翻译：siemenliu　校对：zq54zquan

附属脚本（Subscripts）

本页包含内容：

附属脚本 可以定义在类（Class）、结构体（structure）和枚举（enumeration）这些目标中，可以认为是访问对象、集合或序列的快捷方式，不需要再调用实例的特定的赋值和访问方法。举例来说，用附属脚本访问一个数组(Array)实例中的元素可以这样写 `someArray[index]`，访问字典(Dictionary)实例中的元素可以这样写 `someDictionary[key]`。

对于同一个目标可以定义多个附属脚本，通过索引值类型的不同来进行重载，而且索引值的个数可以是多个。

译者：这里附属脚本重载在本小节中原文并没有任何演示

附属脚本语法

附属脚本允许你通过在实例后面的方括号中传入一个或者多个的索引值来对实例进行访问和赋值。语法类似于实例方法和计算型属性的混合。与定义实例方法类似，定义附属脚本使用`subscript`关键字，显式声明入参（一个或多个）和返回类型。与实例方法不同的是附属脚本可以设定为读写或只读。这种方式又有点像计算型属性的getter和setter：

```
subscript(index: Int) -> Int {
    get {
        // 返回与入参匹配的Int类型的值
    }
```

```
    set(newValue) {
        // 执行赋值操作
    }
}
```
newValue的类型必须和附属脚本定义的返回类型相同。与计算型属性相同的是set的入参声明newValue就算不写，在set代码块中依然可以使用默认的newValue这个变量来访问新赋的值。

与只读计算型属性一样，可以直接将原本应该写在get代码块中的代码写在subscript中：

```
subscript(index: Int) -> Int {
    // 返回与入参匹配的Int类型的值
}
```
下面代码演示了一个在TimesTable结构体中使用只读附属脚本的用法，该结构体用来展示传入整数的*n*倍。

```
struct TimesTable {
    let multiplier: Int
    subscript(index: Int) -> Int {
        return multiplier * index
    }
}
let threeTimesTable = TimesTable(multiplier: 3)
println("3的6倍是\(threeTimesTable[6])")
// 输出 "3的6倍是18"
```
在上例中，通过TimesTable结构体创建了一个用来表示索引值三倍的实例。数值3作为结构体构造函数入参初始化实例成员multiplier。

你可以通过附属脚本来来得到结果，比如threeTimesTable[6]。这句话访问了threeTimesTable的第六个元素，返回18或者6的3倍。

注意：

TimesTable例子是基于一个固定的数学公式。它并不适合开放写权限来对threeTimesTable[someIndex]进行赋值操作，这也是为什么附属脚本只定义为只读的原因。

附属脚本用法

根据使用场景不同附属脚本也具有不同的含义。通常附属脚本是用来访问集合（collection），列表（list）或序列（sequence）中元素的快捷方式。你可以在你自己特定的类或结构体中自由的实现附属脚本来提供合适的功能。

例如，Swift 的字典（Dictionary）实现了通过附属脚本来对其实例中存放的值进行存取操作。在附属脚本中使用和字典索引相同类型的值，并且把一个字典值类型的值赋值给这个附属脚本来为字典设值：

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

上例定义一个名为`numberOfLegs`的变量并用一个字典字面量初始化出了包含三对键值的字典实例。`numberOfLegs`的字典存放值类型推断为`Dictionary<String, Int>`。字典实例创建完成之后通过附属脚本的方式将整型值`2`赋值到字典实例的索引为`bird`的位置中。

更多关于字典（Dictionary）附属脚本的信息请参考读取和修改字典

注意：

Swift 中字典的附属脚本实现中，在`get`部分返回值是`Int?`，上例中的`numberOfLegs`字典通过下边返回的是一个`Int?`或者说“可选的int”，不是每个字典的索引都能得到一个整型值，对于没有设过值的索引的访问返回的结果就是`nil`；同样想要从字典实例中删除某个索引下的值也只需要给这个索引赋值为`nil`即可。

附属脚本选项

附属脚本允许任意数量的入参索引，并且每个入参类型也没有限制。附属脚本的返回值也可以是任何类型。附属脚本可以使用变量参数和可变参

数，但使用写入读出（in-out）参数或给参数设置默认值都是不允许的。

一个类或结构体可以根据自身需要提供多个附属脚本实现，在定义附属脚本时通过入参个类型进行区分，使用附属脚本时会自动匹配合适的附属脚本实现运行，这就是附属脚本的重载。

一个附属脚本入参是最常见的情况，但只要有合适的场景也可以定义多个附属脚本入参。如下例定义了一个`Matrix`结构体，将呈现一个`Double`类型的二维矩阵。`Matrix`结构体的附属脚本需要两个整型参数：

```
struct Matrix {
    let rows: Int, columns: Int
    var grid: Double[]
    init(rows: Int, columns: Int) {
        self.rows = rows
        self.columns = columns
        grid = Array(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) + columns] =
newValue
        }
    }
```

```
}
```
Matrix提供了一个两个入参的构造方法，入参分别是rows和columns，创建了一个足够容纳rows * columns个数的Double类型数组。为了存储，将数组的大小和数组每个元素初始值0.0，都传入数组的构造方法中来创建一个正确大小的新数组。关于数组的构造方法和析构方法请参考创建并且构造一个数组。

你可以通过传入合适的row和column的数量来构造一个新的Matrix实例：

```
var matrix = Matrix(rows: 2, columns: 2)
```
上例中创建了一个新的两行两列的Matrix实例。在阅读顺序从左上到右下的Matrix实例中的数组实例grid是矩阵二维数组的扁平化存储：

```
// 示意图
grid = [0.0, 0.0, 0.0, 0.0]

        col0     col1
row0   [0.0,     0.0,
row1    0.0,     0.0]
```
将值赋给带有row和column附属脚本的matrix实例表达式可以完成赋值操作，附属脚本入参使用逗号分割

```
matrix[0, 1] = 1.5
matrix[1, 0] = 3.2
```
上面两条语句分别让matrix的右上值为 1.5，坐下值为 3.2：

```
[0.0, 1.5,
 3.2, 0.0]
```
Matrix附属脚本的getter和setter中同时调用了附属脚本入参的row和column是否有效的判断。为了方便进行断言，Matrix包含了一个名为indexIsValid的成员方法，用来确认入参的row或column值是否会造成数组越界：

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

```
column < columns
}
```
断言在附属脚本越界时触发:

```
let someValue = matrix[2, 2]
// 断言将会触发，因为 [2, 2] 已经超过了matrix的最大长度
```