Go语言学习(七)闭包和错误处理

原创 2016年03月26日 20:30:58 标签：go语言 708

1.闭包

Go语言中的闭包同样也会引用到函数外的变量。闭包的实现确保只要闭包还被使用，那么

被闭包引用的变量会一直存在，例如:

1

2

3

package main

import "fmt"

func main(){

var j int = 5

a := func()(func()) { //圆括号中的func()表示返回值是一个func()函数

var i int = 10

return func() { //这里返回一个匿名函数

fmt.Printf("i, j: %v, %v\n", i, j)

}

}() //花括号后带参数列表表示调用匿名函数,执行到这里变量a就等于了一个函数了.

a() //调用函数a

j \*= 2 //修改函数外部的变量j

a() //再次调用函数a

}

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

运行结果:

i, j: 10, 5

i, j: 10, 10

1

2

3

4

在上面的例子中，变量 a 指向的闭包函数引用了局部变量 i 和 j ， i 的值被隔离，在闭包外不

能被修改，改变 j 的值以后，再次调用 a ，发现结果是修改过的值。

在变量 a 指向的闭包函数中，只有内部的匿名函数才能访问变量 i ，而无法通过其他途径访问

到，因此保证了 i 的安全性。

1

2

3

4

5

2.错误处理

2.1 error接口

Go语言引入了一个关于错误处理的标准模式，即 error 接口，该接口的定义如下：

type error interface {

Error() string

}

创建error通常如下:

var e error = errors.New("...")//需要使用使用errors包

对于大多数函数，如果要返回错误，大致上都可以定义为如下模式，将 error 作为多种返回

值中的最后一个，但这并非是强制要求：

1

2

3

4

5

6

7

8

9

10

func Foo(param int)(res int,err error){

//....

}

1

2

3

调用时的代码建议按如下方式处理错误情况：

1

2

n, err := Foo(0)

if err != nil {

// 错误处理

} else {

// 使用返回值n

}

1

2

3

4

5

6

2.2 defer关键字

Go语言中有种不错的设计，即延迟（defer）语句，你可以在函数中添加多个defer语句。当函数执行到最后时，

这些defer语句会按照逆序执行，最后该函数返回。特别是当你在进行一些打开资源的操作时，遇到错误需要提前返回，

在返回前你需要关闭相应的资源，不然很容易造成资源泄露等问题。

1

2

3

4

func ReadWrite() bool {

file.Open("file")

// 做一些工作

if failureX {

file.Close()

return false

}

if failureY {

file.Close()

return false

}

file.Close()

return true

}

1

2

3

4

5

6

7

8

9

10

11

12

13

14

我们看到上面有很多重复的代码，Go的defer有效解决了这个问题。使用它后,不但代码量减少了很多，

而且程序变得更优雅。

1

2

3

func ReadWrite() bool {

file.Open("file")

defer file.Close() //保证资源正常关闭

if failureX {

return false

}

if failureY {

return false

}

return true

}

1

2

3

4

5

6

7

8

9

10

11

如果有很多调用defer，那么defer是采用后进先出模式

1

2

for i := 0; i < 5; i++ {

defer fmt.Printf("%d ", i) //输出结果:4 3 2 1 0

}

1

2

3

defer有点类似java中的try{}finall{}

1

2

2.3 panic和recover函数

Go语言有2个内置的函数panic()和recover()，用以报告和捕获运行时发生的程序错误，与error不同，

panic和recover一般用在函数内部。一定要注意不要滥用panic和recover，可能会导致性能问题，

一般只在未知输入和不可靠请求时使用。

Go语言的错误处理流程：当一个函数在执行过程中出现了异常或遇到 panic()，正常语句就会立即终止，

然后执行 defer 语句，再报告异常信息，最后退出 goroutine。如果在defer中使用了recover()函数,

则会捕获错误信息，使该错误信息终止报告。如下示例,例子来自网络

1

2

3

4

5

6

7

8

package main

import (

"log" //log包

"strconv" //字符转换包

)

//捕获因未知输入导致的程序异常

func catch(nums ...int) int {

defer func() {

//recover()可以捕获运行时发生的异常,避免异常时程序直接over,通常用在defer函数内

if r := recover(); r != nil {

log.Println("[E]", r) //将捕获的异常信息通过log打印,而不会导致程序挂掉

}

}()

return nums[1] \* nums[2] \* nums[3] //index out of range

}

//主动抛出 panic，不推荐使用，可能会导致性能问题

func toFloat64(num string) (float64, error) {

defer func() {

if r := recover(); r != nil {

log.Println("[W]", r)

}

}()

if num == "" {

panic("param is null") //主动抛出 panic

}

return strconv.ParseFloat(num, 10)

}

func main() {

catch(2, 8)

toFloat64("")

}

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

运行结果:

2016/03/26 20:16:03 [E] runtime error: index out of range

2016/03/26 20:16:03 [W] param is null

1

2

3

4

最后补充下nil的介绍:

golang的nil在概念上和其它语言的null、None、nil、NULL一样，都指代零值或空值。

nil是预先说明的标识符，也即通常意义上的关键字。在golang中，nil只能赋值给指针、

channel、func、interface、map或slice类型的变量。如果未遵循这个规则，则会引发panic。

1

2

3

4

文章来源：http://blog.csdn.net/mchenys/article/details/50988701

版权声明：本文为博主原创文章，未经博主允许不得转载。