函数function

- Go 函数 **不支持** 嵌套、重载和默认参数
- 但支持以下特性：

  无需声明原型、不定长度变参、多返回值、命名返回值参数
  匿名函数、闭包

- 定义函数使用关键字 func，且左大括号不能另起一行
- 函数也可以作为一种类型使用

- Go 函数 **不支持** **嵌套**、**重载**和**默认参数**
- 但支持以下特性：

  无需声明原型、不定长度变参、多返回值、命名返回值参数
  匿名函数、闭包

- 定义函数使用关键字 func，且左大括号不能另起一行
- 函数也可以作为一种类型使用

function代码例子:

1、普通函数

定义个函数A

```
1.  package main
2.
3.  import (
4.      "fmt"
5.  )
6.
7.  func main() {
8.
9.  }
10.
11. func A() {
12.
13. }
```

首先是一个关键字**func** 然后是函数的名称，然后是一对小括号**()** ，小括号当中是A函数参数列表，这里如果没有参数就为空。如有参数的话，就先写明参数的名称，参数的类型。

```
1.  package main
2.
3.  import (
```

```
    "fmt"
)

func main() {

}

func A(a int, b string) {

}
```

这里就是有俩个参数，int类型的a和string类型的b。这样A函数就需要俩个参数 a 和 b才能成功的对它进行调用。如果你知道他的返回值的话呢，那就在小口号的右边 在输入小括号

```
package main

import (
    "fmt"
)

func main() {

}

func A(a int, b string) (int, string) {

}
```

它可以返回int类型，也可以返回第2个参数string类型，它还可以返回第3个参数int型。如果没有返回值的话，就可以把这个小括号以及小括号的内容去掉。

```
package main

import (
    "fmt"
)

func main() {

}

func A(a int, b string) {

}
```

如果至返回一个返回值参数的话，就不需要小括号，可以直接写上返回值的类型

```
package main

import (
    "fmt"
)

```

```
func main() {

}

func A(a int, b string) int {

}
```

假如还有一种情况，小括号里面的参数全是int型

```
package main

import (
    "fmt"
)

func main() {

}

func A(a int, b int, c int) int {

}
```

这时候三个全都是int型，那么有什么办法可以简写吗？可以把前面俩个int都去掉

```
package main

import (
    "fmt"
)

func main() {

}

func A(a, b, c int) int {

}
```

这样的话 a b c 全都是int类型。
同样的规则也可以适用的返回值当中。如果有3个返回值a b c，这三个返回值都是int类型。

```
package main

import (
    "fmt"
)

func main() {

}

```

```
func A() (a, b, c int) {

}
```

那么命名返回值参数和不命名返回值参数有什么区别呢？
1.如果要使用多个返回值的话，而且是简写返回值的，你就必须命名返回值。不然就无法知道。也可以这样写，这样写就没有必要命名返回值

```
package main

import (
    "fmt"
)

func main() {

}

func A() (int, int, int) {

}
```

假设我这里还有a b c

```
package main

import (
    "fmt"
)

func main() {

}

func A() (int, int, int) {
    a,b,c:=1,2,3
}
```

如果我没有命名返回值，那么**return 后面就必须跟上返回值名称**

```
package main

import (
    "fmt"
)

func main() {

}

func A() (int, int, int) {
    a, b, c := 1, 2, 3
    return a, b, c
```

```
}
```

如果前面已经命名这个返回值。那么return 后面就不需要跟上返回值，因为它知道我要返回的是我前面已经定义好的返回值

```
package main

import (
    "fmt"
)

func main() {

}

func A() (a, b, c int) {
    a, b, c := 1, 2, 3
    return
}
```

这里面有一个小点，就说明这个语句错误了。为什么呢？如果定直接使用命名返回值参数，它其实在这个函数执行代码之前它就已经存在了。作为这个函数的均衡变量。所以收这里我们不需要这个冒号，只需要直接进行赋值。因为在对变量a b c 赋值这行代码之前就已经存在。已经作为变行地址了。

```
package main

import (
    "fmt"
)

func main() {

}

func A() (a, b, c int) {
    a, b, c = 1, 2, 3
    return
}
```

作为一个代码编写要求呢，要求不管有没有定义返回值参数，都要在return 加上返回值名称，这样看着比较清楚

```
package main

import (
    "fmt"
)

func main() {

}
```

```
10.
11. func A() (a, b, c int) {
12.     a, b, c = 1, 2, 3
13.     return a, b, c
14. }
```

有一个需求，不知道这个A函数 接受多少个参数，但是我知道它都是int类型。要计算一串数字的最大值，我可能会有a b c d e f int

```
1.  package main
2.
3.  import (
4.      "fmt"
5.  )
6.
7.  func main() {
8.
9.  }
10.
11. func A(a, b, c, d, e, f int) (a, b, c int) {
12.     a, b, c = 1, 2, 3
13.     return a, b, c
14. }
```

我还可能会有N个，这时候怎么办呢？
这时候就可以利用go语言当值的不定长变参 那么怎么使用呢？使用类型前面加上3个点...

```
1.  package main
2.
3.  import (
4.      "fmt"
5.  )
6.
7.  func main() {
8.
9.  }
10.
11. func A(a, b, c, d, e, f ...int) (a, b, c int) {
12.     a, b, c = 1, 2, 3
13.     return a, b, c
14. }
```

变成这样子就是不定长变参的使用

```
1.  package main
2.
3.  import (
4.      "fmt"
5.  )
6.
```

```
func main() {

}

func A(a ...int) {
    a, b, c = 1, 2, 3
    return a, b, c
}
```

也就是说 这个参数a在接受不定长变参的时候，它就变成了一个slice。
打印一下a的这个slice的数值

```
package main

import (
    "fmt"
)

func main() {

}

func A(a ...int) {
    fmt.Println(a) //打印slice a
}
```

我在main函数中调用A函数的值。输入1 和 2

```
package main

import (
    "fmt"
)

func main() {
    A(1, 2) // 调用A函数
}

func A(a ...int) {
    fmt.Println(a) //打印slice a
}
```

```
/home/jiemin/code/GOlang/go/src/function/function [/home/jiemin/code/GOlang/go/src/function]
[1 2]
成功: 进程退出代码 0.
```

```
package main

import (
    "fmt"
)
```

```
6.
7.  func main() {
8.      A(1, 2, 3, 4, 5, 6, 7, 8, 9, 0) // 调用A函数
9.  }
10.
11. func A(a ...int) {
12.     fmt.Println(a) //打印slice a
13. }
```

```
/home/jiemin/code/GOlang/go/src/function/function [/home/jiemin/code/GOlang/go/src/function]
[1 2 3 4 5 6 7 8 9 0]
成功: 进程退出代码 0.
```

这个就是不定长变参的使用方法，有一个注意事项，它必须是参数列表中的最后一个参数，如果使用了不定长变参，那么后面就不能在加变量

```
1.  package main
2.
3.  import (
4.      "fmt"
5.  )
6.
7.  func main() {
8.      A(1, 2, 3, 4, 5, 6, 7, 8, 9, 0) // 调用A函数
9.  }
10.
11. func A(a ...int,b string) {
12.     fmt.Println(a) //打印slice a
13. }
```

使用`(a ...int,b string)`这样的语法是错误的，使用了不定长变参。这个是最后一个参数才能使用。在不定长变参后面在加上参数。那么就会报错。那麽怎么做才能不报错呢？那就把b string 这个移动到前面

```
1.  package main
2.
3.  import (
4.      "fmt"
5.  )
6.
7.  func main() {
8.      A(1, 2, 3, 4, 5, 6, 7, 8, 9, 0) // 调用A函数
9.  }
10.
11. func A(b string, a ...int) {
12.     fmt.Println(a) //打印slice a
13. }
```

这样就可以使用了。这就是使用不定长变参的硬性规定

然后在注意，传进来是一个slice。这里面还有小小的疑问？slice它是引用传递，引用类型，虽说`A(1, 2, 3, 4, 5, 6, 7, 8, 9, 0)`这串数字传入进来，不是整数1 2 3 4 5 6 7 8 9 0 ，而是几个变量，于是在main函数中有a 和 b 分别等于1和2。然后打印一下a和b。这时候在讲A函数中的a和b变

量参数传递进去。在A函数接收一个int型的slice，然后在赋值的当中进行一个修改，slice的索引0对应的就是a，slice的索引1对应的就是b。然后打印一下slice的值。看一下打印结果。

```go
package main

import (
    "fmt"
)

func main() {
    a, b := 1, 2
    A(a, b)            // 调用A函数。将a,b传进去
    fmt.Println(a, b) //打印a,b
}

func A(s ...int) {
    s[0] = 3       // 把slice的第一个数值1改成3
    s[1] = 4       // 把slice的第一个数值2改成4
    fmt.Println(s) //打印slice s
}
```

```
/home/jiemin/code/GOlang/go/src/function/function [/home/jiemin/code/GOlang/go/src/function]
[3 4]
1 2
成功: 进程退出代码 0.
```

可以看到尽管在A函数中接收到的是一个slice，但是它实际上得到的是一个值拷贝。那么它和直接传递进去的slice有什么区别？我们来看一下。
有一个slice 1，是一个int类型的slice。然后我们讲这个slice 1作为参数传递进去，传到函数当中就是一个s []int
同样对这几个参数对一个修改。

```go
package main

import (
    "fmt"
)

func main() {
    s1 := []int{1, 2, 3, 4} //创建一个slice1
    A(s1)                   // 调用A函数。将是slice作为参数传进去
    fmt.Println(s1)         //打印slice 1
}

func A(s []int) { // 函数中就是一个slice s []int
    s[0] = 5 // 把slice的第一个数值1改成5
    s[1] = 6 // 把slice的第一个数值2改成6
    s[2] = 7 // 把slice的第一个数值3改成7
    s[3] = 8 // 把slice的第一个数值4改成8
    fmt.Println(s) //打印slice s
}
```

```
/home/jiemin/code/GOlang/go/src/function/function [/home/jiemin/code/GOlang/go/src/function]
[5 6 7 8]
[5 6 7 8]
成功: 进程退出代码 0.
```

看到在A函数中修改slice引起了main函数的变量。所以说，如果直接传递一个slice，它就会直接影响slice本身，这里要注意，它并不是传递一个指针进去，而是对这个slice的内存地址进行了一个拷贝，所以说这个变量的值拷贝和这个slice之间的地址拷贝，同样都是本质上的拷贝传递，至不过它拷贝的地方不一样。如果说只是传递一个值类型，int类型的

```go
package main

import (
    "fmt"
)

func main() {
    a := 1          //创建一个a变量
    A(a)            // 调用A函数。将是a参数传进去
    fmt.Println(a) //打印a
}

func A(a int) { // 函数中就是int型的变量a
    a = 2           // 创建一个a变量
    fmt.Println(a) //打印a
}
```

```
/home/jiemin/code/GOlang/go/src/function/function [/home/jiemin/code/GOlang/go/src/function]
2
1
成功: 进程退出代码 0.
```

像上面这种int类型 、string类型 进行一个普通常规参数传入进去，它是一个值拷贝。但是刚才的slice，为什么可以对这个原始的slice进行一个修改呢？因为它穿进去，虽然也是一个拷贝，但是它是一个地址的拷贝，所以说只要拿到这个slice的地址，实际上就是对这个slice本身进行操作，如果想要也用int，string型也进行一个地址的拷贝，怎么办呢？
我们来进行一个类似指针传递 ，比如说这里先取出a的地址。

```go
package main

import (
    "fmt"
)

func main() {
    a := 1          //创建一个a变量
    A(&a)           // 调用A函数。先取出a的内存地址，将是a参数传进去
    fmt.Println(a) //打印a
}

func A(a *int) { // 函数中就是int型的指针变量a
    *a = 2           // 指针int类型的a，在操作都需要取出来它的值，来进行一个操作
    fmt.Println(*a) //打印a指针
```

```
16. }
```

```
/home/jiemin/code/GOlang/go/src/function/function [/home/jiemin/code/GOlang/go/src/function]
2
2
成功: 进程退出代码 0.
```

这个时候函数a，就可以改变原始函数A的值了，这就是一个值类型和引用类型的不同之处，一个是拷贝值，另一个是拷贝地址，在go语言当中，函数也是一种类型。比如说这里讲函数A改造一下。

```
1.  package main
2.
3.  import (
4.      "fmt"
5.  )
6.
7.  func main() {
8.      a := A //创建一个a变量等于函数A
9.      a()    // 调用这个小a
10. }
11.
12. func A() { // 函数A中，没有参数也没有返回值
13.     fmt.Println("Func A") //打印字符串Func A
14. }
```

```
/home/jiemin/code/GOlang/go/src/function/function [/home/jiemin/code/GOlang/go/src/function]
Func A
成功: 进程退出代码 0.
```

调用小a 也会打印出来Func A。说明其实 a() 就是 A 的复制品， 它就是A的类型。这就是一个函数类型的使用。所以说在go语言当中，一切皆类型。

2、匿名函数

像我们这里func 后面紧跟着是这个函数的名称，这个函数不是一个匿名函数。
什么是匿名函数呢？

```
1.  package main
2.
3.  import (
4.      "fmt"
5.  )
6.
7.  func main() {
8.      a := func() { // 匿名函数是没有名称的，创建一个a变量是匿名函数
9.          fmt.Println("Func A")
10.     }
11.     a() // 调用这个小a
12. }
13.
14. func A() { // 函数A中，没有参数也没有返回值
15.     fmt.Println("Func A") //打印字符串Func A
16. }
```

```
/home/jiemin/code/GOlang/go/src/function/function  [/home/jiemin/code/GOlang/go/src/function]
Func A
成功: 进程退出代码 0.
```

这个时候，这个代码块`func() { fmt.Println("Func A") }`本身是一个函数，没有给他命名名称，但是将它变成了函数类型，然后赋值给了a，a是这种类型的变量，所以可以把`a()`当成函数一样调用它。`func() { fmt.Println("Func A") }`就称之为匿名函数。

3、闭包

闭包解释：

闭包是指可以包含自由（未绑定到特定对象）变量的代码块；这些变量不是在这个代码块内或者任何全局上下文中定义的，而是在定义代码块的环境中定义（局部变量）。“闭包”一词来源于以下两者的结合：要执行的代码块（由于自由变量被包含在代码块中，这些自由变量以及它们引用的对象没有被释放）和为自由变量提供绑定的计算环境（作用域）。在PHP、Scala、Scheme、Common Lisp、Smalltalk、Groovy、JavaScript、Ruby、 Python、Go、Lua、objective c、swift 以及 Java（Java8及以上）等语言中都能找到对闭包不同程度的支持。

拓扑概念编辑

集合A的闭包定义为所有包含A的闭集之交。A的闭包是包含A的最小闭集。

本质

集合 S 是闭集当且仅当 Cl(S)=S（这里的cl即closure，闭包）。特别的，空集的闭包是空集，X 的闭包是 X。集合的交集的闭包总是集合的闭包的交集的子集（不一定是真子集）。有限多个集合的并集的闭包和这些集合的闭包的并集相等；零个集合的并集为空集，所以这个命题包含了前面的空集的闭包的特殊情况。无限多个集合的并集的闭包不一定等于这些集合的闭包的并集，但前者一定是后者的父集。

若 A 为包含 S 的 X 的子空间，则 S 在 A 中计算得到的闭包等于 A 和 S 在 X 中计算得到的闭包（$Cl_A(S) = A \cap Cl_X(S)$）的交集。特别的，S在 A 中是稠密的，当且仅当 A 是 $Cl_X(S)$ 的子集。

度量空间中的

对欧几里德空间的子集 S，x 是 S 的闭包点，若所有以 x 为中心的开球都包含 S 的点（这个点也可以是 x）。

这个定义可以推广到度量空间 X 的任意子集 S。具体地说，对具有度量 d 的度量空间 X，x 是 S 的闭包点，若对所有 r > 0，存在 y 属于 S，使得距离 d(x,y) < r（同样的，可以是 x = y）。另一种说法可以是，x 是 S 的闭包点，若距离 d(x,S) := inf{d(x,s) : s 属于 S} = 0（这里 inf 表示下确界）。

这个定义也可以推广到拓扑空间，只需要用邻域替代“开球”。设 S 是拓扑空间 X 的子集，则 x 是 S 的闭包点，若所有 x 邻域都包含 S 的点。注意，这个定义并不要求邻域是开的。

极限点

闭包点的定义非常接近极限点的定义。这两个定义之间的差别非常微小但很重要——在极限点的定义中，点 x 的邻域必须包含和 x 不同的集合的点。

因此，所有极限点都是闭包点，但不是所有的闭包点都是极限点。不是极限点的闭包点就是孤点。也就是说，点 x 是孤点，若它是 S 的元素，且存在 x 的邻域，该邻域中除了 x 没有其他的点属于 S。

对给定的集合 S 和点 x，x 是 S 的闭包点，当且仅当 x 属于 S，或 x 是 S 的极限点。
集合的闭包
集合 S 的闭包是所有 S 的闭包点组成的集合。S 的闭包写作 cl(S），Cl(S) 或 S−。
性质
cl(S) 是 S 的闭父集。
cl(S) 是所有包含 S 的闭集的交集。
cl(S) 是包含 S 的最小的闭集。
集合 S 是闭集，当且仅当 S = cl(S）。
若 S 是 T 的子集，则 cl(S) 是 cl(T) 的子集。
若 A 是闭集，则 A 包含 S 当且仅当 A 包含 cl(S）。
有时候，上述第二或第三条性质会被作为拓扑闭包的定义。
在第一可数空间（如度量空间）中，cl(S) 是所有点的收敛数列的所有极限。
举例说明编辑
闭包 (closure）是个精确但又很难解释的电脑名词。在 Perl 里面，闭包是以 匿名函数的形式来实现，具有持续参照位于该函数范围之外的文字式变数值的能力。这些外部的文字变数会神奇地保留它们在闭包函数最初定义时的值 （ 深连结）。
如果一个程式语言容许函数递回另一个函数的话 （ 像 Perl 就是），闭包便具有意义。要注意的是，有些语言虽提供匿名函数的功能，但却无法正确处理闭包； Python 这个语言便是一例。如果要想多了解闭包的话，建议你去找本功能性程式 设计的教科书来看。Scheme这个语言不仅支持闭包，更鼓励多加使用。
以下是个典型的产生函数的函数：

```
sub add_function_generator {
return sub { shift + shift };
}
$add_sub = add_function_generator();
$sum = &$add_sub(4,5)； # $sum是 9了
```

闭包用起来就像是个函数样板，其中保留了一些可以在稍後再填入的空格。
add_function_generator() 所递回的匿名函数在技术上来讲并不能算是一个闭包， 因为它没有用到任何位在这个函数范围之外的文字变数。
把上面这个例子和下面这个make_adder()函数对照一下，下面这个函数所递回的匿名函数中使用了一个外部的文字变数。这种指明外部函数的作法需要由 Perl递回一个适当的闭包，因此那个文字变数在匿名函数产生之时的值便永久地被锁进闭包里。

```
sub make_adder {
my $addpiece = shift;
return sub { shift + $addpiece };
}
$f1 = make_adder(20)；
$f2 = make_adder(555)；
```

这样一来&$f1($n) 永远会是 20加上你传进去的值$n ，而&$f2($n) 将 永远会是 555加上你传进去的值$n。$addpiece的值会在闭包中保留下来。
闭包在比较实际的场合中也常用得到，譬如当你想把一些程式码传入一个函数时：

```
my $line;
timeout（30,sub { $line = <STDIN> });
```
如果要执行的程式码当初是以字串的形式传入的话，即'$line = <STDIN>'，那么timeout() 这个假想的函数在回到该函数被呼叫时所在的范围後便无法再撷取$line这个文字变数的值了。

语法结构编辑

Groovy的闭包

闭包（Closure）是Java所不具备的语法结构(JAVA8增加了对闭包的支持)。闭包就是一个代码块，用“{ }”包起来。此时，程序代码也就成了数据，可以被一个变量所引用（与C语言的函数指针比较类似）。闭包的最典型的应用是实现回调函数（callback）。Groovy的API大量使用闭包，以实现对外开放。闭包的创建过程很简单，例如：

```
{ 参数 ->
代码...
}
```
参考下面的例子代码，定义了c1和c2两个闭包，并对它们进行调用：

```
def c1 = { println it }
def c2 = { text -> println text }
c1.call("content1") //用call方法调用闭包
c2("content2") //直接调用闭包
```
“->；”之前的部分为闭包的参数，如果有多个参数，之间可用逗号分割；“->；”之后的部分为闭包内的程序代码。如果省略了“->；”和它之前的部分，此时闭包中代码，可以用名为“it”的变量访问参数。

闭包的返回值和函数的返回值定义方式是一样的：如果有return语句，则返回值是return语句后面的内容；如果没有return语句，则闭包内的最后一行代码就是它的返回值。[1]

环境表达编辑

在Javascript中闭包（Closure)

什么是闭包

“官方”的解释是：所谓“闭包”，指的是一个拥有许多变量和绑定了这些变量的环境的表达式（通常是一个函数），因而这些变量也是该表达式的一部分。

相信很少有人能直接看懂这句话，因为他描述的太学术。我想用如何在Javascript中创建一个闭包来告诉你什么是闭包，因为跳过闭包的创建过程直接理解闭包的定义是非常困难的。看下面这段

代码

```
function a(){
var i=0;
function b(){
alert(++i);
}
return b;
}
var c=a();
c();
```
特点

这段代码有两个特点：
1、函数b嵌套在函数a内部；
2、函数a返回函数b。
这样在执行完var c=a( )后，变量c实际上是指向了函数b，再执行c( )后就会弹出一个窗口显示i的值（第一次为1）。这段代码其实就创建了一个闭包，为什么？因为函数a外的变量c引用了函数a内的函数b，就是说：
当函数a的内部函数b被函数a外的一个变量引用的时候，就创建了一个闭包。
作用
简而言之，闭包的作用就是在a执行完并返回后，闭包使得Javascript的垃圾回收机制不会收回a所占用的资源，因为a的内部函数b的执行需要依赖a中的变量。这是对闭包作用的非常直白的描述，不专业也不严谨，但大概意思就是这样，理解闭包需要循序渐进的过程。
在上面的例子中，由于闭包的存在使得函数a返回后，a中的i始终存在，这样每次执行c（ ），i都是自加1后alert出i的值。
那 么我们来想象另一种情况，如果a返回的不是函数b，情况就完全不同了。因为a执行完后，b没有被返回给a的外界，只是被a所引用，而此时a也只会被b引 用，因此函数a和b互相引用但又不被外界打扰（被外界引用），函数a和b就会被回收。（关于Javascript的垃圾回收机制将在后面详细介绍）
另一个例子
模拟私有变量

```
function Counter(start){
var count = start;
    return{
    increment:function(){
    count++;
    },
    get:function(){
    return count;
    }
    }
    }
    var foo =Counter(4);
    foo.increment();
    foo.get();// 5
```

结果
这里，Counter 函数返回两个闭包，函数 increment 和函数 get。 这两个函数都维持着 对外部作用域 Counter 的引用，因此总可以访问此作用域内定义的变量 count.
文法
objective c的闭包（block）
objective c 中的的闭包，是通过block实现的。Apple在C，Objective-C和C++中扩充了Block这种文法的，并且在GCC4.2中进行了支持。你可以把它理解为函数指针，匿名函数，闭包，lambda表达式，这里暂且用块对象来表述，因为它们之间还是有些许不同的。
声明一个块

如果以内联方式使用块对象，则无需声明。块对象声明语法与函数指针声明语法相似，但是块对象应使用脱字符(^)而非星号指针 (*)。下面的代码声明一个aBlock变量，它标识一个需传入三个参数并具有float返回值的块。

```
float (^aBlock)(const int*, int, float);
```

l 创建一个块

块使用脱字符(^)作为起始标志，使用分号作为结束标志。下面的例子声明一个简单块，并且将其赋给之前声明的block变量（ oneFrom ）。

```
int (^oneFrom)(int);
oneFrom = ^(int anInt) {
return anInt - 1;
};
```

微观世界

如 果要更加深入的了解闭包以及函数a和嵌套函数b的关系，我们需要引入另外几个概念：函数的执行环境（ execution context ）、活动对象（ call object ）、作用域（ scope ）、作用域链（ scope chain ）。以函数a从定义到执行的过程为例阐述这几个概念。

1、当定义函数a的时候，js解释器会将函数a的作用域链（ scope chain ）设置为定义a时a所在的“环境”，如果a是一个全局函数，则scope chain中只有window对象。

2、当函数a执行的时候，a会进入相应的执行环境（ execution context ）。

3、在创建执行环境的过程中，首先会为a添加一个scope属性，即a的作用域，其值就为第1步中的scope chain。即a.scope=a的作用域链。

4、然后执行环境会创建一个活动对象（ call object ）。活动对象也是一个拥有属性的对象，但它不具有原型而且不能通过JavaScript代码直接访问。创建完活动对象后，把活动对象添加到a的作用域链的最顶端。此时a的作用域链包含了两个对象：a的活动对象和window对象。

5、下一步是在活动对象上添加一个arguments属性，它保存着调用函数a时所传递的参数。

6、最后把所有函数a的形参和内部的函数b的引用也添加到a的活动对象上。在这一步中，完成了函数b的的定义，因此如同第3步，函数b的作用域链被设置为b所被定义的环境，即a的作用域。

到此，整个函数a从定义到执行的步骤就完成了。此时a返回函数b的引用给c，又函数b的作用域链包含了对函数a的活动对象的引用，也就是说b可以访问到a中定义的所有变量和函数。函数b被c引用，函数b又依赖函数a，因此函数a在返回后不会被GC回收。

当函数b执行的时候亦会像以上步骤一样。因此，执行时b的作用域链包含了3个对象：b的活动对象、a的活动对象和window对象，如下图所示：

如图所示，当在函数b中访问一个变量的时候，搜索顺序是先搜索自身的活动对象，如果存在则返回，如果不存在将继续搜索函数a的活动对象，依 次查找，直到找到为止。如果整个作用域链上都无法找到，则返回undefined。如果函数b存在prototype原型对象，则在查找完自身的活动对象 后先查找自身的原型对象，再继续查找。这就是Javascript中的变量查找机制。

应用场景

1、保护函数内的变量安全。以最开始的例子为例，函数a中i只有函数b才能访问，而无法通过其他途径访问到，因此保护了i的安全性。

2、在内存中维持一个变量。依然如前例，由于闭包，函数a中i的一直存在于内存中，因此每次执行c（ ），都会给i自加1。

以上两点是闭包最基本的应用场景，很多经典案例都源于此。

回收机制
在Javascript中，如果一个对象不再被引用，那么这个对象就会被GC回收。如果两个对象互相引用，而不再被第3者所引用，那么这两个互相引用的对象也会被回收。因为函数a被b引用，b又被a外的c引用，这就是为什么函数a执行后不会被回收的原因。
匿名内部
在Python中的闭包（Closure)
学过Java GUI编程的人都知道定义匿名内部类是注册监听等处理的简洁有效手段，闭包的定义方式有点类似于这种匿名内部类，
但是闭包的作用威力远远超过匿名内部类，这也是很多流行动态语言选择闭包的原因，相信你在JavaScript中已经了解它的神奇功效了。
定义
如果在一个内部函数里，对在外部作用域（但不是在全局作用域）的变量进行引用，那么内部函数就被认为是闭包（closure）。
简单闭包的例子：
下面是一个使用闭包简单的例子，模拟一个计数器，通过将整型包裹为一个列表的单一元素来模拟使看起来更易变：
函数counter（ ）所作的唯一一件事就是接受一个初始化的值来计数，并将该值赋给列表count成员，然后定义一个内部函数incr（ ）。通过内部函数使用变量count，就创建了一个闭包。最魔法的地方是counter（ ）函数返回一个incr（ ），一个可以调用的函数对象。
运行：

```
>>> c = counter(5)
>>> type(c)
<type 'function'>
>>> print c()6
>>> print c()
7
```

代码格式较重要
代码格式较重要

```
>>> c2 = counter（99）
100
>>> print c()
8
```

离散数学中编辑
“关系”的闭包（Closure)
离散数学中，一个关系R的闭包，是指加上最小数目的有序偶而形成的具有自反性，对称性或传递性的新的有序偶集，此集就是关系R的闭包。
设R是集合A上的二元关系，R的自反（对称、传递）闭包是满足以下条件的关系R'：
（i）R'是自反的（对称的、传递的）；
（ii）R'⊇R；
（iii）对于A上的任何自反（对称、传递）关系R"，若R"⊇R，则有R"⊇R'。
R的自反、对称、传递闭包分别记为r(R）、s(R) 和t(R）。

性质1
集合A上的二元关系R的闭包运算可以复合，例如：
ts(R)=t(s(R))
表示R的对称闭包的传递闭包，通常简称为R的对称传递闭包。而tsr(R)则表示R的自反对称传递闭包。
性质2
设R是集合A上的二元关系，则有
（a）如果R是自反的，那么s(R）和t(R）也是自反的；
（b）如果R是对称的，那么r(R）和t(R）也是对称的；
（c）如果R是传递的，那么r(R）也是传递的。
性质3
设R是集合A上的二元关系，则有
（a）rs(R)=sr(R）；
（b）rt(R)=tr(R）；
（c）ts(R)⊇ st(R）。
Lua中编辑
包
当一个函数内部嵌套另一个函数定义时，内部的函数体可以访问外部的函数的局部变量，这种特征在lua中我们称作词法定界。虽然这看起来很清楚，事实并非如此，词法定界加上第一类函数在编程语言里是一个功能强大的概念，很少语言提供这种支持。
下面看一个简单的例子，假定有一个学生姓名的列表和一个学生名和成绩对应的表；想根据学生的成绩从高到低对学生进行排序，
可以这样做

```
names = {"Peter","Paul","Mary"}
grades = {Mary = 10,Paul = 7,Peter = 8}
table.sort(names,function (n1,n2 )
return grades[n1] > grades[n2] -- compare the grades
end)
```

假设
假定创建一个函数实现此功能：

```
function sortbygrade (names,grades)
table.sort(names,function (n1,n2 )
return grades[n1] > grades[n2] --compare the grades
end)
```

外部局部变量
例子中包含在sortbygrade函数内部的sort中的匿名函数可以访问sortbygrade的参数grades，在匿名函数内部grades不是全局变量也不是局部变量，我们称作外部的局部变量（external local variable）或者upvalue。（upvalue意思有些误导，然而在Lua中他的存在有历史的根源，还有他比起external local variable简短）。
看下面的代码：

```
function newCounter()
```

```
local i = 0
return function() -- anonymous function
i = i + 1
return i
end
end
c1 = newCounter()
print(c1()) --> 1
print(c1()) --> 2
```

匿名函数使用upvalue i保存他的计数，当我们调用匿名函数的时候i已经超出了作用范围，因为创建i的函数newCounter已经返回了。然而Lua用闭包的思想正确处理了这种情况。简单的说，闭包是一个函数以及它的upvalues。如果我们再次调用newCounter，将创建一个新的局部变量i，因此我们得到了一个作用在新的变量i上的新闭包。

```
c2 = newCounter()
print(c2()) --> 1
print(c1()) --> 3
print(c2()) --> 2
```

c1、c2是建立在同一个函数上，但作用在同一个局部变量的不同实例上的两个不同的闭包。

技术上来讲，闭包指值而不是指函数，函数仅仅是闭包的一个原型声明；尽管如此，在不会导致混淆的情况下我们继续使用术语函数代指闭包。

闭包在上下文环境中提供很有用的功能，如前面我们见到的可以作为高级函数（sort）的参数；作为函数嵌套的函数（newCounter）。这一机制使得我们可以在Lua的函数世界里组合出奇幻的编程技术。闭包也可用在回调函数中，比如在GUI环境中你需要创建一系列button，但用户按下button时回调函数被调用，可能不同的按钮被按下时需要处理的任务有点区别。具体来讲，一个十进制计算器需要10个相似的按钮，每个按钮对应一个数字，可以使用下面的函数创建他们：

```
function digitButton (digit)
return Button{ label = digit,
action = function ()
add_to_display(digit)
end
}
end
```

这个例子中我们假定Button是一个用来创建新按钮的工具，label是按钮的标签，action是按钮被按下时调用的回调函数。（实际上是一个闭包，因为他访问upvalue digit）。digitButton完成任务返回后，局部变量digit超出范围，回调函数仍然可以被调用并且可以访问局部变量digit。

闭包在完全不同的上下文中也是很有用途的。因为函数被存储在普通的变量内我们可以很方便的重定义或者预定义函数。通常当你需要原始函数有一个新的实现时可以重定义函数。例如你可以重定义sin使其接受一个度数而不是弧度作为参数：

```
oldSin = math.sin
math.sin = function (x)
return oldSin(x*math.pi/180 )
```

```
end
```
更清楚的方式：
```
do
local oldSin = math.sin
local k = math.pi/180
math.sin = function (x)
return oldSin(x*k)
end
end
```
这样我们把原始版本放在一个局部变量内，访问sin的唯一方式是通过新版本的函数。
利用同样的特征我们可以创建一个安全的环境（也称作沙箱，和java里的沙箱一样），当我们运行一段不信任的代码（比如我们运行网络服务器上获取的代码）时安全的环境是需要的，比如我们可以使用闭包重定义io库的open函数来限制程序打开的文件。
```
do
local oldOpen = io.open
io.open = function (filename,mode)
if access_OK(filename,mode) then
return oldOpen(filename,mode)
else
return nil,"access denied"
end
end
```
二种意义
Scheme中的闭包
其他编程的语言主要采用的是闭包的第二种意义（一个与闭包毫不相干的概念）：闭包也算一种为表示带有自由变量的过程而用的实现技术。但Scheme的术语“闭包”来自抽象代数。在抽象代数里，一集元素称为在某个运算（操作）之下封闭，如果将该运算应用于这一集合中的元素，产生出的仍然是该集合里的元素。
用Scheme的序对举例，为了实现数据抽象，Scheme提供了一种称为序对的复合结构。这种结构可以通过基本过程cons构造出来。过程cons取两个参数，返回一个包含这两个参数作为其成分的复合数据对象。请注意，一个序对也算一个数据对象。进一步说，还可以用cons去构造那种其元素本身就是序对的序对，并继续这样做下去。
```
（define x (cons 1 2)) //构造一个x序对，有1，2组成
（define y (cons 3 4))
(define z (cons x y))
```
Scheme可以建立元素本身也算序对的序对，这就是表结构得以作为一种表示工具的根本基础。我们将这种能力称为cons的闭包性质。一般说，某种组合数据对象的操作满足闭包性质，那就是说，通过它组合起数据对象得到的结果本身还可以通过同样的操作再进行组合。闭包性质是任何一种组合功能的威力的关键要素，因为它使我们能够建立起层次性结构，这种结构由一些部分构成，而其中的各个部分又是由它们的部分构成，并且可以如此继续下去。

一个比较经典的闭包操作：

代码详解：

有一个命名为`func closure()`的函数，它有一个参数`(x int)` 它会返回一个函数(`func(int) int`)，这个`func(int)`函数的它的返回值类型是`int`，`func(int) int` 。这个闭包函数作用是什么呢？这个闭包函数的作用是返回一个匿名函数，这里就进行一个`return`，`return`返回一个`func` 接受一个int类型的参数，返回一个int类型的函数 `(int)int` ,这里就需要用这个参数命名`func(y int) int{}` 。这个匿名函数返回什么呢？它返回 `return x + y` ,这时候就会问了，这里只有一个参数y，那么x在哪里来呢？x它是从外围函数`func closure(x int)` 中的这个x参数 中得来的。在main函数中声明一个参数f,它是通过调用外围`closure` 这个函数，里面传进去一个数值10，比如这样写`f := closure(10)` ，这个时候它会`return`一个匿名函数`func(y int) int { return x + y}`，那么根据刚才的实例，这个f，其实就是类似刚才的小a，它是一个这种类型`func(y int) int { return x + y}`的变量，它代表一个函数。然后我们来调用一下f这个函数，打印一下 `fmt.Println(f())`，我们进入进去一个数值`fmt.Println(f(1))` 和 `fmt.Println(f(2))` ，我们看到打印出来11和12。

很显然，这里的`x` 它用到了我们一开始传入进去的`closure(10)` 的10，所以说`x` 它一直都是10，之后又传入的`fmt.Println(f(1))`和`fmt.Println(f(2))`，怎么证明我们这样就形成一个闭包呢？那么可以打印一下这个地址。我们

可以在`return func(y int) int { fmt.Printf("%p\n", &x) return x + y }`加入`fmt.Printf("%p\n", &x)` 这个一行代码，打印x参数的内存地址。在匿名函数中打印这个x地址，同时也需要在闭包当中也需要打印一

下`func closure(x int) func(int) int { fmt.Printf("%p\n", &x) return func(y int) int { fmt.Printf("%p\n", &x) return x + y }}`

加入`fmt.Printf("%p\n", &x)`在来看一下。可以看到打印出来三次x的内存地址。哪三次？第一次使我们调用`f := closure(10)`这个外层函数打印`fmt.Printf("%p\n", &x)`出来的x的地址。第二次是`f(1)` 这里调用它。打印出来x的地址。第三次是在`f(2)`这里调用它打印出来x的地址。可以看到三次x的地址都是一样的。所以说我们三次调用的时候指向x，都是同一个x，而不是这个值的拷贝。而是直至指向x的原始地址。

```go
package main

import (
    "fmt"
)

func main() {
    f := closure(10)  // 定一个参数，把外层函数closure 赋值给f,它会返回一个匿名函数
    fmt.Println(f(1)) //调用这个f函数，传入数值
    fmt.Println(f(2)) //调用这个f函数，传入数值
}

func closure(x int) func(int) int { /* 创建closure函数 闭包，床将一个int型参数x,
    它返回一个函数，一个参数是int,返回值是int，它的作用是返回一个匿名函数*/
    fmt.Printf("%p\n", &x) // 打印x 的内存地址
    //return func (int)int  // 返回是接收一个int的参数，返回是int类型的函数
    return func(y int) int { // 为这个int参数命名为y
        fmt.Printf("%p\n", &x) // 打印x 的内存地址
```

```
        return x + y            // 这个匿名函数返回的是x+y,这个x是在哪里来的呢？
        // 这个x 就是从外层函数x得来的
    }
}
```

```
/home/jiemin/code/GOlang/go/src/function/function  [/home/jiemin/code/GOlang/go/src/function]
0xc42000a340
0xc42000a340
11
0xc42000a340
12
成功: 进程退出代码 0.
```

defer

- 的执行方式类似其它语言中的析构函数，在函数体执行结束后按照调用顺序的相反顺序逐个执行
- 即使函数发生严重错误也会执行
- 支持匿名函数的调用
- 常用于资源清理、文件关闭、解锁以及记录时间等操作
- 通过与匿名函数配合可在return之后修改函数计算结果
- 如果函数体内某个变量作为defer时匿名函数的参数，则在定义defer时即已经获得了拷贝，否则则是引用某个变量的地址

- Go 没有异常机制，但有 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

先进后出，后进先出的原则。

我们先打印一个a，然后使用defer这个关键字，也打印一下，打印b。再来一个defer关键字打印一个c。我们来看一下结果

```
package main

import (
    "fmt"
)

func main() {
    fmt.Println("a")
    defer fmt.Println("b")
    defer fmt.Println("c")
}
```

```
/home/jiemin/code/GOlang/go/src/function/function  [/home/jiemin/code/GOlang/go/src/function]
a
c
b
成功: 进程退出代码 0.
```

它打印的顺序是a c b。正常打印的顺序应该是a b c，它打印 a c b 呢。就是刚才说的 后定义的先调

用。它是逆序向上调用。所以它先调用了`defer fmt.Println("c")` 打印c，然后在调用`defer fmt.Println("b")`打印b。

下面用一个循环的例子

```
1.  package main
2.
3.  import (
4.      "fmt"
5.  )
6.
7.  func main() {
8.      for i := 0; i < 3; i++ {
9.          defer fmt.Println(i)
10.     }
11. }
```

```
/home/jiemin/code/GOlang/go/src/function/function [/home/jiemin/code/GOlang/go/src/function]
2
1
0
成功: 进程退出代码 0.
```

打印出来是2 1 0，赋值进入的是0 1 2，这就是defer的普通调用

在来一个例子：

`defer func (){fmt.Println(i)}()`

最后俩个括号是什么意思呢？它是调用这个函数。比如说上面的代码里面有大 A() 的时候，后面都要加上一对小括号，没有参数就是一对空着的小括号，有参数的呢，将参数列表写到这对小括号里面当中。所以说defer它的作用是调用某一个函数。所以必须要在后面加上一对小括号，这样看起来肯定有点奇怪，前面讲过的`func() { fmt.Println(i) }`将这个类型赋值给某一个变量。可以这样理解成`defer a()` 。代码里面`defer func() { fmt.Println(i) }()`只不过是一个比较大的整体。

```
1.  package main
2.
3.  import (
4.      "fmt"
5.  )
6.
7.  func main() {
8.      for i := 0; i < 3; i++ {
9.          defer func() { fmt.Println(i) }() //最后俩个括号是什么意思呢？它是调用这个函数。
10.     }
11. }
```

```
1.  package main
2.
3.  import (
4.      "fmt"
5.  )
6.
```

```
func main() {
    for i := 0; i < 3; i++ {
        defer func() {
            fmt.Println(i)
        }() //最后俩个括号是什么意思呢？它是调用这个函数。
    }
}
```

```
/home/jiemin/code/GOlang/go/src/function/function  [/home/jiemin/code/GOlang/go/src/function]
3
3
3
成功: 进程退出代码 0.
```

我们和上面的代码的区别是，上面代码直接打印`fmt.Println(i)`的值。前面这个代码把`fmt.Println(i)`放在一个匿名函数当中。来打印这个 i 的值。我们看到输出结果。三次输出都是 3，那么这个又是怎么回事呢？这段代码`defer func() { fmt.Println(i) }()`实际上用到一个闭包。闭包呢，它就是`fmt.Println(i)`这个 i 就是一直在引用`i := 0; i < 3; i++` 这个 i 的局部变量。而我们上段代码直接打印`fmt.Println(i)` 这个 i 作为参数直接传入进去，所以说呢，它在运行到`defer fmt.Println(i)`语句的时候呢，它就已经对这个 i 的值进行一个拷贝。所以输出 2 1 0，而在`defer func() { fmt.Println(i) }()`这个时候，这个 i 它一直都是作为地址的引用，引用了`i := 0; i < 3; i++`这个局部变量。所以说它在退出这个循环体的时候，这个 i 实际上已经点成了 3，在这个main函数return的时候，开始执行这个defer语句，defer语句当时这个 i 就是引用了这个3的情况，所以三次打印出来都是3，这个就是defer与闭包使用的需要注意的地方。

- Go 没有异常机制，但有 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

来引发一个panic 来进行recover的列子：
创建A() B() C()函数，使用main主函数进行调用。在B()函数中引发panic，但是并没有使用recover进行处理。程序执行到B()函数时候遇到panic时候就会终止。输出结果：

```
package main

import (
    "fmt"
)

func main() {
    A()
    B()
    C()
}

func A() {
    fmt.Print("Func A\n")
}

func B() {
```

```
18.     panic("Panic in B\n")    //使用panic引发程序崩溃退出
19. }
20. 
21. func C() {
22.     fmt.Print("Func C\n")
23. }
```

```
/home/jiemin/code/GOlang/go/src/function/function  [/home/jiemin/code/GOlang/go/src/function]
Func A
panic: Panic in B


goroutine 1 [running]:
panic(0x48a560, 0xc42000a360)
        /usr/local/go/src/runtime/panic.go:500 +0x1a1
main.B()
        /home/jiemin/code/GOlang/go/src/function/function.go:18 +0x6d
main.main()
        /home/jiemin/code/GOlang/go/src/function/function.go:9 +0x19
错误: 进程退出代码 2.
```

那我们怎么让程序恢复呢？那么在B()函数中，进行一个recover。recover只有在defer调用的函数有效。所以要使用defer关键字来调用一个函数。`defer func(){}` 这里又是一个匿名函数。在这个defer函数中要进行一个check，check这个panic这个是否存在。那么用到if `err := recover(){}`这个语句，我们调用这个 **recover()** 它就会返回一个值。返回这个panic的间隙，如果说它 **recover()**返回不是new的话，说明引发了panic，程序属于恐慌状态了。如果返回的是new的话。说明就什么都不需要做。还需要做一个判断`err != nil`是否等于new，确实不等于new的话，确实引发了panic，那么recover是有效的。在这里输出一下Recover in B，输出一下：

```
1.  package main
2.  
3.  import (
4.      "fmt"
5.  )
6.  
7.  func main() {
8.      A()
9.      B()
10.     C()
11. }
12. 
13. func A() {
14.     fmt.Print("Func A\n")
15. }
16. 
17. func B() {
18.     panic("Panic in B\n") //使用panic引发程序崩溃退出
19.     defer func() {
20.         if err := recover(); err != nil { // check一下panic是否存在
21.             /*调用这个 recover() 它就会返回一个值。返回这个panic的间隙，
22.             如果说它 recover()返回不是new的话，说明引发了panic，程序属于恐慌状态了。
23.             如果返回的是new的话。说明就什么都不需要做。
24.             还需要做一个判断err != nil是否等于new，
25.             确实不等于new的话，确实引发了panic，那么recover是有效的*/
26.             fmt.Println("Recover in B\n")
```

```
        }
    }()
}

func C() {
    fmt.Print("Func C\n")
}
```

```
/home/jiemin/code/GOlang/go/src/function/function [/home/jiemin/code/GOlang/go/src/function]
Func A
panic: Panic in B


goroutine 1 [running]:
panic(0x48a560, 0xc42000a360)
        /usr/local/go/src/runtime/panic.go:500 +0x1a1
main.B()
        /home/jiemin/code/GOlang/go/src/function/function.go:18 +0x6d
main.main()
        /home/jiemin/code/GOlang/go/src/function/function.go:9 +0x19
错误: 进程退出代码 2.
```

执行错误，还有一个要注意的地方，我们的defer语句不应该要放到panic之后的。应该放在panic语句之前，为什么呢？当程序执行到panic语句，它输出了Panic in B，在这个时候程序已经发生报错，不在执行了，而我们还需要defer语句当中的这个recover来进行一个恢复。但是它执行到了panic，它就不执行了。也就是说在函数B当中，它根本就没有注册这个defer函数，所以说，defer函数必须要放到panic语句之前。对defer函数进行一个注册。它在panic的时候，我已经知道我有了defer函数了，所以说这个时候，它才会执行这个defer函数，这个defer函数当中就进行了一个recover，现在来输出一个结果：

```
package main

import (
    "fmt"
)

func main() {
    A()
    B()
    C()
}

func A() {
    fmt.Print("Func A\n")
}

func B() {
    defer func() {
        if err := recover(); err != nil { // check一下panic是否存在
            /*调用这个 recover() 它就会返回一个值。返回这个panic的间隙，
            如果说它 recover()返回不是new的话，说明引发了panic，程序属于恐慌状态了。
            如果返回的是new的话。说明就什么都不需要做。
            还需要做一个判断err != nil是否等于new，
            确实不等于new的话，确实引发了panic，那么recover是有效的*/
            fmt.Println("Recover in B\n")
```

```
        }
    }()
    panic("Panic in B") //使用panic引发程序崩溃退出
}

func C() {
    fmt.Print("Func C\n")
}
```

```
/home/jiemin/code/GOlang/go/src/function/function [/home/jiemin/code/GOlang/go/src/function]
Func A
Recover in B

Func C
成功: 进程退出代码 0.
```

看到了Recover in B，Panic in B 就没有输出了，在输出了Recover in B，有执行了C()函数。通过defer函数将程序在panic状态进行recover回来。恢复到正常运行状态。

课堂作业

- 运行以下程序并分析输出结果。

```
func main() {
    var fs = [4]func(){}

    for i := 0; i < 4; i++ {
        defer fmt.Println("defer i = ", i)
        defer func() { fmt.Println("defer_closure i = ", i) }()
        fs[i] = func() { fmt.Println("closure i = ", i) }
    }

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


```
func main() {
    var fs = [4]func(){}

    for i := 0; i < 4; i++ {
        defer fmt.Println("defer i = ", i)
        defer func() { fmt.Println("defer_closure i = ", i) }()
        fs[i] = func() { fmt.Println("closure i = ", i) }
    }

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


作业：

```
package main

import (
    "fmt"
```

```
)

func main() {
    var fs = [4]func(){}

    for i := 0; i < 4; i++ {
        defer fmt.Println("defer i = ", i)
        defer func() {
            fmt.Println("defer_closure i = ", i)
        }()
        fs[i] = func() {
            fmt.Println("closure i = ", i)
        }
    }

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

```
/home/jiemin/code/GOlang/go/src/function/function  [/home/jiemin/code/GOlang/go/src/function]
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
成功: 进程退出代码 0.
```

首先输出是闭包 i 等于4，它输出的四个都是等于4。我们看这个`fs[i] = func() { fmt.Println("closure i = ", i)}`是这个函数。首先在这个for循环当中，将这些function 类型`func() { fmt.Println("closure i = ", i)}`匿名函数存到 `fs[i]` 类型的selice当中。然后我们在`for i := 0; i < 4; i++ { defer fmt.Println("defer i = ", i) defer func() { fmt.Println("defer_closure i = ", i) }() fs[i] = func() { fmt.Println("closure i = ", i) }}` 循环结束在用一个for循环`for _, f := range fs { f() }` for range语句来对这个函数fs 进行一个调用。然后输出这个 i 的值。但是为什么都是4呢？如果按照正常思想不知道闭包呢，就会坚定的认为就应该是0 1 2 3。但是这里就是用到了一个闭包的思想。怎么看呢？我们看到`func() { fmt.Println("closure i = ", i)}`这里是一个匿名函数，然后呢我们这个`func()` 里面也没有参数，而是直接使用了`func() { fmt.Println("closure i = ", i)}`这个 i ，但是明显的看到`fmt.Println("closure i = ", i)` 在这个匿名函数当中并没有在那个地方定义这个 i 。所以说这个 i 就是从它的外层函数当中夺过来的。所以就得到了一个引用地址。那么这个引用地址是什么呢？就是这个for循环`for i := 0; i < 4; i++` 当中的这个 i 的引用地址。 在最后执行完`for i := 0; i < 4; i++ { defer fmt.Println("defer i = ", i) defer func() { fmt.Println("defer_closure i = ", i) }() fs[i] = func() { fmt.Println("closure i = ", i) }}` 这个for循环之后它这个 i 实际上就等于4了。然后在来输出`for _, f := range fs { f() }` 这个 i 的

值。也就相当于输出了这个 i 的内存地址所存的那个值，那么显然它存的值都是4。

接下来看`defer func() { fmt.Println("defer_closure i = ", i) }()`这个匿名函数。同样它这里`func()`也没有参数，同样也用到`fmt.Println("defer_closure i = ", i)` 这个 i ，所以说它也是得到了这个 i 的内存地址，那不同的呢，它是通过defer 定义的。所以说它是先进后出，后进先出。因为这里输出都是4，但是我们要知道它这里还是使用了一个闭包的思想。

最后来看第一个`defer fmt.Println("defer i = ", i)` ，这里并没有使用匿名函数，而是直接使用了 `fmt.Println("defer i = ", i)` 然后将这个 i 作为了一个参数。要注意：这里这个 i 已经变成了一个参数，它遵循正常的一个规则。什么规则呢？那就正常的值拷贝。因为 i 是什么类型？ i 是int 类型，它遵循一个正常规则它是值拷贝。所以说在 `defer` 定义的时候，它已经得到了这个 i 的值拷贝。在输出的时候，它输出的是 0 1 2 3 。因为是`defer`定义的那么它就是逆序输出是 3 2 1 0 。