Error / Panic

```
严重错误 -> Panic异常 -> 程序崩溃
panic ~ abort (类似但不完全一样)
普通错误—— error 处理
严重错误—— panic ,程序逻辑已无法继续
经典panic场景
 // 1. 代码逻辑的严重错误(本不该发生)
 func MustGetConfig() Config {
    if config == nil {
       panic("config 未初始化") // 如果调用此函数, 预期 config 必须存在
    }
    return config
 }
 // 2. 不可恢复的外部错误
 func ConnectToDatabase() {
    if err := db.Ping(); err != nil {
       panic("数据库连接失败,服务无法启动") // 启动依赖失败,程序无法运行
    }
 }
```

会导致程序立即停止当前函数的执行! And then ...

defer延迟函数

defer?

defer关键字 延迟 (函数的执行)

```
package main

import "fmt"

func main() {
    defer fmt.Println("这是最后执行的语句")
    fmt.Println("这是第一个执行的语句")
    fmt.Println("这是第二个执行的语句")
}

被defer的会延迟执行
结果:

这是第一个执行的语句
这是第二个执行的语句
```

how to defer?

这是最后执行的语句

延迟+栈

defer 语句会将函数调用推入一个栈中,在函数返回(或终止)时,这些函数按照栈的风格(后进先出)的顺序执行

```
func deferStack() {
    defer fmt.Println("第一个 defer")
    defer fmt.Println("第二个 defer")
    defer fmt.Println("第三个 defer")
}

第三个 defer
第二个 defer
第一个 defer
```

参数立即求值

最后才执行但结果(相关参数的值)早已确定

```
func valInDefer() {
    i := 1
    defer fmt.Println("i in Defer:", i)
    i++
    fmt.Println("i in Normal:", i)
}

i in Normal: 2
i in Defer: 1
```

应用?

e.g. 确保文件关闭

```
func readFile(filename string) (string, error) {
    f, err := os.Open(filename)
    if err != nil {
        return "", err
    }
    defer f.Close() // 确保文件会被关闭
    // 兜底 or 殿后

    content, err := io.ReadAll(f)
    if err != nil {
        return "", err
    }

    return string(content), nil
}
```

panic异常

panic?痛太痛了我的程序 😇

抛出严重异常 & 终止程序?

```
package main

func main() {
    panic("发生了严重错误!") // 程序在此处中断
    println("这行不会执行") // unreachable code
}

panic: 发生了严重错误!

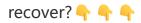
goroutine 1 [running]:
main.panicTest(...)
    e:/mati/go-demos/ch5-2/lostInPanic.go:34
main.main()
    e:/mati/go-demos/ch5-2/lostInPanic.go:10 +0x31
exit status 2
```

注意!不要滥用panic!可预见的轻微的异常—— error,然后接正常的异常处理;应该"不可能发生"的严重异常—— panic ,因为一旦panic程序可能就寄了

panic我程序真终止了吗? 😘

panic 的工作流程

- 1. 程序执行到 panic 时立即停止当前函数的执行
- 2. 开始执行当前函数中所有的 defer 语句(按后进先出顺序) // defer还没有倒下! 🐎
- 3. 向上回溯调用栈,重复这个过程 // 记住这句话~ 😘
- 4. 如果没有被 recover 捕获,程序最终会崩溃并打印堆栈跟踪信息



Recover捕获异常

recover?真能复活吗?

C++ throw异常有catch捕获

那Go的panic有没有什么人来捕获/接收/兜底呢? 应该鼓励panic快快趋势 😂

《通常来说,不应该对panic异常做任何处理,但有时,也许我们可以从异常中恢复,至少我们可以在程序崩溃前,做一些操作》

Recover! 返回吧! 我的函数!

因为确实救不了,函数触发panic异常,recover会使程序从panic中恢复,并返回panic value。导致panic异常的函数不会继续运行,但能正常返回

但是panic被recover捕获之后,就不会霍霍其他人了(详见示例)

限制

recover 返回 panic 传递的值,如果没有发生 panic 则返回 nil

必须在 defer 函数中调用才有效

只能捕获同一goroutine (类似"线程",后面的章节会介绍)中的 panic

示例

带type不,老panic?

```
func handlePanic() {
   defer func() {
       if r := recover(); r != nil {
           switch x := r.(type) { // panic(r) 也是有type的!
           case string:
               fmt.Println("String panic:", x)
           case error:
               fmt.Println("Error panic:", x)
           default:
               fmt.Println("Unknown panic:", x)
               panic(r) // 重新抛出无法处理的 panic
           }
       }
   }()
   panic(errors.New("an error occurred"))
}
```

嵌套panic-recover

```
func inner() {
    defer fmt.Println("inner defer")
    panic("inner panic")
 }
 func outer() {
    defer func() {
        if r := recover(); r != nil {
           fmt.Println("outer recovered:", r)
        }
    }()
     inner() // defer 栈 后进先出
    // 但是panic已经被inner()函数的recover逮住了,为什么outer()的也跟着defer / recover了?
    // 还记得那句话吗?向上回溯调用栈,重复这个过程
    // defer那一刻,你就已经进栈了,哼,想逃?闪电stack劈!
 }
 func main() {
    outer()
    fmt.Println("你说我能不能执行?")
 }
输出:
 inner defer
 outer recovered: inner panic
 你说我能不能执行?
```

除非满朝尽忠error之时,不可轻易乱用panic大法呀 😉