The Swift Programming Language in Chinese

https://github.com/letsswift

http://letsswift.com/

June 15, 2014

目录

1 Swift 中文教程（十二）下标

类，结构和枚举类型都可以通过定义下标来访问一组或者一个序列中的成员元素。通过下标索引就可以方便地检索和设置相应的值，而不需要其他的额外操作。比如你可以通过 someArray[index] 来访问数组中的元素，或者 someDictionary[key] 来对字典进行索引。

你可以为一个类型定义多个下标，以及适当的下标重载用来根据传递给下标的索引来设置相应的值。下标不仅可以定义为一维的，还可以根据需要定义为多维的，多个参数的。

1.1 下标语法

下标可以让你通过实例名后加中括号内一个或多个数值的形式检索一个元素。语法和方法语法和属性语法类似，通过使用 subscript 关键定义，一个或多个输入参数以及一个返回值。不同于实例方法的是，下标可以是可读写的或者只读的。这种行为通过一个 getter 和 setter 语句联通，就像是计算属性一样。

```
subscript(index: Int) -> Int {
    get {
        // return an appropriate subscript value here
    }
    set(newValue) {
        // perform a suitable setting action here
    }
}
```

newValue 的类型和下标返回的类型一样。和计算属性一样，你可以选择不指定 setter 的参数，因为当你不指定的时候，默认参数 newValue 会被提供给 setter。

和计算属性一样，只读下标可以不需要 get 关键词：

```
subscript(index: Int) -> Int {
    // return an appropriate subscript value here
}
```

下面是一个只读下标的实现，定义了一个 TimesTable 结构来表示一个整数的倍数表：

```
struct TimesTable {
    let multiplier: Int
    subscript(index: Int) -> Int {
        return multiplier * index
    }
}
let threeTimesTable = TimesTable(multiplier: 3)
println("six times three is \(threeTimesTable[6])")
// prints "six times three is 18"
```

在这个例子中，实例 TimesTable 被创建为 3 倍数表，这是通过在初始化的时候为 multiplier 参数传入的数值 3 设置的。

注意:

倍数表是根据特定的数学规则设置的,所以不应该为 threeTimeTable[someIndex] 元素设置一个新值，所以 TimesTable 的下标定义为只读。

1.2 下标的使用

下标的具体含义由使用它时的上下文来确定。下标主要用来作为集合，列表和序列的元素快捷方式。你可以自由的为你的类或者结构定义你所需要的下标。

比如说，Swift 中字典类型实现的下标是设置和检索字典实例中的值。可以通过分别给出下标中的关键词和值来设置多个值，也可以通过下标来设置单个字典的值:

```
var numberOfLegs = ["spider": 8, "ant": 6, "cat": 4]
numberOfLegs["bird"] = 2
```

上面的例子中定义了一个变量 numberOfLegs，然后通过键值对初始化。numberOfLegs 的类型是字典类型 Dictionary。在字典创建之后，例子使用了下标赋值方法添加了一个类型为字符串的键”bird” 和 Int 值 2 到字典中。

更多关于字典的下标可以参考：访问和修改字典这一章节

注意:

Swift 中字典类型实现的键值对下标是可选类型。对于 numberOfLges 字典来说，返回的值是 Int?，也就是可选 Int 值。字典的这种使用可选类型下标的方式说明不是所有的键都有对应的值。同样也可以通过给键赋值 nil 来删除这个键。

1.3 下标选项

下标可以接收任意数量的参数，参数的类型也可以各异。下标还可以返回任何类型的值。下标可以使用变量参数或者可变参数，但是不能够使用输入输出参数或者提供默认参数的值。

类或者结构可以根据需要实现各种下标方式，可以在需要的时候使用合适的下标通过中括号中的参数返回需要的值。这种多下标的定义被称作下标重载。

当然，最常见的下标用法是单个参数，也可以定义多个参数的下标。下面的例子演示了一个矩阵 Matrix 结构，它含有二维的 Double 值。矩阵结构的下标包括两个整形参数:

```
struct Matrix {
    let rows: Int, columns: Int
    var grid: Double[]
    init(rows: Int, columns: Int) {
        self.rows = rows
        self.columns = columns
        grid = Array(count: rows * columns, repeatedValue: 0.0)
    }
    func indexIsValidForRow(row: Int, column: Int) -> Bool {
      return row >= 0 && row < rows && column >= 0 && column < columns
    }
    subscript(row: Int, column: Int) -> Double {
        get {
        assert(indexIsValidForRow(row, column: column), "Index out of range")
            return grid[(row * columns) + column]
        }
        set {
        assert(indexIsValidForRow(row, column: column), "Index out of range")
            grid[(row * columns) + column] = newValue
        }
    }
}
```

矩阵 Matrix 提供了一个初始化方法，使用两个参数 rows 和 columns，然后建立了一个数组来存储类型为 Double 的值 rows*columns。每个矩阵中的位置都被设置了一个初始值 0.0。通过传递初始值 0.0 和数组长度给数组初始化方法完成上述操作。数组的初始化方法在：创建和初始化数组中有更详细的叙述。

你可以传递两个参数 row 和 column 来完成 Matrix 的初始化：

```
var matrix = Matrix(rows: 2, columns: 2)
```

上面的初始化操作创建了一个两行两列的矩阵 Matrix 实例。这个矩阵实例的 grid 数组看起来是平坦的，但是实际上是矩阵从左上到右下的一维存储形式。

```
![](pic/subscriptMatrix01_2x.png "subscriptMatrix01_2x")
```

矩阵中的值可以通过使用包含 row 和 column 以及逗号的下标来设置：

```
matrix[0, 1] = 1.5
matrix[1, 0] = 3.2
```

这两个语句调用了下标的 setter 方法为右上和左下角的两个元素分别赋值 1.5 和 3.2

$$\begin{bmatrix} 0.0 & 1.5 \\ 3.2 & 0.0 \end{bmatrix}$$

矩阵下标的 getter 和 setter 方法都包括了一个断言语句来检查下标 row 和 column 是否有效。通过 indexIsValid 方法来判断 row 和 column 是否在矩阵的范围内：

```
func indexIsValidForRow(row: Int, column: Int) -> Bool {
    return row >= 0 && row < rows && column >= 0 && column < columns
}
```

如果访问的矩阵越界的时候，断言就会被触发：

```
let someValue = matrix[2, 2]
// this triggers an assert, because [2, 2] is outside of the matrix bounds
```

参考文献