

字符串和字符

字符串是一系列字符，如的"hello, world"或"albatross"。Swift字符串由String类型表示。a的内容String可以以各种方式访问，包括作为Character值的集合。

Swift String和Character类型提供了一种快速，符合Unicode的方式来处理代码中的文本。字符串创建和操作的语法是轻量级且可读的，具有类似于C的字符串文字语法。字符串连接非常简单，只需将两个字符串与+操作符组合在一起，而字符串可变性通过在常量或变量之间进行选择来管理，就像Swift中的任何其他值一样。您也可以使用字符串将常量，变量，文字和表达式插入到更长的字符串中，这个过程称为字符串插值。这使得创建显示，存储和打印的自定义字符串值变得很容易。

尽管语法简单，但Swift的String类型是一种快速，现代的字符串实现。每个字符串都由与编码无关的Unicode字符组成，并为访问各种Unicode表示中的字符提供支持。

注意

斯威夫特的String类型与基金会的NSString班级桥接。基础还扩展String到公开的方法NSString。这意味着，如果您导入Foundation，则可以NSString在String不投射的情况下访问这些方法。

有关String在Foundation和Cocoa中使用的更多信息，请参阅在[Cocoa和Objective-C中使用Swift使用 Cocoa 数据类型 \(Swift 4.1\)](#)。

字符串文字

您可以将预定义的String值作为字符串文字包含在您的代码中。字符串文字是由双引号 (") 包围的一系列字符。

使用字符串文字作为常量或变量的初始值：

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

需要注意的是斯威夫特，推测型String的someString，因为它有一个字符串值初始化不变。

多行字符串文字

如果您需要跨越多行的字符串，请使用多行字符串文字 - 由三个双引号括起来的字符序列：

```
1 let quotation = """
2 The White Rabbit put on his spectacles. "Where shall I begin,
3 please your Majesty?" he asked.
4
5 "Begin at the beginning," the King said gravely, "and go on
6 till you come to the end; then stop."
7 """
```

多行字符串文字包括其开始和结束引号之间的所有行。该字符串从打开引号 (""") 后的第一行开始，并在结尾引号之前的行上结束，这意味着下面的任何字符串都不以换行符开头或结尾：

```
1 let singleLineString = "These are the same."
2 let multilineString = """
3 These are the same.
4 """
```

当源代码在多行字符串文字中包含换行符时，该换行符也会出现在字符串的值中。如果要使用换行符使源代码更易于阅读，但不希望换行符成为字符串值的一部分，请在\这些行的末尾写入反斜杠 ()：

```
1 let softWrappedQuotation = """
2 The White Rabbit put on his spectacles. "Where shall I begin, \
3 please your Majesty?" he asked.
```

```

4
5 "Begin at the beginning," the King said gravely, "and go on \
6 till you come to the end; then stop."
7 ""

```

要制作以换行符开始或结束的多行字符串文字，请将空行写为第一行或最后一行。例如：

```

1 let lineBreaks = ""
2
3 This string starts with a line break.
4 It also ends with a line break.
5
6 ""

```

多行字符串可以缩进以匹配周围的代码。关闭引号之前的空格""告诉Swift在所有其他行之前要忽略哪些空白。但是，除了在引号之前的内容之前，如果在一行的开始处写入空格，则会包含空白。

```

let linesWithIndentation = ""
This line doesn't begin with whitespace.
This line begins with four spaces.
This line doesn't begin with whitespace.
""

```

Space ignored
Appears in string

在上面的示例中，即使整行多行字符串文字是缩进的，字符串中的第一行和最后一行也不以任何空格开头。中间行的缩进比关闭引号更多，所以它以该额外的四空间缩进开始。

字符串文字中的特殊字符

字符串文字可以包含以下特殊字符：

- 转义的特殊字符\0（空字符），\\（反斜杠），\t（水平制表符），\n（换行符），\r（回车），\"（双引号）和\'（单引号）
- 任意的Unicode标，写为\u{N}，其中N是具有值的1-8位十六进制数等于一个有效的Unicode代码点（Unicode的在讨论的Unicode下文）

下面的代码显示了这些特殊字符的四个示例。该wiseWords常数包含两个逃脱双引号。该dollarSign，blackHeart和sparklingHeart常量演示了Unicode标量格式：

```

1 let wiseWords = "\"Imagination is more important than knowledge\" - Einstein"
2 // "Imagination is more important than knowledge" - Einstein
3 let dollarSign = "\u{24}" // $, Unicode scalar U+0024
4 let blackHeart = "\u{2665}" // ♥, Unicode scalar U+2665
5 let sparklingHeart = "\u{1F496}" // 💖, Unicode scalar U+1F496

```

因为多行字符串文字使用三个双引号而不是一个，所以可以"在多行字符串文字中包含双引号（），而不会将其转义。要将文本包含""在多行字符串中，请至少转义其中一个引号。例如：

```

1 let threeDoubleQuotationMarks = ""
2 Escaping the first quotation mark \"
3 Escaping all three quotation marks "\"\"\"
4 ""

```

初始化一个空字符串

要创建一个空String值作为构建更长字符串的起点，可以将空字符串文字分配给变量，或String使用初始化语法初始化新实例：

```

1 var emptyString = "" // empty string literal
2 var anotherEmptyString = String() // initializer syntax

```

```
3 // these two strings are both empty, and are equivalent to each other
```

String通过检查其布尔isEmpty属性 来确定值是否为空:

```
1 if emptyString.isEmpty {
2     print("Nothing to see here")
3 }
4 // Prints "Nothing to see here"
```

字符串可变性

您String可以通过将某个特定对象分配给一个变量（在这种情况下可以对其进行修改）或者对一个常量（在这种情况下不能对其进行修改）来指示某个特定对象是否可以进行修改（或变异）：

```
1 var variableString = "Horse"
2 variableString += " and carriage"
3 // variableString is now "Horse and carriage"
4
5 let constantString = "Highlander"
6 constantString += " and another Highlander"
7 // this reports a compile-time error - a constant string cannot be modified
```

注意

这种方法与Objective-C和Cocoa中的字符串变异不同，您可以在两个类（NSString和NSMutableString）之间进行选择以指示字符串是否可以发生变异。

字符串是值类型

Swift的String类型是一个值类型。如果你创建一个新的String值，那么当它被传递给一个函数或方法时，或者当它被分配给一个常量或变量时，该String值被复制。在每种情况下，String都会创建现有值的新副本，并且传递或分配新副本，而不是原始版本。值类型在[结构和枚举是值类型](#)中进行了介绍。

Swift的默认复制String行为确保了当一个函数或方法向您传递一个String值时，显然您拥有该确切String值，而不管它来自哪里。你可以确信你所传递的字符串不会被修改，除非你自己修改它。

在幕后，Swift的编译器优化了字符串的使用，以便仅在必要时才进行实际的复制。这意味着在使用字符串作为值类型时，您总能获得出色的性能。

使用字符

您可以访问个人Character的价值观String通过遍历一个字符串for- in循环：

```
1 for character in "Dog!🐶" {
2     print(character)
3 }
4 // D
5 // o
6 // g
7 // !
8 // 🐶
```

的for- in环中描述[对于-在循环中](#)。

或者，您可以Character通过提供Character类型注释来从单字符字符串文字创建独立常量或变量：

```
let exclamationMark: Character = "!"
```

String值可以通过将一个Character值数组作为参数传递给它的初始化程序来构造：

```
1 let catCharacters: [Character] = ["C", "a", "t", "!", "🐱"]
2 let cat
3 print(catString,
4 // Prints "Cat!🐱"
```

连接字符串和字符

String值可以与添加运算符 (+) 一起添加 (或连接+) 以创建新String值：

```
1 let string1 = "hello"
2 let string2 = " there"
3 var welcome = string1 + string2
4 // welcome now equals "hello there"
```

您还可以使用附加赋值运算符 (+=) 将String值附加到现有String变量+=：

```
1 var instruction = "look over"
2 instruction += string2
3 // instruction now equals "look over there"
```

您可以使用类型的方法将Character值附加到String变量中：String.append()

```
1 let exclamationMark: Character = "!"
2 welcome.append(exclamationMark)
3 // welcome now equals "hello there!"
```

注意

你不能追加一个String或Character到一个现有的Character变量，因为一个Character值只能包含单个字符。

如果您使用多行字符串文字构建较长字符串的行，则需要字符串中的每一行以换行符结束，包括最后一行。例如：

```
1 let badStart = ""
2 one
3 two
4 ""
5 let end = ""
6 three
7 ""
8 print(badStart + end)
9 // Prints two lines:
10 // one
11 // twothree
12
13 let goodStart = ""
14 one
15 two
16
17 ""
18 print(goodStart + end)
19 // Prints three lines:
20 // one
21 // two
22 // three
```

在上面的代码中，级联badStart与end产生两行字符串，这是不期望的结果。由于最后一行badStart不以换行符结束，因此该行与第一行组合end。相反，两条线goodStart都有一个换行符，所以当它与end结果组合在一起时，就有三条线，如预期的那样。

字符串插值

字符串插值是一种通过将String常量，变量，文字和表达式的值包含在字符串文字中来构造新值的方法。您可以在单行和多行字符串文字中使用字符串插值。插入到字符串文字中的每个项目都包含在一对括号中，并以反斜杠 (\) 作为前缀：

```
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，值multiplier被插入到字符串文字中\ (multiplier)。这个占位符被替换为multiplier当字符串插值被评估以创建实际字符串时的实际值。

该值multiplier也是稍后在字符串中的较大表达式的一部分。该表达式计算值Double(multiplier) * 2.5并将结果 (7.5) 插入到字符串中。在这种情况下，表达式被写入为\ (Double(multiplier) * 2.5) 它包含在字符串文字中的时候。

注意

您在插入字符串中括号内写入的表达式不能包含未转义的反斜杠 (\)，回车符或换行符。但是，它们可以包含其他字符串文字。

统一

Unicode是在不同书写系统中编码，表示和处理文本的国际标准。它使您可以以任何语言以标准形式表示几乎任何字符，并可以从外部源（如文本文件或网页）读取和写入这些字符。如本节所述，Swift String和Character类型完全符合Unicode。

Unicode标量

在幕后，Swift的本机String类型是由Unicode标量值构建的。甲Unicode标为字符或改性剂，例如唯一的21位的数U+0061为LATIN SMALL LETTER A ("a")，或U+1F425为FRONT-FACING BABY CHICK ("🐣")。

注意

甲Unicode标是任何Unicode 代码点的范围内U+0000，以U+D7FF包含性的或U+E000，以U+10FFFF包容。Unicode的标量不包括Unicode 代理对码点，这是该范围内的代码点U+D800来U+DFFF包容。

请注意，并非所有的21位Unicode标量都分配给一个字符 - 有些标量保留给将来分配。已分配给一个字符标量通常还具有一个名字，如LATIN SMALL LETTER A和FRONT-FACING BABY CHICK在上面的实施例。

扩展的字形集群

Swift Character类型的每个实例都表示一个扩展的字形集群。扩展字形集群是一个或多个Unicode标量的序列，它们在组合时会产生一个可读的字符。

这是一个例子。该字母é可以表示为单个Unicode标量é (LATIN SMALL LETTER E WITH ACUTE或U+00E9)。但是，同一个字母也可以表示为一对标量 - 一个标准字母e (LATIN SMALL LETTER E或U+0065)，后跟COMBINING ACUTE ACCENT标量 (U+0301)。该COMBINING ACUTE ACCENT标量图形应用到它前面，把一个标量e到é时它是由一个支持Unicode的文本的渲染系统渲染。

在这两种情况下，该字母é都表示为Character代表扩展字形集群的单个Swift 值。在第一种情况下，集群包含一个标量; 在第二种情况下，它是由两个标量组成的集群：

```

1 let eAcute: Character = "\u{E9}" // é
2 let combinedEAcute: Character = "\u{65}\u{301}" // e followed by ´
3 // eAcute is é, combinedEAcute is é

```

扩展字形集群是一种将许多复杂脚本字符表示为单一Character值的灵活方式。例如，米日韩文字母的韩文首节可以表示为预先分解或分解的序列。这两种表示Character在Swift中都是合格的：

```

1 let precomposed: Character = "\u{D55C}" // 한
2 let decomposed: Character = "\u{1112}\u{1161}\u{11AB}" // ㅎ, ㅏ, ㄴ
3 // precomposed is 한, decomposed is 한

```

扩展字形集群可以使用标量来封闭标记（例如COMBINING ENCLOSING CIRCLE或U+20DD），以将其他Unicode标量作为单个Character值的一部分进行封装：

```

1 let enclosedEAcute: Character = "\u{E9}\u{20DD}"
2 // enclosedEAcute is ☐

```

区域指标符号的Unicode标量可以成对组合，以构成单个Character值，例如REGIONAL INDICATOR SYMBOL LETTER U (U+1F1FA) 和REGIONAL INDICATOR SYMBOL LETTER S (U+1F1F8) 的这种组合：

```

1 let regionalIndicatorForUS: Character = "\u{1F1FA}\u{1F1F8}"
2 // regionalIndicatorForUS is 🇺🇸

```

计数字符

要检索Character字符串中值的计数，请使用字符串的count属性：

```

1 let unusualMenagerie = "Koala 🐨, Snail 🐌, Penguin 🐧, Dromedary 🐪"
2 print("unusualMenagerie has \(unusualMenagerie.count) characters")
3 // Prints "unusualMenagerie has 40 characters"

```

请注意，Swift使用扩展字形集群的Character值意味着字符串连接和修改可能并不总是会影响字符串的字符数。

例如，如果使用四个字符的单词初始化一个新字符串cafe，然后在该字符串的末尾附加一个COMBINING ACUTE ACCENT (U+0301)，则结果字符串的字符数仍然4为第四个字符é，而不是e：

```

1 var word = "cafe"
2 print("the number of characters in \(word) is \(word.count)")
3 // Prints "the number of characters in cafe is 4"
4
5 word += "\u{301}" // COMBINING ACUTE ACCENT, U+0301
6
7 print("the number of characters in \(word) is \(word.count)")
8 // Prints "the number of characters in café is 4"

```

注意

扩展字形集群可以由多个Unicode标量组成。这意味着不同的字符以及同一个字符的不同表示可能需要不同数量的内存来存储。正因为如此，Swift中的字符在字符串表示中并不占用相同数量的内存。因此，如果不迭代字符串以确定其扩展字形集群边界，则无法计算字符串中的字符数。如果您使用的字符串值特别长，请注意该count属性必须迭代整个字符串中的Unicode标量以确定该字符串的字符。

该count属性返回的字符数并不总是与包含相同字符的length属性NSString相同。an的长度NSString取决于字符串的UTF-16表示中的16位代码单元的数量，而不是字符串中的Unicode扩展字形集群的数量。

访问和修改字符串

您可以通过其方法和属性或使用下标语法来访问和修改字符串。

字符串索引

每个String值都有一个关联的索引/类型，String.Index它对应于每个Character字符串中的位置。

如上所述，不同

须遍历每个Unicode标量的开头或结尾String。由于这个原因，Swift字符串不能被整数值索引。

使用该startIndex属性访问Character a 的第一个位置String。该endIndex属性是a中最后一个字符之后的位置String。因此，该endIndex属性不是字符串下标的有效参数。如果a String是空的，startIndex并且endIndex相等。

您可以使用index(before:)和的index(after:)方法在给定索引之前和之后访问索引String。要访问远离给定索引的索引，可以使用该index(_:offsetBy:)方法而不是多次调用其中一种方法。

您可以使用下标语法来访问Character特定String索引处的。

```
1 let greeting = "Guten Tag!"
2 greeting[greeting.startIndex]
3 // G
4 greeting[greeting.index(before: greeting.endIndex)]
5 // !
6 greeting[greeting.index(after: greeting.startIndex)]
7 // u
8 let index = greeting.index(greeting.startIndex, offsetBy: 7)
9 greeting[index]
10 // a
```

尝试访问字符串范围Character之外的索引或字符串范围之外的索引将触发运行时错误。

```
1 greeting[greeting.endIndex] // Error
2 greeting.index(after: greeting.endIndex) // Error
```

使用该indices属性可以访问字符串中所有单个字符的索引。

```
1 for index in greeting.indices {
2     print("\(greeting[index]) ", terminator: "")
3 }
4 // Prints "G u t e n   T a g ! "
```

注意

您可以使用startIndex与endIndex属性和index(before:), index(after:)以及index(_:offsetBy:)对符合任何类型的方法Collection的协议。这包括String，如下图所示，以及集合类型，如Array, Dictionary和Set。

插入和删除

要将单个字符插入指定索引处的字符串中，请使用该insert(_:at:)方法，并将另一个字符串的内容插入到指定索引处，请使用该insert(contentsOf:at:)方法。

```
1 var welcome = "hello"
2 welcome.insert("!", at: welcome.endIndex)
3 // welcome now equals "hello!"
4
5 welcome.insert(contentsOf: " there", at: welcome.index(before: welcome.endIndex))
6 // welcome now equals "hello there!"
```

要从指定索引处的字符串中删除单个字符，请使用该remove(at:)方法，并删除指定范围内的子字符串，请使用以下removeSubrange(_:):方法:

```
1 welcome.remove(at: welcome.index(before: welcome.endIndex))
2 // welcome now equals "hello there"
3
```

```

4 let range = welcome.index(welcome.endIndex, offsetBy: -6)..

```

注意

您可以使用`insert(_:at:)`、`insert(contentsOf:at:)`、`remove(at:)`、和`removeSubrange(_:)`对符合任何类型的方法`RangeReplaceableCollection`的协议。这包括`String`，如下图所示，以及集合类型，如`Array`、`Dictionary`和`Set`。

子

当你从一个字符串中得到一个子字符串时 - 例如，使用下标或类似`prefix(_:)`的方法 - 结果是一个实例`Substring`，而不是另一个字符串。Swift中的子串与字符串的大部分方法相同，这意味着您可以像使用字符串一样处理子字符串。但是，与字符串不同，在对字符串执行操作时，只需很短的时间就可以使用子字符串。当您准备好将结果存储较长时间时，可以将子字符串转换为一个实例`String`。例如：

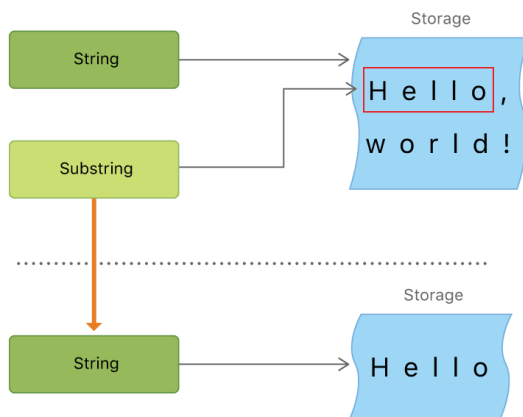
```

1 let greeting = "Hello, world!"
2 let index = greeting.index(of: ",") ?? greeting.endIndex
3 let beginning = greeting[..<index]
4 // beginning is "Hello"
5
6 // Convert the result to a String for long-term storage.
7 let newString = String(beginning)

```

像字符串一样，每个子字符串都有一个内存区域，构成子字符串的字符被存储。字符串和子字符串之间的区别在于，作为性能优化，子字符串可以重用用于存储原始字符串的部分内存，或者用于存储另一个子字符串的部分内存。（字符串有类似的优化，但是如果两个字符串共享内存，它们是相等的。）这种性能优化意味着在修改字符串或子字符串之前，您不必支付复制内存的性能成本。如上所述，子字符串不适合长期存储 - 因为它们会重用原始字符串的存储空间，只要使用任何子字符串，整个原始字符串就必须保存在内存中。

在上面的例子中，`greeting`是一个字符串，这意味着它有一个内存区域，构成字符串的字符被存储。因为`beginning`是一个子字符串`greeting`，它会重用使用的内存`greeting`。相反，它`newString`是一个字符串 - 当它从子字符串创建时，它有自己的存储空间。下图显示了这些关系：



注意

二者`String`并`Substring`符合`StringProtocol`协议，这意味着它的常方便的字符串操作函数接受`StringProtocol`的值。您可以使用`a String`或`Substring`来调用这些函数。

比较字符串

Swift提供了三种比较文本值的方法：字符串和字符相等，前缀相等和后缀相等。

字符串和字符平等

字符串和字符平等是指“等”字符（相等）和“不等”字符（不等）如左图所示。

```
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     print("These two strings are considered equal")
5 }
6 // Prints "These two strings are considered equal"
```

如果两个String值（或两个Character值）的扩展字形群是正则等价的，则认为它们是相等的。如果扩展字形集具有相同的语言含义和外观，即使它们是在幕后使用不同的Unicode标量组成的，它们也具有正常的等价性。

例如，LATIN SMALL LETTER E WITH ACUTE (U+00E9) 是规范等价于LATIN SMALL LETTER E (U+0065)，随后加入COMBINING ACUTE ACCENT (U+0301)。这两个扩展字形集都是表示字符的有效方式é，因此它们被认为是正则等价的：

```
1 // "Voulez-vous un café?" using LATIN SMALL LETTER E WITH ACUTE
2 let eAcuteQuestion = "Voulez-vous un caf\u{E9}?"
3
4 // "Voulez-vous un café?" using LATIN SMALL LETTER E and COMBINING ACUTE ACCENT
5 let combinedEAcuteQuestion = "Voulez-vous un caf\u{65}\u{301}?"
6
7 if eAcuteQuestion == combinedEAcuteQuestion {
8     print("These two strings are considered equal")
9 }
10 // Prints "These two strings are considered equal"
```

相反，英文中使用的LATIN CAPITAL LETTER A (U+0041或"A") 不等于于俄文中使用的CYRILLIC CAPITAL LETTER A (U+0410或"А")。字符在外观上相似，但不具有相同的语言含义：

```
1 let latinCapitalLetterA: Character = "\u{41}"
2
3 let cyrillicCapitalLetterA: Character = "\u{0410}"
4
5 if latinCapitalLetterA != cyrillicCapitalLetterA {
6     print("These two characters are not equivalent.")
7 }
8 // Prints "These two characters are not equivalent."
```

注意

Swift中的字符串和字符比较不是区域敏感的。

前缀和后缀平等

要检查一个字符串是否具有特定的字符串前缀或后缀，请调用字符串的方法hasPrefix(:)和hasSuffix(:)方法，它们都采用一个类型的参数String并返回一个布尔值。

下面的例子考虑了一组代表莎士比亚罗密欧与朱丽叶前两幕的场景位置：

```
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 ]

```

您可以使用该数组的hasPrefix(_:)方法romeoAndJuliet来计算剧本的第1幕中的场景数量:

```

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

```

同样, 使用该hasSuffix(_:)方法来计算Capulet大厦和Friar Lawrence小区内或周围发生的场景数量:

```

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

```

注意

该hasPrefix(_:)和hasSuffix(_:)在每一个串的方法执行所述扩展字形簇之间的字符逐字符规范等价比较, 如在所述的[字符串和字符平等](#)。

字符串的Unicode表示

将Unicode字符串写入文本文件或其他存储时, 该字符串中的Unicode标量将使用几种Unicode定义的编码形式之一进行编码。每个表单都将这些字符串编码为称为代码单元的小块。这些包括UTF-8编码形式(将字符串编码为8位代码单元), UTF-16编码形式(将字符串编码为16位代码单元)和UTF-32编码形式(编码一个字符串为32位代码单元)。

Swift提供了几种不同的方法来访问字符串的Unicode表示。您可以使用for-in语句迭代字符串, 以CharacterUnicode扩展字形群集的形式访问其各个值。这个过程在[使用字符](#)中描述。

或者, 可以使用String其他三种符合Unicode的表示法之一访问值:


- UTF-8代码单元的集合 (使用字符串的utf8属性访问)
- UTF-16代码单元的集合 (使用字符串的utf16属性访问)
- 21位Unicode标量值的集合, 等效于字符串的UTF-32编码形式 (通过字符串的unicodeScalars属性进行访问)

下面各实施例中示出了下面的字符串, 它是由所述字符的向上的不同表示D, o, g, !! (DOUBLE EXCLAMATION MARK或Unicode标U+203C), 以及🐶字符 (DOG FACE或Unicode标U+1F436):

```
let dogString = "Dog!!🐶"
```

UTF-8表示

您可以String通过迭代其utf8属性来访问a的UTF-8表示。该属性的类型String.UTF8View是一个无符号8位（UInt8）值的集合，对于字符串的UTF-8表示中的每个字节都有一个值：


Character	D U+0044	o U+006F	g U+0067	!! U+203C			 U+1F436			
UTF-8 Code Unit	68	111	103	226	128	188	240	159	144	182
Position	0	1	2	3	4	5	6	7	8	9

```
1 for codeUnit in dogString.utf8 {
2     print("\(codeUnit) ", terminator: "")
3 }
4 print("")
5 // Prints "68 111 103 226 128 188 240 159 144 182 "
```

在上面的例子中，前三个十进制codeUnit值（68，111，103）所表示的字符D，o和g，其UTF-8表示相同的ASCII表示。接下来的三个十进制codeUnit值（226，128，188）是一个三字节UTF-8表示DOUBLE EXCLAMATION MARK的字符。最后四个codeUnit值（240，159，144，182）是一个四字节UTF-8表示DOG FACE的字符。

UTF-16表示

您可以String通过迭代其utf16属性来访问a的UTF-16表示。该属性是一个类型String.UTF16View，它是无符号16位（UInt16）值的集合，对于字符串的UTF-16表示中的每个16位代码单元都有一个值：

Character	D U+0044	o U+006F	g U+0067	!! U+203C	 U+1F436	
UTF-16 Code Unit	68	111	103	8252	55357	56374
Position	0	1	2	3	4	5

```
1 for codeUnit in dogString.utf16 {
2     print("\(codeUnit) ", terminator: "")
3 }
4 print("")
5 // Prints "68 111 103 8252 55357 56374 "
```

再次，前三个codeUnit值（68，111，103）所表示的字符D，o和g，其UTF-16代码单元具有如在字符串的UTF-8表示相同的值（因为这些Unicode标量表示ASCII字符）。

第四个codeUnit值（8252）是十六进制值的十进制等值203C，它代表了Unicode标U+203C为DOUBLE EXCLAMATION MARK字符。该字符可以表示为UTF-16中的单个代码单元。

第五个和第六个codeUnit值（55357和56374）是DOG FACE字符的UTF-16代理对表示。这些值的高代理值U+D83D（十进制值55357）和一个低代理值U+DC36（十进制值56374）。

Unicode标量表示

您可以String遍历其unicodeScalars属性来访问值的Unicode标量表示。该属性是类型的UnicodeScalarView，它是类型值的集合UnicodeScalar。

每个元素UnicodeScalar都有一个value属性，用于返回标量的21位值。该值表示为UInt+32：

Character	D U+0044	o U+006F	g U+0067	!! U+203C	🐶 U+1F436
Unicode Scalar Code Unit	68	111	103	8252	128054
Position	0	1	2	3	4

```
1 for scalar in dogString.unicodeScalars {
2     print("\(scalar.value) ", terminator: "")
3 }
4 print("")
5 // Prints "68 111 103 8252 128054 "
```

在本页

的value前三个属性UnicodeScalar值（68，111，103）再次表示字符D，o和g。

第四codeUnit值（8252）再次是十六进制值的十进位等值203C，它代表了Unicode标U+203C为DOUBLE EXCLAMATION MARK字符。

在value第五和最后的属性UnicodeScalar，128054是十六进制值的十进位等值1F436，它代表了Unicode标U+1F436为DOG FACE字符。

作为查询value属性的替代方法，UnicodeScalar还可以使用每个值构造一个新String值，例如使用字符串插值：

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