类和结构

*类*和*结构*是通用的，灵活的构造，它们成为程序代码的构建块。您可以使用与常量，变量和函数完全相同的语法来定义属性和方法，从而为您的类和结构添加功能。

与其他编程语言不同，Swift不要求您为自定义类和结构创建单独的接口和实现文件。在Swift中，您可以在单个文件中定义一个类或结构，并且该类或结构的外部接口会自动提供给其他代码使用。

> 注意
>
> 传统上将*类*的实例称为*对象*。然而，斯威夫特类和结构在功能上比其他语言更接近，而很多本章介绍了可以应用到的实例功能，*无论是*类或结构类型。因此，使用更一般的术语*实例*。

比较类和结构

Swift中的类和结构有许多共同之处。两者都可以：

- 定义属性以存储值
- 定义提供功能的方法
- 使用下标语法定义下标以提供对其值的访问
- 定义初始化程序以设置其初始状态
- 扩展到超出默认实现范围的功能
- 符合协议以提供某种标准功能

有关更多信息，请参阅属性，方法，下标，初始化，扩展和协议。

类具有结构不具有的其他功能：

- 继承使一个类能够继承另一个类的特性。
- 类型转换使您能够在运行时检查和解释类实例的类型。
- 去初始化器使类的一个实例释放它分配的任何资源。
- 引用计数允许对一个类实例的多个引用。

有关更多信息，请参阅继承，类型转换，取消初始化和自动引用计数。

> 注意
>
> 结构在代码中传递时总是被复制，并且不使用引用计数。

定义语法

类和结构具有类似的定义语法。您可以使用`class`关键字引入具有关键字和结构的类`struct`。两者都将其整个定义放在一对大括号中：

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

> 注意
>
> 每当你定义一个新的类或结构时，你都可以有效地定义一个全新的Swift类型。给出类型`UpperCamelCase`名称（如`SomeClass`和`SomeStructure`这里）来匹配标准斯威夫特类型的资本（如`String`，`Int`和`Bool`）。相反，总是给属性和方法`lowerCamelCase`名称（如`frameRate`和`incrementCount`）来区分它们和类型名称。

在本页

下面是一个结构定义和类定义的例子：

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

上面的例子定义了一个称为的新结构`Resolution`来描述基于像素的显示分辨率。这个结构有两个存储的属性，称为`width`和`height`。存储属性是常量或变量，它们被捆绑并存储为类或结构的一部分。通过将这两个属性`Int`设置为初始整数值，可以将这两个属性推断为类型`0`。

上面的例子还定义了一个新类`VideoMode`，用于描述视频显示的特定视频模式。这个类有四个变量存储的属性。第一个，`resolution`是用一个新的`Resolution`结构实例初始化的，它推断出一个属性类型`Resolution`。对于其他三个属性，`VideoMode`将使用（意思是“非隔行视频”）`interlaced`设置来初始化新实例`false`，播放帧速率为`0.0`，以及可选`String`值为`name`。该`name`属性会自动给出默认值`nil`或“无`name`值”，因为它是可选类型。

类和结构实例

该`Resolution`结构定义和`VideoMode`类定义只说明什么`Resolution`或`VideoMode`看起来像。他们自己没有描述特定的分辨率或视频模式。要做到这一点，你需要创建一个结构或类的实例。

创建实例的语法对于结构和类都非常相似：

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

结构和类都为新实例使用初始化语法。最简单的初始化语法形式使用类或名称的结构，后跟空括号，如`Resolution()`或`VideoMode()`。这将创建类或结构的新实例，并将任何属性初始化为默认值。类和结构初始化在初始化中有更详细的描述。

访问属性

您可以使用*点语法*访问实例的属性。在点语法中，可以在实例名称后面立即写入属性名称，并用句点（`.`）分隔，而不带任何空格：

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

在这个例子中，`someResolution.width`指的是`width`属性`someResolution`，并返回它的默认初始值`0`。

您可以深入查看子属性，例如a `width`属性中的`resolution`属性`VideoMode`：

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

您也可以使用点语法为变量属性指定一个新值：

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

> 注意
>
> 与Objective-C不同的是，Swift使您能够直接设置结构属性的子属性。在上面的最后一个例子中，`width`属性`resolution`property `someVideoMode`是直接设置的，不需要将整个`resolution`属性设置为新值。

结构类型的成员初始化程序

所有结构都有一个自动生成的*成员初始化程序*，您可以使用它初始化新结构实例的成员属性。新实例属性的初始值可以按名称：

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

与结构不同，类实例不会接收默认的成员初始值设定项。初始化中更详细地描述在初始化。

结构和枚举是值类型

甲*值类型*是一个类型，其值被*拷贝*时，它被分配给一个变量或常数，或当它被传递给函数。

在前面的章节中，您实际上已经广泛使用了值类型。事实上，Swift整数，浮点数，布尔值，字符串，数组和字典中的所有基本类型都是值类型，并在后台实现为结构。

所有结构和枚举都是Swift中的值类型。这意味着您创建的任何结构和枚举实例以及它们作为属性具有的任何值类型都会在您的代码中传递时始终进行复制。

考虑这个例子，它使用Resolution了前面例子中的结构：

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

此示例声明了一个常量hd，并将其设置为使用Resolution全高清视频的宽度和高度（1920像素宽1080高像素）初始化的实例。

然后它声明一个变量cinema，并将其设置为当前值hd。因为Resolution是一个结构，现有实例的一个*副本*被创建，并且这个新副本被分配给cinema。虽然hd和cinema现在有相同的宽度和高度，他们是幕后两种完全不同的情况。

接下来，将width属性cinema修改为用于数字电影投影的宽度稍宽的2K标准（2048像素宽和1080像素高）：

```
cinema.width = 2048
```

检查width属性cinema显示它确实已更改为2048：

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

但是，width原始hd实例的属性仍具有以下旧值1920：

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

当cinema给出当前值时hd，存储在其中的*值*hd被复制到新cinema实例中。最终结果是两个完全分离的实例，它们恰好包含相同的数值。由于它们是单独的实例，因此设置宽度cinema以2048不影响存储的宽度hd。

相同的行为适用于枚举：

```
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赋值的时候currentDirection，它实际上被设置为该值的一个副本。currentDirection此后更改此值不会影响存储在其中的原始值的副本rememberedDirection。

类是引用类型

与值类型不同，
同现有实例的引用。

下面是一个使用VideoMode上面定义的类的示例：

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

本示例声明了一个新的常量tenEighty，并将其设置为引用VideoMode该类的新实例。视频模式被分配的HD分辨率的副本，1920通过1080从之前。它被设置为隔行扫描，并被命名为"1080i"。最后，它被设置为25.0每秒帧数的帧速率。

接下来，tenEighty将其分配给一个新的常量，并调用alsoTenEighty帧速率alsoTenEighty：

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

因为类是引用类型，tenEighty并且alsoTenEighty实际上都引用*同*一个VideoMode实例。实际上，它们只是同一个实例的两个不同名称。

检查frameRate属性tenEighty显示它正确报告30.0来自底层VideoMode实例的新帧速率：

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

请注意，tenEighty并alsoTenEighty声明为*常量*，而不是变量。但是，你仍然可以改变tenEighty.frameRate，alsoTenEighty.frameRate因为常量tenEighty和alsoTenEighty常量的值本身并没有改变。tenEighty并且alsoTenEighty他们自己不“存储” VideoMode实例 - 相反，它们都*指向*VideoMode幕后的实例。它是frameRate底层的属性VideoMode被更改，而不是常量引用的值VideoMode。

身份运营商

由于类是引用类型，因此多个常量和变量可能会在幕后引用同一个类的单个实例。（结构和枚举也是如此，因为它们在分配给常量或变量或传递给函数时总是被复制。）

找出两个常量或变量是否指向一个类的完全相同的实例有时会很有用。为了实现这一点，Swift提供了两个身份运算符：

- 与（===） 相同
- 与（!==） 不相同

使用这些运算符来检查两个常量或变量是否引用同一个单一实例：

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

请注意，“与.....相同”（用三个等号表示，或者===）并不等同于“等于”（用两个等号表示==）：

- “与.....相同”表示类型的两个常量或变量指向完全相同的类实例。
- “等于”意味着两个实例在值中被认为是“相等的”或“等价的”，对于类型的设计者定义的“相等”的某些适当的含义。

当您定义自己的自定义类和结构时，您有责任决定两个“平等”实例的合格性。在等价运算符中描述定义您自己的“等于”和“不等于”运算符的实现的过程。

指针

如果您有使用C，C ++或Objective-C的经验，您可能会知道这些语言使用*指针*来引用内存中的地址。引用某个引用类型的实例的Swift常量或变量类似于C中的指针，但不是指向内存中某个地址的直接指针，也不需要写一个asterisk（*）来指示您是创造一个参考。相反，这些引用是像Swift中的其他常量或变量一样定义的。

选择类和结构

您可以使用类和结构来定义自定义数据类型，以用作程序代码的构建块。

但是，结构实例总是按*值*传递，而类实例总是按*引用*传递。这意味着它们适合于不同类型的任务。当您考虑项目所需的数据结构和功能时，请决定是将每个数据结构定义为类还是结构。

作为一般指导原则，考虑在适用以下一个或多个条件时创建结构：

- 该结构的主要目的是封装一些相对简单的数据值。
- 当分配或传递该结构的实例时，期望封装值将被复制而不是被引用是合理的。
- 结构存储的任何属性都是它们自己的值类型，也可能被复制而不是引用。
- 该结构不需要继承其他现有类型的属性或行为。

良好的结构候选人的例子包括：

- 几何形状的大小，也许封装一个`width`属性和一个`height`属性，都是类型`Double`。
- 一种引用一系列范围内的范围的方法，也许封装一个`start`属性和一个`length`属性，两者都是类型`Int`。
- 3D坐标系中的一个点，可能是封装`x`，`y`以及`z`每个类型的属性`Double`。

在所有其他情况下，定义一个类，并创建该类的实例，以便通过引用进行管理和传递。实际上，这意味着大多数自定义数据结构应该是类而不是结构。

字符串，数组和字典的赋值和复制行为

在夫特，许多基本的数据类型，如`String`，`Array`以及`Dictionary`被实现为结构。这意味着如果将字符串，数组和字典等数据分配给新的常量或变量，或者将它们传递给函数或方法时，它们将被复制。

此行为是不同的基金：`NSString`，`NSArray`，和`NSDictionary`为类，而不是结构来实现。基金会中的字符串，数组和字典始终作为对现有实例的引用进行分配和传递，而不是作为副本。

> 注意
>
> 上面的描述涉及字符串，数组和字典的“复制”。您在代码中看到的行为总是会像发生副本一样。但是，当绝对必要时，Swift仅在幕后执行*实际的*副本。Swift管理所有值复制以确保最佳性能，并且您不应该避免分配尝试抢占此优化。