Swift 3 的第一印象

忘记旧有的"C风格"字符串吧 (https://www.boxueio.com/series/swift-up-and-running/ebook/106)

为什么String不是一个Collection类型? (https://www.boxueio.com/series/swift-up-and-running/ebook/121)

String和NSString处理Unicode上的差异

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

字号
字号
默认主题
金色主题
暗色主题

4. String和NSString处理Unicode时的差异

上一节，我们了解了关于unicode的基本知识。其中，我们提到了unicode长度是可变的。除了code unit的长度可变之外，在这一节，我们将看到unicode另外一个可变的特性，即组成同一个字符的code unit组合也是可变的。而区分 `String` 和 `NSString` 的一个重要方式，就是它们对unicode的这个特性的处理方式，是不同的。为了理解这个事情，我们从unicode grapheme clusters说起。

Unicode grapheme clusters

首先，我们定义一个字符串：

```
let cafe = "Caf\u{00e9}"
```

```
5 let cafe = "Caf\u{00e9}"    "Café"
```

上个视频中我们已经提到过，Swift里，我们可以使用 `\u{}` 这样的方式使用unicode scalar定义unicode字符。

对于单词Café中的最后一个字符来说，它的unicode scalar是 `U+00E9` ，名字是 `LATIN SMALL LETTER E WITH ACUTE` 。每一个unicode都有一个scalar值以及一个全大写字母表示的名称。

为了表示这个字符é，除了使用它的unicode scalar外，我们可以用两个其它的unicode字符拼起来：

- 英文字母 `e` ，它的unicode scalar是 `U0065` ，name是 `LATIN SMALL LETTER E` ；
- 声调字符 `'` ，它的unicode scalar是 `U0301` ，name是 `COMBINING ACUTE ACCENT` ；

当我们把这两个字符像下面这样组合起来的时候：

```
let cafee = "Caf\u{0065}\u{0301}"
```

```
5 let cafe = "Caf\u{00e9}"    "Café"
6 let cafee = "Caf\u{0065}\u{0301}"    "Café"
```

尽管cafee的定义中貌似有5个字符，但实际显示出来的最后一个字符和之前用unicode scalar定义是一样的。我们管 `\u{0065}\u{0301}` 就叫做grapheme cluster。

既然同一个unicode字符可以有多种表现形式，那么由不同code unit构成的字符串相等么？对此，Swift中的 `String` 和Objective-C中的 `NSString` 处理方式却是有差别的，这种差别，是区分它们最明显的地方之一。

Canonically equivalent

为了能识别上面 `cafe` 和 `cafee` 的情况，unicode规范中提出了一个概念：canonically equivalent。如何理解它呢？我们先来看Swift是如何识别 `cafe` 和 `cafee` 的。

Swift String

当我们要读取一个字符串中所有的字符时，可以访问 `String` 对象的 `characters` 属性，在下个视频中，我们会更多讲到它的用法：

```
8 cafe.characters.count    4
9 cafee.characters.count    4
```

尽管 cafee 中最后一个字符的定义使用了两个code unit，Swift可以识别的 Character 中字符的个数也是4。

但是，当我们查看 cafe 和 cafee 的UTF-8和UTF-16编码的个数时，就能看到它们的区别了：

```
cafe.utf8.count
cafee.utf8.count

cafe.utf16.count
cafee.utf16.count
```

为什么会这样呢？我们用UTF-8编码举例：

对于 cafe 来说， é 的UTF-8编码是 C3 A9 ，加上前面 Caf 的编码是 43 61 66 ，因此 cafe 的UTF-8编码个数是5；

对于 cafee 来说，声调字符 ' 的UTF-8编码是 CC 81 ，加上前面 Cafe 的UTF-8编码是 43 61 66 65 ，因此是6个，它相当于 Cafe' ；

理解了之后，你可以自己去推算一下UTF-16的情况。

尽管cafe和cafee的编码方式不同，当我们在Swift中，比较cafe和cafee时，结果会是 true ：

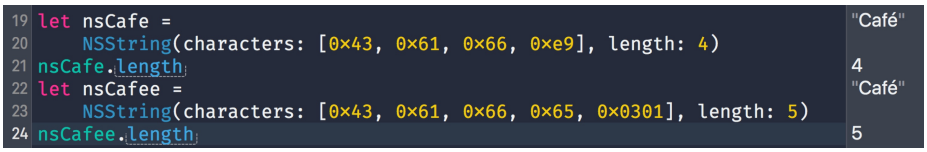
```
cafe == cafee
```

这就是unicode canonically equivalent的含义，通过这些例子，你也能更好的了解到Swift在unicode表意正确上作出的努力。

NSString

而当我们把这些例子用在 NSString 上，情况就会有些不同。用同样的code unit定义下面两个 NSString 对象：

```
let nsCafe =
    NSString(characters: [0x43, 0x61, 0x66, 0xe9], length: 4)
nsCafe.length
let nsCafee =
    NSString(characters: [0x43, 0x61, 0x66, 0x65, 0x0301], length: 5)
nsCafee.length
```



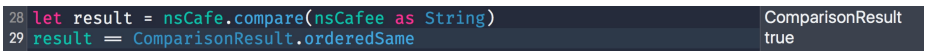
从图中可以看到，同样是使用不同的code unit构建字符串"Café"，在 NSString 看来，它们是长度不同的两个字符串。

因此，当我们比较 nsCafe 和 nsCafee 的时候，结果也是没有意外的 false ：

```
nsCafe == nsCafee
```

因此， == 对 NSString 来说，并没有执行canonically equivalent的语义。为了在不同的 NSString 对象之间进行语义比较，我们只能这样：

```
let result = nsCafe.compare(nsCafee as String)
result == ComparisonResult.orderedSame
```



从图中可以看到，这样就能按照canonically equivalent的方式判断相等了。

Unicode 9.0?

除了这个视频里提到了用两个code unit组合来实现一个字符之外，我们还可以使用多个code unit组合成一个“字符”，例如：

给 é 外围再套个圈：

```
let circleCafee = cafee + "\u{20dd}"
circleCafee.characters.count
```

```
31 let circleCafee = cafee + "\u{20dd}"                    "Caf⃝"
32 circleCafee.characters.count                             4
```

可以看到，尽管我们用3个code unit拼接了一个奇怪的字符， `circleCafee` 的字符个数，仍旧是4。

至此，事情看似一切正常。基于表达语意的计算方式也很符合人们的直觉。但事情并没有我们想象的这么简单，如果你挖掘一些emoji，就会发现一些有趣的现象：

```
35 "💁".characters.count                                    2
36 "👭".characters.count                                    4
```

一个亚洲姑娘的字符数是2，而一群小伙伴的字符个数是4。为什么会这样呢？其实它们和字符 é 构成的原理是类似的。都是通过多个code unit组合而成的字符。但是，它们的字符计算方式却和我们之前看到的不太一样。

你可以把4个小伙伴单独的字符都找出来，然后用 `\u{200d}` 把它们粘合起来，在Swift中就会看到，它们是相等的：

```
38 "👩\u{200D}👩\u{200D}👧\u{200D}👧" == "👭"                true
```

甚至，如果你把中国和美国的国旗定义成一个字符串，然后统计字符串中字符个数的时候，好像Swift都不会在意后者的存在（LoL）：

```
40 let flags = "🇨🇳🇺🇸"                                     "🇨🇳🇺🇸"
41 flags.characters.count                                   1
```

因此，拼接字符 é 在语义上表达的一致性之外，后面我们举的这些例子，都是既有的unicode编码方式中还需要解决问题。在2016年6月更新的Unicode 9.0 (http://www.unicode.org/versions/Unicode9.0.0/) 中，对于grapheme cluster的边界问题作出了修正，相信Swift 3在不久也会对上面的问题作出调整。

What's next?

以上就是这一节的全部内容，我们了解了unicode在比较字符串时执行的语意，也看到了Swift `String` 在对unicode支持上，相比 `NSString` 做出的改进。当然，我们还了解到了一些既有的unicode存在的问题。接下来，我们将了解当 `String` 不再是一个集合类型的时候，如何正确遍历其中的内容。

忘记旧有的"C风格"字符串吧
(https://www.boxueio.com/series/swift-up-and-running/ebook/106)

为什么String不是一个Collection类型？
(https://www.boxueio.com/series/swift-up-and-running/ebook/121)

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/)