

The Swift Programming Language in Chinese

<https://github.com/letsswift>

<http://letsswift.com/>

June 15, 2014

目录

1	Swift 中文教程 (二十) 扩展	1
1.1	扩展语法 (Extension Syntax)	1
1.2	计算型属性 (Computed Properties)	2
1.3	构造器 (Initializers)	3
1.3.1	方法 (Methods)	4
1.3.2	修改实例方法 (Mutating Instance Methods)	5
1.3.3	下标 (Subscripts)	5
1.3.4	嵌套类型 (Nested Types)	6

1 Swift 中文教程 (二十) 扩展

扩展就是向一个已有的类、结构体或枚举类型添加新功能 (functionality)。这包括在没有权限获取原始源代码的情况下扩展类型的能力 (即逆向建模)。扩展和 Objective-C 中的分类 (categories) 类似。(不过与 Objective-C 不同的是, Swift 的扩展没有名字。)

Swift 中的扩展可以:

- 添加计算型属性和计算静态属性
- 定义实例方法和类型方法
- 提供新的构造器
- 定义下标
- 定义和使用新的嵌套类型
- 使一个已有类型符合某个接口

注意: 如果你定义了一个扩展向一个已有类型添加新功能, 那么这个新功能对该类型的所有已有实例中都是可用的, 即使它们是在你的这个扩展的前面定义的。

1.1 扩展语法 (Extension Syntax)

声明一个扩展使用关键字 `extension`:

```
extension SomeType {
    // 在 SomeType 中定义成员
}
```

一个扩展可以扩展一个已有类型, 使其能够适配一个或多个协议 (protocol)。当这种情况发生时, 接口的名字应该完全按照类或结构体的名字的方式进行书写:

```
extension SomeType: SomeProtocol, AnotherProtocol {
    // 实现协议方法
}
```

按照这种方式添加的协议遵循者 (protocol conformance) 被称之为在扩展中添加协议遵循者

1.2 计算型属性 (Computed Properties)

扩展可以向已有类型添加计算型实例属性和计算型类型属性。下面的例子向 Swift 的内建 Double 类型添加了 5 个计算型实例属性，从而提供与距离单位协作的基本支持。

```
extension Double {
    var km: Double { return self * 1_000.0 }
    var m : Double { return self }
    var cm: Double { return self / 100.0 }
    var mm: Double { return self / 1_000.0 }
    var ft: Double { return self / 3.28084 }
}

let oneInch = 25.4.mm
println("One inch is \(oneInch) meters")
// 输出: "One inch is 0.0254 meters"

let threeFeet = 3.ft
println("Three feet is \(threeFeet) meters")
// 输出: "Three feet is 0.914399970739201 meters"
```

这些计算属性表达的含义是把一个 Double 型的值看作是某单位下的长度值。即使它们被实现为计算型属性，但这些属性仍可以接一个带有 dot 语法的浮点型字面值，而这恰恰是使用这些浮点型字面量实现距离转换的方式。

在上述例子中，一个 Double 型的值 1.0 被用来表示“1 米”。这就是为什么 m 计算型属性返回 self——表达式 1.m 被认为是计算 1.0 的 Double 值。

其它单位则需要一些转换来表示在米下测量的值。1 千米等于 1,000 米，所以 km 计算型属性要把值乘以 1_000.00 来转化成单位米下的数值。类似地，1 米有 3.28024 英尺，所以 ft 计算型属性要把对应的 Double 值除以 3.28024 来实现英尺到米的单位换算。

这些属性是只读的计算型属性，所有从简考虑它们不用 get 关键字表示。它们的返回值是 Double 型，而且可以用于所有接受 Double 的数学计算中：

```
let aMarathon = 42.km + 195.m
println("A marathon is \(aMarathon) meters long")
// 输出: "A marathon is 42495.0 meters long"
```

注意：扩展可以添加新的计算属性，但是不可以添加存储属性，也不可以向已有属性添加属性观测器 (property observers)。

1.3 构造器 (Initializers)

扩展可以向已有类型添加新的构造器。这可以让你扩展其它类型，将你自己的定制类型作为构造器参数，或者提供该类型的原始实现中没有包含的额外初始化选项。

注意：如果你使用扩展向一个值类型添加一个构造器，该构造器向所有的存储属性提供默认值，而且没有定义任何定制构造器 (custom initializers)，那么对于来自你的扩展构造器中的值类型，你可以调用默认构造器 (default initializers) 和成员级构造器 (memberwise initializers)。正如在值类型的构造器授权中描述的，如果你已经把构造器写成值类型原始实现的一部分，上述规则不再适用。

下面的例子定义了一个用于描述几何矩形的定制结构体 Rect。这个例子同时定义了两个辅助结构体 Size 和 Point，它们都把 0.0 作为所有属性的默认值：

```
struct Size {  
    var width = 0.0, height = 0.0  
}  
  
struct Point {  
    var x = 0.0, y = 0.0  
}  
  
struct Rect {  
    var origin = Point()  
    var size = Size()  
}
```

因为结构体 Rect 提供了其所有属性的默认值，所以正如默认构造器中描述的，它可以自动接受一个默认的构造器和一个成员级构造器。这些构造器可以用于构造新的 Rect 实例：

```
let defaultRect = Rect()  
let memberwiseRect = Rect(origin: Point(x: 2.0, y: 2.0),  
    size: Size(width: 5.0, height: 5.0))
```

你可以提供一个额外的使用特殊中心点和大小的构造器来扩展 Rect 结构体：

```
extension Rect {  
    init(center: Point, size: Size) {  
        let originX = center.x - (size.width / 2)  
        let originY = center.y - (size.height / 2)  
        self.init(origin: Point(x: originX, y: originY), size: size)  
    }  
}
```

这个新的构造器首先根据提供的 `center` 和 `size` 值计算一个合适的原点。然后调用该结构体自动的成员构造器 `init(origin:size:)`，该构造器将新的原点和大小存到了合适的属性中：

```
let centerRect = Rect(center: Point(x: 4.0, y: 4.0),  
    size: Size(width: 3.0, height: 3.0))  
// centerRect 的 origin 是 (2.5, 2.5)，size 是 (3.0, 3.0)
```

注意：如果你使用扩展提供了一个新的构造器，你依旧有责任保证构造过程能够让所有实例完全初始化。

1.3.1 方法 (Methods)

扩展可以向已有类型添加新的实例方法和类型方法。下面的例子向 `Int` 类型添加一个名为 `repetitions` 的新实例方法：

```
extension Int {  
    func repetitions(task: () -> ()) {  
        for i in 0..  
            self {  
                task()  
            }  
    }  
}
```

这个 `repetitions` 方法使用了一个 `() -> ()` 类型的单参数 (single argument)，表明函数没有参数而且没有返回值。

定义该扩展之后，你就可以对任意整数调用 `repetitions` 方法，实现的功能则是多次执行某任务：

```
3.repetitions({
    println("Hello!")
})
// Hello!
// Hello!
// Hello!
```

可以使用 trailing 闭包使调用更加简洁:

```
3.repetitions{
    println("Goodbye!")
}
// Goodbye!
// Goodbye!
// Goodbye!
```

1.3.2 修改实例方法 (Mutating Instance Methods)

通过扩展添加的实例方法也可以修改该实例本身。结构体和枚举类型中修改 self 或其属性的方法必须将该实例方法标注为 mutating, 正如来自原始实现的修改方法一样。

下面的例子向 Swift 的 Int 类型添加了一个新的名为 square 的修改方法, 来实现一个原始值的平方计算:

```
extension Int {
    mutating func square() {
        self = self * self
    }
}
var someInt = 3
someInt.square()
// someInt 现在等于 9
```

1.3.3 下标 (Subscripts)

扩展可以向一个已有类型添加新下标。这个例子向 Swift 内建类型 Int 添加了一个整型下标。该下标 [n] 返回十进制数字从右向左数的第 n 个数字

- 123456789[0] 返回 9
- 123456789[1] 返回 8
- ...等等

```
extension Int {
    subscript(digitIndex: Int) -> Int {
        var decimalBase = 1
        for _ in 1...digitIndex {
            decimalBase *= 10
        }
        return (self / decimalBase) % 10
    }
}

746381295[0]
// returns 5
746381295[1]
// returns 9
746381295[2]
// returns 2
746381295[8]
// returns 7
```

如果该 Int 值没有足够的位数，即下标越界，那么上述实现的下标会返回 0，因为它会在数字左边自动补 0：

```
746381295[9]
//returns 0, 0 0 0 0 :
0746381295[9]
```

1.3.4 嵌套类型 (Nested Types)

扩展可以向已有的类、结构体和枚举添加新的嵌套类型：

```
extension Character {
    enum Kind {
        case Vowel, Consonant, Other
    }
}
```



```

var kind: Kind {
    switch String(self).lowercaseString {
    case "a", "e", "i", "o", "u":
        return .Vowel
    case "b", "c", "d", "f", "g", "h", "j", "k", "l", "m",
         "n", "p", "q", "r", "s", "t", "v", "w", "x", "y", "z":
        return .Consonant
    default:
        return .Other
    }
}

```

该例子向 `Character` 添加了新的嵌套枚举。这个名为 `Kind` 的枚举表示特定字符的类型。具体来说，就是表示一个标准的拉丁脚本中的字符是元音还是辅音（不考虑口语和地方变种），或者是其它类型。

这个类子还向 `Character` 添加了一个新的计算实例属性，即 `kind`，用来返回合适的 `Kind` 枚举成员。

现在，这个嵌套枚举可以和一个 `Character` 值联合使用了：

```

func printLetterKinds(word: String) {
    println("'\\(word)' is made up of the following kinds of letters:")
    for character in word {
        switch character.kind {
        case .Vowel:
            print("vowel ")
        case .Consonant:
            print("consonant ")
        case .Other:
            print("other ")
        }
    }
    print("\\n")
}

printLetterKinds("Hello")
// 'Hello' is made up of the following kinds of letters:
// consonant vowel consonant consonant vowel

```

函数 `printLetterKinds` 的输入是一个 `String` 值并对其字符进行迭代。在每次迭代过程中，考虑当前字符的 `kind` 计算属性，并打印出合适的类别描述。所以 `printLetterKinds` 就可以用来打印一个完整单词中所有字母的类型，正如上述单词 “hello” 所展示的。

注意：由于已知 `character.kind` 是 `Character.Kind` 型，所以 `Character.Kind` 中的所有成员值都可以使用 `switch` 语句里的形式简写，比如使用 `Vowel` 代替 `Character.Kind.Vowel`

本文部分原文来自于 <http://www.swiftguide.cn/> 翻译小组的译文，共同校对中。

感谢翻译小组成员：李起攀 (微博)、若晨 (微博)、YAO、粽子、山有木兮木有枝、渺 -Bessie、墨离、Tiger 大顾 (微博)，校对：CXH_ME(微博), Joshua 孟思拓 (微博)

本文由翻译小组成员原创发布，个人转载请注明出处和原始链接，商业转载请联系我们 感谢您对我们工作的支持