The Swift Programming Language in Chinese

https://github.com/letsswift

http://letsswift.com/

June 15, 2014

目录

1 Swift 中文教程（三）字符串和字符

一个字符串 String 就是一个字符序列，像”hello,world”,”albatross” 这样的。Swift 中的字符串是用 String 关键词来定义的，同时它也是一些字符的集合，用 Character 定义。

Swift 的 String 和 Character 类型为代码提供了一个快速的，兼容 Unicode 的字符解决方案。String 类型的初始化和使用都是可读的，并且和 C 中的 strings 类似。同时 String 也可以通过使用 + 运算符来组合，使用字符串就像使用 Swift 中的其他基本类型一样简单。

1.1 字符串常量

在代码中可以使用由 String 预先定义的字符串常量，定义方式非常简单：

```
let someString = “ Some string literal value ”
```

字符串常量可以包括下面这些特殊字符：

- 空字符 \0，反斜杠 \，制表符,换行符 \n，回车符 \r，双引号 \” 和单引号 \’
- 单字节 Unicode 字符，\xnn，其中 nn 是两个十六进制数
- 双字节 Unicode 字符，\unnnn，其中 nnnn 是四个十六进制数
- 四字节 Unicode 字符，\Unnnnnnnn，其中 nnnnnnnn 是八个十六进制数

下面的代码给出了这四种字符串的例子：

```
let wiseWords = "\"Imagination is more important than knowledge\" - Einstein"
// "Imagination is more important than knowledge" - Einstein
let dollarSign = "\x24" // $, Unicode scalar U+0024
let blackHeart = "\u2665" //  , Unicode scalar U+2665
let sparklingHeart = "\U0001F496" // , Unicode scalar U+1F496
```

1.2 初始化一个空串

初始化一个空串时有两种形式，但是两种初始化方法的结果都一样，表示空串

```
var emptyString = "" // empty string literal
var anotherEmptyString = String() // initializer syntax
// these two strings are both empty, and are equivalent to each other
```

通过 isEmpty 属性可以检查一个字符串是否为空

```
if emptyString.isEmpty {
    println("Nothing to see here")
}
// prints "Nothing to see here"
```

1.3 变长字符串

如果使用 var 关键词定义的字符串即为可修改的变长字符串，而 let 关键词定义的字符串是常量字符串，不可修改。

```
var variableString = "Horse"
variableString += " and carriage"
// variableString is now "Horse and carriage"
let constantString = "Highlander"
constantString += " and another Highlander"
// this reports a compile-time error - a constant string cannot be modified
```

1.4 字符串不是指针，而是实际的值

在 Swift 中，一个 String 类型就是一个实际的值，当定义一个新的 String，并且将之前的 String 值拷贝过来的时候，是实际创建了一个相等的新值，而不是仅仅像指针那样指向过去。

同样在函数传递参数的时候，也是传递的实际值，并且创建了一个新的字符串，后续的操作都不会改变原有的 String 字符串。

1.5 字符

Swift 的字符串 String 就是由字符 Character 组成的，每一个 Character 都代表了一个特定的 Unicode 字符。通过 for-in 循环，可以遍历字符串中的每一个字符：

```
for character in "Dog!" {
    println(character)
}
// D
// o
// g
// !
//
```

你也可以仅仅定义一个单独的字符：

```
let yenSign: Character = "￥"
```

1.6 字符计数

使用全局函数 countElements 可以计算一个字符串中字符的数量：

```
let unusualMenagerie = "Koala , Snail , Penguin , Dromedary "
println("unusualMenagerie has \(countElements(unusualMenagerie)) characters")
// prints "unusualMenagerie has 40 characters"
```

1.7 组合使用字符和字符串

String 和 Character 类型可以通过使用 + 号相加来组合成一个新的字符串

```
let string1 = "hello"
let string2 = " there"
let character1: Character = "!"
let character2: Character = "?"
let stringPlusCharacter = string1 + character1 // equals "hello!"
let stringPlusString = string1 + string2 // equals "hello there"
let characterPlusString = character1 + string1 // equals "!hello"
let characterPlusCharacter = character1 + character2 // equals "!?"
```

也可以使用 += 号来组合：

```
var instruction = "look over"
instruction += string2
// instruction now equals "look over there"
var welcome = "good morning"
welcome += character1
// welcome now equals "good morning!"
```

1.8 使用字符串生成新串

通过现有的字符串，可以使用如下方法来生成新的字符串：

```
let multiplier = 3
let message = "\(multiplier) times 2.5 is \(Double(multiplier) * 2.5)"
// message is "3 times 2.5 is 7.5"
```

在上面这个例子中，首先使用 multiplier 这个字符串 3，来作为新串的一部分，用 (multiplier) 添加，同时上面的例子还用到了类型转换 Double(multiplier)，将计算结果和字符串本身都作为元素添加到了新的字符串中。

1.9 字符串比较

Swift 提供三种方法比较字符串的值：字符串相等，前缀相等，和后缀相等

1.9.1 字符串相等

当两个字符串的包含完全相同的字符时，他们被判断为相等。

```
let quotation = "We're a lot alike, you and I."
let sameQuotation = "We're a lot alike, you and I."
if quotation == sameQuotation {
    println("These two strings are considered equal")
}
// prints "These two strings are considered equal"
// □ □ ”These two strings are considered equal”
```

1.9.2 前缀（prefix）相等和后缀 (hasSuffix) 相等

使用 string 类的两个方法 hasPrefix 和 hasSuffix, 来检查一个字符串的前缀或者后缀是否包含另外一个字符串，它需要一个 String 类型型的参数以及返回一个布尔类型的值。两个方法都会在原始字符串和前缀字符串或者后缀字符串之间做字符与字符之间的。

下面一个例子中, 用一个字符串数组再现了莎士比亚的罗密欧与朱丽叶前两幕的场景。

```
let romeoAndJuliet = [
    "Act 1 Scene 1: Verona, A public place",
    "Act 1 Scene 2: Capulet's mansion",
    "Act 1 Scene 3: A room in Capulet's mansion",
    "Act 1 Scene 4: A street outside Capulet's mansion",
    "Act 1 Scene 5: The Great Hall in Capulet's mansion",
    "Act 2 Scene 1: Outside Capulet's mansion",
    "Act 2 Scene 2: Capulet's orchard",
    "Act 2 Scene 3: Outside Friar Lawrence's cell",
    "Act 2 Scene 4: A street in Verona",
    "Act 2 Scene 5: Capulet's mansion",
    "Act 2 Scene 6: Friar Lawrence's cell"
]
```

你可以使用 hasPrefix 方法和 romeoAndJuliet 数组计算出第一幕要表演多少个场景。

```
var act1SceneCount = 0
for scene in romeoAndJuliet {
    if scene.hasPrefix("Act 1 ") {
        ++act1SceneCount
    }
}
println("There are \(act1SceneCount) scenes in Act 1")
// □ □ "There are 5 scenes in Act 1"
```

同理，使用 hasSuffix 方法去计算有多少个场景发生在 Capulet 公馆和 Friar Lawrence 牢房

```
var mansionCount = 0
var cellCount = 0
for scene in romeoAndJuliet {
    if scene.hasSuffix("Capulet's mansion") {
        ++mansionCount
    } else if scene.hasSuffix("Friar Lawrence's cell") {
        ++cellCount
    }
}
println("\(mansionCount) mansion scenes; \(cellCount) cell scenes")
// □ □ "6 mansion scenes; 2 cell scenes”
```

1.10 大小写字符串

你可以从一个 String 类型的 uppercaseString 和 lowercaseString 中获得一个字符串的大写或小写。

```
let normal = "Could you help me, please?"
let shouty = normal.uppercaseString
// shouty is equal to "COULD YOU HELP ME, PLEASE?"
let whispered = normal.lowercaseString
// whispered is equal to "could you help me, please?"
```

1.11 Unicode

Unicode 是编码和表示文本的国际标准。它几乎可以显示所有语言的所有字符的标准形态。还可以从类似于文本文件或者网页这样的外部源文件中读取和修改他们的字符。

1.11.1 Unicode 术语

每一个 Unicode 字符都能被编码为一个或多个 unicode scalar。一个 unicode scalar 是一个唯一的 21 位数（或者名称），对应着一个字符或者标识。例如 U+0061 是一个小写的 A (“a”), 或者 U+1F425 是一个面向我们的黄色小鸡

当一个 Unicode 字符串写入文本或者其他储存时，unicode scalar 会根据 Unicode 定义的格式来编码。每一个格式化编码字符都是小的代码块，称成为

code units. 他包含 UTF-8 格式 (每一个字符串由 8 位的 code units 组成)。和 UTF-16 格式（每一个字符串由 16 位的 code units 组成)

1.11.2 Unicode 字符串

Swift 支持多种不同的方式取得 Unicode 字符串.

你可以使用 for-in 语句遍历字符串，来获得每一个字符的 Unicode 编码值。这个过程已经在字符（Working with Characters）描述过了。

或者，下面三个描述中使用合适的一个来获得一个字符串的值

- UTF-8 字符编码单元集合使用 String 类型的 utf-8 属性
- UTF-16 字符编码单元集合使用 String 类型的 utf-16 属性
- 21 位 Unicode 标量集合使用 String 类型的 unicodeScalars 属性

下面的每一个例子展示了不同编码显示由 D , o , g , !

(DOG FACE, 或者 Unicode 标量 U+1F436）字符组成的字符串

1.11.3 UTF-8

你可以使用 String 类型的 utf8 属性遍历一个 UTF-8 编码的字符串。这个属性是 UTF8View 类型，UTF8View 是一个 8 位无符号整形 (UInt8) 的集合, 集合中的每一个字节都是 UTF-8 编码。

```
for codeUnit in dogString.utf8 {
    print("\(codeUnit) ")
}
print("\n")
// 68 111 103 33 240 159 144 182
```

在上面的例子中，前 4 个十进制 codeunit 值（68,111,103,33）显示为字符串 D , o ,g 和! ，和他们的 ASCII 编码相同一样。后面 4 个 codeunit 的值 (240,159,144,182）是 DOG FACE 字符的 4 字节 UTF-8 编码。

1.11.4 UTF-16

你可以使用 String 类型的 utf16 属性遍历一个 UTF-16 编码的字符串。这个属性是 UTF16View 类型，UTF16View 是一个 16 位无符号整形 (UInt16) 的集合, 集合中的每一个字节都是 UTF-16 编码。

```
for codeUnit in dogString.utf16 {
    print("\(codeUnit) ")
}
print("\n")
// 68 111 103 33 55357 56374
```

同理，前 4 个十进制 codeunit 值（68,111,103,33）显示为字符串 D , o ,g 和! ，他们的 UTF-16 的 codeunit 和他们 UTF-8 的编码值相同。

第 5 和第 6 个 codeunit 值（55357 和 56374）是 DOG FACE 字符的 UTF-16 的代理对编码。他们的值是由值为 U+D83D（十进制 55357）的高位代理 (lead surrogate) 和值为 U+DC36 (十进制 56374) 的低位代理（trail surrogate）组成。

1.11.5 Unicode 标量

你可以使用 String 类型的 unicodeScalars 属性遍历一个 Unicode 标量编码的字符串。这个属性是 UnicodeScalarsView 类型，UnicodeScalarsView 是一个 UnicodeScalar 类型的集合。每一个 Unicode 标量都是一个任意 21 位 Unicode 码位，没有高位代理，也没有低位代理。

每一个 UnicodeScalar 使用 value 属性，返回标量的 21 位值，每一位都是 32 位无符号整形 (UInt32) 的值:

```
for scalar in dogString.unicodeScalars {
    print("\(scalar.value) ")
}
print("\n")
// 68 111 103 33 128054
```

value 属性在前 4 个 UnicodeScalar 值（68,111,103,33）再一次展示编码了字符 D , o , g 和！。第五个也是最后一个 UnicodeScalar 是 DOG FACE 字符，十进制为 128054，等价于 16 进制的 1F436，相当于 Unicode 标量的 U+1F436。

每一个 UnicodeScalar 可以被构造成一个新的字符串来代替读取他们的 value 属性，类似于插入字符串。

```
for scalar in dogString.unicodeScalars { println("\(scalar) ") }
// D
// o
// g
```

```
// !
//
```

参考文献