String 是一个有序的字符集合，例如 "hello, world", "albatross"。Swift 字符串通过 String 类型来表示，也可以表示为 Character 类型值的集合。

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的操作方式相似，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，并提供了用于访问这些字符在不同Unicode表示的支持。

String也可以用来在长字符串中插入常量、变量、字面量和表达式，这一过程称为字符串插值。这使得创建用于展示、存储和打印的自定义字符串变得轻松自如。

|  |
| --- |
| 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，整个 NSString API 都可以调用您创建的任意 String 类型的值，除了本章介绍的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 类型的值进行替换。 |

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](https://developer.apple.com/library/prerelease/ios/documentation/Swift/Conceptual/BuildingCocoaApps/index.html#//apple_ref/doc/uid/TP40014216)。

**字符串字面量**

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。

字符串字面量可以用于为常量和变量提供初始值。

1. let someString = "Some string literal value"

|  |
| --- |
| 注意：someString 常量通过字符串字面量进行初始化，Swift 因此推断其为 String 类型。 |

字符串字面量可以包含以下特殊字符：

1.转义特殊字符 \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)。

2.单字节 Unicode 标量，写成 \xnn，其中 nn 为两位十六进制数。

3.双字节 Unicode 标量，写成 \unnnn，其中 nnnn 为四位十六进制数。

4.四字节 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：

1. let wiseWords = "\"Imagination is more important than knowledge\" - Einstein"
2. // "Imagination is more important than knowledge" - Einstein
3. let dollarSign = "\x24"        // $,  Unicode scalar U+0024
4. let blackHeart = "\u2665"      // ♥,  Unicode scalar U+2665
5. let sparklingHeart = "\U0001F496"  // ????, Unicode scalar U+1F496

**初始化空字符串**

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：

1. **var** emptyString = ""               // empty string literal
2. **var** anotherEmptyString = String()  // initializer syntax
3. // 这两个字符串都为空，并且两者等价

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：

1. **if** emptyString.isEmpty {
2. println("Nothing to see here")
3. }
4. // 打印 "Nothing to see here"

**字符串可变性**

您可以通过将一个特定字符串分配给一个变量（对其进行修改）或者常量（保证其不会被修改）来指定该字符串是否可以被修改：

1. **var** variableString = "Horse"
2. variableString += " and carriage"
3. // variableString 现在为 "Horse and carriage"
4. let constantString = "Highlander"
5. constantString += " and another Highlander"
6. // 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**

Swift 的 String 类型是值类型。如果您创建了一个新的字符串值，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。在不同情况下，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。值类型在 [Structures and Enumerations Are Value Types](https://developer.apple.com/library/prerelease/ios/documentation/Swift/Conceptual/Swift_Programming_Language/ClassesAndStructures.html#//apple_ref/doc/uid/TP40014097-CH13-XID_104) 中进行了说明。

|  |
| --- |
| 注意：和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，或者赋给一个变量，您永远都是传递或赋值同一个 NSString 实例的一个引用。除非您特别要求其进行值拷贝，否则字符串不会进行赋值新副本操作。 |

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，其明确您独有该字符串的值，无论它来自哪里。您可以放心您传递的字符串本身不会被更改，除非是你自己更改它。

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，这意味着您始终可以将字符串作为值类型的同时获得极高的性能。

**使用字符(Characters)**

Swift 的 String 类型表示特定序列的字符值的集合。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：

1. **for** character **in** "Dog!????" {
2. println(character)
3. }
4. // D
5. // o
6. // g
7. // !
8. // ????

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：

1. let yenSign: Character = "¥"

**计算字符数量**

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。

1. let unusualMenagerie = "Koala ????, Snail ????, Penguin ????, Dromedary ????"
2. println("unusualMenagerie has \(countElements(unusualMenagerie)) characters")
3. // prints "unusualMenagerie has 40 characters"

|  |
| --- |
| 注意：    1.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表示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。  2.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code units数目，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值访问时会被称为utf16count。 |

**连接字符串和字符**

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：

1. let string1 = "hello"
2. let string2 = " there"
3. let character1: Character = "!"
4. let character2: Character = "?"
6. let stringPlusCharacter = string1 + character1        // 等于 "hello!"
7. let stringPlusString = string1 + string2              // 等于 "hello there"
8. let characterPlusString = character1 + string1        // 等于 "!hello"
9. 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：

1. **var** instruction = "look over"
2. instruction += string2
3. // instruction 现在等于 "look over there"
5. **var** welcome = "good morning"
6. welcome += character1
7. // 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**

字符串插值是一种全新的构建字符串的方式，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：

1. let multiplier = 3
2. let message = "\(multiplier) times 2.5 is \(Double(multiplier) \* 2.5)"
3. // message is "3 times 2.5 is 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**

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**

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：

1. let quotation = "We're a lot alike, you and I."
2. let sameQuotation = "We're a lot alike, you and I."
3. **if** quotation == sameQuotation {
4. println("These two strings are considered equal")
5. }
6. // prints "These two strings are considered equal"

**前缀/后缀相等**

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返回 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：

1. let romeoAndJuliet = [
2. "Act 1 Scene 1: Verona, A public place",
3. "Act 1 Scene 2: Capulet's mansion",
4. "Act 1 Scene 3: A room in Capulet's mansion",
5. "Act 1 Scene 4: A street outside Capulet's mansion",
6. "Act 1 Scene 5: The Great Hall in Capulet's mansion",
7. "Act 2 Scene 1: Outside Capulet's mansion",
8. "Act 2 Scene 2: Capulet's orchard",
9. "Act 2 Scene 3: Outside Friar Lawrence's cell",
10. "Act 2 Scene 4: A street in Verona",
11. "Act 2 Scene 5: Capulet's mansion",
12. "Act 2 Scene 6: Friar Lawrence's cell"
13. ]

您可以利用 hasPrefix 方法使用romeoAndJuliet数组来计算话剧中第一幕的场景数：

1. **var** act1SceneCount = 0
2. **for** scene **in** romeoAndJuliet {
3. **if** scene.hasPrefix("Act 1 ") {
4. ++act1SceneCount
5. }
6. }
7. println("There are \(act1SceneCount) scenes in Act 1")
8. // prints "There are 5 scenes in Act 1"

同样，可使用hasSuffix方法来计算发生在Capulet公馆和Lawrence牢房内以及周围的场景数。

1. **var** mansionCount = 0
2. **var** cellCount = 0
3. **for** scene **in** romeoAndJuliet {
4. **if** scene.hasSuffix("Capulet's mansion") {
5. ++mansionCount
6. } **else** **if** scene.hasSuffix("Friar Lawrence's cell") {
7. ++cellCount
8. }
9. }
10. println("\(mansionCount) mansion scenes; \(cellCount) cell scenes")
11. // prints "6 mansion scenes; 2 cell scenes"

**大写和小写字符串**

您可以通过字符串的 uppercaseString 和 lowercaseString 属性来访问一个字符串的大写/小写版本。

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

**Unicode**

Unicode 是文本编码和表示的国际标准。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 的，它支持如下所述的一系列不同的 Unicode 编码。

**Unicode 术语(Terminology)**

Unicode 中每一个字符都可以被解释为一个或多个 unicode 标量。一个unicode 标量是字符或者修饰符的唯一21位数(和名称)，例如 U+0061 表示小写的拉丁字母A ("a")，U+1F425 表示正面站立的鸡宝宝 ("????")

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。每个格式将字符串编码为小的代码块-code units，其包括UTF-8格式(以8位代码单元进行编码)和UTF-16格式(以16位代码单元进行编码)。

**字符串的 Unicode 表示**

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 [Working with Characters](https://developer.apple.com/library/prerelease/ios/documentation/Swift/Conceptual/Swift_Programming_Language/StringsAndCharacters.html#//apple_ref/doc/uid/TP40014097-CH7-XID_376) 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：

1.UTF-8 代码单元集合 (利用字符串的 utf8 属性进行访问)

2.UTF-16 代码单元集合 (利用字符串的 utf16 属性进行访问)

3.21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http://www.cocoachina.com/cms/uploads/allimg/140606/4196_140606144525_1.png字符(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：

1. 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 UIn8 都是一个字符的 UTF-8 表示：

1. **for** codeUnit **in** dogString.utf8 {
2. print("\(codeUnit) ")
3. }
4. print("\n")
5. // 68 111 103 33 240 159 144 182

上面的例子中，前四个10进制codeUnit值 (68, 111, 103, 33) 代表了字符 D o g 和 ! ，他们的 UTF-8 表示与其ASCII 表示相同。后四个codeUnit值 (240, 159, 144, 182) 是DOG FACE字符的4位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，是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：

1. **for** codeUnit **in** dogString.utf16 {
2. print("\(codeUnit) ")
3. }
4. print("\n")
5. // 68 111 103 33 55357 56374

同样，前四个codeUnit值 (68, 111, 103, 33) 代表了字符 D o g 和 ! ，他们的 UTF-16 code units值和 UTF-8 表示完全相同。

第五和第六个codeUnit值 (55357 and 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。一个Unicode Scalar是任意的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值来表示。

1. **for** scalar **in** dogString.unicodeScalars {
2. print("\(scalar.value) ")
3. }
4. print("\n")
5. // 68 111 103 33 128054

同样，前四个UnicodeScalar值 (68, 111, 103, 33) 再次代表了字符D, o, g和 ! 。第五个也是最后一个UnicodeScalar的值属性为128054，是一个十六进制1F436的十进制表示。其等同于DOG FACE 字符的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：

1. **for** scalar **in** dogString.unicodeScalars {
2. println("\(scalar) ")
3. }
4. // D
5. // o
6. // g
7. // !
8. // ????