Go 语言接口 Go 语言开发工具

Go 错误处理

Go 语言通过内置的错误接口提供了非常简单的错误处理机制。

error类型是一个接口类型，这是它的定义：

```
type error interface {
    Error() string
}
```

我们可以在编码中通过实现 error 接口类型来生成错误信息。

函数通常在最后的返回值中返回错误信息。使用errors.New 可返回一个错误信息：

```
func Sqrt(f float64) (float64, error) {
    if f < 0 {
        return 0, errors.New("math: square root of negative number")
    }
    // 实现
}
```

在下面的例子中，我们在调用Sqrt的时候传递的一个负数，然后就得到了non-nil的error对象，将此对象与nil比较，结果为true，所以fmt.Println(fmt包在处理error时会调用Error方法)被调用，以输出错误，请看下面调用的示例代码：

```
result, err:= Sqrt(-1)

if err != nil {
   fmt.Println(err)
}
```

实例

```
package main

import (
    "fmt"
)

// 定义一个 DivideError 结构
type DivideError struct {
    dividee int
    divider int
}
```

```
// 实现 `error` 接口
func (de *DivideError) Error() string {
    strFormat := `
    Cannot proceed, the divider is zero.
    dividee: %d
    divider: 0
`
    return fmt.Sprintf(strFormat, de.dividee)
}

// 定义 `int` 类型除法运算的函数
func Divide(varDividee int, varDivider int) (result int, errorMsg string) {
    if varDivider == 0 {
        dData := DivideError{
            dividee: varDividee,
            divider: varDivider,
        }
        errorMsg = dData.Error()
        return
    } else {
        return varDividee / varDivider, ""
    }

}

func main() {

    // 正常情况
    if result, errorMsg := Divide(100, 10); errorMsg == "" {
        fmt.Println("100/10 = ", result)
    }
    // 当被除数为零的时候会返回错误信息
    if _, errorMsg := Divide(100, 0); errorMsg != "" {
        fmt.Println("errorMsg is: ", errorMsg)
    }

}
```

执行以上程序，输出结果为：

```
100/10 =  10
errorMsg is:
    Cannot proceed, the divider is zero.
    dividee: 100
    divider: 0
```

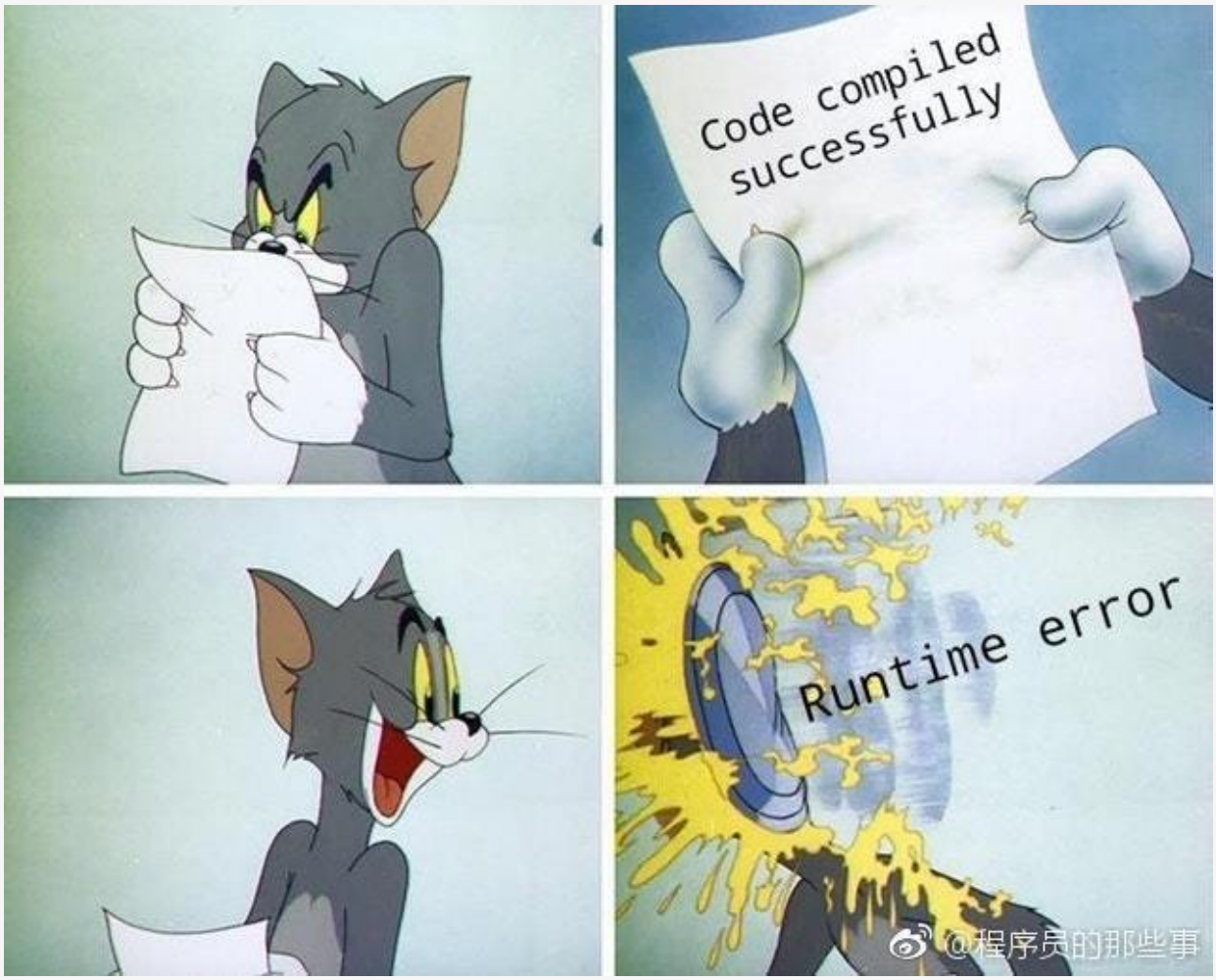
Go 语言接口 Go 语言开发工具

2 篇笔记 写笔记

这里应该介绍一下 panic 与 recover,一个用于主动抛出错误，一个用于捕获panic抛出的错误。

概念

panic 与 recover 是 Go 的两个内置函数，这两个内置函数用于处理 Go 运行时的错误，panic 用于主动抛出错误，recover 用来捕获 panic 抛出的错误。



- 引发panic有两种情况，一是程序主动调用，二是程序产生运行时错误，由运行时检测并退出。
- 发生panic后，程序会从调用panic的函数位置或发生panic的地方立即返回，逐层向上执行函数的defer语句，然后逐层打印函数调用堆栈，直到被recover捕获或运行到最外层函数。
- panic不但可以在函数正常流程中抛出，在defer逻辑里也可以再次调用panic或抛出panic。defer里面的panic能够被后续执行的defer捕获。
- recover用来捕获panic，阻止panic继续向上传递。recover()和defer一起使用，但是defer只有在后面的函数体内直接被掉用才能捕获panic来终止异常，否则返回nil，异常继续向外传递。

例子1

```
//以下捕获失败
defer recover()
defer fmt.Prinntln(recover)
defer func(){
    func(){
        recover() //无效，嵌套两层
    }()
}()

//以下捕获有效
defer func(){
    recover()
}()

func except(){
    recover()
}
func test(){
    defer except()
    panic("runtime error")
}
```

例子2

多个panic只会捕捉最后一个：

```
package main
import "fmt"
func main(){
    defer func(){
        if err := recover() ; err != nil {
            fmt.Println(err)
        }
    }()
    defer func(){
        panic("three")
    }()
    defer func(){
        panic("two")
    }()
    panic("one")
}
```

使用场景

一般情况下有两种情况用到：

- 程序遇到无法执行下去的错误时，抛出错误，主动结束运行。
- 在调试程序时，通过 panic 来打印堆栈，方便定位错误。

若小叶 2个月前 (01-05)

```
if result, errorMsg := Divide(100, 10); errorMsg == "" {
    fmt.Println("100/10 = ", result)
}

if _, errorMsg := Divide(100, 0); errorMsg != "" {
    fmt.Println("errorMsg is: ", errorMsg)
}
```

等价于:

```
result, errorMsg := Divide(100,10)
if errorMsg == ""{
    fmt.Println("100/10 = ", result)
}

result, errorMsg = Divide(100,0)
if errorMsg != ""{
    fmt.Println("errorMsg is: ", errorMsg)
}
```

GG 2周前 (03-01)