通过closure参数化对数组元素的变形操作

Back to series (/series/collection-types)

字号
字号
默认主题
金色主题
暗色主题

就像我们在前几节中提到的一样，当你要对 Array 做一些处理的时候，像C语言中类似的循环和下标，都不是理想的选择。Swift有一套自己的“现代化”手段。简单来说，就是用closure来参数化对数组的操作行为。这听着有点儿抽象，我们从一个最简单的例子开始。

从循环到map

假设我们有一个简单的Fibonacci序列：`[0, 1, 1, 2, 3, 5]`。如果我们要计算每个元素的平方，怎么办呢?

一个最朴素的做法是 for 循环：

```
var fibonacci = [0, 1, 1, 2, 3, 5]
var squares = [Int]()

for value in fibonacci {
    squares.append(value * value)
}
```

也许，现在你还觉得这样没什么不好理解，但是，想象一下这段代码在几十行代码中间的时候，或者当这样类似的逻辑反复出现的时候，整体代码的可读性就不那么强了。

如果你觉得这还不是个足够引起你注意的问题，那么，当我们要定义一个常量 squares 的时候，上面的代码就完全无法胜任了。怎么办呢？先来看解决方案：

```
// [0, 1, 1, 4, 9, 25]
let constSquares = fibonacci.map { $0 * $0 }
```

上面这行代码，和之前那段 for 循环执行的结果是相同的。显然，它比 for 循环更具表现力，并且也能把我们期望的结果定义成常量。当然， map 并不是什么魔法，无非就是把 for 循环执行的逻辑，封装在了函数里，这样我们就可以把函数的返回值赋值给常量了。我们可以通过 extension 很简单的自己来实现 map：

```
extension Array {
    func myMap<T>(_ transform: (Element) -> T) -> [T] {
        var tmp: [T] = []
        tmp.reserveCapacity(count)

        for value in self {
            tmp.append(transform(value))
        }

        return tmp
    }
}
```

虽然和Swift标准库相比， myMap 的实现中去掉了和异常声明相关的部分。但它已经足以表现 map 的核心实现过程了。除了在 append 之前使用了 reserveCapacity 给新数组预留了空间之外，它的实现过程和一开始我们使用的 for 循环没有任何差别。

> 如果你还不了解 Element 也没关系，把它理解为 Array 中元素类型的替代符就好了。在后面我们讲到 Sequence 类型的时候，会专门提到它。

完成后，当我们在playground里测试的时候：

```
// [0, 1, 1, 4, 9, 25]
let constSequence1 = fibonacci.myMap { $0 * $0 }
```

就会发现执行结果和之前的 constSequence 是一样的了。

参数化数组元素的执行动作

其实，仔细观察 myMap 的实现，就会发现它最大的意义，就是保留了遍历 Array 的过程，而把要执行的动作留给了 myMap 的调用者通过参数去定制。而这，就是我们一开始提到的**用closure来参数化对数组的操作行为**的含义。

有了这种思路之后，我们就可以把各种常用的带有遍历行为的操作，定制成多种不同的遍历“套路”，而把对数组中每一个元素的处理动作留给函数的调用者。但是别急，在开始自动动手造轮子之前，Swift library已经为我们准备了一些，例如：

首先，是找到最小、最大值，对于这类操作来说，只要数组中的元素实现了 Equatable protocol，我们甚至无需定义对元素的具体操作：

```
fibonacci.min() // 0
fibonacci.max() // 5
```

使用 min 和 max 很安全，因为当数组为空时，这两个方法将返回 nil 。

其次，过滤出满足特定条件的元素，我们只要通过参数指定筛选规则就好了：

```
fibonacci.filter { $0 % 2 == 0 }
```

第三，比较数组相等或以特定元素开始。对这类操作，我们需要提供两个内容，一个是要比较的数组，另一个则是比较的规则：

```
// false
fibonacci.elementsEqual([0, 1, 1], by: { $0 == $1 })
// true
fibonacci.starts(with: [0, 1, 1], by: { $0 == $1 })
```

第四，最原始的 for 循环的替代品：

```
fibonacci.forEach { print($0) }
// 0
// 1
// ...
```

要注意它和 map 的一个重要区别： **forEach 并不处理closure参数的返回值**。因此它只适合用来对数组中的元素进行一些操作，而不能用来产生返回结果。

第五、对数组进行排序，这时，我们需要通过参数指定的是排序规则：

```
// [0, 1, 1, 2, 3, 5]
fibonacci.sorted()
// [5, 3, 2, 1, 1, 0]
fibonacci.sorted(by: >)

let pivot = fibonacci.partition(by: { $0 < 1 })
fibonacci[0 ..< pivot] // [5, 1, 1, 2, 3]
fibonacci[pivot ..< fibonacci.endIndex] // [0]
```

其中， sorted(by:) 的用法是很直接的，它默认采用升序排列。同时，也允许我们通过 by 自定义排序规则。在这里 > 是 { $0 > $1 } 的简写形式。Swift中有很多在不影响语义的情况下的简写形式。

而 partition(by:) 则会**先对传递给它的数组进行重排**，然后根据指定的条件在重排的结果中返回一个分界点位置。这个分界点分开的两部分中，前半部分的元素都不满足指定条件；后半部分都满足指定条件。而后，我们就可以使用range operator来访问这两个区间形成的 Array 对象。大家可以根据例子中注释的结果，来理解partition的用法。

第六，是把数组的所有内容，“合并”成某种形式的值，对这类操作，我们需要指定的，是合并前的初始值，以及“合并”的规则。例如，我们计算 fibonacci 中所有元素的和：

```
fibonacci.reduce(0, +) // 12
```

在这里，初始值是0，和第二个参数 + ，则是 { $0 + $1 } 的缩写。

通过这些例子，你应该能感受到了，这些通过各种形式封装了遍历动作的方法，它们之中的任何一个，都比直接通过 for 循环实现具有更强的表现力。这些API，开始让我们的代码从面向机器的，转变成面向业务需求的。因此，在Swift里，你应该试着让自己转变观念，当你面对一个 Array 时，你真的几乎可以忘记下标和循环了。

区分修改外部变量和保存内部状态

当我们使用上面提到的这些带有closure参数的 Array 方法时，一个不好的做法就是通过closure去修改外部变量，并依赖这种副作用产生的结果。来看一个例子：

```
var sum = 0
let constSquares2 = fibonacci.map { (fib: Int) -> Int in
    sum += fib
    return fib * fib
}
```

在这个例子里， map 的执行产生了一个副作用，就是对 fibonacci 中所有的元素求和。这不是一个好的方法，我们应该避免这样。你应该单独使用 reduce 来完成这个操作，或者如果一定要在closure参数里修改外部变量，哪怕用 forEach 也是比 map 更好的方案。

但是，在函数实现内部，专门用一个外部变量来保存closure参数的执行状态，则是一个常用的实现技法。例如，我们要创建一个新的数组，其中每个值，都是数组当前位置和之前所有元素的和，可以这样：

```
extension Array {
    func accumulate<T>(_ initial: T,
                       _ nextSum: (T, Element) -> T) -> [T] {
        var sum = initial

        return map { next in
            sum = nextSum(sum, next)
            return sum
        }
    }
}
```

在上面这个例子里，我们利用 map 的closure参数捕获了 sum ，这样就保存了每一次执行 map 时，之前所有元素的和。

```
// [0, 1, 2, 4, 7, 12]
fibonacci.accumulate(0, +)
```

What's next?

在这一节中，我们向大家介绍了Swift中，使用 Array 最重要的一个思想：**通过closure来参数化对数组的操作行为**。在Swift标准库中，基于这个思想，为我们提供了在各种常用数组操作场景中的API。因此，当你下意识的开始用一个循环处理数组时，让自己停一下，去看看 Array 的官方文档，你一定可以找到更现代化的处理方法。在下一节，我们将着重了解一下标准库中的三个API： filter 、 reduce 和 flatMap 。之所以选择它们，是因为 filter 和 map 是构成其它各种API的基础，而 flatMap 则不太容易理解。

◀ 用Swift的方式使用Array
(https://www.boxueio.com/series/collection-types/ebook/126)

Filter / Reduce / FlatMap的实现和扩展 ▶
(https://www.boxueio.com/series/collection-types/ebook/128)

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/)