



类和结构体

4.2 ▾

Swift 编程语言 / 类和结构体

这是一篇社区协同翻译的文章，你可以点击右边区块信息里的『改进』按钮向译者提交改进建议。

结构体和类是通用，灵活的设计，是你程序代码的结构基础。使用与你定义常量，变量和函数相同的语法来定义属性和方法以此为你的结构体和类添加功能。

并不像其他语言，Swift 不需要你为自定义结构体和类创建单独的接口与实现文件，Swift 中定义结构体和类在一个文件里，并且该类或结构体对其他代码会自动创建外部接口。

注意

一个类的实例通常被称为对象。Swift 的结构体和类在功能性上要比其他语言更接近，本章大部分叙述的功能都可以应用于类或结构体的实例。因此，使用了类和结构体都通用的术语实例。

比较结构体和类

Swift 中结构体和类有很多共同点，二者皆可：

- 定义属性以存储值
- 定义方法以提供功能
- 定义下标以提供下标语法访问其值
- 定义构造器以设置其初始化状态
- 通过扩展以增加默认实现功能
- 遵循协议以提供某种标准功能

更多信息请参阅 [属性](#)，[方法](#)，[下标](#)，[构造过程](#)，[扩展](#) 和 [协议](#)。

类有有一些结构体没有的额外功能：

- 继承让一个类可以继承另一个类的特征

Infinity

翻译进度

12

分块数量

3

参与人数



MorningStar 翻译于 5 个
前

0

重译

由 Summer 审阅



MorningStar 翻译于 5 个
前

0

重译

由 Summer 审阅



- 类型转换让你在运行时可以检查和解释一个类实例
- 析构器让一个类的实例可以释放任何被其所分配的资源
- 引用计数允许对一个类实例进行多次引用

更多信息请参阅 [继承](#)，[类型转换](#)，[析构过程](#) 和 [自动引用计数](#)。

使用类的额外功能其代价就是增加了复杂性。一般来说，更推荐结构体和枚举，因为他们更容易进行推断，并且适当或必要时使用类。实际上，这意味着将会使用结构体和枚举来定义大多数自定义数据类型。更多比较详情请参阅 [抉择在结构体和类之间](#)。

定义语法

结构体和类有相似的定义语法。使用 `struct` 关键字定义结构体、使用 `class` 关键字定义类。二者都在大括号中定义其具体内容：

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

注意

每当你定义一个新的结构体或类都是定义一个全新的 Swift 类型。请使用 `UpperCamelCase` 命名法（例如 `SomeStructure` 和 `SomeClass`）以符合大写命名风格的标准 Swift 类型（例如 `String`，`Int` 和 `Bool`）。对于属性和方法使用 `lowerCamelCase` 命名法（例如 `frameRate` 和 `incrementCount`）以此和类名区分。

以下是定义结构体和类的示例：

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



MorningStar 翻译于 5 个月前

0

重译

由 Summer 审阅

```
}
}
```

上面的例子定义了一个名为 `Resolution` 的结构体，用于描述基于像素的显示分辨率。这个结构体有两个存储属性分别名为 `width` 和 `height`。存储属性是与结构体或类绑定并存储为其一部分的常量或变量。由于设置这两个属性的初始值为 0 所以其类型被推断为 `Int`。

上面的例子还定义了一个名为 `VideoMode` 的类，用于描述视频的指定显示模式，这个类有四个为存储属性的变量。第一个，`resolution`，是 `Resolution` 结构体初始化的一个实例，其类型被推断为 `Resolution`。至于其他三个属性，新的 `VideoMode` 实例会把 `interlaced` 设置为 `false`（逐行扫描视频），播放帧数 `frameRate` 设置为 `0.0`，和一个名为 `name` 关联值类型是 `String` 的可选类型。由于是可选类型，`name` 默认值自动为 `nil`，或说『无 `name` 值』。

结构体与类实例

`Resolution` 结构体和 `VideoMode` 类的定义只描述了 `Resolution` 和 `VideoMode` 是什么样的。他们并没有描述一个特定的分辨率或视频模式。为此，我们需要创建其实例来指定分辨率或视频模式。

结构体和类创建实例的语法非常相似：

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

结构体和类都可以使用构造语法创建新的实例，最简单形式的构造语法就是类或结构体的类型名称之后尾随空括号，例如 `Resolution()` 或 `VideoMode()`。这样就创建了一个类或结构体的实例，并且所有属性全部初始化为默认值。[构造过程](#) 中更详尽的叙述了类和结构体构造过程。

访问属性

你可以使用点语法来访问一个实例的属性。点语法中，在实例名的后面直接写属性名，用（`.`）来分割。

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

这个例子中，`someResolution.width` 就是 `someResolution` 中的 `width`，返回其默认值 `0`。



MorningStar 翻译于 5 个月前

👍 0 重译

由 Summer 审阅



MorningStar 翻译于 5 个月前

👍 0 重译

由 Summer 审阅

你也可深入访问子属性，例如 `VideoMode` 的 `resolution` 属性的 `width` 的属性：

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

你也可以使用点语法给变量属性赋值：

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

结构体类型的成员构造器

所有结构体都有一个用于初始化结构体实例的成员属性，并且是自动生成的*成员构造器*。实例属性的初始化值通过属性名称传递到成员构造器中：

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

与结构体不同，类没有默认的成员构造器，关于构造器更多详情请参阅 [构造过程](#)。

值类型的结构体和枚举

*值类型*是一种赋值给变量或常量，或传递给函数时，值会被*拷贝*的类型。

其实你在之前的章节中已广泛的使用了值类型。其实 Swift 中的所有基本类型 --- 整数，浮点数，布尔，字符串，数组和字典 --- 它们都是值类型，其底层也是以结构体实现的。

Swift 中所有的结构体和枚举都是值类型。这意味着在代码中你创建的任何结构体或枚举的实例 --- 及其任何值类型的属性 --- 都会在传递时被拷贝。

注意

标准库所定义的集合例如数组，字典和字符串都进行了优化以减少拷贝时的性能开销。这些集合不是直接复制，而是在原始实例和所有副本之间共享内存。如果集合的任意一个副本被修改，则会在修改之前复制该元素。代码中这种行为看似好像立即发生。

这个示例用了上面的 `Resolution` 结构体：



MorningStar 翻译于 5 个月前

0

重译

由 Aufree 审阅



MorningStar 翻译于 5 个月前

👍 0 重译

由 Aufree 审阅

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

声明了一个名为 `hd` 的常量并使用全高清视频的宽高（1920 像素宽，1080 像素高）将其初始化为 `Resolution` 的实例。

还声明了一个名为 `cinema` 的变量并使用当前 `hd` 的值为其赋值。因为 `Resolution` 是一个结构体，所以会制作一个当前实例的副本赋值给 `cinema`。虽然 `hd` 和 `cinema` 现在有同样的宽高，但是他们在底层是完全不同的两个实例。

接下来，将 `cinema` 的属性 `width` 修改为略宽一点的数字影院放映的 2 K 标准宽度（2048 像素宽和 1080 像素高）

```
cinema.width = 2048
```

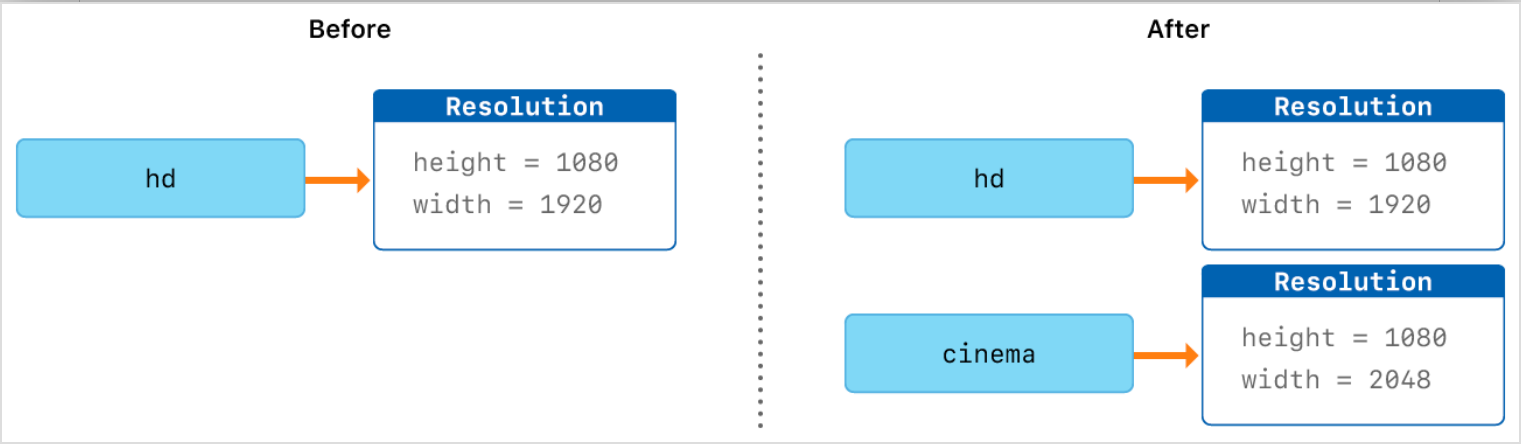
查看 `cinema` 的属性 `width` 会发现已经改成了 `2048`：

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

而原始 `hd` 实例的 `width` 属性还是之前的值 `1920`：

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

当 `hd` 赋值给 `cinema` 时，存储在 `hd` 中的值就拷贝给了新的 `cinema` 实例。最终结果就是相同数值但完全独立的两个实例。由于他们完全独立，所以设置 `cinema` 的宽度为 2048 并不会影响 `hd` 中的存储，如下图所示：



枚举也适用于同样的行为准则：

```
enum CompassPoint {
```



MorningStar 翻译于 5 个月前

👍 0 重译

由 Aufree 审阅


```
case north, south, east, west

mutating func turnNorth() {
    self = .north
}

}

var currentDirection = CompassPoint.west
let rememberedDirection = currentDirection
currentDirection.turnNorth()

print("The current direction is \(currentDirection)")
print("The remembered direction is \(rememberedDirection)")

// 打印 "The current direction is north"
// 打印 "The remembered direction is west"
```

当 `currentDirection` 的值赋值给 `rememberedDirection`，实际上也是值拷贝。因此改变 `currentDirection` 的值并不会影响存储在 `rememberedDirection` 中原始值的副本。

类是引用类型

与值类型不同，赋值给变量或常量，或是传递给函数时，引用类型并不会拷贝。引用的不是副本而是已经存在的实例。

下面这个例子使用了之前定义的 `VideoMode` 类。

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

这个例子声明了一个名为 `tenEighty` 的常量并将其设置为引用 `VideoMode` 类的实例。用之前的 `1920 * 1080` 的高清分辨率的副本赋给视频模式。将其命名为 `"1080i"` 并设置为隔行扫描。最后设置帧率为每秒 `25.0` 帧。

然后将 `tenEighty` 赋值给一个名为 `alsoTenEighty` 的新常量，同时修改其帧率：

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

因为类是引用类型，所以其实 `tenEighty` and `alsoTenEighty` 引用了同一



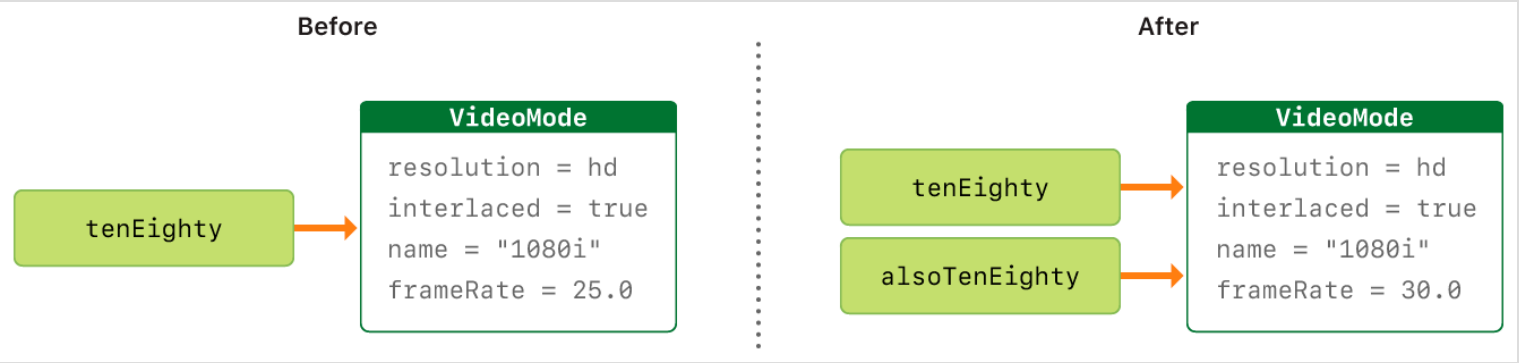
MorningStar 翻译于 5 个月前

👍 0

重译

由 Aufree 审阅

个 `VideoMode` 的实例。实际上，他们只是两个不同名字的*相同*实例，如下图所示：



查看 `frameRate` 的属性 `tenEighty`，会发现它正确的引用了 `VideoMode` 实例的新帧率 `30.0`：

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

以上的例子还显示了引用类型推断有多费劲。如果 `tenEighty` 和 `alsoTenEighty` 在你的代码中相距甚远，那么可能很难找到改变视频模式的所有地方。无论你在哪里使用 `tenEighty`，都需要考虑用到 `alsoTenEighty` 的代码，反之亦然。相反，值类型就很好推断，因为在你的源文件中相同值相互作用的所有代码是紧密相连的。

注意 `tenEighty` 和 `alsoTenEighty` 声明的是*常量*而不是变量。但是你仍然可以改变 `tenEighty.frameRate` 和 `alsoTenEighty.frameRate`，因为常量 `tenEighty` 和 `alsoTenEighty` 的值自身实际上没有改变。`tenEighty` 和 `alsoTenEighty` 本身并不存储 `VideoMode` 的实例，他们只是在底层引用了 `VideoMode` 的实例。改变的是 `VideoMode` 的属性 `frameRate`，而不是引用 `VideoMode` 的常量的值。

恒等运算符

因为类是引用类型，在底层可能多个常量和变量引用同一个类的实例。（同样的理论对结构体和枚举来说并不有效，因为它们赋值给常量或变量，或是传递给函数时，总是拷贝的。）

有时找出两个常量或变量是否引用同一个类的实例很有帮助。为此，Swift 提供了恒等运算符：

- 等价于 (`===`)
- 非等价 (`!==`)

使用他们来检查两个常量或变量是否引用同一个实例：



MorningStar 翻译于 5 个月前

0

重译

由 Aufree 审阅



MorningStar 翻译于 5 个月前

0

重译

由 Aufree 审阅

```
if tenEighty === alsoTenEighty {  
    print("tenEighty and alsoTenEighty refer to the same VideoMode instance.")  
}  
  
// 打印 "tenEighty and alsoTenEighty refer to the same VideoMode instance."
```

注意等价于（用三个等号表示 `===`）和等于（用两个等号表示 `==`）完全不是一回事。等价于意思是两个常量或变量完全引用相同的类实例。等于的意思是两个实例某种意义上的值相等或相同，就像类型设计者定义的那样。

当你自定义结构体或类时，你有责任决定两个实例相等的标准。在 [等价运算符](#) 中介绍了实现自定义『等于』和『非等于』的流程。

指针

如果你有过 C，C++，或 Objective-C 的经验，你或许知道这些语言使用 *指针* 来指向内存中的地址。指向某种引用类型实例的 Swift 常量或变量和 C 中的指针类似，但是并不直接指向内存地址，你也不需要写星号（`*`）来表示创建了一个引用。定义引用和 Swift 中的其他 常量或变量一样。如果你需要直接与指针交互标准库提供了指针和 `buffer` 类型 --- 请参阅 [手动内存管理](#)。

本文中的所有译文仅用于学习和交流目的，转载请务必注明文章译者、出处、和本文链接

我们的翻译工作遵照 [CC 协议](#)，如果我们的工作有侵犯到您的权益，请及时联系我们。



MorningStar 翻译于 5 个月前

👍 0

重译

由 Aufree 审阅

← 上一篇

下一篇 →

👍 点赞



👤 参与译者：3



更多职位





讨论数量: 0

发起讨论

☐ 只看当前版本讨论

暂无话题~


兄弟社区

-  Laravel China
-  PythonCaff.com
-  GolangCaff.com
-  VuejsCaff.com

资源推荐

资源推荐

其他信息

-  软件外包
-  商务合作
-  联系站长

由 Summer 设计和编码 ❤