The Swift Programming Language in Chinese

https://github.com/letsswift

http://letsswift.com/

June 15, 2014

目录

1 Swift 中文教程（八）枚举类型

枚举定义了一个常用的具有相关性的一组数据，并在你的代码中以一个安全的方式使用它们。

如果你熟悉 C 语言，你就会知道，C 语言中的枚举指定相关名称为一组整数值。在 Swift 中枚举更为灵活，不必为枚举的每个成员提供一个值。如果一个值（被称为 “原始” 的值）被提供给每个枚举成员，则该值可以是一个字符串，一个字符，或者任何整数或浮点类型的值。

另外，枚举成员可以指定任何类型，每个成员都可以存储的不同的相关值，就像其他语言中使用集合或变体。你还可以定义一组通用的相关成员为一个枚举，每一种都有不同的一组与它相关的适当类型的值的一部分。

在 Swift 中枚举类型是最重要的类型。它采用了很多以前只有类才具有的特性，如计算性能，以提供有关枚举的当前值的更多信息，方法和实例方法提供的功能相关的枚举表示的值传统上支持的许多功能。枚

举也可以定义初始化，以提供一个初始成员值; 可以在原有基础上扩展扩大它们的功能; 并使用协议来提供标准功能。

欲了解更多有关这些功能，请参见 Properties, Methods, Initialization, Extensions, Protocols

1.1 枚举语法

使用枚举 enum 关键词并把他们的整个定义在一对大括号内：

```
enum SomeEnumeration {
    // enumeration definition goes here
}
```

下面是一个指南针的四个点一个例子：

```
enum CompassPoint {
    case North
    case South
    case East
    case West
}
```

在枚举中定义的值（如 North，South，East 和 West）是枚举的成员值（或成员)。这个例子里 case 关键字表示成员值一条新的分支将被定义。

> Note 不像 C 和 Objective-C，Swift 枚举成员在创建时不分配默认整数值。在上面的例子 CompassPoints 中 North，South，Eath，West 不等于隐含 0，1，2 和 3，而是一种与 CompassPoint 明确被定义的类型却各不相同的值。

多个成员的值可以出现在一行上，用逗号分隔:

```
enum Planet {
   case Mercury, Venus, Earth, Mars, Jupiter, Saturn, Uranus, Neptune
}
```

每个枚举定义中定义了一个全新的类型。像其他 Swift 的类型，它们的名称(如 CompassPoint 和 Planet）应为大写字母。给枚举类型单数而不是复数的名字，这样理解起来更加容易如:

```
var directionToHead = CompassPoint.West
```

使用 directionToHead 的类型时，用 CompassPoint 的一个可能值初始化的推断。一旦 directionToHead 被声明为一个 CompassPoint，您可以将其设置为使用更短的. 语法而不用再书写枚举 CompassPoint 值本身:

```
directionToHead = .East
```

directionToHead 的类型是已知的，所以你可以在设定它的值时，不写该类型。使用类型明确的枚举值可以让代码具有更好的可读性。

1.2 匹配枚举值与 **switch** 语句

你可以使用单个枚举值匹配 switch 语句:

```
directionToHead = .South
switch directionToHead {
    case .North:
        println("Lots of planets have a north")
    case .South:
```

```
        println("Watch out for penguins")
    case .East:
        println("Where the sun rises")
    case .West:
        println("Where the skies are blue")
}
// prints "Watch out for penguins"
```

你可以理解这段代码:

“考虑 directionToHead 的价值。当它等于 North，打印 “Lots of planets have a north”。当它等于 South，打印 “Watch out for penguins” 等等。

正如控制流所描述，Switch 语句考虑枚举的成员, 如果省略了 West 时，这段代码无法编译，因为它没有考虑 CompassPoint 成员的完整性。Switch 语句要求全面性确保枚举成员，避免不小心漏掉情况发生。

当它不需要为每一个枚举成员都匹配的情况下，你可以提供一个默认 default 分支来涵盖未明确提到的任何成员:

```
let somePlanet = Planet.Earth
switch somePlanet {
    case .Earth:
        println("Mostly harmless")
    default:
        println("Not a safe place for humans")
}
// prints "Mostly harmless"
```

1.3 关联值

在上一节中的示例延时了一个枚举的成员是如何被定义（分类）的。你可以为 Planet.Earth 设置一个常量或变量，然后在代码中检查这个值。但是，它有时是有用的才能存储其它类型的关联值除了这些成员的值。这让你随着成员值存储额外的自定义信息，并允许在你的代码中来使用该信息。

Swift 的枚举类型可以由一些数据类型相关的组成，如果需要的话，这些数据类型可以是各不相同的。枚举的这种特性跟其它语言中的奇异集合，标签集合或者变体相似

例如，假设一个库存跟踪系统需要由两种不同类型的条形码来跟踪产品。有些产品上标有 UPC-A 代码格式，它使用数字 0 到 9 的一维条码，每一个条码都有一个 “数字系统” 的数字，后跟十 “标识符” 的数字。最后一位是 “检查” 位，以验证代码已被正确扫描：

其他产品都贴有二维条码 QR 码格式，它可以使用任何的 ISO8859-1 字符，并可以编码字符串，最多 2,953 个字符：

这将是方便的库存跟踪系统能够存储 UPC-A 条码作为三个整数的元组，和 QR 代码的条形码的任何长度的字符串。

在 Swift 中可以使用一个枚举来定义两种类型的产品条形码，结构可以是这样的：

```
enum Barcode {
    case UPCA(Int, Int, Int)
    case QRCode(String)
}
```

这可以被理解为：

“定义一个名为条形码枚举类型，它可以是 UPC-A 的任一值类型的关联值 (Int，Int，Int)，或 QRCode 的一个类型为 String 的关联值。”

这个定义不提供任何实际的 Int 或 String 值，它只是定义了条形码常量和变量当等于 Barcode.UPCA 或 Barcode.QRCode 关联值的类型的时候的存储形式。

然后可以使用任何一种类型来创建新的条码：

```
var productBarcode = Barcode.UPCA(8, 85909_51226, 3)
```

此示例创建一个名为 productBarcode 新的变量，并与相关联的元组值赋给它 Barcode.UPCA 的值（8，8590951226，3）。提供的 “标识符” 值都有整数加下划线的文字，85909_51226，使其更易于阅读的条形码。

同一产品可以分配不同类型的条形码：

```
productBarcode = .QRCode("ABCDEFGHIJKLMNOP")
```

在这一点上，原来 Barcode.UPCA 和其整数值被新的 Barcode.QRCode 及其字符串值代替。*条形码的常量和变量可以存储任何一个* UPCA 或 QRCode 的（连同其关联值），但它们只能存储其中之一在任何指定时间。

不同的条码类型像以前一样可以使用一个 switch 语句来检查，但是这一次相关的值可以被提取作为 switch 语句的一部分。您提取每个相关值作为常数（let 前缀）或变量（var 前缀）不同的情况下，在 switch 语句的 case 代码内使用：

```
switch productBarcode {
    case .UPCA(let numberSystem, let identifier, let check):
```

```
     println("UPC-A with value of \(numberSystem), \(identifier), \(check).")
    case .QRCode(let productCode):
        println("QR code with value of \(productCode).")
}
// prints "QR code with value of ABCDEFGHIJKLMNOP."
```

如果所有的枚举成员的关联值的提取为常数，或者当所有被提取为变量，为了简洁起见，可以放置一个 var，或 let 标注在成员名称前：

```
switch productBarcode {
    case let .UPCA(numberSystem, identifier, check):
     println("UPC-A with value of \(numberSystem), \(identifier), \(check).")
    case let .QRCode(productCode):
        println("QR code with value of \(productCode).")
}
// prints "QR code with value of ABCDEFGHIJKLMNOP."
```

1.4 原始值

在关联值的条形码的例子演示了一个枚举的成员如何能声明它们存储不同类型的关联值。作为替代关联值，枚举成员可以拿出预先填入缺省值（称为原始值），从而具有相同的类型。

这里是一个存储原始的 ASCII 值命名枚举成员的一个例子：

```
enum ASCIIControlCharacter: Character {
    case Tab = "\t"
    case LineFeed = "\n"
    case CarriageReturn = "\r"
}
```

在这里，原始值被定义为字符类型的枚举叫做 ASCIIControlCharacter，并设置了一些比较常见的 ASCII 控制字符。字符值的字符串和字符的描述。

注意，原始值是不相同关联值。原始值设置为预填充的值时，应先在你的代码中定义枚举，像上述三个 ASCII 码。对于一个特定的枚举成员的原始值始终是相同的。当你创建一个基于枚举的常量或变量的新成员的关联值设置，每次当你这样做的时候可以是不同的。

原始值可以是字符串，字符，或任何整数或浮点数类型。每个原始值必须在它的枚举中唯一声明。当整数被用于原始值，如果其他 枚举成员没有值时，它们自动递增。

下面列举的是一个细化的早期 Planet 枚举，使用原始整数值来表示每个 Planet 的太阳系的顺序：

```
enum Planet: Int {
   case Mercury = 1, Venus, Earth, Mars, Jupiter, Saturn, Uranus, Neptune
}
```

自动递增意味着 Planet.Venus 具有 2 的原始值，依此类推。

访问其 toRaw 方法枚举成员的原始值：

```
let earthsOrder = Planet.Earth.toRaw()
// earthsOrder is 3
```

使用枚举的 fromRaw 方法来试图找到一个特定的原始值枚举成员。这个例子识别 Uranus 的位置通过原始值为 7：

```
let possiblePlanet = Planet.fromRaw(7)
// possiblePlanet is of type Planet? and equals Planet.Uranus
```

然而，并非所有可能的 Int 值都会找到一个匹配的星球。正因如此，该 fromRaw 方法返回一个可选的枚举成员。在上面的例子中，是 possiblePlanet 类型 Planet? 或 “可选的 Planet”。

如果你试图找到一个 Planet 为 9 的位置，通过 fromRaw 返回可选的 Planet 值将是无：

```
let positionToFind = 9
if let somePlanet = Planet.fromRaw(positionToFind) {
    switch somePlanet {
    case .Earth:
        println("Mostly harmless")
    default:
        println("Not a safe place for humans")
    }
} else {
```

```
    println("There isn't a planet at position \(positionToFind)")
}
// prints "There isn't a planet at position 9"
```

这个范例使用 somePlanet= Planet.fromRaw(9) 来尝试访问可选集合 Planet，在可选 Planet 集合中设置检索条件 somePlanet，在原始值为 9 的情况下，不能检索到位置为 9 的星球，所有 else 分支被执行。

参考文献