翻译：JaySurplus　校对：sg552

类和结构体

本页包含内容：

类和结构体是人们构建代码所用的一种通用且灵活的构造体。为了在类和结构体中实现各种功能，我们必须要严格按照对于常量，变量以及函数所规定的语法规则来定义属性和添加方法。

与其他编程语言所不同的是，Swift 并不要求你为自定义类和结构去创建独立的接口和实现文件。你所要做的是在一个单一文件中定义一个类或者结构体，系统将会自动生成面向其它代码的外部接口。

注意：

通常一个类的实例被称为对象。然而在Swift 中，类和结构体的关系要比在其他语言中更加的密切，本章中所讨论的大部分功能都可以用在类和结构体上。因此，我们会主要使用实例而不是对象。

类和结构体对比

Swift 中类和结构体有很多共同点。共同处在于：

- 定义属性用于储存值
- 定义方法用于提供功能
- 定义下标用于通过下标语法访问值
- 定义初始化器用于生成初始化值
- 通过扩展以增加默认实现的功能
- 符合协议以对某类提供标准功能

更多信息请参见 属性，方法，下标，初始过程，扩展，和协议。

与结构体相比，类还有如下的附加功能：

- 继承允许一个类继承另一个类的特征
- 类型转换允许在运行时检查和解释一个类实例的类型
- 取消初始化器允许一个类实例释放任何其所被分配的资源
- 引用计数允许对一个类的多次引用

更多信息请参见继承，类型转换，初始化，和自动引用计数。

注意：

结构体总是通过被复制的方式在代码中传递，因此请不要使用引用计数。

定义

类和结构体有着类似的定义方式。我们通过关键字`class`和`struct`来分别表示类和结构体，并在一对大括号中定义它们的具体内容：

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

注意：

在你每次定义一个新类或者结构体的时候，实际上你是有效地定义了一个新的 Swift 类型。因此请使用 `UpperCamelCase` 这种方式来命名（如 `SomeClass` 和`SomeStructure`等），以便符合标准Swift 类型的大写命名风格（如`String`，`Int`和`Bool`）。相反的，请使用`lowerCamelCase`这种方式为属性和方法命名（如`framerate`和`incrementCount`），以便和类区分。

以下是定义结构体和定义类的示例：

```
struct Resolution {
    var width = 0
    var heigth = 0
```

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

在上面的示例中我们定义了一个名为`Resolution`的结构体，用来描述一个显示器的像素分辨率。这个结构体包含了两个名为`width`和`height`的储存属性。储存属性是捆绑和储存在类或结构体中的常量或变量。当这两个属性被初始化为整数`0`的时候，它们会被推断为`Int`类型。

在上面的示例中我们还定义了一个名为`VideoMode`的类，用来描述一个视频显示器的特定模式。这个类包含了四个储存属性变量。第一个是分辨率，它被初始化为一个新的`Resolution`结构体的实例，具有`Resolution`的属性类型。新`VideoMode`实例同时还会初始化其它三个属性，它们分别是，初始值为`false`(意为“non-interlaced video”)的`inteflaced`，回放帧率初始值为`0.0`的`frameRate`和值为可选`String`的`name`。`name`属性会被自动赋予一个默认值`nil`，意为“没有`name`值”，因它是一个可选类型。

类和结构体实例

`Resolution`结构体和`VideoMode`类的定义仅描述了什么是`Resolution`和`VideoMode`。它们并没有描述一个特定的分辨率（resolution）或者视频模式（video mode）。为了描述一个特定的分辨率或者视频模式，我们需要生成一个它们的实例。

生成结构体和类实例的语法非常相似：

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

结构体和类都使用初始化器语法来生成新的实例。初始化器语法的最简单形式是在结构体或者类的类型名称后跟随一个空括弧，如`Resolution()`或`VideoMode()`。通过这种方式所创建的类或者结构体实例，其属均会被初始化为默认值。构造过程章节会对类和结构体的初始化进行更详细的讨论。

属性访问

通过使用点语法（*dot syntax*）,你可以访问实例中所含有的属性。其语法规则是，实例名后面紧跟属性名，两者通过点号(.)连接：

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

在上面的例子中，`someResolution.width`引用`someResolution`的`width`属性，返回`width`的初始值`0`。

你也可以访问子属性，如何`VideoMode`中`Resolution`属性的`width`属性：

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

你也可以使用点语法为属性变量赋值：

```
someVideoMode.resolution.width = 12880
println("The width of someVideoMode is now \
(someVideoMode.resolution.width)")
// 输出 "The width of someVideoMode is now 1280"
```

注意：

与 Objective-C 语言不同的是，Swift 允许直接设置结构体属性的子属性。上面的最后一个例子，就是直接设置了`someVideoMode`中`resolution`属性的`width`这个子属性，以上操作并不需要从新设置`resolution`属性。

结构体类型的成员逐一初始化器

//Memberwise Initializers for structure Types

所有结构体都有一个自动生成的成员逐一初始化器，用于初始化新结构体实例中成员的属性。新实例中各个属性的初始值可以通过属性的名称传递到成员逐一初始化器之中：

```
let vga = resolution（width:640, heigth: 480）
```

与结构体不同，类实例没有默认的成员逐一初始化器。构造过程章节会对

初始化器进行更详细的讨论。

结构体和枚举是值类型

值类型被赋予给一个变量，常数或者本身被传递给一个函数的时候，实际上操作的是其的拷贝。

在之前的章节中，我们已经大量使用了值类型。实际上，在 Swift 中，所有的基本类型：整数(Integer)、浮点数(floating-point)、布尔值(Booleans)、字符串(string)、数组(array)和字典(dictionaries)，都是值类型，并且都是以结构体的形式在后台所实现。

在 Swift 中，所有的结构体和枚举都是值类型。这意味着它们的实例，以及实例中所包含的任何值类型属性，在代码中传递的时候都会被复制。

请看下面这个示例，其使用了前一个示例中`Resolution`结构体：

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

在以上示例中，声明了一个名为`hd`的常量，其值为一个初始化为全高清视频分辨率(1920 像素宽，1080 像素高)的`Resolution`实例。

然后示例中又声明了一个名为`cinema`的变量，其值为之前声明的`hd`。因为`Resolution`是一个结构体，所以`cinema`的值其实是`hd`的一个拷贝副本，而不是`hd`本身。尽管`hd`和`cinema`有着相同的宽(width)和高(height)属性，但是在后台中，它们是两个完全不同的实例。

下面，为了符合数码影院放映的需求(2048 像素宽，1080 像素高)，`cinema`的`width`属性需要作如下修改：

```
cinema.width = 2048
```

这里，将会显示`cinema`的`width`属性确已改为了`2048`：

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

然而，初始的hd实例中width属性还是1920：

```
println("hd is still \(hd.width    ) pixels wide")
// 输出 "hd is still 1920 pixels wide"
```

在将hd赋予给cinema的时候，实际上是将hd中所储存的值(values)进行拷贝，然后将拷贝的数据储存到新的cinema实例中。结果就是两个完全独立的实例碰巧包含有相同的数值。由于两者相互独立，因此将cinema的width修改为2048并不会影响hd中的宽(width)。

枚举也遵循相同的行为准则：

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

上例中rememberedDirection被赋予了currentDirection的值(value)，实际上它被赋予的是值(value)的一个拷贝。赋值过程结束后再修改currentDirection的值并不影响rememberedDirection所储存的原始值(value)的拷贝。

类是引用类型

与值类型不同，引用类型在被赋予到一个变量，常量或者被传递到一个函数时，操作的并不是其拷贝。因此，引用的是已存在的实例本身而不是其拷贝。

请看下面这个示例，其使用了之前定义的VideoMode类：

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

以上示例中，声明了一个名为tenEighty的常量，其引用了一个VideoMode类的新实例。在之前的示例中，这个视频模式(video mode)被赋予了HD分辨率(1920*1080)的一个拷贝(hd)。同时设置为交错(interlaced),命名为“1080i”。最后，其帧率是25.0帧每秒。

然后，tenEighty 被赋予名为alsoTenEighty的新常量，同时对alsoTenEighty的帧率进行修改：

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

因为类是引用类型，所以tenEight和alsoTenEight实际上引用的是相同的VideoMode实例。换句话说，它们只是同一个实例的两种叫法。

下面，通过查看tenEighty的frameRate属性，我们会发现它正确的显示了基本VideoMode实例的新帧率，其值为30.0：

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

需要注意的是tenEighty和alsoTenEighty被声明为常量*(constants)*而不是变量。然而你依然可以改变tenEighty.frameRate和alsoTenEighty.frameRate,因为这两个常量本身不会改变。它们并不储存这个VideoMode实例，在后台仅仅是对VideoMode实例的引用。所以，改变的是被引用的基础VideoMode的frameRate参数，而不改变常量的值。

恒等运算符

因为类是引用类型，有可能有多个常量和变量在后台同时引用某一个类实例。(对于结构体和枚举来说，这并不成立。因为它们作值类型，在被赋

予到常量，变量或者传递到函数时，总是会被拷贝。）

如果能够判定两个常量或者变量是否引用同一个类实例将会很有帮助。为了达到这个目的，Swift 内建了两个恒等运算符：

- 等价于（===）
- 不等价于（!==）

以下是运用这两个运算符检测两个常量或者变量是否引用同一个实例：

```
if tenEighty === alsoTenTighty {
    println("tenTighty and alsoTenEighty refer to the same Resolution instance.")
}
//输出 "tenEighty and alsoTenEighty refer to the same Resolution instance."
```

请注意“等价于”(用三个等号表示，===) 与“等于”(用两个等号表示，==)的不同：

- “等价于”表示两个类类型(class type)的常量或者变量引用同一个类实例。
- “等于”表示两个实例的值“相等”或“相同”，判定时要遵照类设计者定义定义的评判标准，因此相比于“相等”，这是一种更加合适的叫法。

当你在定义你的自定义类和结构体的时候，你有义务来决定判定两个实例“相等”的标准。在章节运算符函数(Operator Functions)中将会详细介绍实现自定义“等于”和“不等于”运算符的流程。

指针

如果你有 C，C++ 或者 Objective-C 语言的经验，那么你也许会知道这些语言使用指针来引用内存中的地址。一个 Swift 常量或者变量引用一个引用类型的实例与C语言中的指针类似，不同的是并不直接指向内存中的某个地址，而且也不要求你使用星号(*)来表明你在创建一个引用。Swift 中这些引用与其它的常量或变量的定义方式相同。

类和结构体的选择

在你的代码中，你可以使用类和结构体来定义你的自定义数据类型。

然而，结构体实例总是通过值传递，类实例总是通过引用传递。这意味两者适用不同的任务。当你的在考虑一个工程项目的数据构造和功能的时候，你需要决定每个数据构造是定义成类还是结构体。

按照通用的准则，当符合一条或多条以下条件时，请考虑构建结构体：

- 结构体的主要目的是用来封装少量相关简单数据值。
- 有理由预计一个结构体实例在赋值或传递时，封装的数据将会被拷贝而不是被引用。
- 任何在结构体中储存的值类型属性，也将会被拷贝，而不是被引用。
- 结构体不需要去继承另一个已存在类型的属性或者行为。

合适的结构体候选者包括：

- 几何形状的大小，封装一个`width`属性和`height`属性，两者均为`Double`类型。
- 一定范围内的路径，封装一个`start`属性和`length`属性，两者均为`Int`类型。
- 三维坐标系内一点，封装`x`，`y`和`z`属性，三者均为`Double`类型。

在所有其它案例中，定义一个类，生成一个它的实例，并通过引用来管理和传递。实际中，这意味着绝大部分的自定义数据构造都应该是类，而非结构体。

集合(Collection)类型的赋值和拷贝行为

Swift 中`数组(Array)`和`字典(Dictionary)`类型均以结构体的形式实现。然而当数组被赋予一个常量或变量，或被传递给一个函数或方法时，其拷

贝行为与字典和其它结构体有些许不同。

以下对数组和结构体的行为描述与对NSArray和NSDictionary的行为描述在本质上不同，后者是以类的形式实现，前者是以结构体的形式实现。NSArray和NSDictionary实例总是以对已有实例引用,而不是拷贝的方式被赋值和传递。

注意:

以下是对于数组，字典，字符串和其它值的拷贝的描述。 在你的代码中，拷贝好像是确实是在有拷贝行为的地方产生过。然而，在Swift 的后台中，只有确有必要，实际(actual)拷贝才会被执行。Swift 管理所有的值拷贝以确保性能最优化的性能，所以你也没有必要去避免赋值以保证最优性能。(实际赋值由系统管理优化)

字典类型的赋值和拷贝行为

无论何时将一个字典实例赋给一个常量或变量，或者传递给一个函数或方法，这个字典会即会在赋值或调用发生时被拷贝。在章节结构体和枚举是值类型中将会对此过程进行详细介绍。

如果字典实例中所储存的键(keys)和/或值(values)是值类型(结构体或枚举)，当赋值或调用发生时，它们都会被拷贝。相反，如果键(keys)和/或值(values)是引用类型，被拷贝的将会是引用，而不是被它们引用的类实例或函数。字典的键和值的拷贝行为与结构体所储存的属性的拷贝行为相同。

下面的示例定义了一个名为ages的字典，其中储存了四个人的名字和年龄。ages字典被赋予了一个名为copiedAges的新变量，同时ages在赋值的过程中被拷贝。赋值结束后，ages和copiedAges成为两个相互独立的字典。

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

这个字典的键(keys)是字符串(String)类型，值(values)是整(Int)类型。这两种类型在Swift 中都是值类型(value types)，所以当字典被拷贝时，两

者都会被拷贝。

我们可以通过改变一个字典中的年龄值(age value)，检查另一个字典中所对应的值，来证明ages字典确实是被拷贝了。如果在copiedAges字典中将Peter的值设为24，那么ages字典仍然会返回修改前的值23：

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

数组的赋值和拷贝行为

在Swift 中，数组(Arrays)类型的赋值和拷贝行为要比字典(Dictionary)类型的复杂的多。当操作数组内容时，数组(Array)能提供接近C语言的的性能，并且拷贝行为只有在必要时才会发生。

如果你将一个数组(Array)实例赋给一个变量或常量，或者将其作为参数传递给函数或方法调用，在事件发生时数组的内容不会被拷贝。相反，数组公用相同的元素序列。当你在一个数组内修改某一元素，修改结果也会在另一数组显示。

对数组来说，拷贝行为仅仅当操作有可能修改数组长度时才会发生。这种行为包括了附加(appending),插入(inserting),删除(removing)或者使用范围下标(ranged subscript)去替换这一范围内的元素。只有当数组拷贝确要发生时，数组内容的行为规则与字典中键值的相同，参见章节[集合（collection）类型的赋值与复制行为](#assignment_and_copy_behavior_for_collection_types。

下面的示例将一个整数(Int)数组赋给了一个名为a的变量，继而又被赋给了变量b和c：

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

我们可以在a,b,c上使用下标语法以得到数组的第一个元素：

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

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

如果通过下标语法修改数组中某一元素的值，那么`a`,`b`,`c`中的相应值都会发生改变。请注意当你用下标语法修改某一值时，并没有拷贝行为伴随发生，因为下表语法修改值时没有改变数组长度的可能：

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

然而，当你给`a`附加新元素时，数组的长度会改变。 当附加元素这一事件发生时，Swift 会创建这个数组的一个拷贝。从此以后，`a`将会是原数组的一个独立拷贝。

拷贝发生后，如果再修改`a`中元素值的话，`a`将会返回与`b`，`c`不同的结果，因为后两者引用的是原来的数组：

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

确保数组的唯一性

在操作一个数组，或将其传递给函数以及方法调用之前是很有必要先确定这个数组是有一个唯一拷贝的。通过在数组变量上调用`unshare`方法来确定数组引用的唯一性。(当数组赋给常量时，不能调用`unshare`方法)

如果一个数组被多个变量引用，在其中的一个变量上调用`unshare`方法，

则会拷贝此数组，此时这个变量将会有属于它自己的独立数组拷贝。当数组仅被一个变量引用时，则不会有拷贝发生。

在上一个示例的最后，b和c都引用了同一个数组。此时在b上调用unshare方法则会将b变成一个唯一个拷贝：

```
b.unshare()
```

在unshare方法调用后再修改b中第一个元素的值，这三个数组(a,b,c)会返回不同的三个值：

```
b[0] = -105
println(a[0])
// 77
println(b[0])
// -105
println(c[0])
// 42
```

判定两个数组是否共用相同元素

我们通过使用恒等运算符(identity operators)(=== and !==)来判定两个数组或子数组共用相同的储存空间或元素。

下面这个示例使用了“恒等于(identical to)” 运算符(===) 来判定b和c是否共用相同的数组元素：

```
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.")
}

// 输出 "b and c now refer totwo independent sets of
array elements."
```

此外，我们还可以使用恒等运算符来判定两个子数组是否共用相同的元素。下面这个示例中，比较了b的两个相等的子数组，并且确定了这两个

子数组都引用相同的元素：

```
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.")
}
// 输出 "These two subarrays share the same elements."
```

强制复制数组

我们通过调用数组的`copy`方法进行强制显性复制。这个方法对数组进行了浅拷贝(shallow copy),并且返回一个包含此拷贝的新数组。

下面这个示例中定义了一个`names`数组，其包含了七个人名。还定义了一个`copiedNames`变量，用以储存在`names`上调用`copy`方法所返回的结果：

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

我们可以通过修改一个数组中某元素，并且检查另一个数组中对应元素的方法来判定`names`数组确已被复制。如果你将`copiedNames`中第一个元素从`"Mohsen"`修改为`"Mo"`,则`names`数组返回的仍是拷贝发生前的`"Mohsen"`：

```
copiedName[0] = "Mo"
println(name[0])
// 输出 "Mohsen"
```

注意：

如果你仅需要确保你对数组的引用是唯一引用，请调用`unshare`方法，而不是`copy`方法。`unshare`方法仅会在确有必要时才会创建数组拷贝。`copy`方法会在任何时候都创建一个新的拷贝，即使引用已经是唯一引用。