Swift中的异常和错误处理

如何处理closure参数会发生的错误? (https://www.boxueio.com/series/error-handling/ebook/202)

返回视频 (/series/error-handling)

使用defer以及串联either type

Back to series (/series/error-handling)

字号 字号 默认主题 金色主题 暗色主题

在很多编程语言的错误处理机制中，除了“捕获”并处理异常之外，通常还会有一个 `finally` 的环节，让我们编写无论函数的执行是否顺利，在离开函数作用域的时候一定会执行的代码。Swift中也有一个类似的机制，叫做 `defer` ，只不过，它只保证在离开它所在的作用域时，内部的代码一定会被执行。你不用把它和 `try...catch` 放在一起。

通过defer来计数

例如，我们要统计所有 `Car` 对象的启动次数，就可以这样。先在 `Car` 中添加一个静态属性：

```
struct Car {
    // ...
    static var startCounter: Int = 0
    // ...
}
```

然后，把 `start` 方法修改成这样：

```
/// - Throws: `CarError` if the car is out of fuel
func start() throws -> String {
    guard fuel > 5 else {
        throw CarError.outOfFuel(no: no, fuel: fuel)
    }

    defer { Car.startCounter += 1 }

    return "Ready to go"
}
```

注意到 `defer` 的用法了么？这样，无论 `start()` 是“抛出”了错误，还是正常返回， `defer` 中的代码都会被执行，于是 `startCounter` 都会被加1。

因此，只要你的函数有可能因为发生错误提前返回，就可以考虑使用这种模式来进行资源清理或回收的工作。

串联多个either type

接下来，我们回到之前提到的either type，相比 `do...catch` ，它有一个用法上的缺陷。在日常的编程中，我们经常会调用多个返回 `Result<T>` 类型的方法来完成一个复杂的操作。但这通常会导致嵌套很深的代码，我们得不断在上一个方法的 `.success` 情况里，继续后续的方法调用。于是，用不了太多方法，你就受不了自己编写的代码了。

例如，对于上一节中提到的更新Car OS的例子，我们把更新的步骤更具体的表现出来。首先，添加一个新的 `CarError` ，表示文件校验错误：

```
enum CarError: Error {
    case outOfFuel(no: String, fuel: Double)
    case updateFailed
    case integrationError
}
```

然后，为了通过编译，我们给 `Car` 添加两个函数，表示下载和校验文件，按照约定，它们都返回一个 `Result<T>` 表示执行结果：

```
func downLoadPackage() -> Result<String> {
        return .failure(CarError.updateFailed)
    }

func checkIntegration(of path: String) -> Result<Int> {
    return .failure(CarError.integrationError)
}
```

最后，我们来看下面这个让人痛苦的 osUpdate 实现方式：

```
func osUpdate(postUpdate: @escaping (Result<Int>) -> Void) {
    DispatchQueue.global().async {
        // 1. Download package
        switch self.downLoadPackage() {
        case let .success(filePath):
            // 2. Check integration
            switch self.checkIntegration(of: filePath) {
            case let .success(checksum):
                // 3. Do you want to continue from here?
                // ...
            case let .failure(e):
                postUpdate(.failure(e))
            }
        case let .failure(e):
            postUpdate(.failure(e))
        }
    }
}
```

写到这里，我们仅仅完成了两个工作，在注释的第三步，你还有勇气继续再写下去么？为了解决either type的这个问题，我们得给它添加一些扩展。其实，这个问题和我们连续使用optional变量是非常类似的。既然为了不断使用optional的非 `nil` 值，我们可以使用 `flatMap` 串联，对于 `Result<T>` ，我们可以如法炮制一个：

```
extension Result {
    func flatMap<U>(transform: (T) -> Result<U>) -> Result<U> {
        switch self {
        case let .success(v):
            return transform(v)
        case let .failure(e):
            return .failure(e)
        }
    }
}
```

之后，我们的 osUpdate 就可以改成这样：

```
func osUpdate(postUpdate: @escaping (Result<Int>) -> Void) {
    DispatchQueue.global().async {
        let result = self.downLoadPackage()
            .flatMap {
                self.checkIntegration(of: $0)
            }
            // Chain other processes here

        postUpdate(result)
    }
}
```

然后，我们可以使用 `flatMap` 继续串联任意多个后续需要执行的方法了，如果其中某个环节发生了错误，整个流程自然就结束了。这时，调用 osUpdate 的代码可以改成这样：

```
Car(fuel: 10, no: "1").osUpdate(postUpdate: {
    switch $0 {
    case let .success(checksum):
        print("Update success: \(checksum)")
    case let .failure(e):
         if let e = e as? CarError {
            switch e {
            case .integrationError:
                print("Checksum error")
            default:
                break
            }
        }

        print("Update failed")
    }
})
```

最终，在 postUpdate 这个回调函数里，无论是成功还是失败，我们都可以得到 osUpdate 最后一步操作返回的结果。

◀ 如何处理closure参数会发生的错误?
(https://www.boxueio.com/series/error-handling/ebook/202)

返回视频 ▶
(/series/error-handling)