链滴

Golang 入门笔记 -11- 错误处理

作者：zyk
原文链接：https://ld246.com/article/1606705325953
来源网站：链滴

Go 没有像 Java 那样的 try/catch 异常处理机制，而是用 defer/panic/recover 机制来处理异常。

Go 语言的设计者认为 try/catch 机制使用过于泛滥，而且从底层向高层抛出错误太耗费资源，因此给 Go 语言设计了一种**返回值处理错误方式**：通过在函数和方法中返回错误对象，这个错误对象一般多个返回值的最后；如果返回 nil，则表明没有错误，且主调函数应该检查并处理每一个错误。

我们通过调用 pack1 包中的 Func1 函数来了解 Go 语言中的错误处理方式：

Func1返回了两个值，一个 value和 err，err 是错误对象，若 err 不为 nil（nil 是空的意思，类似于 Java 中的 null），则进行错误处理，打印出具体错误信息。

```
if value, err := pack1.Func1(param1); err != nil {
    fmt.Printf("Error %s in pack1.Func1 with parameter %v", err.Error(), param1)
    return    // or: return err
}
```

Go 有一个预先定义的 error 接口类型：

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

错误处理

定义错误

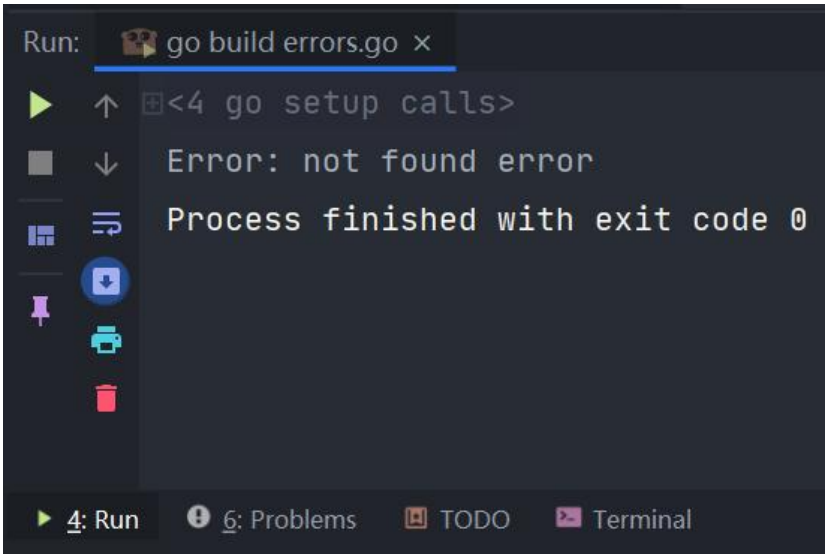
可以通过 errors 包中 New() 函数传递错误信息，从而自定义错误，如下：

```
err := errors.New("square root of negative number")
```

我们来看一个例子：

```go
package main

import (
    "errors"
    "fmt"
)

var errNotFound = errors.New("not found error")

func main() {
    fmt.Printf("Error: %v", errNotFound)
}
```

上述代码运行结果为：

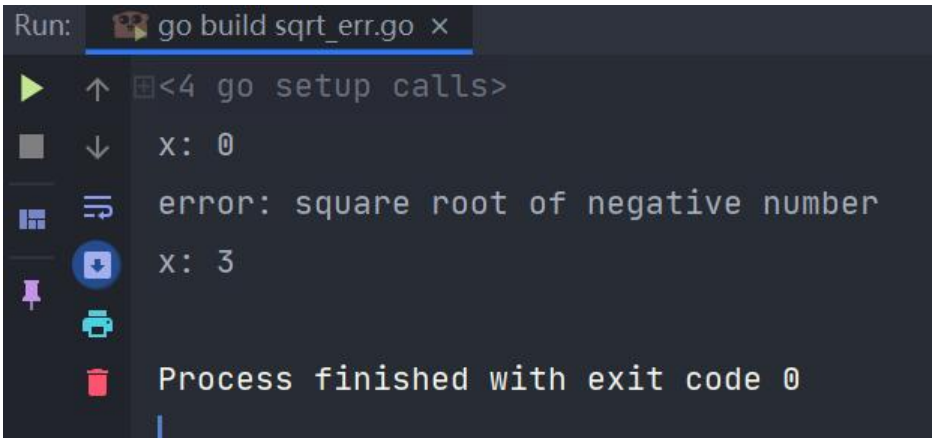


我们通过测试平方根函数，来了解错误处理机制：

```go
package main

import (
    "errors"
    "fmt"
    "math"
)

// 自定义错误
var errSqrtNegative = errors.New("square root of negative number")

/*
求平方根：
若传入数据小于 0，返回 0 和错误；否则，返回平方根结果和 nil
*/
func sqrt(x float64) (float64, error) {
    if x < 0 {
        return 0, errSqrtNegative
    }
    return math.Sqrt(x), nil
}
```

```go
func main() {
    x, err := sqrt(-1)
    fmt.Printf("x: %v\n", x)
    if err != nil { // 若错误不为空，进行错误处理
        fmt.Printf("error: %v\n", err.Error())
    }

    x, err = sqrt(9)
    fmt.Printf("x: %v\n", x)
    if err != nil {
        fmt.Printf("error: %v\n", err.Error())
    }
}
```

上述代码运行结果为：



注意：一般错误信息都会有 Error 前缀，因此错误信息不要以大写字母开头。

自定义错误类型中可以包含错误信息以外的其他信息，我们来看下 os.Open 操作触发的 PathError 误：

```go
// PathError records an error and the operation and file path that caused it.
type PathError struct {
    Op string    // “open”，“unlink”, etc.
    Path string  // The associated file.
    Err error  // Returned by the system call.
}

func (e *PathError) String() string {
    return e.Op + " " + e.Path + ": "+ e.Err.Error()
}
```

如果在一个操作中会发生多种错误，可以用**类型断言**或者**类型判断**对错误进行判断：

```go
if err, ok := err.(*os.PathError); ok {
    // ...
}
```

或：

```go
switch err := err.(type) {
```

```
    case ParseError:
        // ...
    case PathError:
        // ...
    default:
        fmt.Printf("not a special error, just %v\n", err)
}
```

用 fmt 创建错误对象

可以通过 fmt.Errorf() 来创建错误对象，用法类似于 fmt.Printf()，可以用占位符来格式化输出，例如：

```
if f < 0 {
    return 0, fmt.Errorf("math: square root of negative number %g", f)
}
```

运行时异常和 panic

当发生**数组下标越界**等运行错误时，会触发 panic，程序会崩溃并抛出一个 runtime.Error 接口类型。

panic 一般用在**错误条件很苛刻**且**不可恢复**时，当程序无法继续运行时，比如连接数据库时密码错误可以使用 panic 函数产生一个中止运行的错误。

panic 可以接收一个任意类型的参数，一般是字符串，当程序执行到 panic 时，会打印出来，同时 panic 之后的语句将不会被执行， 我们来看一个例子：

```
package main

import "fmt"

func main() {
    fmt.Println("Starting the program")
    panic("A severe error occurred: stopping the program!")
    fmt.Println("Ending the program")
}
```

上述代码运行结果为：

```
Run:  go build panic_1.go ×
<4 go setup calls>
Starting the program
panic: A severe error occurred: stopping the program!

goroutine 1 [running]:
main.main()
    D:/projects/go_projects/goBasic/err/panic_1.go:7 +0xa5

Process finished with exit code 2
4: Run   6: Problems   TODO   Terminal
```

不能随意用 panic 来中止程序，必须尽力补救错误以便让程序能够继续正常运行。

从 panic 中恢复

recover 函数可以让程序从 panic 场景下恢复，停止终止过程而恢复正常。

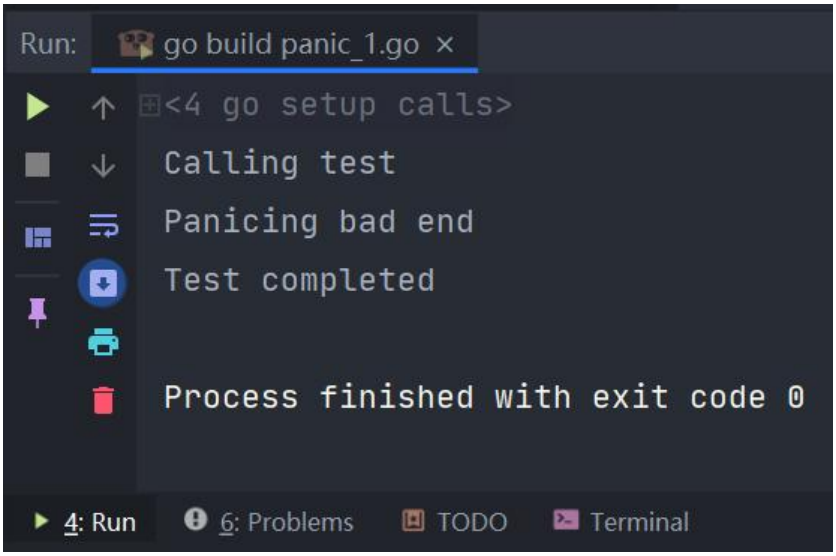
recover 只能在 defer 修饰的函数中使用，可以获取从 panic 中传递过来的错误信息，若是正常调用 recover 只返回 nil。

我们来看一个例子：

```go
package main

import (
    "fmt"
)

func badCall() {
    panic("bad end")
}

func test() {
    // 用 defer 调用匿名函数
    defer func() {
        // 用 recover 接收 panic 错误信息
        if e := recover(); e != nil {
            fmt.Printf("Panicing %s\r\n", e)
        }
    }()
    badCall()
    fmt.Printf("After bad call\r\n")
}

func main() {
```

```
    fmt.Printf("Calling test\r\n")
    test()
    fmt.Printf("Test completed\r\n")
}
```

上述代码运行结果为：



闭包处理错误

每当函数返回错误时，我们都应该去处理，但这样会导致代码重复。结合 defer/panic/recover 机制闭包我们可以创造一种更优雅的错误处理模式。但是这个模式只适用于函数签名相同的情况，一个很的例子就是在 Web 应用中的处理函数，它的一般形式如下：

```
func handler(w http.ResponseWriter, r *http.Request) {
    // ...
}
```

假设所有函数都是这样的签名：

```
func f(a type1, b type2)
```

给这个函数类型一个名称：

```
fType1 = func f(a type1, b type2)
```

在该模式中使用两个帮助函数：

- check：用来检查是否有错误和 panic 发生。

```
func check(err error) {
    if err != nil {
        panic(err)
    }
}
```

- errorhandler：一个包装函数，接收一个 fType1 类型的函数并返回一个 fType1 类型的函数，在中包含 defer/recover 机制。

```
func errorHandler(fn fType1) fType1 {
```

```
    return func(a type1, b type2) {
        defer func() {
            if e, ok := recover().(error); ok {
                log.Printf("run time panic: %v", err)
            }
        }()
        fn(a, b)
    }
}
```

当有错误时会 recover 并打印日志，check 函数会在所有的被调函数中调用：

```
func f1(a type1, b type2) {
    // ...
    f, _, err := handler1() // 调用函数
    check(err)              // 错误处理
    t, err := handler2()
    check(err)
    _, err2 := handler3()
    check(err2)
    // ...
}
```

在这种机制下，所有的错误都会被 recover，且错误处理代码也简化为调用 check(err) 。在这种模式，不同的错误处理必须对应不同的函数类型；错误处理可能被隐藏在处理包内部。