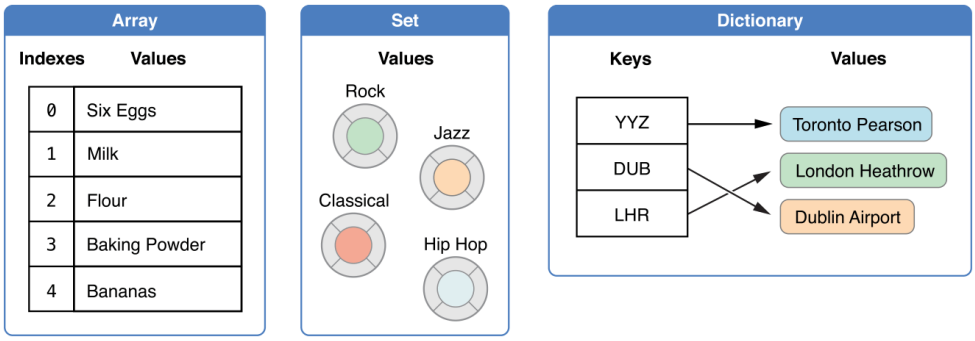
# 集合类型

Swift提供了三种主要的*集合类型*，称为数组，集合和字典，用于存储值的集合。数组是有序的值集合。集合是唯一值的无序集合。字典是键值关联的无序集合。




Swift中的数组，集合和字典总是清楚它们可以存储的值和键的类型。这意味着您不能错误地将错误类型的值插入到集合中。这也意味着您可以确信您将从集合中检索的值的类型。

> 注意
>
> Swift的数组，集合和字典类型被实现为*通用集合*。有关泛型类型和集合的更多信息，请参阅泛型。

## 集合的可变性

如果您创建数组，集合或字典，并将其分配给一个变量，则创建的集合将是*可变的*。这意味着，你可以改变（或*变异*它是由添加，删除或改变集合中的项目创建后）的集合。如果将数组，集合或字典分配给常量，那么该集合是*不可变的*，并且其大小和内容不能更改。

> 注意
>
> 在集合不需要改变的所有情况下创建不可变集合是一个好习惯。这样做可以让您更轻松地推理代码，并使Swift编译器能够优化您创建的集合的性能。

## 数组

一个*阵列*存储在有序列表中的相同类型的值。相同的值可以在不同位置多次出现在阵列中。

> 注意
>
> 斯威夫特的`Array`类型被桥接到基金会的`NSArray`班级。
>
> 有关`Array`在Foundation和Cocoa中使用的更多信息，请参阅在*Cocoa和Objective-C*中*使用Swift使用* Cocoa数据类型（*Swift 4.1*）。

### 数组类型速记语法

Swift数组的类型完全写成`Array<Element>`，其中`Element`数组允许存储的值的类型在哪里。你也可以用简写形式写出数组的类型`[Element]`。虽然这两种形式在功能上是相同的，但是在引用数组类型时，速记形式是优选的，并且贯穿本指南。

### 创建一个空数组

您可以使用初始化语法创建一个特定类型的空数组：

```
var som
print("someInts is of type [Int] with \(someInts.count) items.")
// Prints "someInts is of type [Int] with 0 items."
```

请注意，someInts变量的类型被推断为[Int]来自初始值设定项的类型。

或者，如果上下文已经提供了类型信息，例如函数参数或已经存在类型的变量或常量，则可以创建一个空数组，其中包含一个空数组文本，该文本写为[]（一对空方括号）：

```
someInts.append(3)
// someInts now contains 1 value of type Int
someInts = []
// someInts is now an empty array, but is still of type [Int]
```

### 用默认值创建一个数组

Swift的Array类型还提供了一个初始化器，用于创建一个具有特定大小的数组，并将其所有值设置为相同的默认值。您将此初始值设定项传递给适当类型（调用repeating）的默认值：以及在新数组（调用count）中重复该值的次数：

```
var threeDoubles = Array(repeating: 0.0, count: 3)
// threeDoubles is of type [Double], and equals [0.0, 0.0, 0.0]
```

### 通过一起添加两个数组来创建一个数组

您可以通过使用添加运算符（+）添加具有兼容类型的两个现有数组来创建新数组。新数组的类型是根据您添加在一起的两个数组的类型推断的：

```
var anotherThreeDoubles = Array(repeating: 2.5, count: 3)
// anotherThreeDoubles is of type [Double], and equals [2.5, 2.5, 2.5]

var sixDoubles = threeDoubles + anotherThreeDoubles
// sixDoubles is inferred as [Double], and equals [0.0, 0.0, 0.0, 2.5, 2.5, 2.5]
```

### 使用数组字面值创建数组

您还可以使用*数组文本*初始化数组，这是将一个或多个值作为数组集合写入的简写方法。数组文字被写为一列值，用逗号分隔，并由一对方括号包围：

```
[ 值1, 值2, 值3 ]
```

下面的示例创建一个调用shoppingList以存储String值的数组：

```
var shoppingList: [String] = ["Eggs", "Milk"]
// shoppingList has been initialized with two initial items
```

该shoppingList变量被声明为“字符串值的数组”，写为[String]。由于此特定数组已指定值类型String，因此只允许存储String值。在这里，shoppingList数组被初始化为两个String值（"Eggs"和"Milk"），写入数组文字中。

> 注意
>
> 该shoppingList数组被声明为一个变量（与var介绍人），而不是一个常量（与let介绍人），因为更多的项目被添加到下面的例子中的购物清单。

在这种情况下，数组文字包含两个`String`值，没有别的。这与`shoppingList`变量声明的类型（一个只能包含`String`值的数组）相匹配，因此允许使用数组文本的赋值作为初始化`shoppingList`两个初始项的方法。

感谢Swift的类型推断，如果使用包含相同类型值的数组字面值初始化数组，则不必编写数组的类型。初始化`shoppingList`本

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

因为数组文本中的所有值都是相同的类型，所以Swift可以推断出这`[String]`是用于`shoppingList`变量的正确类型。

## 访问和修改数组

您可以通过其方法和属性或使用下标语法来访问和修改数组。

要找出数组中项目的数量，请检查其只读`count`属性：

```
print("The shopping list contains \(shoppingList.count) items.")
// Prints "The shopping list contains 2 items."
```

使用Boolean `isEmpty`属性作为检查`count`属性是否等于的快捷方式`0`：

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

您可以通过调用数组的`append(_:)`方法将新项添加到数组的末尾：

```
shoppingList.append("Flour")
// shoppingList now contains 3 items, and someone is making pancakes
```

或者，使用添加赋值运算符（`+=`）添加一个或多个兼容项目的数组：

```
shoppingList += ["Baking Powder"]
// shoppingList now contains 4 items
shoppingList += ["Chocolate Spread", "Cheese", "Butter"]
// shoppingList now contains 7 items
```

使用*subscript语法*从数组中检索一个值，并在数组名称后立即传递要检索的方括号内的值的索引：

```
var firstItem = shoppingList[0]
// firstItem is equal to "Eggs"
```

> 注意
> 数组中的第一个项目的索引是`0`，而不是`1`。Swift中的数组始终为零索引。

您可以使用下标语法来更改给定索引处的现有值：

```
shoppingList[0] = "Six eggs"
// the first item in the list is now equal to "Six eggs" rather than "Eggs"
```

当使用下标语法时，您指定的索引需要有效。例如，编写`shoppingList[shoppingList.count] = "Salt"`试图将项添加到数组的末尾会导致运行时错误。

即使替换值的长度与要替换的范围不同，您也可以使用下标语法一次更改一系列值。下面的示例替换`"Chocolate Spread"`，`"Cheese"`以及`"Butter"`与`"Bananas"`和`"Apples"`：

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

```
// shoppingList now contains 6 items
```

要将一个项目插入到指定索引处的数组中，请调用该数组的`insert(_:at:)`方法：

```
shoppingList.insert("Maple Syrup", at: 0)
// shoppingList now contains 7 items
// "Maple Syrup" is now the first item in the list
```

这个`insert(_:at:)`方法的调用插入一个新的项目，其值为`"Maple Syrup"`购物清单的最开始处，由索引为`0`。

同样，您可以使用该`remove(at:)`方法从数组中移除一个项目。此方法删除指定索引处的项目并返回删除的项目（但如果不需要，您可以忽略返回的值）：

```
let mapleSyrup = shoppingList.remove(at: 0)
// the item that was at index 0 has just been removed
// shoppingList now contains 6 items, and no Maple Syrup
// the mapleSyrup constant is now equal to the removed "Maple Syrup" string
```

> 注意
>
> 如果您尝试访问或修改数组现有边界之外的索引的值，则会触发运行时错误。通过将索引与数组的`count`属性进行比较，您可以检查索引是否有效。数组中最大的有效索引是`count - 1`因为数组从零开始索引 - 但是，当`count`is `0`（意味着数组为空）时，没有有效的索引。

当一个项目被删除时，数组中的任何间隙都会关闭，因此index `0`处的值再次等于`"Six eggs"`：

```
firstItem = shoppingList[0]
// firstItem is now equal to "Six eggs"
```

如果要从数组中移除最后一项，请使用`removeLast()`方法而不是`remove(at:)`方法来避免查询数组的`count`属性。像该`remove(at:)`方法一样，`removeLast()`返回已删除的项目：

```
let apples = shoppingList.removeLast()
// the last item in the array has just been removed
// shoppingList now contains 5 items, and no apples
// the apples constant is now equal to the removed "Apples" string
```

## 迭代数组

可以遍历整个集合值与数组`for- in`循环：

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

如果您需要每个项目的整数索引及其值，请使用该`enumerated()`方法遍历数组。对于数组中的每个项目，该`enumerated()`方法返回一个由整数和项目组成的元组。整数从零开始，每个项目加1; 如果您枚举整个数组，这些整数与项目的索引匹配。作为迭代的一部分，您可以将元组分解为临时常量或变量：

```
for (index, value) in shoppingList.enumerated() {
    print("Item \(index + 1): \(value)")
}
// Item 1: Six eggs
// Item 2: Milk
// Item 3: Flour
// Item 4: Baking Powder
```

```
// Item 5: Bananas
```

有关for- in循环的更多信息，请参阅For-In循环。

## 集

一个*集合*在同一个*集合中*存储相同类型的不同值并且没有定义的顺序。当项目顺序不重要时，或者需要确保项目只出现一次时，您可以使用一个集合而不是一个数组。

> 注意
>
> 斯威夫特的Set类型被桥接到基金会的NSSet班级。
>
> 有关Set在Foundation和Cocoa中使用的更多信息，请参阅在*Cocoa和Objective-C*中*使用Swift使用* Cocoa数据类型*（Swift 4.1）*。

### 集合类型的哈希值

一个类型必须是可*散列*的才能存储在一个集合中 - 也就是说，该类型必须提供一种为自己计算*散列值*的方法。哈希值是Int是对于同样比较所有对象，例如，如果相同的值a == b，它遵循a.hashValue == b.hashValue。

所有斯威夫特的基本类型（例如String，Int，Double，和Bool）默认情况下可哈希，并可以作为设定值类型或字典密钥类型。没有关联值的枚举大小写值（如枚举中所述）默认情况下也是可哈希的。

> 注意
>
> 您可以使用自己的自定义类型作为设置值类型或字典键类型，使其符合HashableSwift标准库中的协议。符合Hashable协议的类型必须提供一个Int名为gettable的属性hashValue。类型hashValue属性返回的值不需要在同一程序的不同执行过程中或在不同的程序中相同。
>
> 由于Hashable协议符合Equatable，符合类型还必须提供equals运算符（==）的实现。该Equatable协议要求任何符合实现的==是等价关系。也就是说，一个实现==必须满足以下三个条件，所有值a，b以及c：
>
> - a == a （自反）
> - a == b暗示b == a（对称）
> - a == b && b == c意味着a == c（传递性）
>
> 有关符合协议的更多信息，请参阅协议。

### 设置类型语法

Swift集合的类型被写为Set<Element>，Element集合允许存储的类型在哪里。与数组不同，集合不具有等效的速记形式。

### 创建并初始化一个空集

您可以使用初始化语法创建一个特定类型的空集：

```
var letters = Set<Character>()
print("letters is of type Set<Character> with \(letters.count) items.")
// Prints "letters is of type Set<Character> with 0 items."
```

> 注意
>
> 根据初始值设定项的类型letters推断变量Set<Character>的类型是。

或者，如果上下文已经提供了类型信息，例如函数参数或已经存在类型的变量或常量，则可以使用空数组文本创建一个空集：

```
letters.insert("a")
```

```
// letters now contains 1 value of type Character
letters = []
// letters is now an empty set, but is still of type Set<Character>
```

## 使用数组文字创建一个集合

您还可以使用数组文本初始化一个集合，作为将一个或多个值作为集合集合写入的简写方法。

下面的示例创建一个调用favoriteGenres存储String值的集合：

```
var favoriteGenres: Set<String> = ["Rock", "Classical", "Hip hop"]
// favoriteGenres has been initialized with three initial items
```

该favoriteGenres变量被声明为“一组String值”，写为Set<String>。因为这个特定的集合已经指定了一个值类型String，所以*只*允许存储String值。在此，favoriteGenres集合被初始化具有三个String值（"Rock"，"Classical"，和"Hip hop"），阵列字面内写入。

> 注意
>
> 该favoriteGenres集被声明为一个变量（与var介绍人），而不是一个常量（与let介绍人），因为在下面的例子中添加和删除了项目。

集合类型不能单独从数组文本中推断出来，所以Set必须明确声明类型。但是，由于Swift的类型推断，如果使用包含相同类型值的数组字面值初始化它，则不必编写该类型的集合。初始化favoriteGenres本来可以用较短的形式写成：

```
var favoriteGenres: Set = ["Rock", "Classical", "Hip hop"]
```

因为数组文本中的所有值都是相同的类型，所以Swift可以推断出这Set<String>是用于favoriteGenres变量的正确类型。

## 访问和修改集合

你可以通过它的方法和属性来访问和修改一个集合。

要找出一组中的项目数量，请检查其只读count属性：

```
print("I have \(favoriteGenres.count) favorite music genres.")
// Prints "I have 3 favorite music genres."
```

使用Boolean isEmpty属性作为检查count属性是否等于的快捷方式0：

```
if favoriteGenres.isEmpty {
    print("As far as music goes, I'm not picky.")
} else {
    print("I have particular music preferences.")
}
// Prints "I have particular music preferences."
```

您可以通过调用set的insert(_:)方法将新项目添加到集合中：

```
favoriteGenres.insert("Jazz")
// favoriteGenres now contains 4 items
```

您可以通过调用set的remove(_:)方法从集合中删除一个项目，如果该项目是该集合的成员，则删除该项目并返回已删除的值; nil如果该集合未包含该项目，则返回该值。或者，可以使用其removeAll()方法删除集合中的所有项目。

```
if let removedGenre = favoriteGenres.remove("Rock") {
    print("\(removedGenre)? I'm over it.")
} else {
```

```
    print("I never much cared for that.")
}
// Prints "Rock? I'm over it."
```

要检查一个集合是否包含特定项目，请使用该contains(_:)方法。

```
if favoriteGenres.contains("Funk") {
    print("I get up on the good foot.")
} else {
    print("It's too funky in here.")
}
// Prints "It's too funky in here."
```

### 迭代集合

您可以使用for- in循环遍历集合中的值。

```
for genre in favoriteGenres {
    print("\(genre)")
}
// Jazz
// Hip hop
// Classical
```

有关for- in循环的更多信息，请参阅For-In循环。

Swift的Set类型没有定义的顺序。要按特定顺序迭代集合的值，请使用该sorted()方法，该方法将集合的元素作为使用<运算符排序的数组返回。

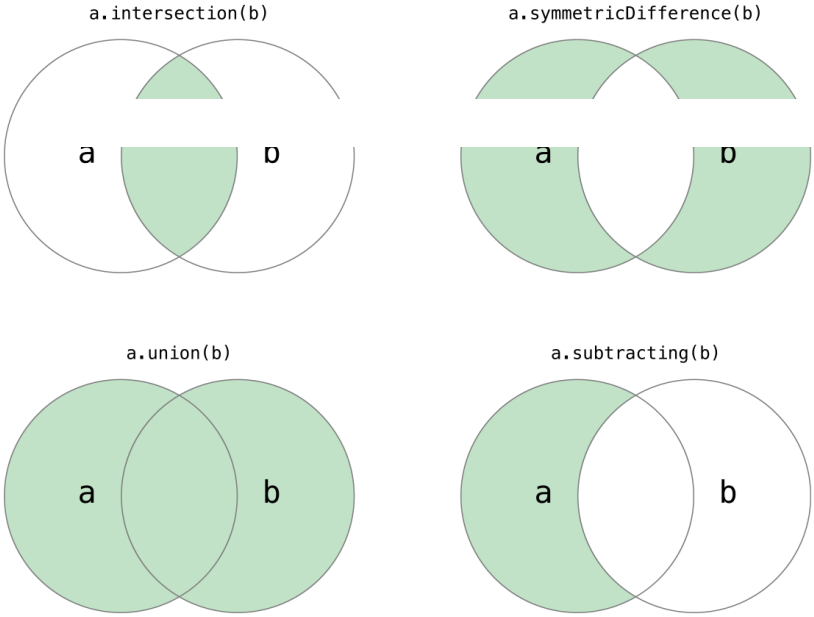
```
for genre in favoriteGenres.sorted() {
    print("\(genre)")
}
// Classical
// Hip hop
// Jazz
```

## 执行集合操作

您可以高效地执行基本集合操作，例如将两个集合组合在一起，确定两个集合具有哪些值，或确定两个集合是包含全部，部分还是不包含相同的值。

### 基本设置操作

下图描绘了两个集- a和b-附由阴影区域表示的各种设定操作的结果。



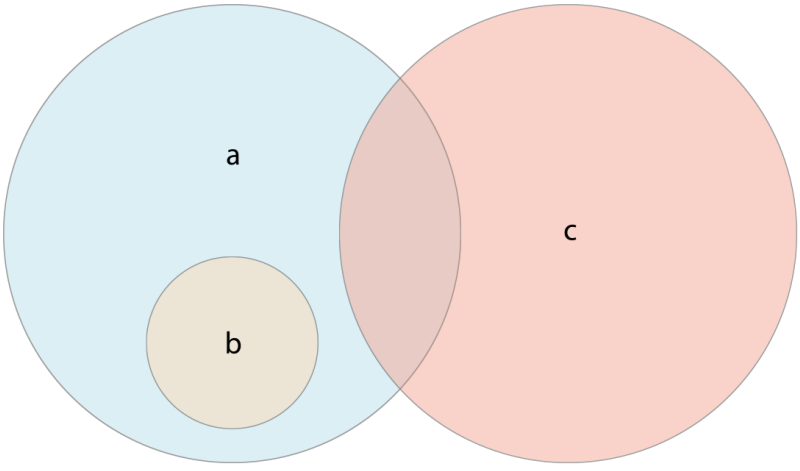

- 使用该`intersection(_:)`方法创建一个仅具有两个集合通用值的新集合。
- 使用该`symmetricDifference(_:)`方法创建一个新的集合，其中任何一个集合中都有值，但不能同时包含两个值
- 使用该`union(_:)`方法创建一个包含两个集合中所有值的新集合。
- 使用该`subtracting(_:)`方法创建一个新的集合，其值不在指定的集合中。

```
let oddDigits: Set = [1, 3, 5, 7, 9]
let evenDigits: Set = [0, 2, 4, 6, 8]
let singleDigitPrimeNumbers: Set = [2, 3, 5, 7]

oddDigits.union(evenDigits).sorted()
// [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
oddDigits.intersection(evenDigits).sorted()
// []
oddDigits.subtracting(singleDigitPrimeNumbers).sorted()
// [1, 9]
oddDigits.symmetricDifference(singleDigitPrimeNumbers).sorted()
// [1, 2, 9]
```

## 设置成员资格和平等

下面的插图描述了三个集合a，b以及c表示在集合之间共享元素的重叠区域。Set a是*集合的超*集b，因为a包含了所有元素b。相反，set b是集合的一个*子*集a，因为所有元素b都包含在内a。Set b和Set c是彼此*分离的*，因为它们没有共同的元素。



- 使用“is equal”运算符（==）来确定两个集合是否包含所有相同的值。
- 使用该`isSubset(of:)`方法确定一个集合的所有值是否都包含在指定的集合中。
- 使用该`isSuperset(of:)`方法确定一个集合是否包含指定集合中的所有值。
- 使用`isSt`
  一个指定的集合。
- 使用该`isDisjoint(with:)`方法确定两个集合是否没有共同的值。

```
let houseAnimals: Set = ["🐶", "🐱"]
let farmAnimals: Set = ["🐮", "🐔", "🐑", "🐶", "🐱"]
let cityAnimals: Set = ["🐦", "🐭"]

houseAnimals.isSubset(of: farmAnimals)
// true
farmAnimals.isSuperset(of: houseAnimals)
// true
farmAnimals.isDisjoint(with: cityAnimals)
// true
```

## 字典

甲*字典*存储相同类型的密钥和一个集合中的相同类型的值与没有定义排序之间的关联。每个值都与一个唯一*键*相关联，该*键*用作字典中该值的标识符。与数组中的项目不同，字典中的项目没有指定的顺序。当需要根据标识符查找值时，您可以使用字典，这与使用真实世界字典查找特定字词的定义的方式大致相同。

> 注意
>
> 斯威夫特的`Dictionary`类型被桥接到基金会的`NSDictionary`班级。
>
> 有关`Dictionary`在Foundation和Cocoa中使用的更多信息，请参阅在*Cocoa和Objective-C*中*使用Swift使用Cocoa数据类型（Swift 4.1）*。

### 字典类型速记语法

Swift字典的类型完全写成`Dictionary<Key, Value>`，其中`Key`是可以用作字典键`Value`的值的类型，并且是字典为这些键存储的值的类型。

> 注意
>
> 字典`Key`类型必须符合`Hashable`协议，就像集合的值类型一样。

您也可以用简写形式书写字典的类型`[Key: Value]`。尽管这两种形式在功能上是相同的，但是在引用字典类型时，缩写形式是优选的，并且贯穿本指南。

### 创建一个空字典

与数组一样，您可以`Dictionary`使用初始化语法创建一个空的某种类型：

```
var namesOfIntegers = [Int: String]()
// namesOfIntegers is an empty [Int: String] dictionary
```

这个例子创建一个空的字典类型`[Int: String]`来存储可读的整数值名称。它的键是类型的`Int`，它的值是类型的`String`。

如果上下文已经提供了类型信息，则可以创建一个空字典，其中包含一个空字典文字，该文字被写为`[:]`（一对方括号内的冒号）：

```
namesOfIntegers[16] = "sixteen"
// namesOfIntegers now contains 1 key-value pair
```

```
namesOfIntegers = [:]
// namesOfIntegers is once again an empty dictionary of type [Int: String]
```

## 使用字典文字创建字典

您还可以使用*字典文字*来初始化字典，它具有与前面所看到的数组字面相似的语法。字典文字是将一个或多个键值对写入Dictionary集合的简写方式。

甲*键值对*是一个键和值的组合。在字典文字中，每个键 - 值对中的键和值由冒号分隔。键值对写成一个列表，用逗号分隔，并用一对方括号包围：

```
[ 键1 : 值1 , 键2 : 值2 , 键3 : 值3 ]
```

下面的示例创建一个字典来存储国际机场的名称。在这本词典中，关键字是三个字母的国际航空运输协会代码，其值是机场名称：

```
var airports: [String: String] = ["YYZ": "Toronto Pearson", "DUB": "Dublin"]
```

该airports字典被声明为具有式的[String: String]，意思是“—Dictionary键均为类型的String，并且其值是类型的也String”。

> 注意
>
> 该airports字典被声明为一个变量（与var导引器），而不是一个常数（与let导引器），因为更多的机场被添加到词典中下面的例子。

该airports字典被初始化与含有两个键-值对的字典字面值。第一对有一个键"YYZ"和一个值"Toronto Pearson"。第二对有一个键"DUB"和一个值"Dublin"。

这个字典文字包含两String: String对。这个键值类型与airports变量声明的类型相匹配（只有String键和只有String值的词典），因此字典文本的赋值可以用来初始化airports带有两个初始化项的字典。

与数组一样，如果使用键和值具有一致类型的字典文字进行初始化，则不必编写字典的类型。初始化airports本来可以用较短的形式写成：

```
var airports = ["YYZ": "Toronto Pearson", "DUB": "Dublin"]
```

由于文本中的所有键都是彼此相同的类型，并且同样所有的值都是相同类型的，所以Swift可以推断出这[String: String]是用于airports字典的正确类型。

## 访问和修改字典

您可以通过其方法和属性或使用下标语法来访问和修改字典。

与数组一样，您可以Dictionary通过检查其只读count属性来找出a中的项目数量：

```
print("The airports dictionary contains \(airports.count) items.")
// Prints "The airports dictionary contains 2 items."
```

使用Boolean isEmpty属性作为检查count属性是否等于的快捷方式0：

```
if airports.isEmpty {
    print("The airports dictionary is empty.")
} else {
    print("The airports dictionary is not empty.")
}
// Prints "The airports dictionary is not empty."
```

您可以使用下标语法将新项目添加到字典中。使用适当类型的新键作为下标索引，并分配适当类型的新值：

```
airports["LHR"] = "London"
// the airports dictionary now contains 3 items
```

您还可以使用下标语法来更改与特定键关联的值：

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

作为下标的替代`updateValue(_:forKey:)`方法，使用字典的方法来设置或更新特定键的值。就像上面的下标示例一样，`updateValue(_:forKey:)`如果某个键不存在，该方法将为该键设置一个值，或者如果该键已经存在，则更新该值。然而，与下标不同的是，该`updateValue(_:forKey:)`方法在执行更新后返回*旧*值。这使您可以检查是否发生更新。

该`updateValue(_:forKey:)`方法返回字典值类型的可选值。例如，对于存储`String`值的字典，该方法返回一个类型值`String?`，或“可选`String`”。如果更新前存在该值，则此可选值包含该值的旧值，或者`nil`没有值：

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

您还可以使用下标语法从字典中检索特定键的值。因为可以请求不存在值的键，所以字典的下标返回字典值类型的可选值。如果字典包含请求键的值，则下标返回包含该键现有值的可选值。否则，下标返回`nil`：

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

您可以使用下标语法通过指定该键的值来从字典中删除键值对`nil`：

```
airports["APL"] = "Apple International"
// "Apple International" is not the real airport for APL, so delete it
airports["APL"] = nil
// APL has now been removed from the dictionary
```

或者，使用该`removeValue(forKey:)`方法从字典中移除键值对。如果键值对存在并返回已除去的值，则该方法将移除键值对，`nil`如果没有值，则返回该值：

```
if let removedValue = airports.removeValue(forKey: "DUB") {
    print("The removed airport's name is \(removedValue).")
} else {
    print("The airports dictionary does not contain a value for DUB.")
}
// Prints "The removed airport's name is Dublin Airport."
```

## 迭代字典

您可以用字典遍历键值对`for-in`环。字典中的每一项都作为`(key, value)`元组返回，并且可以将元组的成员分解为临时常量或变量，作为迭代的一部分：

在本页

```
for (airportCode, airportName) in airports {
    print("\(airportCode): \(airportName)")
}
// YYZ: Toronto Pearson
// LHR: London Heathrow
```

有关`for-in`循环的更多信息，请参阅For-In循环。

您还可以通过访问其属性`keys`和`values`属性来检索字典键或值的可迭代集合：

```
for airportCode in airports.keys {
    print("Airport code: \(airportCode)")
```

```
}
// Airport code: YYZ
// Airport code: LHR

for airportName in airports.values {
    print("Airport name: \(airportName)")
}
// Airport name: Toronto Pearson
// Airport name: London Heathrow
```

如果您需要使用带有Array实例的API的字典键或值，请使用keys或values属性初始化新数组：

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

let airportNames = [String](airports.values)
// airportNames is ["Toronto Pearson", "London Heathrow"]
```

Swift的Dictionary类型没有定义的顺序。要按特定顺序遍历字典的键或值，请sorted()在其keys或values属性上使用该方法。