Swift 3 的第一印象

String和NSString处理Unicode上的差异
(https://www.boxueio.com/series/swift-up-and-running/ebook/107)

基于unicode的字符串常用操作
(https://www.boxueio.com/series/swift-up-and-running/ebook/122)

为什么String不是一个Collection类型?

Back to series (/series/swift-up-and-running)

字号
字号
默认主题
金色主题
暗色主题

在上几节中，我们了解了通过不同的code unit构建unicode字符串的过程，也提到了Swift中的 `String` 类型不再是传统意义上的“定宽字符数组”。但是，也许我们还是一时难以在执行层面接受 `String` 不是一个集合类型这样的现实。毕竟，如果 `String` 不是一个集合类型，我该怎么遍历一个字符串呢？该怎么查找字符的位置呢？我们所有对字符串的操作逻辑都告诉我们，把 `String` 理解为一个集合才是对的。

如果String是一个Collection类型...

借助于protocol oriented programming，似乎我们可以很容易强制让 `String` 类型是一个 `Collection`，让它通过 `extension` 遵从 `Collection` protocol就好了：

```
extension String: Collection {
    // Do not need to implement anything.
    // Collection has enough default implementation.
}
```

然后，我们就可以给 `String` 对象用上各种 `Collection` 工具了，例如：

```
var swift = "Swift is fun" // Swift is fun
swift.dropFirst(9) // fun
```

甚至，下面的代码都可以正常工作：

```
let f = "👨‍👩‍👧‍👦"
f.dropFirst(1)
```

这看上去不很好么？我们似乎更有理由相信，把 `String` 理解为一个 `Collection` 是没问题的。那么，为什么Swift 3中，`String` 不是一个 `Collection` 呢?

因为一旦如此，所有在 `Collection` 中定义的算法用在 `String` 对象的时候，我们都会默认它们是正确的。但实际上，我们很难同时做到算法的逻辑正确和unicode语义正确。尽管在上面的例子中，我们看到了 `String.Character` 类型已经尽可能在逻辑层面上忽略掉unicode的各种细节，但有些问题，仍旧难以处理，例如：

```
let cafee = "caf\u{0065}\u{0301}" // café
cafee.dropFirst(4) // ""
cafee.dropLast(1)  // !!! Runtime error !!!
```

在上面的例子里：

- 删掉前面4个“元素”会留下最后的声调符么？我相信，如果问到足够多的人，一定可以得到不同的答案，但实际的情况是，`cafee` 的值会变成一个空字符串，这和你想象的一样么？
- 调用 `dropLast(1)` 会删掉字符é，还是会删掉声调符呢？这次，无论你选择哪个，都不对，因为这个调用会直接引发运行时错误；

这就是把 `String` 作为一个集合类型带来的问题：**面对unicode复杂的组合规则，我们很难保证所有的集合算法都是安全并且语义正确的。**

还得让String用起来像一个集合...

理解表达字符串的view

尽管前面我们说了这么一大堆，但不可否认的一个事实就是，我们对字符串的操作仍旧和集合是非常类似的，因为字符串看上去就一个一连串单个字符的组合。所以，Swift的开发者还是决定让这个类型用起来像一个集合类型。

为了达到这个目的，第一个要解决的就是，如何让Swift理解字符串中的“字符集合”。为了不存在歧义，`String` 为开发者提供了一些不同的"view"。简单来说，就是告诉 `String` 类型如何去理解字符串中的内容，它们是 `String` 的不同属性。

首先，是根据unicode scalar的编码方式划分的“view”：

- **unicodeScalar**：按照字符串中每一个字符的unicode scalar来形成集合；
- **utf8**：按照字符串中每一个字符的UTF-8编码来形成集合；
- **utf16**：按照字符串中每一个字符的UTF-16编码来形成集合；

当我们明确指定了上面的属性之后，字符串就可以安全的当做集合来处理了。例如：

```
cafee.unicodeScalars.forEach { print($0) }
cafee.utf8.forEach { print($0) }
cafee.utf16.forEach { print($0) }
```

我们就可以在控制台得到字符串“café”的每一个unicode scalar / UTF-8 / UTF-16编码的数值。或者，对于上面会带来运行时错误的 `dropLast(:)` 方法，现在也可以正常工作了：

```
cafee.unicodeScalars.dropLast(1) // cafe
cafee.utf16.dropLast(1)          // cafe
cafee.utf8.dropLast(1)           // cafe
```

在上面的例子里，当我们使用 `dropLast(:)` 删掉最后一个unicode scalar和UTF-16编码时，“café”的声调字符就去掉变成了“cafe”，而我们删掉UTF-8编码的最后一个元素时，由于声调符需要两个UTF-8编码，因此最后一个会留意下无法识别的乱码。但无论如何，当我们指定了字符串中元素的构成方式之后，它们至少都可以像集合一样安全的使用了。

其次，是一个需要额外多讲一些的“view”： `characters` ，它是一个 `String.CharacterView` 类型的属性。这个“view”是按照unicode grapheme clusters计算字符串的字符个数，也就是最接近我们肉眼看到的字符的view。因此 `String.characters` 形式上就可以理解为“由我们看到的字符构成的字符数组”。一个最简单直接的例子就是我们之前用过的统计字符个数：

```
let cafee = "caf\u{0065}\u{0301}"
cafee.characters.count // 4
```

`String.characters` 还提供了两个索引位置：

```
cafee.characters.startIndex // 0
cafee.characters.endIndex   // 5
```

它们是 `String.CharacterView.Index` 类型的属性，表示字符串开始，以及最后一个字符的下一个位置。在 `String` 的实现中，我们可以找到下面的类型定义：

```
public struct String {
    /// The index type for subscripting a string.
    public typealias Index = String.CharacterView.Index
}
```

也就是说， `characters` 属性中的索引，是可以直接用来索引字符串特定位置的。但是由于你无法确定两个字符之间到底相隔了多少字符，因此，你并不能像访问一个 `Array` 一样去使用 `characters` ：

```
cafee.characters[2] // !!! This is WRONG !!!
```

`String.CharacterView` 只遵从了 `BidirectionalCollection` protocol，因此，它只能顺序向前，或者向后移动，而不能随机指定位置移动。如果我们要获得特定位置的字符，只能使用 `index(_:offsetBy:)` 这个方法。例如：

```
let index = cafee.index(cafee.startIndex, offsetBy: 3) // 3
cafee[index] // é
```

> 在上面的例子里，使用 `cafee.index(cafee.characters.startIndex, offsetBy: 3)` 是一样的。

或者，如果你担心 `offsetBy` 参数越界，还可以使用 `index(_:offsetBy:limitedBy:)` 方法添加一个末尾位置的限制，一旦offset越界，这个方法就会返回 `nil` ：

```
let index = cafee.index(
    cafee.startIndex,
    offsetBy: 100,
    limitedBy: cafee.endIndex) // nil
```

让String支持[]是个好点子么?

看到这里，你可能会想了，既然有了 index 方法，那我自己给 String 添加一个 extension 去实现 subscript 方法不就用起来更像个数组了么？来看下面的代码，这样真的好用么?

```
extension String {
    subscript(index: Int) -> Character {
        guard let index = self.index(startIndex,
            offsetBy: index, limitedBy: endIndex) else {

            fatalError("String index out of range.")
        }

        return self[index]
    }
}

cafee[3] // é
```

在上面这个例子里， cafee[3] 可以得到正确的结果，却也给开发者带来了一些在性能上的错觉。因为前面我们提到过。 String.CharacterView 只遵从了 BidirectionalCollection ，也就意味着 index 实际上是一个 O(n) 方法，它需要一个个的挪到我们期望的offset。但是， cafee[3] 这样的用法却不会给我们任何关于性能的提示。我们只会觉得这是一个和数组访问一样的O(1)操作。于是，当我们无意间写下下面的代码时：

```
for i in 0..<4 {
    print(cafee[i])
}
```

我们就制造了一个$O(n^2)$的性能瓶颈。因此，让 String 支持 [] 并不是一个好主意。

What's next?

这就是我们这一节的全部内容。我们从unicode语义正确这个角度，理解了为什么 String 不能被实现成一个 Collection 。也了解到了在unicode这个复杂的场景里， String 如何通过各种view来支持“顺序访问字符”这个操作。

在下一节里，我们将讨论一些常用的字符串操作。了解它们在unicode环境里，是如何通过Swift完成的。

◀ String和NSString处理Unicode上的差异
(https://www.boxueio.com/series/swift-up-and-running/ebook/107)

基于unicode的字符串常用操作 ▶
(https://www.boxueio.com/series/swift-up-and-running/ebook/122)

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/)