类和结构体

快速检索 [点击展开]

作为你程序代码的构建基础，*类*和*结构体*是一种多功能且灵活的构造体。通过使用与现存常量、变量、函数完全相同的语法来在类和结构体当中定义属性和方法以添加功能。

不像其他的程序语言，Swift不需要你为自定义类和结构体创建独立的接口和实现文件。在 Swift 中，你在一个文件中定义一个类或者结构体，则系统将会自动生成面向其他代码的外部接口。

> 注意
>
> 一个类的实例通常被称为对象。总之，Swift 的类和结构体在功能上要比其他语言中的更加相近，并且本章节所讨论的大部分功能都可以同时用在类和结构体的实例上。因此，我们使用更加通用的术语实例。

类与结构体的对比

在 Swift 中类和结构体有很多共同之处，它们都能：

- 定义属性用来存储值；
- 定义方法用于提供功能；
- 定义下标脚本用来允许使用下标语法访问值；
- 定义初始化器用于初始化状态；
- 可以被扩展来默认所没有的功能；
- 遵循协议来针对特定类型提供标准功能。

更多信息，请阅览 属性，方法，下标脚本，初始化，扩展和 协议。

类有而结构体没有的额外功能：

- 继承允许一个类继承另一个类的特征；
- 类型转换允许你在运行检查和解释一个类实例的类型；
- 反初始化器允许一个类实例释放任何其所被分配的资源；
- 引用计数允许不止一个对类实例的引用。

更多信息，请阅览继承，类型转换，反初始化和 自动引用计数。

> 注意
>
> 结构体在你的代码中通过复制来传递，并且并不会使用引用计数。

定义语法

类与结构体有着相似的定义语法，你可以通过使用关键词 `class` 来定义类使用 `struct` 来定义结构体。并在一对大括号内定义它们的具体内容。

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

> 注意
>
> 无论你在何时定义了一个新的类或者结构体，实际上你定义了一个全新的 Swift 类型。请用 `UpperCamelCase` 命名法[1]命名 (比如这里我们说到的 `SomeClass` 和 `SomeStructure`)以符合 Swift 的字母大写风格（比如说 `String` ， `Int` 以及 `Bool` ）。相反，对于属性和方法使用 `lowerCamelCase` 命名法[1] (比如 `frameRate` 和 `incrementCount`)，以此来区别于类型名称。

这里有个类定义和结构体定义的例子：

```
struct Resolution {
```

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

上面这个这个例子定义了一个名叫 `Resolution` 的新结构体，用来描述一个基于像素的显示器分辨率。这个结构体拥有两个存储属性名叫 `width` 和 `height` ，存储属性是绑定并储存在类或者结构体中的常量或者变量。这两个属性因以值 0 来初始化，所以它们的类型被推断为 `Int` 。

上面这个例子也定义了一个名叫 `VideoMode` 的新类，用来描述一个视频显示的特定视频模式。这个类有四个变量存储属性。第一个，`resolution` ,用 `Resolution` 结构体实例来初始化，它使属性的类型被推断为 `Resolution` 。对于其他三个属性来说，新的 `VideoMode` 实例将会以 `interlaced` 为 `false` （意思是“非隔行扫描视频”），回放帧率为 `0.0` ，和一个名叫 `name` 的可选项 `String` 值来初始化 。 `name` 属性会自动被赋予一个空值 `nil` ，或“无 `name` 值”，因为它是一个可选项。

类与结构体实例

`Resolution` 结构体的定义和 `VideoMode` 类的定义仅仅描述了什么是 `Resolution` 或 `VideoMode` 。它们自己并没有描述一个特定的分辨率或视频 模式。对此，你需要创建一个结构体或类的实例。

创建结构体和类的实例的语法是非常相似的：

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

结构体和类两者都能使用初始化器语法来生成新的实例。初始化器语法最简单的是在类或结构体名字后面接一个空的圆括号，例如 `Resolution()` 或者 `VideoMode()` 。这样就创建了一个新的类或者结构体的实例，任何属性都被初始化为它们的默认值。在初始化一章有对类和结构体的初始化更详尽的描述。

访问属性

你可以用*点语法*来访问一个实例的属性。在点语法中，你只需在实例名后面书写属性名，用(.)来分开，无需空格：

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

在这个栗子中， `someResolution.width` 就是 `someResolution` 中的 `width` 属性，返回一个它的默认初始值 `0` 。

你可以继续下去来访问子属性，如 `VideoMode` 中 `resolution` 属性中的 `width` 属性：

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

你亦可以用点语法来指定一个新值到一个变量属性中：

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

> 注意
>
> 不同于 Objective–C，Swift 允许你直接设置一个结构体属性中的子属性。在上述最后一个栗

> 子中， `someVideoMode` 的 `resolution` 属性中的 `width` 这个属性可以直接设置，不用你重新设置整个 `resolution` 属性到一个新值。

结构体类型的成员初始化器

所有的结构体都有一个自动生成的*成员初始化器*，你可以使用它来初始化新结构体实例的成员属性。新实例属性的初始化值可以通过属性名称传递到成员初始化器中：

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

与结构体不同，类实例不会接收默认的成员初始化器，初始化器的更多细节在初始化章节。

结构体和枚举是值类型

*值类型*是一种当它被指定到常量或者变量，或者被传递给函数时会被拷贝的类型。

其实，在之前的章节中我们已经大量使用了值类型。实际上，Swift 中所有的基本类型——整数，浮点数，布尔量，字符串，数组和字典——都是值类型，并且都以结构体的形式在后台实现。

Swift 中所有的结构体和枚举都是值类型，这意味着你所创建的任何结构体和枚举实例——和实例作为属性所包含的任意值类型——在代码传递中总是被拷贝的。

看这个栗子，其使用了前面例子中的 `Resolution` 结构体：

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

这个栗子声明了一个叫 `hd` 的常量,并且赋予它一个以全高清视频(`1920`

像素宽乘以 1080 像素高)宽和高初始化的 Resolution 实例。

之后声明了一个叫 cinema 的变量并且以当前 hd 的值进行初始化。因为 Resolution 是一个结构体，现有实例的拷贝会被制作出来，然后这份新的拷贝就赋值给了 cinema。尽管 hd 和 cinema 有相同的像素宽和像素高，但是在后台中他们是两个完全不同的实例。

接下来，为了适应数字影院的放映需求（ 2048 像素宽和 1080 像素高），我们把 cinema 的属性 width 修改为稍宽一点的 2K 标准：

```
cinema.width = 2048
```

检查 cinema 的 width 属性发现已被改成 2048 ：

```
  println("cinema is now \(cinema.width) pixels wide")
  //println "cinema is now 2048 pixels wide"
```

总之，原始 hd 实例中的 width 属性依旧是 1920：

```
print("hd is still \(hd.width) pixels wide")
// prints "hd is still 1920 pixels wide"
```

当 cinema 被赋予 hd 的当前值，存储在 hd 中的*值*就被拷贝给了新的 cinema 实例。这最终的结果是两个完全不同的实例，它们只是碰巧包含了相同的数字值。由于它们是完全不同的实例， cinema 的宽度被设置 2048 并不影响 hd 中 width 存储的值。

这种行为规则同样适用于枚举：

```
enum CompassPoint {
    case North, South, East, West
}
var currentDirection = CompassPoint.West
let rememberedDirection = currentDirection
```

```
currentDirection = .East
if rememberedDirection == .West {
    print("The remembered direction is still .West")
}
// prints "The remembered direction is still .West"
```

当 `rememberedDirection` 被赋予了 `currentDirection` 中的值，实际上是值的拷贝。之后再改变 `currentDirection` 的值并不影响 `rememberedDirection` 所存储的原版值。

类是引用类型

不同于值类型，在*引用类型*被赋值到一个常量，变量或者本身被传递到一个函数的时候它是*不会*被拷贝的。相对于拷贝，这里使用的是同一个对现存实例的引用。

这里有个栗子，使用上面定义的 `VideoMode` 类：

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

这个栗子声明了一个新的名叫 `tenEighty` 的常量并且设置它引用一个 `VideoMode` 类的新实例，这个视频模式复制了之前的 `1920` 乘 `1080` 的HD分辨率。同时设置为隔行扫描，并且给予了一个名字“ `1080i` ”。最后，设置了 `25.0` 帧每秒的帧率。

然后， `tenEighty` 是赋给了一个名叫 `alsoEighty` 的新常量，并且将其帧率修改：

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

因为类是引用类型， `tenEighty` 和 `alsoTenEighty` 其实都是引用了*相同*的 `VideoMode` 实例。实际上，它们只是相同实例的两个不同命名罢了。

下面， `tenEighty` 的 `frameRate` 属性展示了它正确地显示了来自于 `VideoMode` 实例的新帧率：

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

注意 `tenEighty` 和 `alsoTenEighty` 都被声明为*常量*。然而，你仍然能改变 `tenEighty.frameRate` 和 `alsoTenEighty.frameRate` 因为 `tenEighty` 和 `alsoTenEighty` 常量本身的值不会改变。 `tenEighty` 和 `alsoTenEighty` 本身是并没有存储 `VideoMode` 实例—相反，它们两者都在后台*指向*了 `VideoMode` 实例。这是 `VideoMode` 的 `frameRate` 参数在改变而不是引用 `VideoMode` 的常量的值在改变。

特征运算符

因为类是引用类型，在后台有可能有很多常量和变量都是引用到了同一个类的实例。(相同这词对结构体和枚举来说并不是真的相同，因为它们在赋予给常量，变量或者被传递给一个函数时总是被拷贝过去的。)

有时候找出两个常量或者变量是否引用自同一个类实例非常有用，为了允许这样，Swift提供了两个特点运算符：

- 相同于 (`===`)
- 不相同于(`!==`)

利用这两个恒等运算符来检查两个常量或者变量是否引用相同的实例：

```
if tenEighty === alsoTenEighty {
    print("tenEighty and alsoTenEighty refer to the same Vi
```

```
deoMode instance.")
}
// prints "tenEighty and alsoTenEighty refer to the same VideoMode instance."
```

注意”相同于”(用三个等于号表示，或者说 `===`)这与”等于”的意义不同(用两个等于号表示，或者说 `==`)。

- “相同于”意味着两个类类型常量或者变量引用自相同的实例；
- “等于”意味着两个实例的在值上被视作“相等”或者“等价”，某种意义上的“相等”，就如同类设计者定义的那样。

当你定义了你自己的自定义类和结构体，你有义务来判定两个实例”相等”的标准。这个定义在你自己的”等于”和”不等于”实现的过程在相等运算符(此处应有链接)中有详细的介绍。

指针

如果你有过 C，C++ 或者 Objective-C 的经验，你可能知道这些语言使用可*指针*来引用内存中的地址。一个 Swift 的常量或者变量指向某个实例的引用类型和 C 中的指针类似，但是这并不是直接指向内存地址的指针，也不需要你书写星号(`*`)来明确你建立了一个引用。相反，这些引用被定义得就像 Swift 中其他常量或者变量一样。

类和结构体之间的选择

类和结构体都可以用来定义自定义的数据类型，作为你的程序代码构建块。

总之，结构体实例总是通过*值*来传递，而类实例总是通过*引用*来传递。这意味着他们分别适用于不同类型的任务。当你考虑你的工程项目中数据结构和功能的时候，你需要决定把每个数据结构定义成类还是结构体。

按照通用准则，当符合以下一条或多条情形时应考虑创建一个结构体：

- 结构体的主要目的是为了封装一些相关的简单数据值；
- 当你在赋予或者传递结构实例时，有理由需要封装的数据值被拷贝而不是引用；
- 任何存储在结构体中的属性是值类型，也将被拷贝而不是被引用；
- 结构体不需要从一个已存在类型继承属性或者行为。

合适的结构体候选者包括：

- 几何形状的大小，可能封装了一个 `width` 属性和 `height` 属性，两者都为 `double` 类型；
- 一定范围的路径，可能封装了一个 `start` 属性和 `length` 属性，两者为 `Int` 类型；
- 三维坐标系的一个点，可能封装了 `x` ，`y` 和 `z` 属性，他们都是 `double` 类型。

在其他的情况下，定义一个类，并创建这个类的实例通过引用来管理和传递。事实上，大部分的自定义的数据结构应该是类，而不是结构体。

字符串，数组和字典的赋值与拷贝行为

Swift 的 `String` ，`Array` 和 `Dictionary` 类型是作为结构体来实现的，这意味着字符串，数组和字典在它们被赋值到一个新的常量或者变量，亦或者它们本身被传递到一个函数或方法中的时候，其实是传递了拷贝。

这种行为不同于基础库中的 `NSString` ，`NSArray` 和 `NSDictionary` ，它们是作为类来实现的，而不是结构体。 `NSString` ，`NSArray` 和 `NSDictionary` 实例总是作为一个已存在实例的引用而不是拷贝来赋值和传递。

> 注意
>
> 在上述有关字符串，数组和字典“拷贝”的描述中。你在代码中所见到的行为好像总是拷贝。然

> 而在后台 Swift 只有在需要这么做时才会实际去拷贝。Swift 能够管理所有的值的拷贝来确保最佳的性能，所有你也没必要为了保证最佳性能来避免赋值。

译注

[1] `CamelCase` names： 在给储存器或者函数命名时我们习惯上把多个有意义的单词以开头大写的形式拼接在一起组成一个单一的长单词。这种方法被称为“*驼峰式命名法*”，又分为开头大写和开头小写两种。比如说 `SomeClass` 、 `frameRate` 等。
