Swift中的异常和错误处理

返回视频 (/series/error-handling)

NSError是如何桥接到Swift原生错误处理的? (https://www.boxueio.com/series/error-handling/ebook/200)

理解Swift中的错误处理机制

Back to series (/series/error-handling)

字号
字号
默认主题
金色主题
暗色主题

如何利用Swift的语言机制表达错误呢？你可能最先想到的就是optional。成功的时候，返回Value，错误的时候，返回 `nil` 。甚至，有不少Swift标准库的API就是这样做的。因此，这的确是个不错的选择。但 `nil` 只适合表达非常显而易见的错误，例如：访问 `Dictionary` 中一个不存在的Key `dic["nonExistKey"]` ， `nil` 就只能表示Key不存在。

但如果可能会发生的错误不止一种情况， `nil` 的表现力就很弱了，我们很难知道究竟什么原因导致了错误。怎么办呢?

通过enum和Error封装错误

如果你还记得optional的实现方式，就会发现这个问题很好解决。既然optional通过 `enum` 的两个 `case` （ `.some` 和 `.none` ）表示它的两个状态，我们自然也可以用一个 `enum` 表示操作成功和失败的结果：

```
enum Result<T> {
    case success(T)
    case failure(Error)
}
```

为了能包含不同成功结果， `Result` 得是一个泛型 `enum` 。而 `Error` 则是Swift中的一个 `protocol` ，它没有任何具体的约定，只用来表示一个类型的身份。稍后，我们会看到，只有遵从了 `Error` 的错误，才可以被 `throw` 。

有了这个 `enum` 之后，怎么用呢？假设我们要定义一个表示汽车的 `struct` ：

```
struct Car {
    var fuelInLitre: Double

    func start() -> Result<String> {
        guard fuelInLitre > 5 else {
            // How we express the error here?
        }

        return .success("Ready to go")
    }
}
```

我们设定只有当燃料大于5升的时候，才可以正常启动。 `start` 方法通过返回 `Result<String>` 表达了这个概念。当条件满足时，我们直接返回 `.success("Ready to go")` 。然而，我们该如何处理 `.failure` 的情况呢?

首先，要根据我们有可能遇到的错误，再自定义一个遵从 `Error` 的 `enum` ：

```
enum CarError: Error {
    case outOfFuel
}
```

目前，我们只有一个燃料不足的情况，因此，先定义一个 `case` 就好了。然后，在 `start` 的实现里，燃料不足时，我们直接返回 `CarError.outOfFuel` ：

```
func start() -> Result<String> {
    guard fuel > 5 else {
        return .failure(CarError.outOfFuel)
    }

    // ...
}
```

然后，有了 Result 类型的这种约定，我们就可以采用固定的套路来处理错误：

```
let vw = Car(fuel: 2)

switch vw.start() {
case let .success(message):
    print(message)
case let .failure(error):
    if let carError = error as? CarError,
        carError == .outOfFuel {
        print("Cannot start due to out of fuel")
    }
    else {
        print(error.localizedDescription)
    }
}
```

在 case .failure 里，我们可以通过对 error 类型转换的结果来判断是否发生了特定的错误，进而进行专门的处理。而对于所有不识别 CarError ，我们可以直接打印Swift提供的一个本地化的描述。

理解Swift中的throw和catch

理解了这种处理错误的思路之后就会发现，它哪都好，唯一不好的地方在于我们所有的约定都只能停留在口头上， Result 的名字也好， .success 和 .failure 的名字也好，都是如此。我们无法避免有人使用诸如 ResultType / .succ / .fail 这样的名字，于是，不同的开发者写出的代码可能思路都是一样的，却无法搭配在一起好好工作。

解决这个问题唯一的办法，就是能按照上面的思路，提供一种编程语言层面的保障。而这，就是 throw 关键字的用途。我们的 start 方法可以改成这样：

```
/// - Throws: `CarError` if the car is out of fuel
func start() throws -> String {
    guard fuel > 5 else {
        // How we press the error here?
        // return .failure(CarError.outOfFuel)
        throw CarError.outOfFuel
    }

    // return .success("Ready to go")
    return "Ready to go"
}
```

相比之前的版本， throws 版本有了下面这些改进：

- 通过 throws 关键字表示一个函数有可能发生错误相比 Result 更加统一和明确。并且，通过 throws ，函数可以恢复返回正确情况下要返回的类型；
- 遇到错误的情况时，通过 throw 关键字表示“抛出”一个“异常情况”，它有别于使用 return 返回正确的结果；

如果你有过其他面向对象编程语言的经验，对这种写法可能更为熟悉。但就像我们在注释中对比的那样，在Swift中 throw 一个 Error 和 return .failure(...) 这种写法是没有任何区别的，“抛出”的错误没有明确的类型，这种“异常”也不会带来任何运行时成本。 throw 就是一个语法糖而已。

因此，在Swift里，**凡是声明中带有 throws 关键字的，通常都会在注释中标明这个函数有可能发生的错误**。否则，我们很难知道该如何处理。为此，Swift还在markdown注释中添加了一类关键字，就像我们对 start 的注释一样。

除了在表达方式上更为统一之外，使用 throws 声明函数还有一个好处，就是编译器会强制我们用“标准”的方法来调用可能会发生错误的函数。因此，我们之前的 start 调用就可以变成这样：

```
do {
    let message = try vw.start()
    print(message)
} catch CarError.outOfFuel {
    print("Cannot start due to out of fuel")
} catch {
    print("We have something wrong")
}
```

可以看到，当我们调用 start() 时，要明确使用 try 关键字表示这种调用是个尝试，它有可能失败。然后，对于这种调用，我们必须把它包含在一个 do...catch 里，其中，每个 catch 用来匹配 start 有可能会返回的一种错误。最后，我们用一个不匹配任何具体 Error 的 catch 表示匹配其他未列出的

错误。虽然这并不是必要的，但是一旦 `start()` 返回了我们没有 `catch` 的错误，就会导致运行时错误。

同样，要再次强调的是，这里的 `do...catch` 也是个语法糖，它和Java中的 `try...catch` 只是语法类似。而本质上， `do...catch` 和我们之前使用的 `switch...case` 是没有任何区别的。

What's next?

了解了Swift原生的错误处理方式之后，下一节我们来了解这种机制是如何与Objective-C搭配在一起工作的。对于那些通过 `nil` 或者 `NSError` 返回错误信息的API来说，它们是如何移植到Swift的呢？Swift中的原生错误处理机制又是如何桥接到OC的呢？

返回视频 (/series/error-handling)

NSError是如何桥接到Swift原生错误处理的? (https://www.boxueio.com/series/error-handling/ebook/200)