翻译：wh1100717　校对：Hawstein

字符串和字符（Strings and Characters）

本页包含内容：

- 字符串字面量
- 初始化空字符串
- 字符串可变性
- 字符串是值类型
- 使用字符
- 计算字符数量
- 连接字符串和字符
- 字符串插值
- 比较字符串
- 字符串大小写
- Unicode

`String`是例如“hello, world“”，“海贼王”这样的有序的`Character`（字符）类型的值的集合，通过`String`类型来表示。

Swift 的`String`和`Character`类型提供了一个快速的，兼容 Unicode 的方式来处理代码中的文本信息。 创建和操作字符串的语法与 C 语言中字符串操作相似，轻量并且易读。 字符串连接操作只需要简单地通过`+`号将两个字符串相连即可。 与 Swift 中其他值一样，能否更改字符串的值，取决于其被定义为常量还是变量。

尽管语法简易，但`String`类型是一种快速、现代化的字符串实现。 每一个字符串都是由独立编码的 Unicode 字符组成，并提供了以不同 Unicode 表示（representations）来访问这些字符的支持。

Swift 可以在常量、变量、字面量和表达式中进行字符串插值操作，可以

轻松创建用于展示、存储和打印的自定义字符串。

注意：

Swift 的`String`类型与 Foundation `NSString`类进行了无缝桥接。如果您利用 Cocoa 或 Cocoa Touch 中的 Foundation 框架进行工作。所有 `NSString` API 都可以调用您创建的任意`String`类型的值。除此之外，还可以使用本章介绍的`String`特性。您也可以在任意要求传入`NSString`实例作为参数的 API 中使用`String`类型的值作为替代。

更多关于在 Foundation 和 Cocoa 中使用`String`的信息请查看 Using Swift with Cocoa and Objective-C。

字符串字面量（String Literals）

您可以在您的代码中包含一段预定义的字符串值作为字符串字面量。 字符串字面量是由双引号 ("") 包裹着的具有固定顺序的文本字符集。

字符串字面量可以用于为常量和变量提供初始值。

```
let someString = "Some string literal value"
```

注意：

`someString`变量通过字符串字面量进行初始化，Swift 因此推断该变量为`String`类型。

字符串字面量可以包含以下特殊字符：

- 转义字符`\0`(空字符)、`\\`(反斜线)、`\t`(水平制表符)、`\n`(换行符)、`\r`(回车符)、`\"`(双引号)、`\'`(单引号)。
- 单字节 Unicode 标量，写成`\xnn`，其中`nn`为两位十六进制数。
- 双字节 Unicode 标量，写成`\unnnn`，其中`nnnn`为四位十六进制数。
- 四字节 Unicode 标量，写成`\Unnnnnnnn`，其中`nnnnnnnn`为八位十六进制数。

下面的代码为各种特殊字符的使用示例。 `wiseWords`常量包含了两个转移特殊字符 (双括号)； `dollarSign`、`blackHeart`和`sparklingHeart`常

量演示了三种不同格式的 Unicode 标量：

```
let wiseWords = "\"我是要成为海贼王的男人\" - 路飞"
// "我是要成为海贼王的男人" - 路飞
let dollarSign = "\x24"                 // $,  Unicode
标量 U+0024
let blackHeart = "\u2665"               // ♥,  Unicode
标量 U+2665
let sparklingHeart = "\U0001F496"    // 💖, Unicode
标量 U+1F496
```

初始化空字符串 (Initializing an Empty String)

为了构造一个很长的字符串，可以创建一个空字符串作为初始值。 可以将空的字符串字面量赋值给变量，也可以初始化一个新的`String`实例：

```
var emptyString = ""                // 空字符串字面量
var anotherEmptyString = String()  // 初始化 String 实
例
// 两个字符串均为空并等价。
```

您可以通过检查其`Boolean`类型的`isEmpty`属性来判断该字符串是否为空：

```
if emptyString.isEmpty {
    println("什么都没有")
}
// 打印输出："什么都没有"
```

字符串可变性 (String Mutability)

您可以通过将一个特定字符串分配给一个变量来对其进行修改，或者分配给一个常量来保证其不会被修改：

```
var variableString = "Horse"
variableString += " and carriage"
// variableString 现在为 "Horse and carriage"
let constantString = "Highlander"
constantString += " and another Highlander"
// 这会报告一个编译错误 (compile-time error) - 常量不可以被修改。
```

注意：

在 Objective-C 和 Cocoa 中，您通过选择两个不同的类(`NSString`和`NSMutableString`)来指定该字符串是否可以被修改，Swift 中的字符串是否可以修改仅通过定义的是变量还是常量来决定，实现了多种类型可变性操作的统一。

字符串是值类型（Strings Are Value Types）

Swift 的`String`类型是值类型。 如果您创建了一个新的字符串，那么当其进行常量、变量赋值操作或在函数/方法中传递时，会进行值拷贝。 任何情况下，都会对已有字符串值创建新副本，并对该新副本进行传递或赋值操作。 值类型在 结构体和枚举是值类型 中进行了说明。

注意：

与 Cocoa 中的`NSString`不同，当您在 Cocoa 中创建了一个`NSString`实例，并将其传递给一个函数/方法，或者赋值给一个变量，您传递或赋值的是该`NSString`实例的一个引用，除非您特别要求进行值拷贝，否则字符串不会生成新的副本来进行赋值操作。
Swift 默认字符串拷贝的方式保证了在函数/方法中传递的是字符串的值。很明显无论该值来自于哪里，都是您独自拥有的。 您可以放心您传递的字符串本身不会被更改。

在实际编译时，Swift 编译器会优化字符串的使用，使实际的复制只发生在绝对必要的情况下，这意味着您将字符串作为值类型的同时可以获得极

高的性能。

使用字符（Working with Characters）

Swift 的`String`类型表示特定序列的`Character`（字符） 类型值的集合。每一个字符值代表一个 Unicode 字符。 您可利用`for-in`循环来遍历字符串中的每一个字符：

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

for-in 循环在 For Loops 中进行了详细描述。

另外，通过标明一个`Character`类型注解并通过字符字面量进行赋值，可以建立一个独立的字符常量或变量：

```
let yenSign: Character = "¥"
```

计算字符数量 (Counting Characters)

通过调用全局`countElements`函数，并将字符串作为参数进行传递，可以获取该字符串的字符数量。

```
let unusualMenagerie = "Koala 🐨, Snail 🐌, Penguin 🐧, Dromedary 🐪"
println("unusualMenagerie has \
```

```
(countElements(unusualMenagerie)) characters")
// 打印输出: "unusualMenagerie has 40 characters"
```

注意:

不同的 Unicode 字符以及相同 Unicode 字符的不同表示方式可能需要不同数量的内存空间来存储。所以 Swift 中的字符在一个字符串中并不一定占用相同的内存空间。因此字符串的长度不得不通过迭代字符串中每一个字符的长度来进行计算。如果您正在处理一个长字符串，需要注意`countElements`函数必须遍历字符串中的字符以精准计算字符串的长度。

另外需要注意的是通过`countElements`返回的字符数量并不总是与包含相同字符的`NSString`的`length`属性相同。`NSString`的`length`属性是基于利用 UTF-16 表示的十六位代码单元数字，而不是基于 Unicode 字符。为了解决这个问题，`NSString`的`length`属性在被 Swift 的`String`访问时会成为`utf16count`。

连接字符串和字符 (Concatenating Strings and Characters)

字符串和字符的值可以通过加法运算符（`+`）相加在一起并创建一个新的字符串值:

```
let string1 = "hello"
let string2 = " there"
let character1: Character = "!"
let character2: Character = "?"

let stringPlusCharacter = string1 +
character1        // 等于 "hello!"
let stringPlusString = string1 +
string2              // 等于 "hello there"
let characterPlusString = character1 +
string1        // 等于 "!hello"
let characterPlusCharacter = character1 +
```

```
character2  // 等于 "!?"
```
您也可以通过加法赋值运算符 (+=) 将一个字符串或者字符添加到一个已经存在字符串变量上:

```
var instruction = "look over"
instruction += string2
// instruction 现在等于 "look over there"

var welcome = "good morning"
welcome += character1
// welcome 现在等于 "good morning!"
```
注意:

> 您不能将一个字符串或者字符添加到一个已经存在的字符变量上，因为字符变量只能包含一个字符。

字符串插值 (String Interpolation)

字符串插值是一种构建新字符串的方式，可以在其中包含常量、变量、字面量和表达式。 您插入的字符串字面量的每一项都被包裹在以反斜线为前缀的圆括号中:

```
let multiplier = 3
let message = "\(multiplier) 乘以 2.5 是 \
(Double(multiplier) * 2.5)"
// message 是 "3 乘以 2.5 是 7.5"
```
在上面的例子中，`multiplier`作为`\(multiplier)`被插入到一个字符串字面量中。 当创建字符串执行插值计算时此占位符会被替换为`multiplier`实际的值。

`multiplier`的值也作为字符串中后面表达式的一部分。 该表达式计算`Double(multiplier) * 2.5`的值并将结果 (7.5) 插入到字符串中。 在这个例子中，表达式写为`\(Double(multiplier) * 2.5)`并包含在字符串字面量中。

注意：

您插值字符串中写在括号中的表达式不能包含非转义双引号 (") 和反斜杠 (\)，并且不能包含回车或换行符。

比较字符串 (Comparing Strings)

Swift 提供了三种方式来比较字符串的值：字符串相等、前缀相等和后缀相等。

字符串相等 (String Equality)

如果两个字符串以同一顺序包含完全相同的字符，则认为两者字符串相等：

```
let quotation = "我们是一样一样滴."
let sameQuotation = "我们是一样一样滴."
if quotation == sameQuotation {
    println("这两个字符串被认为是相同的")
}
// 打印输出："这两个字符串被认为是相同的"
```

前缀/后缀相等 (Prefix and Suffix Equality)

通过调用字符串的`hasPrefix`/`hasSuffix`方法来检查字符串是否拥有特定前缀/后缀。 两个方法均需要以字符串作为参数传入并传出`Boolean`值。 两个方法均执行基本字符串和前缀/后缀字符串之间逐个字符的比较操作。

下面的例子以一个字符串数组表示莎士比亚话剧《罗密欧与朱丽叶》中前两场的场景位置：

```
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"
]
```

您可以利用`hasPrefix`方法来计算话剧中第一幕的场景数：

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

相似地，您可以用`hasSuffix`方法来计算发生在不同地方的场景数：

```
“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")
// 打印输出: "6 mansion scenes; 2 cell scenes”
```

大写和小写字符串（Uppercase and Lowercase Strings）

您可以通过字符串的`uppercaseString`和`lowercaseString`属性来访问大写/小写版本的字符串。

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

Unicode

Unicode 是一个国际标准，用于文本的编码和表示。 它使您可以用标准格式表示来自任意语言几乎所有的字符，并能够对文本文件或网页这样的外部资源中的字符进行读写操作。

Swift 的字符串和字符类型是完全兼容 Unicode 标准的，它支持如下所述的一系列不同的 Unicode 编码。

Unicode 术语（Unicode Terminology）

Unicode 中每一个字符都可以被解释为一个或多个 unicode 标量。 字符的 unicode 标量是一个唯一的21位数字(和名称)，例如`U+0061`表示小写的拉丁字母A ("a")，`U+1F425`表示小鸡表情 ("🐥")

当 Unicode 字符串被写进文本文件或其他存储结构当中，这些 unicode 标量将会按照 Unicode 定义的集中格式之一进行编码。其包括`UTF-8`（以8位

代码单元进行编码） 和`UTF-16`（以16位代码单元进行编码）。

字符串的 Unicode 表示（Unicode Representations of Strings）

Swift 提供了几种不同的方式来访问字符串的 Unicode 表示。

您可以利用`for-in`来对字符串进行遍历，从而以 Unicode 字符的方式访问每一个字符值。 该过程在 使用字符 中进行了描述。

另外，能够以其他三种 Unicode 兼容的方式访问字符串的值：

- UTF-8 代码单元集合 (利用字符串的`utf8`属性进行访问)
- UTF-16 代码单元集合 (利用字符串的`utf16`属性进行访问)
- 21位的 Unicode 标量值集合 (利用字符串的`unicodeScalars`属性进行访问)

下面由`D``o``g``!`和🐶(`DOG FACE`，Unicode 标量为`U+1F436`)组成的字符串中的每一个字符代表着一种不同的表示：

```
let dogString = "Dog!🐶"
```

UTF-8

您可以通过遍历字符串的`utf8`属性来访问它的`UTF-8`表示。 其为`UTF8View`类型的属性，`UTF8View`是无符号8位 (`UInt8`) 值的集合，每一个`UInt8`值都是一个字符的 UTF-8 表示：

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

上面的例子中，前四个10进制代码单元值 (68, 111, 103, 33) 代表了字符`D` `o` `g`和`!`，他们的 UTF-8 表示与 ASCII 表示相同。 后四个代码单元值 (240,

159, 144, 182) 是DOG FACE的4字节 UTF-8 表示。

UTF-16

您可以通过遍历字符串的utf16属性来访问它的UTF-16表示。 其为UTF16View类型的属性，UTF16View是无符号16位 (UInt16) 值的集合，每一个UInt16都是一个字符的 UTF-16 表示：

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

同样，前四个代码单元值 (68, 111, 103, 33) 代表了字符D o g和!，他们的 UTF-16 代码单元和 UTF-8 完全相同。

第五和第六个代码单元值 (55357 和 56374) 是DOG FACE字符的UTF-16 表示。 第一个值为U+D83D(十进制值为 55357)，第二个值为U+DC36(十进制值为 56374)。

Unicode 标量 (Unicode Scalars)

您可以通过遍历字符串的unicodeScalars属性来访问它的 Unicode 标量表示。 其为UnicodeScalarView类型的属性， UnicodeScalarView是UnicodeScalar的集合。 UnicodeScalar是21位的 Unicode 代码点。

每一个UnicodeScalar拥有一个值属性，可以返回对应的21位数值，用UInt32来表示。

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

同样，前四个代码单元值 (68, 111, 103, 33) 代表了字符`D` `o` `g`和`!`。 第五位数值，128054，是一个十六进制1F436的十进制表示。 其等同于`DOG FACE`的Unicode 标量 U+1F436。

作为查询字符值属性的一种替代方法，每个`UnicodeScalar`值也可以用来构建一个新的字符串值，比如在字符串插值中使用：

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