Go异常处理——defer、panic、recover

目录：

1. Go语言中的异常处理
2. defer延迟函数
3. defer延迟方法
4. defer延迟参数
5. panic及recover

一、 延迟是什么?

- defer即延迟语句，极个别的情况下，Go才使用defer、panic、recover这种异常处理形式。
- defer可以延迟函数、延迟方法、延迟参数。

（一）、延迟函数

1、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
- 如果有很多调用defer，那么defer是采用后进先出模式
- 在离开所在的方法时，执行（报错的时候也会执行）

2、示例代码1:

```
package main
import "fmt"

func main() {
  defer funA()
  funB()
  funC()
```

```
    fmt.Println("main...over....")
}

func funA() {
    fmt.Println("我是funA()...")
}

func funB() { //
    fmt.Println("我是funB()...")
}

func funC() {
    fmt.Println("我是funC()。。")
}
```

运行结果：

```
我是funB()...
我是funC()。。
main...over....
我是funA()...
```

3、示例代码2：

```
package main
import "fmt"

func main() {
    s1 := []int{78, 109, 2, 563, 300}
    largest(s1)
}

func finished() {
    fmt.Println("结束！")
}
```

```
func largest(s []int) {
  defer finished()
  fmt.Println("开始寻找最大数...")
  max := s[0]
  for _, v := range s {
    if v > max {
      max = v
    }
  }
  fmt.Printf("%v中的最大数为：%v \n", s , max)
}
```

运行结果：

```
开始寻找最大数...
[78 109 2 563 300]中的最大数为：563
结束！
```

（二）、延迟方法

1、延迟并不仅仅局限于函数。延迟一个方法调用也是完全合法的。

2、示例代码：

```
package main
import "fmt"

type person struct {
  firstName string
  lastName  string
}

func (p person) fullName() {
  fmt.Printf("%s %s", p.firstName, p.lastName)
}

func main() {
  p := person{"Steven", "Wang"}
```

```
  defer p.fullName()
  fmt.Printf("Welcome ")
}
```

运行结果：

Welcome Steven Wang

（三）、延迟参数

1、延迟函数的参数在执行延迟语句时被执行，而不是在执行实际的函数调用时执行。

2、示例代码：

```
package main
import "fmt"

func printAdd(a , b int) {
  fmt.Printf("延迟函数中：参数a , b分别为%d,%d ， 两数之和为：%d\n", a , b , a+b)
}
func main() {
  a := 5
  b := 6
  //延迟函数的参数在执行延迟语句时被执行，而不是在执行实际的函数调用时执行。
  defer printAdd(a , b)
  a = 10
  b = 7
  fmt.Printf("延迟函数执行前：参数a , b分别为%d,%d ， 两数之和为：%d\n", a , b , a+b)
}
```

运行结果：

延迟函数执行前：参数a , b分别为10,7 ， 两数之和为：17

延迟函数中：参数a , b分别为5,6 ， 两数之和为：11

（四）、堆栈的推迟

1、当一个函数有多个延迟调用时，它们被添加到一个堆栈中，并在Last In First Out（LIFO）后进先出的顺序中执行。

2、示例代码：利用defer实现字符串倒序。

```
package main
import "fmt"

func main() {
  name := "StevenWang欢迎学习区块链"
  fmt.Printf("原始字符串: %s\n", name)
  fmt.Println("翻转后字符串: ")
  ReverseString(name)
}

func ReverseString(str string) {
  for _, v := range []rune(str) {
    defer fmt.Printf("%c", v)
  }
}
```

返回结果：

```
原始字符串: StevenWang欢迎学习区块链
翻转后字符串:
链块区习学迎欢gnaWnevetS
```

（五）、延迟的应用【后续课程讲解】

1、到目前为止，我们所写的示例代码，并没有实际的应用。现在看一下关于延迟的应用。在不考虑代码流的情况下，延迟被执行。让我们以一个使用WaitGroup的程序示例来理解这个问题。我们将首先编写程序而不使用延迟，然后我们将修改它以使用延迟，并理解延迟是多么有用。

2、示例代码：

```
package main
import (
    "fmt"
    "sync"
)

type rect struct {
    length int
    width  int
}

func (r rect) area(wg *sync.WaitGroup) {
    if r.length < 0 {
        fmt.Printf("rect %v's length should be greater than zero\n", r)
        wg.Done()
        return
    }
    if r.width < 0 {
        fmt.Printf("rect %v's width should be greater than zero\n", r)
        wg.Done()
        return
    }
    area := r.length * r.width
    fmt.Printf("rect %v's area %d\n", r, area)
    wg.Done()
}
func main() {
    var wg sync.WaitGroup
    r1 := rect{-67, 89}
    r2 := rect{5, -67}
    r3 := rect{8, 9}
    rects := []rect{r1, r2, r3}
    for _, v := range rects {
        wg.Add(1)
        go v.area(&wg)
    }
    wg.Wait()
    fmt.Println("All go routines finished executing")
```

}

修改以上代码：

```
package main

import (
    "fmt"
    "sync"
)

type rect struct {
    length int
    width  int
}

func (r rect) area(wg *sync.WaitGroup) {
    defer wg.Done()
    if r.length < 0 {
        fmt.Printf("rect %v's length should be greater than zero\n", r)
        return
    }
    if r.width < 0 {
        fmt.Printf("rect %v's width should be greater than zero\n", r)
        return
    }
    area := r.length * r.width
    fmt.Printf("rect %v's area %d\n", r, area)
}
func main() {
    var wg sync.WaitGroup
    r1 := rect{-67, 89}
    r2 := rect{5, -67}
    r3 := rect{8, 9}
    rects := []rect{r1, r2, r3}
    for _, v := range rects {
        wg.Add(1)
        go v.area(&wg)
    }
    wg.Wait()
    fmt.Println("All go routines finished executing")
```

```
}
```

程序运行结果：

```
rect {8 9}'s area 72
rect {-67 89}'s length should be greater than zero
rect {5 -67}'s width should be greater than zero
All go routines finished executing
```

二、panic和recover（宕机和宕机恢复）

（一）、panic和recover机制

1、概述：

- panic：词义"恐慌"，recover："恢复"
- Go语言追求简洁优雅，Go没有像Java那样的 try...catch...finally 异常处理机制。Go语言设计者认为，将异常与流程控制混在一起会让代码变得混乱。
- Go语言中，使用多值返回来返回错误。不用异常代替错误，更不用异常来控制流程。
- go语言利用panic()，recover()，实现程序中的极特殊的异常处理。换句话说，极个别的情况下，Go才使用defer、panic、recover这种异常处理形式。
    - panic()，让当前的程序进入恐慌，中断程序的执行。或者说，panic是一个内建函数，可以中断原有的控制流程，进入一个令人恐慌的流程中。
    - 当函数F调用panic，函数F的执行被中断，但是F中的延迟函数会正常执行，然后F返回到调用它的地方。在调用的地方，F的行为就像调用了panic。这一过程继续向上，直到发生panic的goroutine中所有调用的函数返回，此时程序退出。
    - 恐慌可以直接调用panic产生。也可以由运行时错误产生，例如访问越界的数组。
    - recover 是一个内建的函数，可以让进入令人恐慌的流程中的goroutine恢复过来。
    - recover()，让程序恢复，必须在defer函数中执行。换句话说，recover仅在延迟函数中有效。
    - 在正常 的执行过程中，调用recover会返回nil，并且没有其它任何效

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。

- 一定要记住，应当把它作为最后的手段来使用，也就是说，我们的代码中应当没有，或者很少有panic这样的东西。

（二）、示例代码

```
package main
import "fmt"

func main() {
  /*
  panic：词义"恐慌",
  recover："恢复"
  go语言利用panic()，recover()，实现程序中的极特殊的异常的处理
    panic(),让当前的程序进入恐慌，中断程序的执行
    recover(),让程序恢复，必须在defer函数中执行
   */
  funA()
  funB()
  funC()
  fmt.Println("main...over....")
}

func funA() {
  fmt.Println("我是函数funA()...")
}

func funB() { //外围函数
  defer func() {
    if msg := recover(); msg != nil {
      fmt.Println(msg, "恢复啦。。。")
    }
  }()
  fmt.Println("我是函数funB()...")
  for i := 1; i <= 10; i++ {
    fmt.Println("i:", i)
    if i == 5 {
      //让程序中断
```

```
        panic("funB函数，恐慌啦。。。") //打断程序的执行。。
      }
    }
    //当外围函数中的代码引发运行恐慌时，只有其中所有的延迟函数都执行完毕后，该运行时恐慌才会真正被扩展至调用函数。
}

func funC() {
    defer func() {
        fmt.Println("func的延迟函数。。。")
        //if msg := recover(); msg != nil {
        // fmt.Println(msg, "恢复啦。。。")
        //}
        fmt.Println("recover执行了" , recover())
    }()
    fmt.Println("我是函数funC()。。")
    panic("funC恐慌啦。。")
}
```