它叫Optional, 却必不可少

使用guard简化optional unwrapping

(https://www.boxueio.com/series/optional-is-not-an-option/ebook/141)

为什么需要双层嵌套的Optional?

(https://www.boxueio.com/series/optional-is-not-an-option/ebook/143)

Chaining and Nil coalescing

Back to series (/series/optional-is-not-an-option)

字号
字号
默认主题
金色主题
暗色主题

通常，当我们要调用一个包含在optional中的对象的方法时，我们可能会像下面这样把两种情况分开处理：

```
var swift: String? = "Swift"
let SWIFT: String

if let swift = swift {
    SWIFT = swift.uppercased()
}
else {
    fatalError("Cannot uppercase a nil")
}
```

但是，当我们仅仅想获得一个包含结果的optional类型时，上面的写法就显得有点儿啰嗦了。实际上，我们有更简单的用法：

```
let SWIFT = swift?.uppercased() // Optional("SWIFT")
```

这样，我们就会得到一个新的 `Optional` 。并且，我们还可以把optional对象的方法调用串联起来：

```
let SWIFT = swift?.uppercased().lowercased()
// Optional("swift")
```

上面的形式，在Swift里，就叫做optional chaining。只要前一个方法返回optional类型，我们就可以一直把调用串联下去。但是，如果你仔细观察上面的串联方法，却可以发现一个有趣的细节：对于第一个optional，我们调用 `uppercased()` 方法使用的是 `?.` 操作符，并得到了一个新的 `Optional` ，然后，当我们继续串联 `lowercased()` 的时候，却直接使用了 `.` 操作符，而没有继续使用 `swift?.uppercased()?.lowercased()` 这样的形式，这说明什么呢?

这也就是说，optional在串联的时候，可以对前面方法返回的optional进行unwrapping，如果结果非 `nil` 就继续调用，否则就返回 `nil` 。

但是.....

这也有个特殊情况，就是如果调用的方法自身也返回一个optional（注意：作为调用方法自身，是指的诸如 `uppercased()` 这样的方法，而不是整个 `swift?.uppercased()` 表达式），那么你必须老老实实在每一个串联的方法前面使用 `?.` 操作符，来看下面这个例子。我们自己给 `String` 添加一对 `toUppercased / toLowercased` 方法，只不过，它们都返回一个 `String?` ，当 `String` 为空字符串时，它们返回 `nil` ：

```
extension String {
    func toUppercase() -> String? {
        guard self.isEmpty != 0 else {
            return nil
        }

        return self.uppercased()
    }

    func toLowercase() -> String? {
        guard self.characters.count != 0 else {
            return nil
        }

        return self.lowercased()
    }
}
```

然后，还是之前optional chaining的例子，这次，我们只能这样写：

```
let SWIFT1 = swift?.toUppercase()?.toLowercase()
```

注意到第二个 `?.` 了么，由于前面的 `toUppercase()` 返回了一个 `Optional` ，我们只能用 `?.` 来连接多个调用。而之前的 `uppercased()` 则返回了一个 `String` ，我们就可以直接使用 `.` 来串联多个方法了。

除此之外，一种不太明显的optional chaining用法，就是用来访问 `Dictionary` 中某个Value的方法，因为 `[]` 操作符本身也是通过函数实现的，它既然返回一个optional，我们当然也可以chaining：

```
let numbers = ["fibo6": [0, 1, 1, 2, 3, 5]]
numbers["fibo6"]?[0] // 0
```

因此，绝大多数时候，如果你只需要在optional不为 `nil` 时执行某些动作，optional chaining可以让你的代码简单的多，当然，如果你还了解了在chaining中执行的unwrapping语义，就能在更多场景里，灵活的使用这个功能。

Nil coalescing

除了optional chaining之外，Swift还为optional提供了另外一种语法上的便捷。如果我们希望在optional的值为 `nil` 时设定一个默认值，该怎么做呢？可能你马上就会想起Swift中的三元操作符：

```
var userInput: String? = nil
let username = userInput != nil ? userInput! : "Mars"
```

但就像你看到的， `?:` 操作符用在optional上的时候显得有些啰嗦，除此之外，为了实现同样的逻辑，你还无法阻止一些开发者把默认的情况写在 `:` 左边：

```
let username = userInput == nil ? "Mars" : userInput!
```

如此一来，事情就不那么让人开心了，当你穿梭在不同开发者编写的代码里，这种逻辑的转换迟早会把你搞疯掉。

于是，为了表意清晰的同时，避免上面这种顺序上的随意性，Swift引入了nil coalescing，于是，之前 `username` 的定义可以写成这样：

```
let username = userInput ?? "Mars"
```

其中， `??` 就叫做nil coalescing操作符，optional的值必须写在左边， `nil` 时的默认值必须写在右边。这样，就同时解决了美观和一致性的问题。相比之前的用法，Swift再一次从语言设计层面履行了更容易用对，更不容易用错的准则。

除了上面这种最基本的用法之外， `??` 也是可以串联的，我们主要在下面这些场景里，串联多个 `??` ：

首先，当我们想找到多个optional中，第一个不为 `nil` 的变量：

```
let a: String? = nil
let b: String? = nil
let c: String? = "C"

let theFirstNonNilString = a ?? b ?? c
// Optional("C")
```

在上面的例子里，我们没有在表达式最右边添加默认值。这在我们串联多个 `??` 时是允许的，只不过，这样的串联结果，会导致 `theFirstNonNilString` 的类型变成 `Optional` ，当abc都为 `nil` 时，整个表达式的值，就是 `nil` 。

而如果我们这样：

```
let theFirstNonNilString = a ?? b ?? "C"
```

`theFirstNonNilString` 的类型，就是 `String` 了。理解了这个机制之后，我们就可以把它用在 `if` 分支里，通过 `if let` 绑定第一个不为 `nil` 的optional变量：

```
if let theFirstNonNilString = a ?? b ?? c {
    print(theFirstNonNilString) // C
}
```

这样的方式，要比你在 `if` 条件分支中，写上一堆 `||` 直观和美观多了。

其次，当我们把一个双层嵌套的optional用在nil coalescing操作符的串联里时，要格外注意变量的评估顺序。来看下面的例子：

假设，我们有三个optional，第一个是双层嵌套的optional：

```
let one: Int?? = nil
let two: Int? = 2
let three: Int? = 3
```

当我们把one / two / three串联起来时，整个表达式的结果是2。这个很好理解，因为，整个表达式中，第一个非 `nil` 的optional的值是2：

```
one ?? two ?? three // 2
```

当我们把 one 的值修改成 `.some(nil)` 时，上面这个表达式的结果是什么呢?

```
let one: Int?? = .some(nil)
let two: Int? = 2
let three: Int? = 3

one ?? two ?? three // nil
```

此时，这个表达式的结果会是 `nil` ，为什么呢？这是因为：

1. 评估到 one 时，它的值是 `.some(nil)` ，但是 `.some(nil)` 并不是 `nil` ，于是它自然就被当作第一个非 `nil` 的optional变量被采纳了；
2. 被采纳之后，Swift会unwrapping这个optional的值作为整个表达式的值，于是就得到最终 `nil` 的结果了；

理解了这个过程之后，我们再来看下面的表达式，它的值又是多少呢?

```
(one ?? two) ?? three // 3
```

正确的答案是3。这是因为我们要先评估 `()` 内的表达式，按照刚才我们提到的规则， `(one ?? two)` 的结果是 `nil` ，于是 `nil ?? three` 的结果，自然就是3了。

当你完全理解了双层嵌套的optional在上面三个场景中的评估方式之后，你就明白为什么要对这种类型的串联保持高度警惕了。因为，optional的两种值 `nil` 和 `.some(nil)` ，以及表达式中是否存在 `()` 改变优先级，都会影响整个表达式的评估结果。

What's next?

在这一节里，我们了解了optional chaining以及nil coalescing的常见用法，这些看似只是更方便的写法，却有着它们自己实际而又重要的意义。因此，我们应该记住它们，并有效加以利用。接下来，我们将着重讨论双层嵌套的optional，它当然也不是一个凭空想象出来的偶发情况，Swift编译器甚至还为它做了诸多自动化的操作，在下一节中，我们就来一探究竟。

使用guard简化optional unwrapping
(https://www.boxueio.com/series/optional-is-not-an-option/ebook/141)

为什么需要双层嵌套的Optional?
(https://www.boxueio.com/series/optional-is-not-an-option/ebook/143)

猎云网采访报道泊学 (http://www.lieyunwang.com/archives/144329)
Dec 31, 2015

What most schools do not teach (https://www.boxueio.com/what-most-schools-do-not-teach)
Dec 21, 2015

一个工作十年PM终创业的故事（一） (https://www.boxueio.com/founder-story)
May 8, 2015

泊学相关

关于泊学 ›

加入泊学 ›

泊学用户隐私以及服务条款 (HTTPS://WWW.BOXUEIO.COM/TERMS-OF-SERVICE)

版权声明 (HTTPS://WWW.BOXUEIO.COM/COPYRIGHT-STATEMENT)

联系泊学

Email: 10[AT]boxue.io (mailto:10@boxue.io)
QQ: 2085489246

友情链接 SwiftV (http://www.swiftv.cn) | Seay信息安全博客 (http://www.cnseay.com) | Swift.gg (http://swift.gg/) | Laravist (http://laravist.com/) | SegmentFault (https://segmentfault.com) | 靛青K的博客 (http://blog.dianqk.org/)