOARCH=amd64 go tool compile -S c2\_7\_4a.go >c2\_7\_4a.s 2&1

关键汇编代码及分析如下：  
  
//函数 a 和函数 main 对应的汇编代码

1. "".a STEXT size=91 args=0x10 locals=0x18
2. 0x0000 00000 (c2\_7\_4a.go:3) TEXT "".a(SB), $24-16
3. 0x0000 00000 (c2\_7\_4a.go:3) MOVQ (TLS), CX
4. 0x0009 00009 (c2\_7\_4a.go:3) CMPQ SP, 16(CX)
5. 0x000d 00013 (c2\_7\_4a.go:3) JLS 84
6. 0x000f 00015 (c2\_7\_4a.go:3) SUBQ $24, SP
7. 0x0013 00019 (c2\_7\_4a.go:3) MOVQ BP , 16(SP)
8. 0x0018 00024 (c2\_7\_4a.go:3) LEAQ 16(SP), BP
9. 0x001d 00029 (c2\_7\_4a.go:3) FUNCDATA $0, gclocals·f207267fbf96a0178e8758c6e3e0ce28(SB)
10. 0x001d 00029 (c2\_7\_4a.go:3) FUNCDATA $1, gclocals·33cdeccccebe80329flfdbee7f5874cb (SB)
11. 0x001d 00029 (c2\_7\_4a.go:4) LEAQ type.noalg.struct{ F uintptr; "".i int}(SB), AX
12. 0x0024 00036 (c2\_7\_4a.go:4) MOVQ AX, (SP)
13. 0x0028 00040 (c2\_7\_4a.go:4) PCDATA $0, $0
14. 0x0028 00040 (c2\_7\_4a.go:4) CALL runtime.newobject(SB)
15. 0x002d 00045 (c2\_7\_4a.go:4) MOVQ 8(SP), AX
16. 0x0032 00050 (c2\_7\_4a.go:4) LEAQ "".a.funcl(SB), CX
17. 0x0039 00057 (c2\_7\_4a.go:4) MOVQ CX, (AX)
18. 0x003c 00060 (c2\_7\_4a.go:3) MOVQ "".i+32(SP), CX
19. 0x0041 00065 (c2\_7\_4a.go:4) MOVQ CX, 8(AX)
20. 0x0045 00069 (c2\_7\_4a.go:4) MOVQ AX, "".~r1+40(SP)
21. 0x004a 00074 (c2\_7\_4a.go:4) MOVQ 16(SP), BP
22. 0x004f 00079 (c2\_7\_4a.go:4) ADDQ $24, SP
23. "".main STEXT size=69 args=0x0 locals=0x18
24. 0x0000 00000 (c2\_7\_4a.go:9) TEXT "".main(SB), $24-0
25. 0x0000 00000 (c2\_7\_4a.go:9) MOVQ (TLS), CX
26. 0x0009 00009 (c2\_7\_4a.go:9) CMPQ SP, 16(CX)
27. 0x000d 00013 (c2\_7\_4a.go:9) JLS 62
28. 0x000f 00015 (c2\_7\_4a.go:9) SUBQ $24, SP
29. 0x0013 00019 (c2\_7\_4a.go:9) MOVQ BP, 16(SP)
30. 0x0018 00024 (c2\_7\_4a.go:9) LEAQ 16(SP), BP
31. 0x00ld 00029 (c2\_7\_4a.go:9) FUNCDATA $0, gclocals·33cdeccccebe80329flfdbee7f5874cb(SB)
32. 0x00ld 00029 (c2\_7\_4a.go:9) FUNCDATA $1, gclocals·33cdeccccebe80329flfdbee7f5874cb(SB)
33. 0x00ld 00029 (c2\_7\_4a.go:10) MOVQ $1, (SP)
34. 0x0025 00037 (c2\_7\_4a.go:10) PCDATA $0, $0
35. 0x0025 00037 (c2\_7\_4a.go:10) CALL "".a(SB)
36. 0x002a 00042 (c2\_7\_4a.go:10) MOVQ 8(SP), DX
37. 0x002f 00047 (c2\_7\_4a.go:11) MOVQ (DX), AX
38. 0x0032 00050 (c2\_7\_4a.go:11) PCDATA $0, $0
39. 0x0032 00050 (c2\_7\_4a.go:11) CALL AX
40. 0x0034 00052 (c2\_7\_4a.go:15) MOVQ 16(SP), BP
41. 0x0039 00057 (c2\_7\_4a.go:15) ADDQ $24, SP
42. 0x003d 00061 (c2\_7\_4a.go:15) RET

#### func a() 函数分析

* 第 1～10 行环境准备。
* 第 11 行这里我们看到type.noalg.struct { F uintptr; "".i int }(SB)这个符号是一个闭包类型的数据，闭包类型的数据结构如下：

type Closure struct {  
    F uintptr  
    i int  
}

闭包的结构很简单，一个是函数指针，另一个是对外部环境的引用。注意，这里仅仅是打印 i，并没有修改 i，Go语言编译器并没有传递地址而是传递值。

* 第 11 行将闭包类型元信息放到 (SP) 位置，(SP) 地址存放的是 CALL 函数调用的第一个参数。
* 第 14 行创建闭包对象，我们来看一下 runtime.newobject 的函数原型，该函数的输入参数是一个类型信息，返回值是根据该类型信息构造出来的对象地址。

// src/runtime/malloc.go  
func newobject(typ \*\_type) unsafe.Pointer

* 第 15 行将 newobject 返回的对象地址复制给 AX 寄存器。
* 第 16 行将 a 函数里面的匿名函数 a.func 指针复制到 CX 寄存器。
* 第 17 行将 CX 寄存器中存放的 a.func 函数指针复制到闭包对象的函数指针位置。
* 第 18、19 行将外部闭包变量 i 的值复制到闭包对象的 i 处。
* 第 20 行复制闭包对象指针值到函数返回值位置 "".～r1+40(SP)。

#### main() 函数分析

* 第 23～32 行准备环境。
* 第 33 行将立即数 1 复制到 (SP) 位置，为后续的 CALL 指令准备参数。
* 第 35 行调用函数 a()。
* 第 36 行复制函数返回值到 DX 寄存器。
* 第 37 行间接寻址，复制闭包对象中的函数指针到 AX 寄存器。
* 第 39 行调用 AX 寄存器指向的函数。
* 第 40～42 行恢复环境，并返回。

通过汇编代码的分析，我们清楚地看到 Go 实现闭包是通过返回一个如下的结构来实现的。

type Closure struct {  
    F uintptr  
    env \*Type  
}

F 是返回的匿名函数指针，env 是对外部环境变量的引用集合，如果闭包内没有修改外部变量，则 Go 编译器直接优化为值传递，如上面的例子中的代码所示，反之则是通过指针传递的。

# Go语言Test功能测试函数详解

< [Go语言函数的底层实现](http://c.biancheng.net/view/vip_7321.html)[Go语言结构体](http://c.biancheng.net/golang/struct/) >

[C语言中文网推出辅导班啦，包括「C语言辅导班、C++辅导班、算法/数据结构辅导班」，全部都是一对一教学：一对一辅导 + 一对一答疑 + 布置作业 + 项目实践 + 永久学习。QQ在线，随时响应！](http://fudao.biancheng.net/" \t "http://c.biancheng.net/view/_blank)

Go语言自带了 testing 测试包，可以进行自动化的单元测试，输出结果验证，并且可以测试性能。

## 为什么需要测试

完善的测试体系，能够提高开发的效率，当项目足够复杂的时候，想要保证尽可能的减少 bug，有两种有效的方式分别是代码审核和测试，Go语言中提供了 testing 包来实现单元测试功能。

## 测试规则

要开始一个单元测试，需要准备一个 go 源码文件，在命名文件时文件名必须以\_test.go结尾，单元测试源码文件可以由多个测试用例（可以理解为函数）组成，每个测试用例的名称需要以 Test 为前缀，例如：

func TestXxx( t \*testing.T ){  
    //......  
}

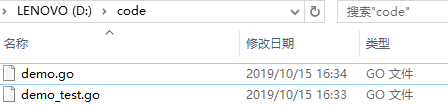
编写测试用例有以下几点需要注意：

* 测试用例文件不会参与正常源码的编译，不会被包含到可执行文件中；
* 测试用例的文件名必须以\_test.go结尾；
* 需要使用 import 导入 testing 包；
* 测试函数的名称要以Test或Benchmark开头，后面可以跟任意字母组成的字符串，但第一个字母必须大写，例如 TestAbc()，一个测试用例文件中可以包含多个测试函数；
* 单元测试则以(t \*testing.T)作为参数，性能测试以(t \*testing.B)做为参数；
* 测试用例文件使用go test命令来执行，源码中不需要 main() 函数作为入口，所有以\_test.go结尾的源码文件内以Test开头的函数都会自动执行。

Go语言的 testing 包提供了三种测试方式，分别是单元（功能）测试、性能（压力）测试和覆盖率测试。

## 单元（功能）测试

在同一文件夹下创建两个Go语言文件，分别命名为 demo.go 和 demt\_test.go，如下图所示：



具体代码如下所示：

#### demo.go：

1. package demo
2. // 根据长宽获取面积
3. func GetArea(weight int, height int) int {
4. return weight \* height
5. }

#### demo\_test.go：

1. package demo
2. import "testing"
3. func TestGetArea(t \*testing.T) {
4. area := GetArea(40, 50)
5. if area != 2000 {
6. t.Error("测试失败")
7. }
8. }

执行测试命令，运行结果如下所示：

PS D:\code> go test -v  
=== RUN   TestGetArea  
--- PASS: TestGetArea (0.00s)  
PASS  
ok      \_/D\_/code       0.435s

## 性能（压力）测试

将 demo\_test.go 的代码改造成如下所示的样子：

1. package demo
2. import "testing"
3. func BenchmarkGetArea(t \*testing.B) {
4. for i := 0; i < t.N; i++ {
5. GetArea(40, 50)
6. }
7. }

执行测试命令，运行结果如下所示：

PS D:\code> go test -bench="."  
goos: windows  
goarch: amd64  
BenchmarkGetArea-4      2000000000               0.35 ns/op  
PASS  
ok      \_/D\_/code       1.166s

上面信息显示了程序执行 2000000000 次，共耗时 0.35 纳秒。

## 覆盖率测试

覆盖率测试能知道测试程序总共覆盖了多少业务代码（也就是 demo\_test.go 中测试了多少 demo.go 中的代码），可以的话最好是覆盖100%。  
  
将 demo\_test.go 代码改造成如下所示的样子：

1. package demo
2. import "testing"
3. func TestGetArea(t \*testing.T) {
4. area := GetArea(40, 50)
5. if area != 2000 {
6. t.Error("测试失败")
7. }
8. }
9. func BenchmarkGetArea(t \*testing.B) {
10. for i := 0; i < t.N; i++ {
11. GetArea(40, 50)
12. }
13. }

执行测试命令，运行结果如下所示：

PS D:\code> go test -cover  
PASS  
coverage: 100.0% of statements  
ok      \_/D\_/code       0.437s