The Swift Programming Language in Chinese

https://github.com/letsswift

http://letsswift.com/

June 15, 2014

目录

1 Swift 中文教程（九）类与结构

类与结构是编程人员在代码中会经常用到的代码块。在类与结构中可以像定义常量，变量和函数一样，定义相关的属性和方法以此来实现各种功能。

和其它的编程语言不太相同的是，Swift 不需要单独创建接口或者实现文件来使用类或者结构。Swift 中的类或者结构可以在单文件中直接定义，一旦定义完成后，就能够被直接其它代码使用。

注意：一个类的实例一般被视作一个对象，但是在 Swift 中，类与结构更像是一个函数方法，在后续的章节中更多地是讲述类和结构的功能性。

1.1 类和结构的异同

类和结构有一些相似的地方，它们都可以：

- 定义一些可以赋值的属性；
- 定义具有功能性的方法
- 定义下标，使用下标语法
- 定义初始化方法来设置初始状态
- 在原实现方法上的可扩展性
- 根据协议提供某一特定类别的基本功能

更多内容可以阅读：属性，方法，下标，初始化，扩展和协议等章节

类还有一些结构不具备的特性：

- 类的继承性
- 对类实例实时的类型转换
- 析构一个类的实例使之释放空间
- 引用计数，一个类实例可以有多个引用

更多内容可以阅读：继承，类型转换，初始化自动引用计数

> 注意：结构每次在代码中传递时都是复制了一整个，所以不要使用引用计数

1.1.1 定义语法

类和结构拥有相似的定义语法，使用 class 关键词定义一个类，struct 关键词定义结构。每个定义都由一对大括号包含：

```
class SomeClass {
    // class definition goes here
}
struct SomeStructure {
    // structure definition goes here
}
```

> 注意：在定义类和结构时，一般使用 UpperCamelCase 命名法来定义类和结构的名称，比如 SomeClass 和 SomeStructure，这样也符合 Swift 其它类型的标准。而给属性和方法命名时，一般时候 lowerCamelCase 命名法，比如 frameRate 和 incrementCount 等。

下面是一个结构和一个类的定义示例：

```
struct Resolution {
    var width = 0
    var height = 0
}
class VideoMode {
    var resolution = Resolution()
    var interlaced = falsevar
    frameRate = 0.0
    var name: String?
}
```

上面的例子首先定义了一个叫 Resolution 的结构，用来描述一个像素显示的分辨率，它有两个属性分别叫 width 和 height。这两个属性被默认定义为 Int 类型，初始化为 0.

之后定义了一个叫 VideoMode 的类，为视频显示的显示方式。这个类有四个属性，第一个属性 resolution 本身又是一个结构，然后是另外两个属性。最后一个属性用到了可选字符串类型 String?，表示这个属性可以存在，或者不存在为 nil。

1.1.2 类和结构的实例

上面的两个定义仅仅是定义了结构 Resolution 和类 VideoMode 的整体样式，它们本身不是一个特定的分辨率或者显示方式，这时候就需要实例化这个结构和类。

实例化的语法相似：

```
let someResolution = Resolution()
let someVideoMode = VideoMode()
```

类和结构都使用实例语法来完成实例化。最简单的实例语法就是用两个括号 () 完成。在这种情况下定义的实例中的属性都会完成默认初始化。更多内容可以参考初始化一章。

1.1.3 访问属性

使用. 语法就可以方便地访问一个实例的属性。在. 语法中，在实例名之后加上 (.) 再加上属性名即可，不需要空格：

```
println("The width of someResolution is \(someResolution.width)")
// prints "The width of someResolution is 0"
```

在这个例子中，someResolution.width 表示 someResolution 的 width 属性，返回了它的初始值 0

也可以使用. 语法连续地获取属性的属性，比如 VideoMode 中 resolution 属性的 width 属性

```
println("The width of someVideoMode is \(someVideoMode.resolution.width)")
// prints "The width of someVideoMode is 0"
```

使用这种方法不仅可以访问，也可以赋值：

```
someVideoMode.resolution.width = 1280
println("The width of someVideoMode is now \(someVideoMode.resolution.width)")
// prints "The width of someVideoMode is now 1280"
```

注意：和 Objective-C 不同，Swift 能够直接设置一个结构属性的子属性，就像上面这个例子一样。

1.1.4 结构类型的成员初始化方法

每个结构都有一个成员初始化方法，可以在初始化的时候通过使用属性名称来指定每一个属性的初始值：

```
let vga = Resolution(width: 640, height: 480)
```

但是和结构不同，类实例不能够使用成员初始化方法，在初始化一章有专门的介绍。

1.2 结构和枚举类型是数值类型

数值类型是说当它被赋值给一个常量或者变量，或者作为参数传递给函数时，是完整地复制了一个新的数值，而不是仅仅改变了引用对象。

事实上读到这里你已经在前面几章见过数值类型了，所有 Swift 中的基础类型 -整型，浮点型，布尔类型，字符串，数组和字典都是数值类型。它们也都是由结构来实现的。

在 Swift 中所有的结构和枚举类型都是数值类型。这意味这你实例化的每个结构和枚举，其包含的所有属性，都会在代码中传递的时候被完整复制。

下面的这个例子可以说明这个特性：

```
let hd = Resolution(width: 1920, height: 1080)
var cinema = hd
```

声明了一个常量 hd，是 Resolution 的实例化，宽度是 1920，高度是 1080，然后声明了一个变量 cinema，和 hd 相同。这个时候表明，cinema 和 hd 是两个实例，虽然他们的宽度都是 1920，高度都是 1080。

如果把 cinema 的宽度更改为 2048，hd 的宽度不会变化，依然是 1920

```
cinema.width = 2048
println("cinema is now \(cinema.width) pixels wide")
// prints "cinema is now 2048 pixels wide"
println("hd is still \(hd.width) pixels wide")
// prints "hd is still 1920 pixels wide"
```

这表明当 hd 被赋值给 cinema 时，是完整地复制了一个全新的 Resolution 结构给 cinema，所以当 cinema 的属性被修改时，hd 的属性不会变化。

下面的例子演示的是枚举类型：

```
enum CompassPoint {
    case North, South, East, West
}
var currentDirection = CompassPoint.West
let rememberedDirection = currentDirection
currentDirection = .East
if rememberedDirection == .West {
    println("The remembered direction is still .West")
}
// prints "The remembered direction is still .West"
```

尽管经过几次赋值，rememberedDirection 依然没有变化，这是因为在每一次赋值过程中，都是将数值类型完整地复制了过来。

1.3 类是引用类型

和数值类型不同引用类型不会复制整个实例，当它被赋值给另外一个常量或者变量的时候，而是会建立一个和已有的实例相关的引用来表示它。

下面是引用的示例，VideoMode 被定义为一个类：

```
let tenEighty = VideoMode()
tenEighty.resolution = hd
tenEighty.interlaced = true
tenEighty.name = "1080i"
tenEighty.frameRate = 25.0
```

分别将这个实例 tenEighty 的四个属性初始化，然后 tenEighty 被赋值给了另外一个叫 alsoTenEighty 的常量，然后 alsoTenEighty 的 frameRate 被修改了

```
let alsoTenEighty = tenEighty
alsoTenEighty.frameRate = 30.0
```

由于类是一个引用类型，所以 tenEighty 和 alsoTenEighty 实际上是同一个实例，仅仅只是使用了不同的名称而已，我们通过检查 frameRate 可以证明这个问题：

```
println("The frameRate property of tenEighty is now \(tenEighty.frameRate)")
// prints "The frameRate property of tenEighty is now 30.0"
```

注意到 tenEighty 和 alsoTenEighty 是被定义为常量的，而不是变量。但是我们还是可以改变他们的属性值，这是因为它们本身实际上没有改变，它们并没有保存这个 VideoMode 的实例，仅仅只是引用了一个 VideoMode 实例，而我们修改的也是它们引用的实例中的属性。

1.3.1 特征操作

因为类是引用类型，那么就可能存在多个常量或者变量只想同一个类的实例(这对于数值类型的结构和枚举是不成立的)。

可以通过如下两个操作来判断两个常量或者变量是否引用的是同一个类的实例:

- 相同的实例 (===)
- 不同的实例 (!==)

使用这些操作可以检查:

if tenEighty === alsoTenEighty { println(“tenEighty and alsoTenEighty refer to the same Resolution instance.”) } // prints “tenEighty and alsoTenEighty refer to the same Resolution instance.”

注意是相同的实例判断使用三个连续的等号，这和相等（两个等号）是不同的

实例相同表示的是两个变量或者常量所引用的是同一个类的实例

相等是指两个实例在数值上的相等，或者相同。

当你定义一个类的时候，就需要说明什么样的时候是两个类相等，什么时候是两个类不相等。更多内容可以从相等操作一章中获得。

1.3.2 指针

如果你有 C，C++ 或者 Objective-C 的编程经验，你一定知道在这些语言中使用指针来引用一个内存地址。Swift 中引用一个实例的常量或变量跟 C 中的指针类似，但是不是一个直接指向内存地址的指针，也不需要使用 * 记号表示你正在定义一个引用。Swift 中引用和其它变量，常量的定义方法相同。

1.3.3 如何选择使用类还是结构

在代码中可以选择类或者结构来实现你所需要的代码块，完成相应的功能。但是结构实例传递的是值，而类实例传递的是引用。那么对于不同的任务，应该考虑到数据结构和功能的需求不同，从而选择不同的实例。

一般来说，下面的一个或多个条件满足时，应当选择创建一个结构：

- 结构主要是用来封装一些简单的数据值
- 当赋值或者传递的时候更希望这些封装的数据被赋值，而不是被引用过去
- 所有被结构存储的属性本身也是数值类型
- 结构不需要被另外一个类型继承或者完成其它行为

一些比较好的使用结构的例子：

- 一个几何形状的尺寸，可能包括宽度，高度或者其它属性，每个属性都是 Double 类型的
- 一个序列的对应关系，可能包括开始 start 和长度 length 属性，每个属性都是 Int 类型的
- 3D 坐标系中的一个点，包括 x，y 和 z 坐标，都是 Double 类型

在其它情况下，类会是更好的选择。也就是说一般情况下，自定义的一些数据结构一般都会被定义为类。

1.4 集合类型的赋值和复制操作

Swift 中，数组 Array 和字典 Dictionary 是用结构来实现的，但是数组与字典和其它结构在进行赋值或者作为参数传递给函数的时候有一些不同。

并且数组和字典的这些操作，又与 Foundation 中的 NSArray 和 NSDictionary 不同，它们是用类来实现的。

> 注意：下面的小节将会介绍数组，字典，字符串等的复制操作。这些复制操作看起来都已经发生，但是 Swift 只会在确实需要复制的时候才会完整复制，从而达到最优的性能。

1.4.1 字典的赋值和复制操作

每次将一个字典 Dictionary 类型赋值给一个常量或者变量，或者作为参数传递给函数时，字典会在赋值或者函数调用时才会被复制。这个过程在上面的小节：结构和枚举是数值类型中描述了。

如果字典中的键值是数值类型（结构或者枚举），它们在赋值的时候会同时被复制。相反，如果是引用类型（类或者函数），引用本身将会被复制，而不是类实例或者函数本身。字典的这种复制方式和结构相同。

下面的例子演示的是一个叫 ages 的字典，存储了一些人名和年龄的对应关系，当赋值给 copiedAges 的时候，里面的数值同时被完整复制。当改变复制了的数值的时候，原有的数值不会变化，如下例子：

```
var ages = ["Peter": 23, "Wei": 35, "Anish": 65, "Katya": 19]
var copiedAges = ages
```

这个字典的键是字符串 String 类型，值是 Int 类型，都是数值类型，那么在赋值的时候都会被完整复制。

```
copiedAges["Peter"] = 24
println(ages["Peter"])
// prints "23"
```

1.4.2 数组的赋值和复制操作

和字典 Dictionary 类型比起来，数组 Array 的赋值和复制操作就更加复杂。Array 类型和 C 语言中的类似，仅仅只会在需要的时候才会完整复制数组的值。

如果将一个数组赋值给一个常量或者变量，或者作为一个参数传递给函数，复制在赋值和函数调用的时候并不会发生。这两个数组将会共享一个元素序列，如果你修改了其中一个，另外一个也将会改变。

对于数组来说，复制只会在你进行了一个可能会修改数组长度操作时才会发生。包括拼接，添加或者移除元素等等。当复制实际发生的时候，才会像字典的赋值和复制操作一样。

下面的例子演示了数组的赋值操作：

```
var a = [1, 2, 3]
var b = a
var c = a
```

数组 a 被赋值给了 b 和 c，然后输出相同的下标会发现：

```
println(a[0])
// 1
println(b[0])
// 1
println(c[0])
// 1
```

如果改变 a 中的某个值，会发现 b 和 c 中的数值也会跟着改变，因为赋值操作没有改变数组的长度：

```
a[0] = 42
println(a[0])
// 42
println(b[0])
// 42
println(c[0])
// 42
```

但是，如果在 a 中添加一个新的元素，那么就改变了数组的长度，这个时候就会发生实际的复制操作。如果再改变 a 中元素的值，b 和 c 中的元素将不会发生改变:

```
a.append(4)
a[0] = 777
println(a[0])
// 777
println(b[0])
// 42
println(c[0])
// 42
```

1.4.3 设置数组是唯一的

如果可以在对数组进行修改前，将它设置为唯一的就最好了。我们可以通过使用 unshare 方法来将数组自行拷贝出来，成为一个唯一的实体。

如果多个变量引用了同一个数组，可以使用 unshare 方法来完成一次 “独立”

```
b.unshare()
```

这时候如果再修改 b 的值，c 的值也不会再受影响

```
b[0] = -105
println(a[0])
// 777
println(b[0])
// -105
println(c[0])
// 42
```

检查两个数组时候共用了相同的元素

使用实例相等操作符来判断两个数组是否共用了元素（=== 和!===）

下面这个例子演示的就是判断是否共用元素：

```
if b === c {
    println("b and c still share the same array elements.")
} else {
   println("b and c now refer to two independent sets of array elements.")
}
// prints "b and c now refer to two independent sets of array elements."
```

也可以使用这个操作来判断两个子数组是否有共用的元素：

```
if b[0...1] === b[0...1] {
    println("These two subarrays share the same elements.")
} else {
    println("These two subarrays do not share the same elements.")
}
// prints "These two subarrays share the same elements."
```

1.4.4 强制数组拷贝

通过调用数组的 copy 方法来完成强制拷贝。这个方法将会完整复制一个数组到新的数组中。

下面的例子中这个叫 names 的数组会被完整拷贝到 copiedNames 中去。

```
var names = ["Mohsen", "Hilary", "Justyn", "Amy", "Rich", "Graham", "Vic"]
var copiedNames = names.copy()
```

通过改变 copiedNames 的值可以验证，数组已经被完整拷贝，不会影响到之前的数组：

```
copiedNames[0] = "Mo"
println(names[0])
// prints "Mohsen"
```

> 注意：如果你不确定你需要的数组是否是独立的，那么仅仅使用 unshare 就可以了。而 copy 方法不管当前是不是独立的，都会完整拷贝一次，哪怕这个数组已经是 unshare 的了。

参考文献