异常处理

Go的设计者认为其它语言异常处理太过消耗资源，且设计和处理复杂，导致使用者不能很好的处理错误，甚至觉得处理麻烦而忽视、忽略错误，导致程序崩溃。

为了解决这些问题，Go将错误处理设计的非常简单

- 函数调用，返回值可以返回多值，一般最后一个值可以是error接口类型的值
  - 如果调用产生错误，则这个值是一个error接口类型的错误
  - 如果调用成功，则该值是nil
- 检查函数返回值中的错误是否是nil，如果不是nil，进行必要的错误处理

error是Go中声明的接口类型

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

所有实现 `Error() string` 签名的方法，都可以实现错误接口。用Error()方法返回错误的具体描述。

自定义error

通过errors包的New方法返回一个error接口类型的错误实例， `errors.New("错误描述")`

```
type errorString struct {
    s string
}

func (e *errorString) Error() string {
    return e.s
}

func New(text string) error {
    return &errorString{text}
}
```

可以看出New方法返回一个实现了Error接口的结构体实例的指针。

```
package main

import (
    "errors"
    "fmt"
)

var ErrDivisionByZero = errors.New("division by zero") // 构造一个错误实例，建议Err前缀

```

```go
func div(a, b int) (int, error) {
    if b == 0 {
        return 0, ErrDivisionByZero
    }
    return a / b, nil
}

func main() {
    if r, err := div(5, 0); err != nil {
        // fmt.Println(err)
        fmt.Println(err.Error())
    } else {
        fmt.Println(r)
    }
}

```

panic

panic有人翻译成宕机。

panic是不好的，因为它发生时，往往会造成程序崩溃、服务终止等后果，所以没人希望它发生。但是，如果在错误发生时，不及时panic而终止程序运行，继续运行程序恐怕造成更大的损失，付出更加惨痛的代价。所以，有时候，panic导致的程序崩溃实际上可以及时止损，只能两害相权取其轻。

panic虽然不好，体验很差，但也是万不得已，可以马上暴露问题，及时发现和纠正问题。

panic产生

- runtime运行时错误导致抛出panic，比如数组越界、除零
- 主动手动调用panic(reason)，这个reason可以是任意类型

panic执行

- 逆序执行当前已经注册过的goroutine的defer链（recover从这里介入）
- 打印错误信息和调用堆栈
- 调用exit(2)结束整个进程

```go
package main

import (
    "fmt"
)

func div(a, b int) int {
    defer fmt.Println("start")
    defer fmt.Println(a, b)
    r := a / b // 这一行有可能panic
    fmt.Println("end")

```

```
    return r
}

func main() {
    fmt.Println(div(5, 0))
}
```

运行后程序崩溃，因为除零异常，输入如下

```
5 0
start
panic: runtime error: integer divide by zero

goroutine 1 [running]:下面是调用栈，div压着main
main.div(0x5, 0x0)
        O:/pros/main.go:13 +0x1bf 出错的行号
main.main()
        O:/pros/main.go:19 +0x25
exit status 2
```

recover

recover即恢复，defer和recover结合起来，在defer中调用recover来实现对错误的捕获和恢复，让代码在发生panic后通过处理能够继续运行。类似其它语言中try/catch。

`err := recover()`，v就是`panic(reason)`中的reason，reason可以是任意类型。

```
package main

import (
    "errors"
    "fmt"
    "io/fs"
    "os"
)

var ErrDivisionByZero = errors.New("division by zero") // 构造一个错误实例

func div(a, b int) int {
    defer func() {
        err := recover()
        fmt.Println(1, err, "@@@@")
    }()
    defer fmt.Println("start")
    defer fmt.Println(a, b)
    defer func() {
        fmt.Println("错误捕获")
        err := recover()    // 一旦recover了，就相当处理过了错误
        switch err.(type) { // 类型断言
        case *fs.PathError:
            fmt.Println("文件不存在", err)
```

```
        case []int:
            fmt.Println("切片", err)
        }
        fmt.Println("离开")
    }()
    if f, err := os.Open("o:/tttt"); err != nil {
        panic(err)
    } else {
        fmt.Println(f)
    }

    r := a / b // 这一行有可能panic
    // panic([]int{1, 3, 5}) // 手动panic
    fmt.Println("end")
    return r
}

func main() {
    fmt.Println(div(5, 0), "!!!") // 有返回值吗？如果有是什么？
    fmt.Println("main exit")
}

```

上例中，一旦在某函数中panic，当前函数panic之后的语句将不再执行，开始执行defer。如果在defer中错误被recover后，就相当于当前函数产生的错误得到了处理。当前函数执行完defer，当前函数退出执行，程序还可以从当前函数之后继续执行。

可以观察到panic和recover有如下

- 有panic，但没有recover，就没有地方处理错误，程序崩溃
- 有painc，有recover来捕获，相当于错误被处理掉了，当前函数defer执行完后，退出当前函数，从当前函数之后继续执行

如果我们在函数中需要拦截某些特定错误，就可以使用defer func定义一个延时函数，在其中对某些类型错误进行处理，不要把这些错误抛出函数外。