翻译：yankuangshi　校对：shinyzhu

枚举（Enumerations）

本页内容包含：

枚举定义了一个通用类型的一组相关的值，使你可以在你的代码中以一个安全的方式来使用这些值。

如果你熟悉 C 语言，你就会知道，在 C 语言中枚举指定相关名称为一组整型值。Swift 中的枚举更加灵活，不必给每一个枚举成员提供一个值。如果一个值（被认为是“原始”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，或是一个整型值或浮点值。

此外，枚举成员可以指定任何类型的实例值存储到枚举成员值中，就像其他语言中的联合体（unions）和变体（variants）。你可以定义一组通用的相关成员作为枚举的一部分，每一组都有不同的一组与它相关的适当类型的数值。

在 Swift 中，枚举类型是一等（first-class）类型。它们采用了很多传统上只被类（class)所支持的特征，例如计算型属性（computed properties），用于提供关于枚举当前值的附加信息， 实例方法（instance methods），用于提供和枚举所代表的值相关联的功能。枚举也可以定义构造函数（initializers）来提供一个初始成员值；可以在原始的实现基础上扩展它们的功能；可以遵守协议（protocols）来提供标准的功能。

欲了解更多相关功能，请参见属性（Properties），方法（Methods），构造过程（Initialization），扩展（Extensions）和协议（Protocols）。

枚举语法

使用enum关键词并且把它们的整个定义放在一对大括号内：

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

以下是指南针四个方向的一个例子：

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

一个枚举中被定义的值（例如North，South，East和West）是枚举的**成员值**（或者**成员**）。case关键词表明新的一行成员值将被定义。

注意：

不像 C 和 Objective-C 一样，Swift 的枚举成员在被创建时不会被赋予一个默认的整数值。在上面的CompassPoints例子中，North，South，East和West不是隐式的等于0，1，2和3。相反的，这些不同的枚举成员在CompassPoint的一种显示定义中拥有各自不同的值。

多个成员值可以出现在同一行上，用逗号隔开：

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

每个枚举定义了一个全新的类型。像 Swift 中其他类型一样，它们的名字（例如CompassPoint和Planet）必须以一个大写字母开头。给枚举类型起一个单数名字而不是复数名字，以便于读起来更加容易理解：

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

directionToHead的类型被推断当它被CompassPoint的一个可能值初始化。一旦directionToHead被声明为一个CompassPoint，你可以使用更短的点（.）语法将其设置为另一个CompassPoint的值：

```
directionToHead = .East
```

directionToHead的类型已知时，当设定它的值时，你可以不再写类型名。使用显示类型的枚举值可以让代码具有更好的可读性。

匹配枚举值和Switch语句

你可以匹配单个枚举值和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")
}
// 输出 "Watch out for penguins”
```

你可以如此理解这段代码：

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

等等依次类推。

正如在控制流（Control Flow）中介绍，当考虑一个枚举的成员们时，一个switch语句必须全面。如果忽略了.West这种情况，上面那段代码将无法通过编译，因为它没有考虑到CompassPoint的全部成员。全面性的要

求确保了枚举成员不会被意外遗漏。

当不需要匹配每个枚举成员的时候，你可以提供一个默认`default`分支来涵盖所有未明确被提出的任何成员：

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

实例值（Associated Values）

上一小节的例子演示了一个枚举的成员是如何被定义（分类）的。你可以为`Planet.Earth`设置一个常量或则变量，并且在之后查看这个值。然而，有时候会很有用如果能够把其他类型的实例值和成员值一起存储起来。这能让你随着成员值存储额外的自定义信息，并且当每次你在代码中利用该成员时允许这个信息产生变化。

你可以定义 Swift 的枚举存储任何类型的实例值，如果需要的话，每个成员的数据类型可以是各不相同的。枚举的这种特性跟其他语言中的可辨识联合（discriminated unions），标签联合（tagged unions），或者变体（variants）相似。

例如，假设一个库存跟踪系统需要利用两种不同类型的条形码来跟踪商品。有些商品上标有 UPC-A 格式的一维码，它使用数字 0 到 9。每一个条形码都有一个代表“数字系统”的数字，该数字后接 10 个代表“标识符”的数字。最后一个数字是“检查”位，用来验证代码是否被正确扫描：

其他商品上标有 QR 码格式的二维码，它可以使用任何 ISO8859-1 字符，并且可以编码一个最多拥有 2,953 字符的字符串:

对于库存跟踪系统来说，能够把 UPC-A 码作为三个整型值的元组，和把 QR 码作为一个任何长度的字符串存储起来是方便的。

在 Swift 中，用来定义两种商品条码的枚举是这样子的:

```
enum Barcode {
    case UPCA(Int, Int, Int)
    case QRCode(String)
}
```
以上代码可以这么理解：

“定义一个名为Barcode的枚举类型，它可以是UPCA的一个实例值（Int，Int，Int），或者QRCode的一个字符串类型（String）实例值。”

这个定义不提供任何Int或String的实际值，它只是定义了，当Barcode常量和变量等于Barcode.UPCA或Barcode.QRCode时，实例值的类型。

然后可以使用任何一种条码类型创建新的条码，如：

```
var productBarcode = Barcode.UPCA(8, 85909_51226, 3)
```
以上例子创建了一个名为productBarcode的新变量，并且赋给它一个Barcode.UPCA的实例元组值(8, 8590951226, 3)。提供的“标识符”值在整数字中有一个下划线，使其便于阅读条形码。

同一个商品可以被分配给一个不同类型的条形码，如：

```
productBarcode = .QRCode("ABCDEFGHIJKLMNOP")
```
这时，原始的Barcode.UPCA和其整数值被新的Barcode.QRCode和其字符串值所替代。条形码的常量和变量可以存储一个.UPCA或者一个.QRCode（连同它的实例值），但是在任何指定时间只能存储其中之一。

像以前那样，不同的条形码类型可以使用一个 switch 语句来检查，然而这次实例值可以被提取作为 switch 语句的一部分。你可以在switch的 case 分支代码中提取每个实例值作为一个常量（用let前缀）或者作为一个变量（用var前缀）来使用：

```
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).")
}
// 输出 "QR code with value of ABCDEFGHIJKLMNOP.”
```

如果一个枚举成员的所有实例值被提取为常量，或者它们全部被提取为变量，为了简洁，你可以只放置一个`var`或者`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).")
}
// 输出 "QR code with value of ABCDEFGHIJKLMNOP."
```

原始值（Raw Values）

在实例值小节的条形码例子中演示了一个枚举的成员如何声明它们存储不同类型的实例值。作为实例值的替代，枚举成员可以被默认值（称为原始值）预先填充，其中这些原始值具有相同的类型。

这里是一个枚举成员存储原始 ASCII 值的例子：

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

在这里，称为`ASCIIControlCharacter`的枚举的原始值类型被定义为字符型`Character`，并被设置了一些比较常见的 ASCII 控制字符。字符值的描述请详见字符串和字符`Strings and Characters`部分。

注意，原始值和实例值是不相同的。当你开始在你的代码中定义枚举的时候原始值是被预先填充的值，像上述三个 ASCII 码。对于一个特定的枚举成员，它的原始值始终是相同的。实例值是当你在创建一个基于枚举成员

的新常量或变量时才会被设置，并且每次当你这么做得时候，它的值可以是不同的。

原始值可以是字符串，字符，或者任何整型值或浮点型值。每个原始值在它的枚举声明中必须是唯一的。当整型值被用于原始值，如果其他枚举成员没有值时，它们会自动递增。

下面的枚举是对之前`Planet`这个枚举的一个细化，利用原始整型值来表示每个 planet 在太阳系中的顺序：

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

自动递增意味着`Planet.Venus`的原始值是`2`，依次类推。

使用枚举成员的`toRaw`方法可以访问该枚举成员的原始值：

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

使用枚举的`fromRaw`方法来试图找到具有特定原始值的枚举成员。这个例子通过原始值`7`识别`Uranus`：

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

然而，并非所有可能的`Int`值都可以找到一个匹配的行星。正因为如此，`fromRaw`方法可以返回一个可选的枚举成员。在上面的例子中，`possiblePlanet`是`Planet?`类型，或“可选的`Planet`”。

如果你试图寻找一个位置为9的行星，通过`fromRaw`返回的可选`Planet`值将是`nil`：

```
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)")
}
// 输出 "There isn't a planet at position 9
```

这个范例使用可选绑定（optional binding），通过原始值`9`试图访问一个行星。`if let somePlanet = Planet.fromRaw(9)`语句获得一个可选`Planet`，如果可选`Planet`可以被获得，把`somePlanet`设置成该可选`Planet`的内容。在这个范例中，无法检索到位置为`9`的行星，所以`else`分支被执行。