它叫Optional, 却必不可少

为什么需要双层嵌套的Optional?

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

什么时候需要force unwrapping

(https://www.boxueio.com/series/optional-is-not-an-option/ebook/145)

Optional map和flatMap的应用和实现

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

在之前的内容里，我们提到过，当我们要在optional的值不为 `nil` 时，执行一些操作，可以使用 `if let` 绑定optional的值，然后，就可以在 `if` 语句内部，直接访问它了，例如，我们要把一个 `String?` 的内容在非空时，转换为大写，可以这样：

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

if let swift = swift {
    SWIFT = swift.uppercased()
}
```

然后，一个似曾相识的问题就来了，如果需要 `SWIFT` 是一个常量怎么办呢？如果，你把一个optional理解为是一个包含值和 `nil` 的集合类型，就自然会有更好的解决办法了。

Optional map

没错，既然 `map` 可以用在集合类型里转换元素，当然也可以用在 `Optional` 类型上：

```
let SWIFT = swift.map { $0.uppercased() }
// Optional("SWIFT")
```

这样，我们就得到了一个新的 `Optional<String>` ，值是“SWIFT”。对于optional类型来说，如果它的值非 `nil` ， `map` 就会把unwrapping的结果传递给它的closure参数，否则，就直接返回 `nil` 。我们完全可以按照这个思路，自己给 `Optional` 实现一个 `myMap` ：

```
extension Optional {
    func myMap<T>(_ transform: (Wrapped) -> T) -> T? {
        if let value = self {
            return transform(value)
        }

        return nil
    }
}
```

在这个实现里，唯一要说明的，就是 `Wrapped` ，这是 `Optional` 类型的泛型参数，表示optional实际包装的值的类型。理解了这个之后， `myMap` 的实现就完全都是套路了。

然后，我们用之前的例子试一下：

```
let SWIFT = swift.myMap { $0.uppercased() }
// Optional("SWIFT")
```

结果和之前，应该是一样的。

理解了这个方式之后，当你再要返回一个optional的时候，除了使用 `if...else...` 对非空情况单独处理之外，直接使用 `map` 通常会是个更好的方法。

接下来，我们再来看一个map更复杂的应用场景。例如，对于之前我们在集合的内容里提到的对 `Array` 中所有元素求和：

```
let numbers = [1, 2, 3, 4]
let sum = numbers.reduce(0, +) // 10
```

其实，第一个参数0是个很没必要的事情，为了让下面的代码可以正常工作：

```
let sum = numbers.reduce(+) // 10
```

我们可以这样重载 Array.reduce：

```
extension Array {
    func reduce(_ nextResult:
        (Element, Element) -> Element) -> Element? {
        guard let first = first else { return nil }

        return dropFirst().reduce(first, nextResult)
    }
}
```

但这样的实现仍旧有改进的空间，由于 map 方法可以根据optional变量的值自动执行后续的行为，我们可以对 first 属性调用 map 来合并上面的 guard 和 return：

```
extension Array {
    func reduce(_ nextResult:
        (Element, Element) -> Element) -> Element? {

        return first.map {
            dropFirst().reduce($0, nextResult)
        }
    }
}
```

这样，如果 first 为 nil， map 就返回 nil，否则，就从 Array 中的第一个元素开始 reduce 。然后，只接受一个 + 版本的 reduce 就也可以正常工作了。

Optional flatMap

介绍完了 map，我们不难联想到，如果 map 方法返回的也是一个optional，我们是否也应该有 flatMap 来处理双层嵌套optional类型的变换呢?

当然，Swift已经在标准库中，为你实现了一个。来看下面的例子：

```
let stringOne: String? = "1"
let ooo = stringOne.map { Int($0) }
type(of: ooo) // Optional<Optional<Int>>
```

此时，由于 Int($0) 返回一个 Int?，而 map 又会返回一个optional类型，因此， ooo 的类型，就变成了 Int??，也就是 Optional<Optional<Int>> 。但我们只是尝试把 stringOne 变成一个整数，因此，应该是一个把 Optional<String> 变成 Optional<Int> 的操作。这时， flatMap 就派上用场了：

```
let oo = stringOne.flatMap { Int($0) }
type(of: oo) // Optional<Int>
```

相比于 map 来说， flatMap 会对它的closure参数的返回值进行处理，当返回非 nil 时，就直接把这个返回值返回；否则，就返回 nil 。这样，我们就获得了一个新的单层optional对象。

当然，为了避免双层嵌套的optional，我们也可以用 if let 来实现类似的效果：

```
if let stringOne = stringOne, let o = Int(stringOne) {
    print(o)    // 1
    type(of: o) // Int
}
```

在上面的代码里，我们用第一个 if let 绑定了 stringOne 中的非 nil 值，并尝试把这个值转换成整数。由于这个转换结果也是一个optional，我们再次使用了 if let 绑定了转换后的非 nil 结果。

实际上， Optional.flatMap 就完全是基于 if let 来实现的：

```
extension Optional {
    func myFlatMap<T>(_ transform: (Wrapped) -> T?) -> T? {
        if let value = self,
            let mapped = transform(value) {
            return mapped
        }

        return nil
    }
}
```

看到了吧， `flatMap` 和 `if let` 简直如出一辙。

如何遍历一个包含optional的集合

在理解了集合和optional类型各自的 `map` 和 `flatMap` 之后，我们来看一个稍复杂一些的例子：如何遍历一个包含optional的数组，并对每个元素做一些操作呢？

假设，我们有一个包含数字的字符串数组：

```
let ints = ["1", "2", "3", "4", "five"]
```

现在，要把 `ints` 中的元素转换成 `Int` 然后求和，该怎么做呢？最“朴素”的做法，当然是先对 `ints` 调用 `map` 把 `[String]` 变成 `[Int?]`

```
ints.map ({ Int($0) })
```

然后，在 `for...in` 中，使用value binding读取数组中的每一个非 `nil` 值，并且求和：

```
var all = 0

for case let int? in ints.map({ Int($0) }) {
    all += int
}
```

仔细分析上面的过程，实际上分成四个独立的步骤：

1. 把 `ints` 中所有的元素变形，形成新的序列；
2. 在第一步的结果中剔除所有的 `nil` ；
3. 在第二步的结果中unwrapping所有的optional；
4. 对第三步的结果执行 `reduce` 求和；

这里， `reduce` 已经是标配了，我们来自己实现一个 `myFlatMap` 方法解决前三步的问题：

```
extension Sequence {
    func myFlatMap<T>(_ transform:
        (Iterator.Element) -> T?) -> [T] {
        return self.map(transform)
            .filter { $0 != nil }
            .map { $0! }
    }
}
```

有了这个方法之后，我们就可以用：

```
let all = ints.myFlatMap { Int($0) }.reduce(0, +)
//10
```

来计算 `ints` 中，所有非 `nil` 元素的和了。实际上，Swift标准库中，已经为序列类型提供了一个 `flatMap` 方法，专门用来处理“在序列中变换并筛选所有非 `nil` 元素”的任务：

```
let intOnes = ints.flatMap { Int($0) }.reduce(0, +)
//10
```

而它的实现思路，和我们之前自己实现的版本，几乎是一样的。

What's next?

以上，就是针对optional类型 `map` 和 `flatMap` 的常见应用场景以及实现原理。至此，关于optional类型常见的使用方式，我们就介绍完了。在下一节中，我们会专门讨论一下optional类型的force unwrapping，为什么要force unwraping？究竟应该在哪些场景中使用它们呢？

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

什么时候需要force unwrapping ⏭
(https://www.boxueio.com/series/optional-is-not-an-option/ebook/145)

职场漂泊的你，每天多学一点。

从开发、测试到运维，让技术不再成为你成长的绊脚石。我们用打磨产品的精神去传播知识，把最新的移动开发技术，通过简单的图表， 清晰的视频，简明的文字和切实可行的例子一一向你呈现。让学习不仅是一种需求，也是一种享受。

泊学动态

一个工作十年PM终创业的故事（二） (https://www.boxueio.com/after-the-full-upgrade-to-swift3)
Mar 4, 2017

人生中第一次创业的"10有" (https://www.boxueio.com/founder-chat)
Jan 9, 2016

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/)