集合类型 (Collection Types)

本页包含内容:

Swift 语言提供经典的数组和字典两种集合类型来存储集合数据。数组用来按顺序存储相同类型的数据。字典虽然无序存储相同类型数据值但是需要由独有的标识符引用和寻址（就是键值对）。

Swift 语言里的数组和字典中存储的数据值类型必须明确。 这意味着我们不能把不正确的数据类型插入其中。 同时这也说明我们完全可以对获取出的值类型非常自信。 Swift 对显式类型集合的使用确保了我们的代码对工作所需要的类型非常清楚，也让我们在开发中可以早早地找到任何的类型不匹配错误。

注意:

Swift 的数组结构在被声明成常量和变量或者被传入函数与方法中时会相对于其他类型展现出不同的特性。 获取更多信息请参见集合的可变性与集合在赋值和复制中的行为章节。

数组

数组使用有序列表存储相同类型的多重数据。相同的值可以多次出现在一个数组的不同位置中。

Swift 数组对存储数据有具体要求。 不同于 Objective-C 的`NSArray`和`NSMutableArray`类，他们可以存储任何类型的实例而且不提供他们返回对象的任何本质信息。 在 Swift 中，数据值在被存储进入某个数组之前类型必须明确，方法是通过显式的类型标注或类型推断，而且不是必须是

`class`类型。例如： 如果我们创建了一个`Int`值类型的数组，我们不能往其中插入任何不是`Int`类型的数据。 Swift 中的数组是类型安全的，并且它们中包含的类型必须明确。

数组的简单语法

写 Swift 数组应该遵循像`Array<SomeType>`这样的形式，其中`SomeType`是这个数组中唯一允许存在的数据类型。 我们也可以使用像`SomeType[]`这样的简单语法。 尽管两种形式在功能上是一样的，但是推荐较短的那种，而且在本文中都会使用这种形式来使用数组。

数组构造语句

我们可以使用字面语句来进行数组构造，这是一种用一个或者多个数值构造数组的简单方法。字面语句是一系列由逗号分割并由方括号包含的数值。`[value 1, value 2, value 3]`。

下面这个例子创建了一个叫做`shoppingList`并且存储字符串的数组：

```
var shoppingList: String[] = ["Eggs", "Milk"]
// shoppingList 已经被构造并且拥有两个初始项。
```

`shoppingList`变量被声明为“字符串值类型的数组“，记作`String[]`。因为这个数组被规定只有`String`一种数据结构，所以只有`String`类型可以在其中被存取。 在这里，`shoppinglist`数组由两个`String`值（`"Eggs"` 和`"Milk"`）构造，并且由字面语句定义。

注意：

`Shoppinglist`数组被声明为变量（`var`关键字创建）而不是常量（`let`创建）是因为以后可能会有更多的数据项被插入其中。
在这个例子中，字面语句仅仅包含两个`String`值。匹配了该数组的变量声明（只能包含`String`的数组），所以这个字面语句的分配过程就是允许用两个初始项来构造`shoppinglist`。

由于 Swift 的类型推断机制，当我们用字面语句构造只拥有相同类型值数组的时候，我们不必把数组的类型定义清楚。 `shoppinglist`的构造也可以这样写：

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

因为所有字面语句中的值都是相同的类型，Swift 可以推断出`String[]`是`shoppinglist`中变量的正确类型。

访问和修改数组

我们可以通过数组的方法和属性来访问和修改数组，或者下标语法。 还可以使用数组的只读属性`count`来获取数组中的数据项数量。

```
println("The shopping list contains \
(shoppingList.count) items.")
// 输出"The shopping list contains 2 items."（这个数组有2个项）
```

使用布尔项`isEmpty`来作为检查`count`属性的值是否为 0 的捷径。

```
if shoppingList.isEmpty {
    println("The shopping list is empty.")
} else {
    println("The shopping list is not empty.")
}
// 打印 "The shopping list is not
empty."（shoppinglist不是空的）
```

也可以使用`append`方法在数组后面添加新的数据项：

```
shoppingList.append("Flour")
// shoppingList 现在有3个数据项，有人在摊煎饼
```

除此之外，使用加法赋值运算符（`+=`）也可以直接在数组后面添加数据项：

```
shoppingList += "Baking Powder"
// shoppingList 现在有四项了
```

我们也可以使用加法赋值运算符（`+=`）直接添加拥有相同类型数据的数

组。

```
shoppingList += ["Chocolate Spread", "Cheese",
"Butter"]
// shoppingList 现在有7项了
```

可以直接使用下标语法来获取数组中的数据项，把我们需要的数据项的索引值放在直接放在数组名称的方括号中：

```
var firstItem = shoppingList[0]
// 第一项是 "Eggs"
```

注意第一项在数组中的索引值是`0`而不是`1`。 Swift 中的数组索引总是从零开始。

我们也可以用下标来改变某个已有索引值对应的数据值：

```
shoppingList[0] = "Six eggs"
// 其中的第一项现在是 "Six eggs" 而不是 "Eggs"
```

还可以利用下标来一次改变一系列数据值，即使新数据和原有数据的数量是不一样的。下面的例子把`"Chocolate Spread"`，`"Cheese"`，和`"Butter"`替换为`"Bananas"`和`"Apples"`：

```
shoppingList[4...6] = ["Bananas", "Apples"]
// shoppingList 现在有六项
```

注意：

> 我们不能使用下标语法在数组尾部添加新项。如果我们试着用这种方法对索引越界的数据进行检索或者设置新值的操作，我们会引发一个运行期错误。我们可以使用索引值和数组的`count`属性进行比较来在使用某个索引之前先检验是否有效。除了当`count`等于 0 时（说明这是个空数组），最大索引值一直是`count - 1`，因为数组都是零起索引。

调用数组的`insert(atIndex:)`方法来在某个具体索引值之前添加数据项：

```
shoppingList.insert("Maple Syrup", atIndex: 0)
// shoppingList 现在有7项
// "Maple Syrup" 现在是这个列表中的第一项
```

这次`insert`函数调用把值为`"Maple Syrup"`的新数据项插入列表的最开

始位置，并且使用0作为索引值。

类似的我们可以使用removeAtIndex方法来移除数组中的某一项。这个方法把数组在特定索引值中存储的数据项移除并且返回这个被移除的数据项（我们不需要的时候就可以无视它）：

```
let mapleSyrup = shoppingList.removeAtIndex(0)
//索引值为0的数据项被移除
// shoppingList 现在只有6项，而且不包括Maple Syrup
// mapleSyrup常量的值等于被移除数据项的值 "Maple Syrup"
```

数据项被移除后数组中的空出项会被自动填补，所以现在索引值为0的数据项的值再次等于"Six eggs"：

```
firstItem = shoppingList[0]
// firstItem 现在等于 "Six eggs"
```

如果我们只想把数组中的最后一项移除，可以使用removeLast方法而不是removeAtIndex方法来避免我们需要获取数组的count属性。就像后者一样，前者也会返回被移除的数据项：

```
let apples = shoppingList.removeLast()
// 数组的最后一项被移除了
// shoppingList现在只有5项，不包括cheese
//  apples 常量的值现在等于"Apples" 字符串
```

数组的遍历

我们可以使用for-in循环来遍历所有数组中的数据项：

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

如果我们同时需要每个数据项的值和索引值，可以使用全局enumerate函

数来进行数组遍历。`enumerate`返回一个由每一个数据项索引值和数据值组成的键值对组。我们可以把这个键值对组分解成临时常量或者变量来进行遍历：

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

更多关于`for-in`循环的介绍请参见for 循环。

创建并且构造一个数组

我们可以使用构造语法来创建一个由特定数据类型构成的空数组：

```
var someInts = Int[]()
println("someInts is of type Int[] with \(someInts。
count) items。")
// 打印 "someInts is of type Int[] with 0
items。"（someInts是0数据项的Int[]数组）
```

注意`someInts`被设置为一个`Int[]`构造函数的输出所以它的变量类型被定义为`Int[]`。

除此之外，如果代码上下文中提供了类型信息， 例如一个函数参数或者一个已经定义好类型的常量或者变量，我们可以使用空数组语句创建一个空数组，它的写法很简单：`[]`（一对空方括号）：

```
someInts.append(3)
// someInts 现在包含一个INT值
someInts = []
// someInts 现在是空数组，但是仍然是Int[]类型的。
```

Swift 中的`Array`类型还提供一个可以创建特定大小并且所有数据都被默认的构造方法。我们可以把准备加入新数组的数据项数量（`count`）和适当

类型的初始值（repeatedValue）传入数组构造函数：

```
var threeDoubles = Double[](count: 3, repeatedValue: 0.0)
// threeDoubles 是一种 Double[]数组, 等于 [0.0, 0.0, 0.0]
```

因为类型推断的存在，我们使用这种构造方法的时候不需要特别指定数组中存储的数据类型，因为类型可以从默认值推断出来：

```
var anotherThreeDoubles = Array(count: 3, repeatedValue: 2.5)
// anotherThreeDoubles is inferred as Double[], and equals [2.5, 2.5, 2.5]
```

最后，我们可以使用加法操作符（+）来组合两种已存在的相同类型数组。新数组的数据类型会被从两个数组的数据类型中推断出来：

```
var sixDoubles = threeDoubles + anotherThreeDoubles
// sixDoubles 被推断为 Double[], 等于 [0.0, 0.0, 0.0, 2.5, 2.5, 2.5]
```

字典

字典是一种存储相同类型多重数据的存储器。每个值（value）都关联独特的键（key），键作为字典中的这个值数据的标识符。和数组中的数据项不同，字典中的数据项并没有具体顺序。我们在需要通过标识符（键）访问数据的时候使用字典，这种方法很大程度上和我们在现实世界中使用字典查字义的方法一样。

Swift 的字典使用时需要具体规定可以存储键和值类型。不同于 Objective-C 的NSDictionary和NSMutableDictionary 类可以使用任何类型的对象来作键和值并且不提供任何关于这些对象的本质信息。在 Swift 中，在某个特定字典中可以存储的键和值必须提前定义清楚，方法是通过显性类型标注或者类型推断。

Swift 的字典使用Dictionary<KeyType, ValueType>定义,其中KeyType是字典中键的数据类型，ValueType是字典中对应于这些键所存

储值的数据类型。

KeyType的唯一限制就是可哈希的，这样可以保证它是独一无二的，所有的 Swift 基本类型（例如String，Int，Double和Bool）都是默认可哈希的，并且所有这些类型都可以在字典中当做键使用。未关联值的枚举成员（参见枚举）也是默认可哈希的。

字典字面语句

我们可以使用字典字面语句来构造字典，他们和我们刚才介绍过的数组字面语句拥有相似语法。一个字典字面语句是一个定义拥有一个或者多个键值对的字典集合的简单语句。

一个键值对是一个key和一个value的结合体。在字典字面语句中，每一个键值对的键和值都由冒号分割。这些键值对构成一个列表，其中这些键值对由方括号包含并且由逗号分割：

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

下面的例子创建了一个存储国际机场名称的字典。在这个字典中键是三个字母的国际航空运输相关代码，值是机场名称：

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

airports字典被定义为一种Dictionary<String, String>,它意味着这个字典的键和值都是String类型。

注意：

airports字典被声明为变量（用var关键字）而不是常量（let关键字）因为后来更多的机场信息会被添加到这个示例字典中。
airports字典使用字典字面语句初始化，包含两个键值对。第一对的键是TYO，值是Tokyo。第二对的键是DUB，值是Dublin。

这个字典语句包含了两个String: String类型的键值对。他们对应airports变量声明的类型（一个只有String键和String值的字典）所以

这个字典字面语句是构造两个初始数据项的airport字典。

和数组一样，如果我们使用字面语句构造字典就不用把类型定义清楚。airports的也可以用这种方法简短定义：

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

因为这个语句中所有的键和值都分别是相同的数据类型，Swift 可以推断出Dictionary<String, String>是airports字典的正确类型。

读取和修改字典

我们可以通过字典的方法和属性来读取和修改字典，或者使用下标语法。和数组一样，我们可以通过字典的只读属性count来获取某个字典的数据项数量：

```
println("The dictionary of airports contains \
(airports.count) items.")
// 打印 "The dictionary of airports contains 2
items."（这个字典有两个数据项）
```

我们也可以在字典中使用下标语法来添加新的数据项。可以使用一个合适类型的 key 作为下标索引，并且分配新的合适类型的值：

```
airports["LHR"] = "London"
// airports 字典现在有三个数据项
```

我们也可以使用下标语法来改变特定键对应的值：

```
airports["LHR"] = "London Heathrow"
// "LHR"对应的值 被改为 "London Heathrow
```

作为另一种下标方法，字典的updateValue(forKey:)方法可以设置或者更新特定键对应的值。就像上面所示的示例，updateValue(forKey:)方法在这个键不存在对应值的时候设置值或者在存在时更新已存在的值。和上面的下标方法不一样，这个方法返回更新值之前的原值。这样方便我们检查更新是否成功。

updateValue(forKey:)函数会返回包含一个字典值类型的可选值。举例来说：对于存储String值的字典，这个函数会返回一个String?或者“可

选`String`”类型的值。如果值存在，则这个可选值值等于被替换的值，否则将会是`nil`。

```
if let oldValue = airports.updateValue("Dublin
Internation", forKey: "DUB") {
    println("The old value for DUB was \(oldValue).")
}
// 输出 "The old value for DUB was Dublin."（dub原值是
dublin）
```

我们也可以使用下标语法来在字典中检索特定键对应的值。由于使用一个没有值的键这种情况是有可能发生的，可选类型返回这个键存在的相关值，否则就返回`nil`：

```
if let airportName = airports["DUB"] {
    println("The name of the airport is \
(airportName).")
} else {
    println("That airport is not in the airports
dictionary.")
}
// 打印 "The name of the airport is Dublin
INTernation."（机场的名字是都柏林国际）
```

我们还可以使用下标语法来通过给某个键的对应值赋值为`nil`来从字典里移除一个键值对：

```
airports["APL"] = "Apple Internation"
// "Apple Internation"不是真的 APL机场，删除它
airports["APL"] = nil
// APL现在被移除了
```

另外，`removeValueForKey`方法也可以用来在字典中移除键值对。这个方法在键值对存在的情况下会移除该键值对并且返回被移除的value或者在没有值的情况下返回`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."
```

字典遍历

我们可以使用`for-in`循环来遍历某个字典中的键值对。每一个字典中的数据项都由`(key, value)`元组形式返回，并且我们可以使用暂时性常量或者变量来分解这些元组：

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

`for-in`循环请参见For 循环。

我们也可以通过访问他的`keys`或者`values`属性（都是可遍历集合）检索一个字典的键或者值：

```
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
```

如果我们只是需要使用某个字典的键集合或者值集合来作为某个接受`Array`实例 API 的参数，可以直接使用`keys`或者`values`属性直接构造一

个新数组：

```
let airportCodes = Array(airports.keys)
// airportCodes is ["TYO", "LHR"]

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

注意：

Swift 的字典类型是无序集合类型。其中字典键，值，键值对在遍历的时候会重新排列，而且其中顺序是不固定的。

创建一个空字典

我们可以像数组一样使用构造语法创建一个空字典：

```
var namesOfIntegers = Dictionary<Int, String>()
// namesOfIntegers 是一个空的 Dictionary<Int, String>
```

这个例子创建了一个`Int, String`类型的空字典来储存英语对整数的命名。他的键是`Int`型，值是`String`型。

如果上下文已经提供了信息类型，我们可以使用空字典字面语句来创建一个空字典，记作`[:]`（中括号中放一个冒号）：

```
namesOfIntegers[16] = "sixteen"
// namesOfIntegers 现在包含一个键值对
namesOfIntegers = [:]
// namesOfIntegers 又成为了一个 Int, String类型的空字典
```

注意：

在后台，Swift 的数组和字典都是由泛型集合来实现的，想了解更多泛型和集合信息请参见泛型。

集合的可变性

数组和字典都是在单个集合中存储可变值。如果我们创建一个数组或者字典并且把它分配成一个变量，这个集合将会是可变的。这意味着我们可以在创建之后添加更多或移除已存在的数据项来改变这个集合的大小。与此相反，如果我们把数组或字典分配成常量，那么他就是不可变的，它的大小不能被改变。

对字典来说，不可变性也意味着我们不能替换其中任何现有键所对应的值。不可变字典的内容在被首次设定之后不能更改。 不可变行对数组来说有一点不同，当然我们不能试着改变任何不可变数组的大小，但是我们·可以重新设定相对现存索引所对应的值。这使得 Swift 数组在大小被固定的时候依然可以做的很棒。

Swift 数组的可变性行为同时影响了数组实例如何被分配和修改，想获取更多信息，请参见集合在赋值和复制中的行为。

注意：

在我们不需要改变数组大小的时候创建不可变数组是很好的习惯。如此 Swift 编译器可以优化我们创建的集合。