下标

4.2

这是一篇社区协同翻译的文章，你可以点击右边区块信息里的『改进』按钮向译者提交改进建议。

类、结构体和枚举可以定义 *下标*，它是用于访问集合，列表或序列的成员元素的快捷方式。你可以直接使用下标来对值进行读写，而无需设置单独的读写方法。例如，你可以将 `Array` 实例中的元素作为 `someArray [index]` 访问，将 `Dictionary` 实例中的元素作为 `someDictionary [key]` 访问。

你可以为单个类型定义多个下标，并根据传递给下标的索引值的类型，选择要使用的相应下标进行重载。下标不限于单个维度，你可以定义具有多个输入参数的下标以满足你自定义类型的需求。

下标语法

使用下标, 让你可以通过在实例名称后面的方括号中写入一个或多个值来查询类的实例。它们的语法类似于实例方法和计算属性语法。使用 `subscript` 关键字定义下标，并且和实例方法类似，可以指定一个或多个输入参数和返回类型。与实例方法不同，下标可以是读写或只读。和计算属性类似, 读写是由 getter 和 setter 方法实现的:

```
subscript(index: Int) -> Int {
    get {
        // 在这里返回一个对应下标的值
    }
    set(newValue) {
        // 在这里执行对应的赋值操作
    }
}
```

`newValue` 的类型与下标的返回值相同。与计算属性一样，你可以选择不指定 setter 的 `(newValue)` 参数。但是如果你没有指定参数，那么 `newValue` 会成为你 setter 的默

认参数。

与只读计算属性一样，你可以通过删除 `get` 关键字以及大括号来简化只读下标的声明：

```
subscript(index: Int) -> Int {
    // 在这里返回一个对应下标的值
}
```

这是一个只读下标实现的例子，它定义了一个 `TimesTable` 结构体来表示一个 *n* 倍整数表：

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

在这个例子中，创建了一个新的 `TimesTable` 实例来表示三倍表。通过将值 `3` 作为实例的 `multiplier` 参数值来初始化结构体。

你能够通过下标来调用 `threeTimesTable` 实例，比如调用 `threeTimesTable[6]` 。这请求了三倍表中的第六项，返回 `18` ，也就是 `6` 的 `3` 倍。

> 注意
>
> *n* 倍表是基于固定的数学规则定义的。给 `threeTimesTable[someIndex]` 赋新值是不合适的，因此 `TimesTable` 的下标倍定义为只读下标。

下标的用法

下标的准确含义依赖于使用它的上下文。下标通常被用来作为访问集合、列表或序列中的元素的快捷方式。你可以用最合适的方式来为你的特定类或结构体函数实现下标。

例如，Swift 中的 `Dictionary` 类型实现了一个下标来设置和检索存储在 `Dictionary` 实例中的值。你可以通过在下标括号中提供一个符合字典键类型的键来设置


wzshare 翻译于 5个月
0 重译
由 RecherJ 审阅

字典的值，并且把符合字典值类型的值赋给下标：

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

上面的例子定义了一个名为 `numberOfLegs` 的变量并使用含有三个键值对的字典初始化它。字典 `numberOfLegs` 的类型被推断为 `[String: Int]` 。创建字典后，通过下标将一个 `String` 类型的键 `「bird」` 和一个 `Int` 类型的值 `2` 添加到了字典。

关于 `Dictionary` 下标的详细信息可以参考 访问与修改字典。

> 注意
>
> Swift 的 `Dictionary` 类型将其键值作为下标实现并返回一个*可选*类型。对于上面的 `numberOfLegs` 字典，键值下标存取的值类型为 `Int?` ，表示「可选的整数」。 `Dictionary` 类型用可选的下标类型来模拟不是每个键都有值的事实，可以通过为该键指定一个 `nil` 值来删除该键值。

下标选项

下标可以设置任意数量的输入参数，这些输入参数也可以是任意类型。 同时，下标也可以返回任何类型。 下标可以使用可变参数，但它们不能使用输入输出参数或是提供默认参数值。

类或结构体可以提供尽可能多的下标实现，并且能够基于当前下标括号内的值或者值的类型来推断合适的下标。定义多个下标被称为 *下标重载*。

虽然下标采用单参数是最常见的，但在合适的情况下也可以定义带多个参数的下标。下面的例子定义了一个 `Matrix` 结构体，这个结构体表示一个类型为 `Double` 的二维矩阵。 `Matrix` 结构体的下标有两个整型参数：

```
struct Matrix {
    let rows: Int, columns: Int
    var grid: [Double]
    init(rows: Int, columns: Int) {
        self.rows = rows
        self.columns = columns
        grid = Array(repeating: 0.0, count: rows * columns)
    }
    func indexIsValid(row: Int, column: Int) -> Bool {
        return row >= 0 && row < rows && column >= 0 && column < columns
    }
```

wzshare 翻译于 5个月

0 重译

由 RecherJ 审阅

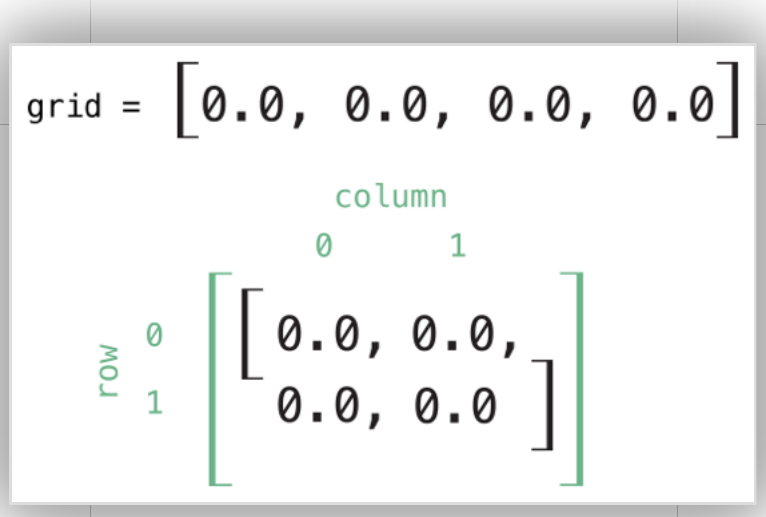
```
    subscript(row: Int, column: Int) -> Double {
        get {
            assert(indexIsValid(row: row, column: column), "Index out of range
            return grid[(row * columns) + column]
        }
        set {
            assert(indexIsValid(row: row, column: column), "Index out of range
            grid[(row * columns) + column] = newValue
        }
    }
}
```

`Matrix` 提供了一个带有两个名为 `rows` 和 `columns` 参数的构造器，并构造一个足够大的数组来存储 `Double` 类型的 `rows * columns` 值。矩阵中的每个值初始化为 `0.0` 。为实现此目的，把数组的大小和初始单元格值 `0.0` 传递给数组构造器，该构造器会构造并初始化一个正确大小的新数组。更多关于构造器的描述参考文档 用默认值创建数组 。

你可以通过传递合适的行数和列数给构造器来构造新的 `Matrix` 实例：

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

上面的示例创建了一个新的带有两行两列的 `Matrix` 实例。 `Matrix` 实例的 `grid` 数组实际上是矩阵从左上角到右下角的展开：



可以通过传递行和列的值到下标来给矩阵赋值，行和列用逗号隔开：

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

这两行代码调用下标的 setter 方法给矩阵的右上角（ `row` 为 `0` 且 `column` 为 `1` ）赋值 `1.5` 和左下角

（ `row` 为 `1` 且 `column` 为 `0` ）赋值 `3.2` 。

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

`Matrix` 下标的 getter 方法和 setter 方法都包含一个断言来检查下标的 `row` 和 `column` 是否有效。为了辅助这些断言， `Matrix` 包含一个名为 `indexIsValid(row:column:)` 的便捷方法，该方法检查请求的 `row` 和 `column` 是否超出矩阵的边界：

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

如果尝试访问矩阵边距之外的下标就会触发断言：

```
let someValue = matrix[2, 2]
// 这里触发了断言，因为下标 [2, 2] 超出了矩阵的边界
```


上一篇

下一篇

更多职位

讨论数量: 0

发起讨论

只看当前版本讨论

暂无话题~