

19 | 错误处理（上）

2018-09-24 郝林



19 | 错误处理（上）

朗读人：黄洲君 09'29'' | 3.27M

提到 Go 语言中的错误处理，我们其实已经在前面接触过几次了。比如，我们声明过 `error` 类型的变量 `err`，也调用过 `errors` 包中的 `New` 函数。今天，我会用这篇文章为你梳理 Go 语言错误处理的相关知识，同时提出一些关键问题并与你一起探讨。

我们说过 `error` 类型其实是一个接口类型，也是一个 Go 语言的内置类型。在这个接口类型的声明中只包含了一个方法 `Error`。这个方法不接受任何参数，但是会返回一个 `string` 类型的结果。

它的作用是返回错误信息的字符串表示形式。我们使用 `error` 类型的方式通常是，在函数声明的结果列表的最后，声明一个该类型的结果，同时在调用这个函数之后，先判断它返回的最后一个结果值是否“不为 `nil`”。

如果这个值“不为 `nil`”，那么就进入错误处理流程，否则就继续进行正常的流程。下面是一个例子，代码在 `demo44.go` 文件中。

```
package main

import (
    "errors"
    "fmt"
)

func echo(request string) (response string, err error) {
    if request == "" {
        err = errors.New("empty content")
        return
    }
    response = fmt.Sprintf("echo: %s", request)
    return
}

func main() {
    for _, req := range []string{"", "hello!"} {
        fmt.Printf("request: %s\n", req)
        resp, err := echo(req)
        if err != nil {
            fmt.Printf("error: %s\n", err)
            continue
        }
        fmt.Printf("response: %s\n", resp)
    }
}
```

□复制代码

我们先看 `echo` 函数的声明。`echo` 函数接受一个 `string` 类型的参数 `request`，并会返回两个结果。

这两个结果都是有名称的，第一个结果`response`也是`string`类型的，它代表了这个函数正常执行后的结果值。第二个结果`err`就是`error`类型的，它代表了函数执行出错时的结果值，同时也包含了具体的错误信息。

当`echo`函数被调用时，它会先检查参数`request`的值。如果该值为空字符串，那么它就会通过调用`errors.New`函数，为结果`err`赋值，然后忽略掉后边的操作并直接返回。

此时，结果`response`的值也会是一个空字符串。如果`request`的值并不是空字符串，那么它就为结果`response`赋一个适当的值，然后返回，此时的结果`err`的值会是`nil`。

再来看`main`函数中的代码。我在每次调用`echo`函数之后都会把它返回的结果值赋给变量`resp`和`err`，并且总是先检查`err`的值是否“不为`nil`”，如果是，就打印错误信息，否则就打印常规的响应信息。

这里值得注意的地方有两个。第一，在`echo`函数和`main`函数中，我都使用到了卫述语句。我在前面讲函数用法的时候也提到过卫述语句。简单地讲，它就被用来检查后续操作的前置条件并进行相应处理的语句。

对于`echo`函数来说，它进行常规操作的前提是：传入的参数值一定要符合要求。而对于调用`echo`函数的程序来说，进行后续操作的前提就是`echo`函数的执行不能出错。

我们在进行错误处理的时候经常会用到卫述语句，以至于有些人会吐槽说：“我的程序满屏都是卫述语句，简直是太难看了！”不过，我倒认为这有可能是程序设计上的问题。每个编程语言的理念和风格几乎都会有明显的不同，我们常常需要顺应它们的纹理去做设计，而不是用其他语言的编程思想来编写当下语言的程序。

再来说第二个值得注意的地方。我在生成`error`类型值的时候用到了`errors.New`函数。这是一种最基本的生成错误值的方式。我们调用它的时候传入一个由字符串代表的错误信息，它会给我们返回一个包含了这个错误信息的`error`类型值。该值的静态类型当然是`error`，而动态类型则是一个在`errors`包中的，包级私有的类型`*errorString`。

显然，`errorString`类型拥有的一个指针方法实现了`error`接口中的`Error`方法。这个方法在被调用后，会原封不动地返回我们之前传入的错误信息。实际上，`error`类型值的`Error`方法就相当于其他类型值的`String`方法。

我们已经知道，通过调用`fmt.Printf`函数，并给定占位符`%s`就可以打印出某个值的字符串表示形式。对于其他类型的值来说，只要我们能在这个类型编写一个`String`方法，就可以自定义它的字符串表示形式。而对于`error`类型值，它的字符串表示形式则取决于它的`Error`方法。

在上述情况下，`fmt.Printf`函数如果发现被打印的值是一个`error`类型的值，那么就会去调用它的`Error`方法。`fmt`包中的这类打印函数其实都是这么做的。

顺便提一句，当我们想通过模板化的方式生成错误信息，并得到错误值时，可以使用 `fmt.Errorf` 函数。该函数所做的其实就是先调用 `fmt.Sprintf` 函数，得到确切的错误信息；再调用 `errors.New` 函数，得到包含该错误信息的 `error` 类型值，最后返回该值。

好了，我现在问一个关于对错误值做判断的问题。我们今天的问题是：**对于具体错误的判断，Go 语言中都有哪些惯用法？**

由于 `error` 是一个接口类型，所以即使同为 `error` 类型的错误值，它们的实际类型也可能不同。这个问题还可以换一种问法，即：怎样判断一个错误值具体代表的是哪一类错误？

这道题的**典型回答**是这样的：

1. 对于类型在已知范围内的一系列错误值，一般使用类型断言表达式或类型 `switch` 语句来判断；
2. 对于已有相应变量且类型相同的一系列错误值，一般直接使用判等操作来判断；
3. 对于没有相应变量且类型未知的一系列错误值，只能使用其错误信息的字符串表示形式来做判断。

问题解析

如果你看过一些 Go 语言标准库的源代码，那么对这几种情况应该都不陌生。我下面分别对它们做个说明。

类型在已知范围内的错误值其实是最容易分辨的。就拿 `os` 包中的几个代表错误的类型 `os.PathError`、`os.LinkError`、`os.SyscallError` 和 `os/exec.Error` 来说，它们的指针类型都是 `error` 接口的实现类型，同时它们也都包含了一个名叫 `Err`，类型为 `error` 接口类型的代表潜在错误的字段。

如果我们得到一个 `error` 类型值，并且知道该值的实际类型肯定是它们中的某一个，那么就可以用类型 `switch` 语句去做判断。例如：

```
func underlyingError(err error) error {
    switch err := err.(type) {
    case *os.PathError:
        return err.Err
    case *os.LinkError:
        return err.Err
    case *os.SyscallError:
        return err.Err
    case *exec.Error:
        return err.Err
    }
}
```

```
        return err
    }
```

□复制代码

函数`underlyingError`的作用是：获取和返回已知的操作系统相关错误的潜在错误值。其中的类型`switch`语句中有若干个`case`子句，分别对应了上述几个错误类型。当它们被选中时，都会把函数参数`err`的`Err`字段作为结果值返回。如果它们都未被选中，那么该函数就会直接把参数值作为结果返回，即放弃获取潜在错误值。

只要类型不同，我们就可以如此分辨。但是在错误值类型相同的情况下，这些手段就无能为力了。在 **Go** 语言的标准库中也有不少以相同方式创建的同类型的错误值。

我们还拿`os`包来说，其中不少的错误值都是通过调用`errors.New`函数来初始化的，比如：`os.ErrClosed`、`os.ErrInvalid`以及`os.ErrPermission`，等等。

注意，与前面讲到的那些错误类型不同，这几个都是已经定义好的、确切的错误值。`os`包中的代码有时候会把它们当做潜在错误值，封装进前面那些错误类型的值中。

如果我们在操作文件系统的时候得到了一个错误值，并且知道该值的潜在错误值肯定是上述值中的某一个，那么就可以用普通的`switch`语句去做判断，当然了，用`if`语句和判等操作符也是可以的。例如：

```
printError := func(i int, err error) {
    if err == nil {
        fmt.Println("nil error")
        return
    }
    err = underlyingError(err)
    switch err {
    case os.ErrClosed:
        fmt.Printf("error(closed) [%d]: %s\n", i, err)
    case os.ErrInvalid:
        fmt.Printf("error(invalid) [%d]: %s\n", i, err)
    case os.ErrPermission:
        fmt.Printf("error(permission) [%d]: %s\n", i, err)
    }
}
```

□复制代码

这个由`printError`变量代表的函数会接受一个`error`类型的参数值。该值总会代表某个文件操作相关的错误，这是我故意地以不正确的方式操作文件后得到的。

虽然我不知道这些错误值的类型的范围，但却知道它们或它们的潜在错误值一定是某个已经在`os`包中定义的值。

所以，我先用`underlyingError`函数得到它们的潜在错误值，当然也可能只得到原错误值而已。然后，我用`switch`语句对错误值进行判等操作，三个`case`子句分别对应我刚刚提到的那三个已存在于`os`包中的错误值。如此一来，我就能分辨出具体错误了。

对于上面这两种情况，我们都有明确的方式去解决。但是，如果我们对一个错误值可能代表的含义知之甚少，那么就只能通过它拥有的错误信息去做判断了。

好在我们总是能通过错误值的`Error`方法，拿到它的错误信息。其实`os`包中就有做这种判断的函数，比如：`os.IsExist`、`os.IsNotExist`和`os.IsPermission`。命令源码文件 **demo45.go** 中包含了对它们的应用，这大致跟前面展示的代码差不太多，我就不在这里赘述了。

总结

今天我们一起初步学习了错误处理的内容。我们总结了错误类型、错误值的处理技巧和设计方式，并一起分享了 **Go** 语言中处理错误的最基本方式。由于错误处理的内容分为上下两篇，在下一次的文章中，我们会站在建造者的角度，一起来探索一下：怎样根据实际情况给予恰当的错误值。

思考题

请列举出你经常用到或者看到的 3 个错误类型，它们所在的错误类型体系都是怎样的？你能画出一棵树来描述它们吗？