# Go语言异常处理

在前面几章中我们。

在本章中我们将学习一下内容：

* Go语言中的错误处理
* 创建error对象的几种方式
* 自定义错误什么是以太坊
* Go语言中的异常处理
* defer延迟函数
* defer延迟方法
* defer延迟参数
* panic及recover

## 错误处理

### 错误是什么?

1、错误指程序中出现不正常的情况，从而导致程序无法正常执行。

● 大多语言中使用try...catch...finally语句执行。

● 假设我们正在尝试打开一个文件，文件系统中不存在这个文件。这是一个异常情况，它表示为一个错误。

● 不要忽略错误。永远不要忽略一个错误。忽视错误会招致麻烦。让我重新编写一个示例，该示例列出了与模式匹配的所有文件的名称，而忽略了错误处理代码。

2、Go语言中没有try...catch

● Go 语言通过内置的错误类型提供了非常简单的错误处理机制。

● 错误值可以存储在变量中，通过函数中返回。

● 如果一个函数或方法返回一个错误，按照惯例，它必须是函数返回的最后一个值。

● 处理错误的惯用方式是将返回的错误与nil进行比较。

● nil值表示没有发生错误，而非nil值表示出现错误。

● 如果不是nil，需打印输出错误。

3、error错误类型的本质

● error本质上是一个接口类型，其中包含一个Error()方法。

type error interface {

Error() string

}

任何实现这个接口的类型都可以作为一个错误使用。这个方法提供了对错误的描述。

### 创建error对象的几种方式

1、errors包下的New()函数返回error对象

● errors.New()创建新的错误。

● 代码分析

// Package errors implements functions to manipulate errors.

package errors

// New returns an error that formats as the given text.

func New(text string) error {

return &errorString{text}

}

// errorString is a trivial implementation of error.

type errorString struct {

s string

}

func (e \*errorString) Error() string {

return e.s

}

2、fmt包下的Errorf()函数返回error对象

● fmt包下的Errorf()函数本质上还是调用errors.New()

// Errorf formats according to a format specifier and returns the string

// as a value that satisfies error.

func Errorf(format string, a ...interface{}) error {

return errors.New(Sprintf(format, a...))

}

3、创建一个自定义错误。

### 自定义错误

1、实现步骤

● 1、定义一个结构体，表示自定义错误的类型

● 2、让自定义错误类型实现error接口的方法 ：Error() string

● 3、定义一个返回error的函数。根据程序实际功能而定。

2、示例代码：

package main

import (

"fmt"

"time"

)

//1.定义一个结构体，表示自定义错误的类型

type MyError struct {

When time.Time

What string

}

//2、自定义错误类型实现error接口的方法 ：Error() string

func (e \*MyError) Error() string {

return fmt.Sprintf("%v : %v", e.When, e.What)

}

//3.定义一个返回error的函数。求矩形的面积

func getArea(width, length float64) (float64, error) {

errorMsg := ""

if width < 0 && length < 0 {

errorMsg = fmt.Sprintf("长度:%v ，宽度:%v ，均为负数", length, width)

} else if length < 0 {

errorMsg = fmt.Sprintf("长度:%v ，出现负数", length)

} else if width < 0 {

errorMsg = fmt.Sprintf("宽度:%v ，出现负数", width)

}

if errorMsg != "" {

return 0, &MyError{time.Now(), errorMsg}

} else {

return width \* length, nil

}

}

func main() {

res1, err := getArea(-4, -6)

if err != nil {

fmt.Printf(err.Error())

} else {

fmt.Println("面积是:", res1)

}

}

## 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。

### 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！

### 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

### 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

### 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

### 延迟的应用

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

## panic和recover（宕机和宕机恢复）

### 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。

### 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。。")

}