枚举

在本页

一个*枚举*定义了一个通用型的一组相关的值，使你在你的代码中的一个类型安全的方式这些值来工作。

如果您熟悉C，您将知道C枚举将相关名称分配给一组整数值。Swift中的枚举更加灵活，并且不必为枚举的每种情况提供值。如果一个值（被称为“原始”的值）*被*提供给每个枚举的情况下，该值可以是一个字符串，一个字符，或任何整数的值或者浮点型。

或者，枚举案例可以指定与每个不同案例值一起存储的*任何*类型的关联值，这与联合会或变体在其他语言中的做法非常相似。您可以将一组相关案例定义为一个枚举的一部分，其中每个案例都有一组与其相关的适当类型的值。

Swift中的枚举本身就是一流的类型。它们采用了传统上仅由类支持的许多功能，例如计算属性以提供有关枚举当前值的附加信息，以及提供与枚举所代表的值相关的功能的实例方法。枚举还可以定义初始化器以提供初始案例值; 可以扩展到超出其原始实现范围的功能; 并且可以符合协议以提供标准功能。

有关这些功能的更多信息，请参阅属性，方法，初始化，扩展和协议。

枚举语法

您可以使用enum关键字引入枚举并将其整个定义放在一对大括号中：

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

以下是一个指南针四个要点的例子：

```
enum CompassPoint {
    case north
    case south
    case east
    case west
}
```

在枚举定义的值（例如north，south，east，和west）是其*枚举的情况下*。您使用case关键字来引入新的枚举案例。

> 注意
>
> 与C和Objective-C不同，Swift枚举案例在创建时未被赋予默认的整数值。在CompassPoint上面的例子，north，south，east和west不等于隐式0，1，2和3。相反，不同的枚举案例本身就是完全成熟的值，具有明确定义的类型CompassPoint。

多个案例可以在一行中出现，用逗号分隔：

```
enum Planet {
    case mercury, venus, earth, mars, jupiter, saturn, uranus, neptune
}
```

每个枚举定义都定义了一种全新的类型。像Swift中的其他类型一样，它们的名称（例如CompassPoint和Planet）应该以大写字母开头。给枚举类型单数而不是复数名称，以便他们阅读不言自明：

```
var directionToHead = CompassPoint.west
```

directionToHead使用其中一个可能值进行初始化时，会推断出该类型CompassPoint。一旦directionToHead声明为a CompassPoint，您可以CompassPoint使用较短的点语法将其设置为不同的值：

```
directionToHead = .east
```

类型`directionToHead`是已知的，因此您可以在设置其值时删除类型。这使得在使用显式类型的枚举值时具有高度可读的代码。

用枚举语句匹配枚举值

您可以将单个枚举值与一个`switch`语句进行匹配：

```
directionToHead = .south
switch directionToHead {
case .north:
    print("Lots of planets have a north")
case .south:
    print("Watch out for penguins")
case .east:
    print("Where the sun rises")
case .west:
    print("Where the skies are blue")
}
// Prints "Watch out for penguins"
```

你可以阅读这段代码：

“考虑价值`directionToHead`。在它相等的情况下`.north`，打印`"Lots of planets have a north"`。在它相等的情况下`.south`，打印`"Watch out for penguins"`。“

...等等。

如控制流程中所述，`switch`在考虑枚举的情况下，声明必须是详尽的。如果省略了`case`for `.west`，则此代码不会编译，因为它不考虑完整的`CompassPoint`案例列表。要求详尽无遗确保枚举案件不会被意外省略。

如果不适合`case`为每个枚举案例提供一个案例，则可以提供一个`default`案例来涵盖任何未明确解决的案例：

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

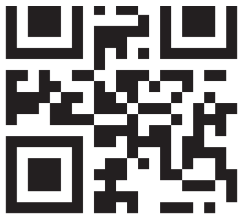
关联值

上一节中的示例显示枚举的情况是如何定义（和键入）的值。您可以设置一个常量或变量`Planet.earth`，并在稍后检查该值。但是，能够在这些案例值旁边存储其他类型的*关联值*有时很有用。这使您可以将额外的自定义信息与案例值一起存储，并且每次在代码中使用该案例时都会允许此信息发生变化。

您可以定义Swift枚举来存储任何给定类型的关联值，并且如果需要，值类型对于枚举的每种情况都可以不同。与这些枚举类似的枚举被称为*区分联合*，*标记联合*或其他编程语言中的*变体*。

例如，假设库存跟踪系统需要通过两种不同类型的条形码跟踪产品。某些产品以UPC格式标记有一维条形码，该格式使用数字`0`来表示`9`。每个条形码都有一个“数字系统”数字，随后是五个“制造商代码”数字和五个“产品代码”数字。这些后面跟着一个“检查”数字以验证代码已被正确扫描：

其他产品采用QR码格式的二维条码进行标注，可以使用任何ISO 8859-1字符，并且可以编码长达2,953个字符的字符串：



库存跟踪系统可以方便地将UPC条形码存储为四个整数的元组，并将QR码条形码存储为任意长度的字符串。

在Swift中，用于定义任一类型的产品条形码的枚举可能如下所示：

```
enum Barcode {
    case upc(Int, Int, Int, Int)
    case qrCode(String)
}
```

这可以理解为：

“定义称为枚举类型Barcode，它可以采取任何的值upc与式（的相关值Int，Int，Int，Int），或的值qrCode与类型的相关联的值String。”

该定义不提供任何实际值Int或String值 - 它只是定义常量和变量在等于或时可以存储的关联值的*类型*。BarcodeBarcode.upcBarcode.qrCode

然后可以使用以下任一类型创建新的条形码：

```
var productBarcode = Barcode.upc(8, 85909, 51226, 3)
```

这个例子创建一个新的变量，productBarcode并为它赋值Barcode.upc一个关联的元组值(8, 85909, 51226, 3)。

相同的产品可以分配不同类型的条形码：

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

此时，原始Barcode.upc值及其整数值将由新Barcode.qrCode值及其字符串值替换。类型的常量和变量Barcode可以存储a .upc或a .qrCode（以及它们的关联值），但是它们只能在任何给定时间存储其中的一个。

像以前一样，可以使用switch语句检查不同的条形码类型。但是，这次可以将关联的值作为switch语句的一部分提取出来。您可以将每个关联值提取为一个常量（带let前缀）或一个变量（带var前缀）以便在switch案例正文中使用：

```
switch productBarcode {
case .upc(let numberSystem, let manufacturer, let product, let check):
    print("UPC: \(numberSystem), \(manufacturer), \(product), \(check).")
case .qrCode(let productCode):
    print("QR code: \(productCode).")
}
// Prints "QR code: ABCDEFGHIJKLMNOP."
```

如果枚举案例的所有关联值都被提取为常量，或者全部提取为变量，则为简洁起见，您可以在案例名称之前放置一个var或let注释。

```
switch productBarcode {
case let .upc(numberSystem, manufacturer, product, check):
    print("UPC : \(numberSystem), \(manufacturer), \(product), \(check).")
case let .qrCode(productCode):
    print("QR code: \(productCode).")
}
// Prints "QR code: ABCDEFGHIJKLMNOP."
```

原始价值

关联值中 的条形码示例显示枚举的案例如何声明它们存储不同类型的关联值。作为关联值的替代，枚举案例可以预先填充所有相同类型的默认值（称为*原始值*）。

下面是一个例子，它将原始ASCII值与名为枚举的情况一起存储：

```
enum ASCIIControlCharacter: Character {
    case tab = "\t"
    case lineFeed = "\n"
    case carriageReturn = "\r"
}
```

这里，被调用的枚举的原始值`ASCIIControlCharacter`被定义为类型`Character`，并被设置为一些更常见的ASCII控制字符。`Character`值在字符串和字符中描述。

原始值可以是字符串，字符或任何整数或浮点数类型。每个原始值在其枚举声明中必须是唯一的。

> 注意
>
> 原始值是*不*一样的关联值。当您首先在您的代码中定义枚举时，将原始值设置为预填充值，如上面的三个ASCII代码。特定枚举个案的原始值始终相同。当您根据某个枚举的情况创建新的常量或变量时，会设置关联值，并且每次都可能会有所不同。

隐式分配的原始值

在处理存储整数或字符串原始值的枚举时，不必为每个个案明确分配一个原始值。当你不这样做时，Swift会自动为你分配值。

例如，当整数用于原始值时，每种情况的隐含值比前一种情况多一个。如果第一个案例没有设置值，则其值为`0`。

下面的枚举是前面`Planet`枚举的一个改进，用整数原始值来表示每个行星从太阳的顺序：

```
enum Planet: Int {
    case mercury = 1, venus, earth, mars, jupiter, saturn, uranus, neptune
}
```

在上面的例子中，`Planet.mercury`有一个明确的原始值`1`，`Planet.venus`有一个隐含的原始值`2`，等等。

当字符串用于原始值时，每个案例的隐含值是该案例名称的文本。

下面的枚举是对前面`CompassPoint`枚举的改进，用字符串原始值来表示每个方向的名称：

```
enum CompassPoint: String {
    case north, south, east, west
}
```

在上面的例子中，`CompassPoint.south`有一个隐含的原始值`"south"`，等等。

您可以使用其`rawValue`属性访问枚举个案的原始值：

```
let earthsOrder = Planet.earth.rawValue
// earthsOrder is 3

let sunsetDirection = CompassPoint.west.rawValue
// sunsetDirection is "west"
```

从原始值初始化

如果使用原始值类型定义枚举，则枚举将自动接收一个初始值设定项，该初始值设定项将采用原始值类型的值（作为参数调用`rawValue`）并返回枚举大小写或`nil`。您可以使用此初始化程序尝试创建枚举的新实例。

这个例子确定了天王星的原始价值`7`：

```
let possiblePlanet = Planet(rawValue: 7)
// possiblePlanet is of type Planet? and equals Planet.uranus
```

但是，并非所有可能的`Int`值都会找到匹配的行星。因此，原始值初始值设定项始终返回一个*可选的*枚举大小写。在上面的例子中，`possiblePlanet`是类型的`Planet?`，或者是“可选的”`Planet`。

> 注意
>
> 原始值初始化器是一个可分解的初始化器，因为不是每个原始值都会返回一个枚举大小写。有关更多信息，请参阅Failable Initializers。

如果您尝试找到位置为的行星，则由原始值初始值设定项返回`11`的可选`Planet`值将为`nil`：

```
let positionToFind = 11
if let somePlanet = Planet(rawValue: positionToFind) {
    switch somePlanet {
    case .earth:
        print("Mostly harmless")
    default:
        print("Not a safe place for humans")
    }
} else {
    print("There isn't a planet at position \(positionToFind)")
}
// Prints "There isn't a planet at position 11"
```

该示例使用可选绑定来尝试访问原始值为的行星`11`。该语句`if let somePlanet = Planet(rawValue: 11)`创建一个可选项`Planet`，如果可以检索`somePlanet`，`Planet`则设置为该可选项的值。在这种情况下，这是不可能检索星球的位置`11`，因此`else`，转而执行分支。

递归枚举

甲*递归枚举*是具有枚举作为一个或一个以上的枚举案件相关联的值的另一个实例的枚举。您通过`indirect`在它之前写入来指示枚举案例是递归的，这会告诉编译器插入必要的间接层。

例如，下面是一个存储简单算术表达式的枚举：

```
enum ArithmeticExpression {
    case number(Int)
    indirect case addition(ArithmeticExpression, ArithmeticExpression)
    indirect case multiplication(ArithmeticExpression, ArithmeticExpression)
}
```

您也可以`indirect`在枚举开始之前写入以启用所有枚举的具有关联值的情况的间接寻址：

```
indirect enum ArithmeticExpression {
    case number(Int)
    case addition(ArithmeticExpression, ArithmeticExpression)
    case multiplication(ArithmeticExpression, ArithmeticExpression)
}
```

该枚举可以存储三种算术表达式：一个普通数，两个表达式的相加，以及两个表达式的相乘。这些`addition`和`multiplication`关联的值具有相关的值，这些值也是算术表达式 - 这些关联值可以嵌套表达式。例如，该表达式`(5 + 4) * 2`在乘法的右侧有一个数字，在乘法的左侧有另一个表达式。因为数据是嵌套的，所以用于存储数据的枚举也需要支持嵌套 - 这意味着枚举需要递归。下面的代码显示了为以下内容`ArithmeticExpression`创建的递归枚举`(5 + 4) * 2`：

```
let five = ArithmeticExpression.number(5)
let four = ArithmeticExpression.number(4)
let sum = ArithmeticExpression.addition(five, four)
let prc
  ArithmeticExpression.number(2))
```

递归函数是处理具有递归结构的数据的直接方式。例如，以下是一个评估算术表达式的函数：

```
func evaluate(_ expression: ArithmeticExpression) -> Int {
    switch expression {
    case let .number(value):
        return value
    case let .addition(left, right):
        return evaluate(left) + evaluate(right)
    case let .multiplication(left, right):
        return evaluate(left) * evaluate(right)
    }
}

print(evaluate(product))
// Prints "18"
```

该函数通过简单地返回关联值来评估一个普通数字。它通过评估左侧的表达式，评估右侧的表达式，然后添加它们或将它们相乘来评估加法或乘法。