|  |
| --- |
| <https://tiancaiamao.gitbooks.io/go-internals/content/zh/03.4.html> |
| <https://studygolang.com/articles/5281> |
| <https://studygolang.com/articles/280> |
| <https://studygolang.com/articles/742> |

Golang的defer行为规则：

|  |
| --- |
| 规则一：延迟函数的参数在defer语句出现时就已经确定  Func a() {  i := 0  defer fmt.Println(i)  i++  return  }  defer语句中fmt.Println()参数值i在defer出现时就已经确定下来，实际上是一份拷贝，后面对变量的修改不会影响defer函数的执行 即任然打印0  注意：对于指针类型参数，规则任然适用，只不过延迟函数的参数时一个地址值，这种情况下defer后面的语句对变量的修改可能会影响延迟函数 |
| 规则二：延迟函数执行按后进先出顺序执行:G中保存一个defer链表，通过链表来实现defer的先进后出 |
| 规则三：延迟函数可能操作主函数的具名返回值  定义defer的函数，即主函数可能有返回值，返回值有没有名字没有关系，defer所作用的函数，即延迟函数可能会影响到返回值  Golang中return不是一个原子操作，实际上return只代理汇编指令ret，即跳转程序执行，比如语句return i，实际上是 将i值存入栈中作为返回值，然后执行跳转，而defer执行时机正是跳转前，所以说defer执行时还是有机会操作返回值 |
| 返回值操作：   1. 主函数拥有匿名返回值，返回字面值： 一个主函数拥有一个匿名的返回值，返回时使用字面值，比如 1 2 “hello”这样的值，这种情况下defer语句是无法操作返回值的。 2. 主函数拥有匿名返回值，返回变量：一个主函数拥有一个匿名的返回值，返回使用本地或全局变量，这种情况下defer语句可以引用到返回值，但不会改变返回值   例如：  Func foo() int {  Var i int  Defer func() {  I++  }()  Return i  }  上面函数返回一个局部变量，同时defer函数会操作这个局部变量，对于匿名返回值来说，可以假定任然有一个变量存储返回值：上述为例，假定返回值变量为 “anony”  拆分为：  anony = i  i++  return  由于将i拷贝给anony，所以defer语句中修改i值，对函数返回值不造成影响   1. 主函数拥有具名返回值：主函数声明语句中带名字的返回值，会初始化成一个局部变量，函数内部可以像使用局部变量一样使用该返回值   func foo() (ret int) {  defer func() { ret++ }()  return 0  }  上述函数可拆解为:  ret = 0  ret++  return |

Golnag中，defer代码块会在函数调用链表中增加一个函数调用，这个函数不是普通函数调用，而是会在函数正常返回，也就是return之后添加一个函数调用。因此，defer通常用来释放函数内部变量。

Defer的思想类似于C++中的析构函数，不过Go语言中“析构”的不是对象，而是函数，defer就是用来添加函数结束时执行的语句。注意是添加，而是不是指定，因为不同于C++中的析构函数时静态的，Go中的defer是动态的。

例1：

func f() (result int) {

defer func() {

result++

}()

return 0

}

例2：

func f() (r int) {

t := 5

defer func() {

t = t + 5

}()

return t

}

例3：

func f() (r int) {

defer func(r int) {

r = r + 5

}(r)

return 1

}

defer是在return之前执行的。这个在 [官方文档](http://golang.org/ref/spec" \l "defer_statements" \t "_blank)中是明确说明了的。要使用defer时不踩坑，最重要的一点就是要明白，**return xxx这一条语句并不是一条原子指令!**

函数返回的过程是这样的：先给返回值赋值，然后调用defer表达式，最后才是返回到调用函数中。defer表达式可能会在设置函数返回值之后，在返回到调用函数之前，修改返回值。

其实使用defer时，用一个简单的转换规则改写一下。改写规则是将return语句拆成两句写，return xxx会被改写成:

返回值 = xxx

调用defer函数

空的return

先看例1，它可以改写成这样：

func f() (result int) {

result = 0 //return语句不是一条原子调用，return xxx其实是赋值＋ret指令

func() { //defer被插入到return之前执行，也就是赋返回值和ret指令之间

result++

}()

return

}

所以这个返回值是1。

再看例2，它可以改写成这样：

func f() (r int) {

t := 5

r = t //赋值指令

func() { //defer被插入到赋值与返回之间执行，这个例子中返回值r没被修改过

t = t + 5

}

return //空的return指令

}

所以这个的结果是5。

最后看例3，它改写后变成：

func f() (r int) {

r = 1 //给返回值赋值

func(r int) { //这里改的r是传值传进去的r，不会改变要返回的那个r值

r = r + 5

}(r)

return //空的return

}

所以这个例子的结果是1。

defer确实是在return之前调用的。但表现形式上却可能不像。本质原因是return xxx语句并不是一条原子指令，defer被插入到了赋值 与 ret之间，因此可能有机会改变最终的返回值。