错误处理

cnswift.org/error-handling

*错误处理*是相应和接收来自你程序中错误条件的过程。Swift 给运行时可恢复错误的抛出、捕获、传递和操纵提供了一类支持。

有些函数和方法不能保证总能完全执行或者产生有用的输出。可选项用来表示不存在值，但是当函数错误，能够了解到什么导致了错误将会变得很有用处，这样你的代码就能根据错误来响应了。

举例来说，假设一个阅读和处理来自硬盘上文件数据的任务。这种情况下有很多种导致任务失败的方法，目录中文件不存在，文件没有读权限，或者文件没有以兼容格式编码。从这些错误中区分不同的状况将能够让程序解决和从这些错误中恢复，并且把不能解决的错误通知给用户。

> 注意
>
> 在 Swift 中的错误处理表示法兼容于 Cocoa 和 Objective-C 中的 NSError 类错误处理模式，参考*与 Cocoa 和 Objective-C 一起使用 Swift (Swift 4.1)*（官方链接）中的错误处理（官方链接）。

表示和抛出错误

在 Swift 中，错误表示为遵循
Error协议类型的值。这个空的协议明确了一个类型可以用于错误处理。

Swift 枚举是典型的为一组相关错误条件建模的完美配适类型，关联值还允许错误错误通讯携带额外的信息。比如说，这是你可能会想到的游戏里自动售货机会遇到的错误条件：

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

抛出一个错误允许你明确某些意外的事情发生了并且正常的执行流不能继续下去。你可以使用
throw 语句来抛出一个错误。比如说，下面的代码通过抛出一个错误来明确自动售货机需要五个额外的金币：

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

处理错误

当一个错误被抛出，周围的某些代码必须为处理错误响应——比如说，为了纠正错误，尝试替代方案，或者把错误通知用户。

在 Swift 中有四种方式来处理错误。你可以将来自函数的错误传递给调用函数的代码中，使用
do-catch 语句来处理错误，把错误作为可选项的值，或者错误不会发生的断言。每一种方法都在下边的章节中有详细叙述。

当函数抛出一个错误，它就改变了你程序的流，所以能够快速定位错误就显得格外重要。要定位你代码中的这些位置，使用
try 关键字——或者
try? 或 try! 变体——放在调用函数、方法或者会抛出错误的初始化器代码之前。这些关键字在下面的章节中有详细的描述。

> 注意
>
> Swift 中的错误处理，
> try,
> catch 和
> throw 的使用与其他语言中的异常处理很相仿。不同于许多语言中的异常处理——包括 Objective-C ——Swift 中的错误处理并不涉及调用堆栈展开，一个高占用过程。因此，
> throw 语句的性能特征与
> return 比不差多少。

使用抛出函数传递错误

为了明确一个函数或者方法可以抛出错误，你要在它的声明当中的形式参数后边写上
throws关键字。使用
throws标记的函数叫做*抛出函数*。如果它明确了一个返回类型，那么
throws关键字要在返回箭头 (
->)之前。

```
1  func canThrowErrors() throws -> String
2  func cannotThrowErrors() -> String
3
```

抛出函数可以把它内部抛出的错误传递到它被调用的生效范围之内。

> 注意
>
> 只有抛出函数可以传递错误。任何在非抛出函数中抛出的错误都必须在该函数内部处理。

在下边的例子中，
VendingMachine类拥有一个如果请求的物品不存在、卖光了或者比押金贵了就会抛出对应的

VendingMachineError错误的
vend(itemNamed:)方法：

```
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)")
    }
}
```

vend(itemNamed:)方法的实现使用了
guard语句来提前退出并抛出错误，如果购买零食的条件不符合的话。因为
throw语句立即传送程序控制，所以只有所有条件都达到，物品才会售出。

由于
vend(itemNamed:)方法传递它抛出的任何错误，所以你调用它的代码要么直接处理错误——使用 `do-catch` 语句，
try?或者
try!——要么继续传递它们。比如说，下边栗子中的
buyFavoriteSnack(person:vendingMachine:)同样是一个抛出函数，任何
vend(itemNamed:)方法抛出的函数都会向上传递给调用
buyFavoriteSnack(person:vendingMachine:)函数的地方。

```
let favoriteSnacks = [
    "Alice": "Chips",
    "Bob": "Licorice",
    "Eve": "Pretzels",
]
func buyFavoriteSnack(person: String, vendingMachine: VendingMachine) throws {
    let snackName = favoriteSnacks[person] ?? "Candy Bar"
    try vendingMachine.vend(itemNamed: snackName)
}
// Dispensing Chips
```

在这个栗子中，
buyFavoriteSnack(person:vendingMachine:)函数查找给定人的最爱零食并且尝试通过调用
vend(itemNamed:)方法来购买它们。由于
vend(itemNamed:) 方法会抛出错误，调用的时候要在前边用
try关键字。

可抛出的初始化器可以像可抛出函数那样传递错误。比如说，上面
PurchasedSnack 结构体的初始化器调用可抛出的函数作为初始化过程的一部分，然后它把遇到的任何错误都传递给它的调用者。

```
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

使用
do-catch语句来通过运行一段代码处理错误。如果do分句中抛出了一个错误，它就会与catch分句匹配，以确定其中之一可以处理错误。

这是
do-catch语句的通常使用姿势：

```
do {
    try expression
    statements
} catch pattern 1 {
    statements
} catch pattern 2 where condition {
    statements
}
```

在

catch后写一个模式来明确分句可以处理哪个错误。如果一个
catch分句没有模式，这个分句就可以匹配所有错误并且绑定这个错误到本地常量
error上。更多关于模式匹配的信息，见模式。
比如说，下面的代码用来匹配
VendingMachineError 枚举里的所有三个错误。

```
var vendingMachine = VendingMachine()
vendingMachine.coinsDeposited = 8
do {
    try buyFavoriteSnack("Alice", vendingMachine: vendingMachine)
    // Enjoy delicious snack
} 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.")
}
// prints "Insufficient funds. Please insert an additional 2 coins."
```

在上面的栗子当中，函数
buyFavoriteSnack(person:vendingMachine:)在
try表达式中被调用，因为它会抛出错误。如果抛出错误，执行会立即切换到
catch分句，它决定是否传递来继续。如果没有错误抛出，
do语句中剩下的语句将会被执行。

catch分句没有处理
do分句可能抛出的所有错误。如果没有
catch分句能处理这个错误，那错误就会传递到周围的生效范围当中。总之，错误必须得在周围*某个*范围内得到处理。

在不抛出错误的函数中，
do-catch 分句就必须处理错误。在可抛出函数中，要么
do-catch 分句处理错误，要么调用者处理。如果错误被传递到了顶层生效范围但还是没有被处理，你就会得到一个运行时错误了。

比如说，上面的例子可以重写一下，这样任何非
VendingMachineError 的错误就会被调用函数捕捉：

```
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
nourish(with:) 函数中，如果
vend(itemNamed:) 抛出
VendingMachineError 枚举中的某一错误，
nourish(with:) 就会打印一个消息以处理错误。否则的话，
nourish(with:) 就会把错误传递给它的调用者。错误就会被通用的
catch 分句捕捉。

转换错误为可选项

使用
try?通过将错误转换为可选项来处理一个错误。如果一个错误在
try?表达式中抛出，则表达式的值为
nil。比如说下面的代码x和y拥有同样的值和行为：

```
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，
x和
y的值就是
nil。另一方面，x和y的值是函数返回的值。注意
x和

y是可选的无论
someThrowingFunction()返回什么类型，这里函数返回了一个整数，所以x和y是可选整数。
当你想要在同一句里处理所有错误时，使用
try?能让你的错误处理代码更加简洁。比如，下边的代码使用了一些方法来获取数据，或者在所有方式都失败后返回
nil。

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

取消错误传递

事实上有时你已经知道一个抛出错误或者方法不会在运行时抛出错误。在这种情况下，你可以在表达式前写
try!来取消错误传递并且把调用放进不会有错误抛出的运行时断言当中。如果错误真的抛出了，你会得到一个运行时错误。

比如说，下面的代码使用了
loadImage(_:)函数，它在给定路径下加载图像资源，如果图像不能被加载则抛出一个错误。在这种情况下，由于图像跟着应用走，运行时不会有错误抛出，所以取消错误传递是合适的。

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

指定清理操作

使用
defer语句来在代码离开当前代码块前执行语句合集。这个语句允许你在以*任何*方式离开当前代码块前执行必须要的清理工作——无论是因为抛出了错误还是因为
return或者
break这样的语句。比如，你可以使用
defer语句来保证文件描述符都关闭并且手动指定的内存到被释放。

defer语句延迟执行直到当前范围退出。这个语句由
defer关键字和需要稍后执行的语句组成。被延迟执行的语句可能不会包含任何会切换控制出语句的代码，比如
break或
return语句，或者通过抛出一个错误。延迟的操作与其指定的顺序相反执行——就是说，第一个
defer语句中的代码会在第二个中代码执行完毕后执行，以此类推。

```
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
defer语句来保证
open(_:)函数能调用
close(_:)。

> 注意
>
> 就算没有涉及错误处理代码，你也可以使用
> defer语句。