The Swift Programming Language in Chinese

https://github.com/letsswift

http://letsswift.com/

June 15, 2014

目录

1 Swift 中文教程（四）集合类型

Swift 提供两种集合类型来存储集合，数组和字典。数组是一个同类型的序列化列表集合。字典是一个能够使用类似于键的唯一标识符来获取值的非序列化集合。

在 Swift 中，数组和字典里的键和值都必须是明确的某个特定类型。这意味这数组和字典不会插入一个错误的类型的值, 以致于出错。这也意味着当你在数组和字典中取回数值的时候能够确定它的类型。

Swift 使用确定的集合类型可以保证代码工作是不会出错，和让你在开发阶段就能更早的捕获错误。

> 注意：Swift 的数组储存不同的类型会展示出不同的行为，例如变量，常量或者函数和方法。更多的信息参见 Mutability of Collections and Assignment 和 Copy Behavior for Collection Types.

1.1 数组

数组是储存同类型多数值的序列化列表。同样的值可以在数组的不同位置出现多次。

Swift 数组只能存储特定的某一类值，这和 Objective-C 中的 NSArray 和 NSMutableArray 类是有区别的。因为它们是储存各种的对象，而且并不提供返回任何有关对象的具体信息。在 Swift 中，无论是确定的声明，还是隐式的声明，数组是非常确定它自身是储存什么样的类型，而且，它并不一定要求储存的是类对象。如果你创建一个存储 int 值的数组，那么这个数组中只能出现 int 类型的值。Swift 数组是类型安全的，而且它一直都清楚它自身所能包含的值的类型。

1.1.1 数组的简略语法

定义数组的完整写法是 Array，其中 SomeType 是你想要包含的类型。你也可以使用类似于 SomeType[] 这样的简略语法。虽然这两种方法在功能上是相同的。但是我们更推荐后者，而且它会一直贯穿于本书。

1.1.2 数组实量（Array Literals）

你可以用一个数组实量（Array Literals）来初始化一个数组，它是用简略写法来创建一个包含一个或多个的值的数组。一个数组实量（Array Literals）是由它包含的值，“,” 分隔符已经包括以上内容的中括号对 “[]” 组成：

```
[value 1, value 2, value 3]
```

下面的例子是创建一个叫 shoppinglist，储存字符串（String）类型的数组。

```
var shoppingList: String[] = ["Eggs", "Milk"]
// □ □ □ □ □ □ □ □ □ □ □ □ □ □ shoppingList
```

shoppinglist 变量被定义为字符串（String）类型的数组，写作 String[]。因为这个数组被确定为字符串类型（String），所以它只能储存字符串（String）类型的值。在这里，我们用两个字符串类型的值（”Eggs” and “Milk”）和数组实量（Array Literals）的写法来初始化 shoppingList 数组。

> 注意 shoppingList 数组是被定义为一个变量（使用 var 标识符）而不是常量（使用 let 标识符），所以在下面的例子可以直接添加元素。

在这个例子中，数组实量（Array Literals）只包含两个字符串类型的值，这符合了 shoppingList 变量的定义（只能包含字符串（String）类型的数组），所以被分配的数组实量（Array Literals）被允许用两个字符串类型的值来初始化。

得益于 Swift 的类型推断，当你用相同类型的值来初始化时，你可以不写明类型。初始化 shoppingList 可以用下面这个方法来代替。

```
var shoppingList = ["Eggs", “ Milk"]
```

因为数组实量（Array Literals）中所有的值都是同类型的，所以 Swift 能够推断 shoppingList 的类型为字符串数组（String[]）。

1.1.3 读取和修改数组

你可以通过方法和属性，或者下标来读取和修改数组。

通过只读属性 count 来读取数组的项数；

```
println("The shopping list contains \(shoppingList.count) items.")
// □ □ □ "The shopping list contains 2 items. ”
```

通过一个返回布尔类型的 isEmpty 属性检查数组的项数是否为 0

```
if shoppingList.isEmpty {
    println("The shopping list is empty.")
} else {
    println("The shopping list is not empty.")
}
// □ □ □ "The shopping list is not empty."
```

在数组末尾增加一项可以通过 append 方法

```
shoppingList.append("Flour")
// shoppingList □ □ □ □3□
```

同理，也可以用（+=）操作符来把一个元素添加到数组末尾

```
shoppingList += "Baking Powder"
// shoppingList □ □ □ □4□
```

你也可以用（+=）操作符来把一个数组添加到另一个数组的末尾

```
shoppingList += ["Chocolate Spread", "Cheese", "Butter"]
// shoppingList □ □ □ □7□ □ □
```

从数组中取出一个值可以使用下标语法。如果你知道一个元素的索引值，你可以数组名后面的中括号中填写索引值来获取这个元素

```
var firstItem = shoppingList[0]
// firstItem □ □ “Eggs"
```

注意，数组的第一个元素的索引值为 0，不为 1，Swift 的数组的索引总是从 0 开始;

你可以使用下标语法通过索引修改已经存在的值。

```
shoppingList[0] = "Six eggs"
//□ □ □ □ □ □ □ □ □ □"Six eggs" □ □ □ □ “Eggs"
```

你可以使用下标语法一次性改变一系列的值，尽管修改的区域远远大于要修改的值。在下面的雷子中，把 “Chocolate Spread”, “Cheese” 和”Butter” 替换为”Bananas” 和”Apples”:

```
shoppingList[4...6] = ["Bananas", "Apples"]
// shoppingList □ □ □ □6□ □ □
```

> 注意，你不能使用下标语法在数组中添加一个元素，如果你尝试使用下标语法来获取或者设置一个元素，你将得到一个运行时的错误。尽管如此，你可以通过 count 属性验证索引是否正确再使用它。除非 count 等于 0（也就是说数组是空的），最大的索引都是 count-1，因为数组的索引从 0 开始计算。

在一个特定的索引位置插入一个值，可以使用 insert(atIndex:) 方法

```
shoppingList.insert("Maple Syrup", atIndex: 0)
// shoppingList □ □ □ □7□ □ □
// "Maple Syrup" □ □ □ □ □ □ □
```

这里调用 insert 方法指明在 shoppingList 的索引为 0 的位置中插入一个新元素 “Maple Syrup”

同理，你可以调用 removeAtIndex 方法移除特定的元素。这个方法移除特定索引位置的元素并返回这个被移除的元素（尽管你可能并不关心这个返回值）。

```
let mapleSyrup = shoppingList.removeAtIndex(0)
// □ □ □ □ □0□ □ □ □ □ □
// shoppingList □ □ □ □6□ □ □ , □ □ □ Maple Syrup
// mapleSyrup □ □ □ □ □ □ □ □ "Maple Syrup" □ □ □
```

当元素被移除的，数组空缺的位置将会被填补，所以现在索引位置为 0 的元素再一次等于”Six eggs”:

```
firstItem = shoppingList[0]
// firstItem □ □ □ □ "Six eggs ”
```

如果你想从数组中移除最后一个元素，使用 removeLast 方法比 removeAtIndex 更方便，因为后者需要通过 count 属性计算数组的长度。和 removeAtIndex 方法一样，removeLast 会返回被移除的元素。

```
let apples = shoppingList.removeLast()
//□ □ □ □ □ □ □ □ □ □ □ □
// shoppingList □ □ □ □5□ □ □ , □ □ □ cheese
// apples □ □ □ □ □ □ □ □ □ "Apples" string
```

1.1.4 遍历数组

可以使用 for − in 循环来遍历数组中的值

```
for item in shoppingList {
    println(item)
}
// Six eggs
// Milk
// Flour
// Baking Powder
// Bananas
```

如果既需要每个元素的值，又需要每个元素的索引值，使用 enumerate 函数代替会更方便，enumerate 函数对于每一个元素都会返回一个包含元素的索引和值的元组（tuple)。你可以在遍历部分分解元素并储存在临时变量或者常量中。

```
for (index, value) in enumerate(shoppingList) {
    println("Item \(index + 1): \(value)")
}
// □ □ 1: Six eggs
// □ □ 2: Milk
// □ □ 3: Flour
// □ □ 4: Baking Powder
// □ □ 5: Bananas
```

如需更多 for-in 循环信息, 参见 For Loops.

1.1.5 创建和初始化数组

创建一个空的数组和确定的类型（不包含初始化值）使用的初始化语法:

```
var someInts = Int[]()
println("someInts is of type Int[] with \(someInts.count) items.")
// □ □ "someInts is of type Int[] with 0 items. ”
```

注意，someInt 的变量类型被确定为 Int[]，因为它使用生成 Int[] 的初始化方法。

或者，如果上下文（context）已经提供类型信息，例如函数参数或者已经确定类型的常量和变量，你可以从空的数组实量（Array Literals）创建一个空数组，写作 []（空的中括号对）。

```
someInts.append(3)
// someInts □ □ □ □1□ Int□ □ □ □
someInts = []
// someInts □ □ □ □ □ □ □ □ □, □ □ □ □ □ □ □Int[];
```

Swift 数组类型也提供初始化方法来创建确定长度和提供默认数值的数组。你可以通过这个初始化方法增加一个新的数组，元素的数量成为 count，合适的默认值为 repeatedValue

```
var threeDoubles = Double[](count: 3, repeatedValue: 0.0)
// threeDoubles □ □ □ □ Double[], □ □ □ [0.0, 0.0, 0.0]
```

得益于类型推断，你并不需要指明这个数组储存的类型就能使用这个初始化方法，因为它从默认值中就能推断出来。

```
var anotherThreeDoubles = Array(count: 3, repeatedValue: 2.5)
// anotherThreeDoubles □ □ □ □ Double[], □ □ □ □ [2.5, 2.5, 2.5]
```

最后，你可以使用（+）操作符就能创建一个新的数组，把两个存在的数组添加进来这个新的数组类型从你添加的两个数组中推断出来

```
var sixDoubles = threeDoubles + anotherThreeDoubles
// sixDoubles □ □ □ □ Double[], □ □ □ [0.0, 0.0, 0.0, 2.5, 2.5, 2.5]
```

1.2 字典

字典是储存同一类型多个值的容器。每一个值都对应这一个唯一的键（Key），就像是字典内的每一个值都有一个标识符。和数组内的元素是有区别的，字典内的元素是没有特殊的序列的。当你需要根据标识符来查找批量的值时，就可以使用字典，和在真实世界的字典中寻找某个字的解释相似。

Swift 字典储存一个特定类型的键和值，与 Objective-C 的 NSDictionary 和 NSMutableDictionary 不同，因为它们是使用各种的对象来作为它们的键和值，而且并不提供任何有关对象的具体信息。在 Swift 中，对于一个特定的字典，它

所能储存的键和值的类型都是确定的，无论是明确声明的类型还是隐式推断的类型。

Swift 的字典写法是 Dictionary,KeyType 是你想要储存的键的类型，Value-Type 是你想要储存的值的类型。

唯一的限制就是 KeyType 必须是可哈希的（hashable）——就是提供一个形式让它们自身是独立识别的。Swift 的所有基础类型（例如字符串（String），整形（Int），双精度（Double）和布尔（Bool））在默认是可哈希的（hashable），和这些类型都常常被当作字典的键。没有协助值（associated values）的枚举成员(具体描述在 Enumerations）默认也是可哈希的（hashable)。

1.2.1 字典实量（Dictionary Literals）

你可以直接用一个字典实量（Dictionary Literals）初始化一个字典。和前面定义一个数组实量（Array Literals）的语法一样。字典实量（Dictionary Literals）就是使用简略写法直接写一个或者多个对应的键和值对来定义一个字典。

一个键值对是一个键和值的组合。在字典实量（Dictionary Literals）里面，每一个键值对总是用一个冒号把键和值分割。键值对的写法就想是一个列表，使用逗号分割，并被一对中括号 [] 包含着:

```
[key 1: value 1, key 2: value 2, key 3: value 3]
```

在下面的例子，将会创建一个字典来储存国际机场的名字。在这个字典里面，键是三个字的国际航空运送协会代码，以及它的值是机场的名称:

```
var airport :Dictionary<String, String> = ["TYO": "Tokyo", "DUB": " Dublin"]
```

airport 字典被定义为一个类型为 Dictionary，这意味这，这个字典的键类型是字符串 String，和它的值的类型也是 String。

> 注意 airport 字典是被定义为一个变量（使用 var 标识符）而不是常量（使用 let 标识符），所以在下面的例子可以直接添加元素。

airport 字典使用一个包含两个键值对的字典实量（Dictionary Literals）来初始化。第一对有一个叫 “TYO” 的键和一个叫 “Tokyo” 的值，第二对有一个叫 “DUB” 的键和一个叫 “Dublin” 的值。

这个字典实量（Dictionary Literals）包含两个字符串（String）: 字符串对。这符合 airport 变量定义的类型（一个字典只包括字符串（String）键和字符串

(String) 值)，所以在分配字典实量（Dictionary Literals）的时候被允许作为 airport 字典的两个初始化元素。

和数组一样，如果你初始化一个字典的时候使用相同的类型，你可以不指明字典的类型。airport 初始化可以用下面这个简略写法来代替：

```
var airports = ["TYO": "Tokyo", "DUB": “ Dublin ”]
```

因为所有的键在字面上都是相同的类型，同样，所有的值也是同样的类型，所以 Swift 可以推断为 Dictionary 是 airports 字典的正确类型。

1.2.2 读取和修改字典

你可以通过属性，方法或者下标来读取和修改字典。和数组一样，你使用只读的 count 属性来检查字典（Dictionary）包含多少个元素。

```
 println("The dictionary of airports contains \(airports.count) items.")
// □ □ "The dictionary of airports contains 2 items."
```

你可以使用下标语法给一个字典添加一个元素。使用合适类型作为新的键，并分配给它一个合适类型的值

```
 airports["LHR"] = "London"
//  airports dictionary □ □ □ 3 items
```

你也可以使用下标语法去改变一个特定键所关联的值。

```
airports["LHR"] = "London Heathrow"
//"LHR" □ □ □ □ □ □ □ □ "London Heathrow"
```

同样, 使用字典的 updateValue(forKey:) 方法去设置或者更新一个特定键的值. 和上面的下标例子一样, updateValue(forKey:) 方法如果键不存在则会设置它的值，如果键存在则会更新它的值, 和下标不一样是, updateValue(forKey:) 方法如果更新时，会返回原来旧的值，意味着你可以使用这个来判断数据是否发生了更新。

updateValue(forKey:) 方法返回一个和字典的值相同类型的可选值. 例如，如果字典的值的类型时 String，则会返回 String? 或者叫 “可选 String“，这个可选值包含一个如果值发生更新的旧值和如果值不存在的 nil 值。

```
if let oldValue = airports.updateValue("Dublin International", forKey: "DUB") {
    println("The old value for DUB was \(oldValue).")
}
// prints "The old value for DUB was Dublin."
```

你也可以使用下标语法通过特定的键去读取一个值。因为如果他的值不存在的时候，字典的下标语法会返回一个字典的值的类型的可选值。如果字典中的键包含对应的值，这字典下标语法会返回这个键所对应的值，否则返回 nil

```
 if let airportName = airports["DUB"] {
     println("The name of the airport is \(airportName).")
 } else {
     println("That airport is not in the airports dictionary.")
 }
// prints "The name of the airport is Dublin International."
```

你可以使用下标语法把他的值分配为 nil，来移除这个键值对。

```
airports["APL"] = "Apple International"
// "Apple International" □ □APL□ □ □ □ □,□ □ □ □ □
airports["APL"] = nil
// APL□ □ □ □ □ □ □ □ □
```

同样，从一个字典中移除一个键值对可以使用 removeValueForKey 方法，这个方法如果存在键所对应的值，则移除一个键值对，并返回被移除的值，否则返回 nil。

```
if let removedValue = airports.removeValueForKey("DUB") {
    println("The removed airport's name is \(removedValue).")
} else {
   println("The airports dictionary does not contain a value for DUB.")
}
// prints "The removed airport's name is Dublin International."
```

1.2.3 遍历字典

你可以使用一个 for-in 循环来遍历字典的键值对。字典中的每一个元素都会返回一个元祖（tuple），你可以在循环部分分解这个元祖，并用临时变量或者常量来储存它。

```
for (airportCode, airportName) in airports {
    println("\(airportCode): \(airportName)")
}
// TYO: Tokyo
// LHR: London Heathrow
```

更多有关 for-in 循环的信息, 参见 For Loops.

你也可以读取字典的 keys 属性或者 values 属性来遍历这个字典的键或值的集合。

```
for airportCode in airports.keys {
    println("Airport code: \(airportCode)")
}
// Airport code: TYO
// Airport code: LHR
for airportName in airports.values {
    println("Airport name: \(airportName)")
}
// Airport name: Tokyo
// Airport name: London Heathrow
```

如果你需要一个接口来创建一个字典的键或者值的数组实例，你可以使用 keys 或者 values 属性来初始化一个数组。

```
let airportCodes = Array(airports.keys)
// airportCodes is ["TYO", "LHR"]
let airportNames = Array(airports.values)
// airportNames is ["Tokyo", "London Heathrow"]
```

> 注意 Swift 中的字典类型是非序列化集合，序列化取回键，值，或者键值对的顺序是不明确的。

1.2.4 创建一个空字典

和字典一样，你可以使用确定类型的语法创建一个空的字典。

```
var namesOfIntegers = Dictionary<Int, String>()
// namesOfIntegers □ □ □ □ □ Dictionary<Int, String> □ □ □ □ □
```

这个例子创建一个 Int，String 类型的字典来储存可读性较好的整数值。它的键是 Int 类型，它的值是 String 类型。

如果上下文（context）中已经提供类型信息，可用一个字典实量（Dictionary Literal）创建一个空的字典，写作 [:]（由一对 □ 包含一个冒号:）

```
namesOfIntegers[16] = "sixteen"
// namesOfIntegers □ □ □ □1 □ □ □ □
namesOfIntegers = [:]
// namesOfIntegers □ □ □ □ □ □Int, String□ □ □ □ □
```

> 注意在这个场景，Swift 数组和字典类型是一个内置的集合。更多的内置类型和集合参见 Generics

1.3 可变集合类型

数组和字典都是在一个集合中一起储存多个变量. 如果你创建一个数组或者字典，再包含一个变量，创建的这个变量被称为可变的（mutable）这意味这，你可以在创建之后增加更多的元素来改变这个集合的长度，或者移除已经包含的。相反的, 如果你把一个数组或者字典定义为常量，则这个数组或者字典不是可变的，他们包含的项数并不能被改变。

在字典中，不可变也意味着你不能替换已经存在的键的值。一个不可变字典，一旦被设置就不能改变。

数组的不可变有一点点的不同。然而，你仍然不能做任何改变项数的操作。但是你可以重新设置一个已经存在的索引，这使得当 Swift 的数组的长度确定时，能更好地优化数组的性能。

拥有可变行为的数组也影响着数组实例的分配和修改，更多内容参见 Assignment and Copy Behavior for Collection Types.

> 注意在一个集合的项数不需要被改变时，创建不可变集合是非常好的尝试。这样的话 Swift 编译器就能充分利用你所创造的集合的性能。

参考文献