错误处理

在本页

*错误处理*是对程序中的错误条件进行响应和恢复的过程。Swift在运行时为抛出，捕获，传播和处理可恢复错误提供了一流的支持。

某些操作不能保证始终完成执行或产生有用的输出。可选项用于表示没有值，但是当操作失败时，了解导致失败的原因通常很有用，以便您的代码可以做出相应的响应。

作为例子，考虑从磁盘上的文件读取和处理数据的任务。此任务可能有多种失败方式，包括文件不存在于指定路径，文件没有读取权限，或文件未以兼容格式编码。通过区分这些不同的情况，程序可以解决一些错误，并向用户传达它无法解决的任何错误。

> 注意
>
> Swift中的错误处理与使用`NSError`Cocoa和Objective-C中的类的错误处理模式进行交互操作。有关此类的更多信息，请参阅在*Cocoa和Objective-C*中*使用Swift中的*错误处理*（Swift 4.1）*。

代表和投掷错误

在Swift中，错误由符合`Error`协议的类型值表示。这个空协议表明一个类型可以用于错误处理。

Swift枚举特别适合于建模一组相关的错误条件，并且相关的值允许提供有关错误性质的附加信息。例如，下面是您如何表示在游戏中操作自动售货机的错误条件：

```
enum VendingMachineError: Error {
    case invalidSelection
    case insufficientFunds(coinsNeeded: Int)
    case outOfStock
}
```

抛出一个错误可以让你指出意外事件发生，并且正常的执行流程无法继续。您使用`throw`语句来引发错误。例如，下面的代码会抛出一个错误，指出自动售货机需要五个额外的硬币：

```
throw VendingMachineError.insufficientFunds(coinsNeeded: 5)
```

处理错误

当引发错误时，一些代码必须负责处理错误 - 例如，纠正问题，尝试替代方法或通知用户错误。

有四种方法可以处理Swift中的错误。您可以将错误从函数传播到调用该函数的代码，使用`do- catch`语句处理错误，将错误作为可选值处理，或声明错误不会发生。下面将介绍每种方法。

当一个函数抛出一个错误时，它会改变程序的流程，所以重要的是你可以在代码中快速识别可能抛出错误的地方。要在您的代码中标识这些位置，请在调用可能引发错误的函数，方法或初始值设定项的代码之前写入`try`关键字`try?`或`try!`变体。这些关键字将在下面的章节中介绍。

> 注意
>
> 在斯威夫特的错误处理类似的异常处理在其他语言中，使用了的`try`，`catch`和`throw`关键字。与许多语言中的异常处理不同 - 包括Swift中的Objective-C错误处理不涉及展开调用堆栈，这是一个计算成本很高的过程。因此，`throw`声明的表现特征与声明的表现特征相当`return`。

使用投掷函数传播错误

为了表明函数，方法或初始值设定项可能会引发错误，可以`throws`在函数声明后的关键字后面写入关键字。标`throws`有的*功能*称为*投掷功能*。如果函数指定返回类型，则`throws`在返回箭头（`->`）之前写入关键字。

```
func canThrowErrors() throws -> String

func cannotThrowErrors() -> String
```

抛出函数将抛出的错误传播给它所调用的作用域。

> 注意
> 只有引发函数会传播错误。任何在非抛出函数内抛出的错误都必须在函数内部处理。

在下面的示例中，如果请求的商品不可用，缺货或成本超过当前存款金额，则`VendingMachine`该类有一个`vend(itemNamed:)`适当的方法`VendingMachineError`：

```
struct Item {
    var price: Int
    var count: Int
}

class VendingMachine {
    var inventory = [
        "Candy Bar": Item(price: 12, count: 7),
        "Chips": Item(price: 10, count: 4),
        "Pretzels": Item(price: 7, count: 11)
    ]
    var coinsDeposited = 0

    func vend(itemNamed name: String) throws {
        guard let item = inventory[name] else {
            throw VendingMachineError.invalidSelection
        }

        guard item.count > 0 else {
            throw VendingMachineError.outOfStock
        }

        guard item.price <= coinsDeposited else {
            throw VendingMachineError.insufficientFunds(coinsNeeded: item.price -
  coinsDeposited)
        }

        coinsDeposited -= item.price

        var newItem = item
        newItem.count -= 1
        inventory[name] = newItem

        print("Dispensing \(name)")
    }
}
```

该`vend(itemNamed:)`方法的实施使用`guard`语句来提前退出该方法，并且如果没有满足购买零食的任何要求则抛出适当的错误。由于`throw`声明立即转移程序控制，所以只有满足所有这些要求时才会出售物品。

因为该`vend(itemNamed:)`方法会传播任何错误，所以调用此方法的任何代码都必须处理错误 - 使用`do`- `catch`语句`try?`，或`try!`- 或继续传播它们。例如，`buyFavoriteSnack(person:vendingMachine:)`在下面的例子中也是一个抛出函数，并且该`vend(itemNamed:)`方法抛出的任何错误都会传播到该`buyFavoriteSnack(person:vendingMachine:)`函数被调用的地方。

```
let favoriteSnacks = [
    "Alice": "Chips",
    "Bob": "Licorice",
```

```
    "Eve": "Pretzels",
]
func buyFavoriteSnack(person: String, vendingMachine: VendingMachine) throws {
    let
    try vendingMachine.vend(itemNamed: snackName)
}
```

在这个例子中，`buyFavoriteSnack(person: vendingMachine:)`函数查找给定人最喜欢的零食，并尝试通过调用该`vend(itemNamed:)`方法为他们购买。由于该`vend(itemNamed:)`方法可能会引发错误，因此会使用`try`前面的关键字调用该方法。

抛出初始化器可以像抛出函数一样传播错误。例如，`PurchasedSnack`下面列表中的结构初始化程序将初始化过程的一部分调用throwing函数，并通过将它传播给调用者来处理它遇到的任何错误。

```
struct PurchasedSnack {
    let name: String
    init(name: String, vendingMachine: VendingMachine) throws {
        try vendingMachine.vend(itemNamed: name)
        self.name = name
    }
}
```

使用Do-Catch处理错误

您使用`do- catch`语句通过运行一段代码来处理错误。如果该`do`子句中的代码引发错误，则将其与`catch`子句进行匹配以确定哪个子句可以处理该错误。

这里是一个的一般形式`do- catch`语句：

```
做 {
    尝试 表达
    声明
} catch  pattern 1  {
    声明
} catch  pattern 2  where  condition  {
    声明
} catch {
    声明
}
```

你在之后编写一个模式`catch`来表明该子句可以处理的错误。如果`catch`子句没有模式，则子句会匹配任何错误并将错误绑定到名为的本地常量`error`。有关模式匹配的更多信息，请参阅模式。

例如，以下代码与`VendingMachineError`枚举的所有三种情况匹配。

```
var vendingMachine = VendingMachine()
vendingMachine.coinsDeposited = 8
do {
    try buyFavoriteSnack(person: "Alice", vendingMachine: vendingMachine)
    print("Success! Yum.")
} catch VendingMachineError.invalidSelection {
    print("Invalid Selection.")
} catch VendingMachineError.outOfStock {
    print("Out of Stock.")
} catch VendingMachineError.insufficientFunds(let coinsNeeded) {
    print("Insufficient funds. Please insert an additional \(coinsNeeded) coins.")
} catch {
    print("Unexpected error: \(error).")
}
// Prints "Insufficient funds. Please insert an additional 2 coins."
```

在上面的例子中，`buyFavoriteSnack(person:vendingMachine:)`函数在`try`表达式中被调用，因为它可能会引发错误。如果发生错误，执行立即转移到`catch`子句，决定是否允许继续传播。如果没有模式匹配，则错误被最终`catch`子句捕获并且绑定到本地`error`常量。如果没有错误发生，`do`则执行语句中的其余语句。

该`catch`条款没有
会传播到周围的范围。但是，传播的错误必须由一些周围的范围来处理。在`nonthrowing`功能，封闭`do- catch`子句必须处理错误。在投掷功能，无论是一个封闭`do- catch`子句或调用者必须处理错误。如果错误传播到顶级作用域而未被处理，则会出现运行时错误。

例如，上面的例子可以写成，所以任何不是`a`的错误都会被`VendingMachineError`调用函数捕获：

```
func nourish(with item: String) throws {
    do {
        try vendingMachine.vend(itemNamed: item)
    } catch is VendingMachineError {
        print("Invalid selection, out of stock, or not enough money.")
    }
}

do {
    try nourish(with: "Beet-Flavored Chips")
} catch {
    print("Unexpected non-vending-machine-related error: \(error)")
}
// Prints "Invalid selection, out of stock, or not enough money."
```

在该`nourish(with:)`函数中，如果`vend(itemNamed:)`抛出错误（这是`VendingMachineError`枚举的其中一种情况），则`nourish(with:)`通过打印消息来处理错误。否则，`nourish(with:)`将错误传播到其呼叫站点。然后错误被通用`catch`条款捕获。

将错误转换为可选值

您可以`try?`通过将其转换为可选值来处理错误。如果在评估`try?`表达式时抛出错误，则表达式的值为`nil`。例如，在下面的代码`x`，并`y`具有相同的价值和行为：

```
func someThrowingFunction() throws -> Int {
    // ...
}

let x = try? someThrowingFunction()

let y: Int?
do {
    y = try someThrowingFunction()
} catch {
    y = nil
}
```

如果`someThrowingFunction()`抛出一个错误，价值`x`和`y`为`nil`。否则，`x`和`y`的值就是函数返回的值。请注意，`x`并且`y`是任何类型`someThrowingFunction()`返回的可选项。在这里，函数返回一个整数，所以`x`和`y`是可选的整数。

`try?`当您想以相同的方式处理所有错误时， 使用可让您编写简洁的错误处理代码。例如，以下代码使用多种方法来获取数据，或者`nil`如果所有方法都失败，则返回。

```
func fetchData() -> Data? {
    if let data = try? fetchDataFromDisk() { return data }
    if let data = try? fetchDataFromServer() { return data }
    return nil
}
```

禁用错误传播

有时你知道抛出函数或方法实际上不会在运行时抛出错误。在这些情况下，您可以`try!`在表达式之前编写代码以禁用错误传播，并将调用包装在运行时断言中，以避免引发错误。如果实际发生错误，则会出现运行时错误。

例如，下面的代
加载时引发错误。在这种情况下，因为图像随应用程序一起提供，所以在运行时不会发生错误，所以适当地禁用错误传播。

```
let photo = try! loadImage(atPath: "./Resources/John Appleseed.jpg")
```

指定清理操作

`defer`在代码执行离开当前代码块之前， 您使用语句来执行一组语句。这个语句可以让你做任何必要的清理工作，而不管执行*如何*离开当前代码块 - 无论是因为抛出错误还是因为诸如`return`or 的语句而离开`break`。例如，您可以使用`defer`语句来确保关闭文件描述符并释放手动分配的内存。

一`defer`直到当前范围退出声明推迟执行。该语句由`defer`关键字和稍后要执行的语句组成。延迟语句可能不包含任何将控制从语句中移出的代码，例如a `break`或`return`语句，或者抛出错误。延迟操作的执行顺序与它们写入源代码的顺序相反。也就是说，第一个`defer`语句中的代码最后执行，第二个`defer`语句中的代码执行倒数第二个，依此类推。`defer`源代码顺序中的最后一条语句首先执行。

```
func processFile(filename: String) throws {
    if exists(filename) {
        let file = open(filename)
        defer {
            close(file)
        }
        while let line = try file.readline() {
            // Work with the file.
        }
        // close(file) is called here, at the end of the scope.
    }
}
```

上面的例子使用一个`defer`语句来确保`open(_:)`函数有相应的调用`close(_:)`。

> 注意
> `defer`即使没有涉及错误处理代码，也可以使用语句。