翻译：shinyzhu　校对：pp-prog

属性 (Properties)

本页包含内容：

属性将值跟特定的类、结构或枚举关联。存储属性存储常量或变量作为实例的一部分，计算属性计算（而不是存储）一个值。计算属性可以用于类、结构体和枚举里，存储属性只能用于类和结构体。

存储属性和计算属性通常用于特定类型的实例，但是，属性也可以直接用于类型本身，这种属性称为类型属性。

另外，还可以定义属性监视器来监控属性值的变化，以此来触发一个自定义的操作。属性监视器可以添加到自己写的存储属性上，也可以添加到从父类继承的属性上。

存储属性

简单来说，一个存储属性就是存储在特定类或结构体的实例里的一个常量或变量，存储属性可以是变量存储属性（用关键字`var`定义），也可以是常量存储属性（用关键字`let`定义）。

可以在定义存储属性的时候指定默认值，请参考构造过程一章的默认属性值一节。也可以在构造过程中设置或修改存储属性的值，甚至修改常量存储属性的值，请参考构造过程一章的在初始化阶段修改常量存储属性一

节。

下面的例子定义了一个名为FixedLengthRange的结构体，他描述了一个在创建后无法修改值域宽度的区间：

```
struct FixedLengthRange {
    var firstValue: Int
    let length: Int
}
var rangeOfThreeItems = FixedLengthRange(firstValue: 0, length: 3)
// 该区间表示整数0，1，2
rangeOfThreeItems.firstValue = 6
// 该区间现在表示整数6，7，8
```

FixedLengthRange的实例包含一个名为firstValue的变量存储属性和一个名为length的常量存储属性。在上面的例子中，length在创建实例的时候被赋值，因为它是一个常量存储属性，所以之后无法修改它的值。

常量和存储属性

如果创建了一个结构体的实例并赋值给一个常量，则无法修改实例的任何属性，即使定义了变量存储属性：

```
let rangeOfFourItems = FixedLengthRange(firstValue: 0, length: 4)
// 该区间表示整数0，1，2，3
rangeOfFourItems.firstValue = 6
// 尽管firstValue诗歌变量属性，这里还是会报错
```

因为rangeOfFourItems声明成了常量（用let关键字），即使firstValue是一个变量属性，也无法再修改它了。

这种行为是由于结构体（struct）属于值类型。当值类型的实例被声明为常量的时候，它的所有属性也就成了常量。

属于引用类型的类（class）则不一样，把一个引用类型的实例赋给一个常

量后，仍然可以修改实例的变量属性。

延迟存储属性

延迟存储属性是指当第一次被调用的时候才会计算其初始值的属性。在属性声明前使用`@lazy`来标示一个延迟存储属性。

注意：

必须将延迟存储属性声明成变量（使用`var`关键字），因为属性的值在实例构造完成之前可能无法得到。而常量属性在构造过程完成之前必须要有初始值，因此无法声明成延迟属性。

延迟属性很有用，当属性的值依赖于在实例的构造过程结束前无法知道具体值的外部因素时，或者当属性的值需要复杂或大量计算时，可以只在需要的时候来计算它。

下面的例子使用了延迟存储属性来避免复杂类的不必要的初始化。例子中定义了`DataImporter`和`DataManager`两个类，下面是部分代码：

```
class DataImporter {
    /*
    DataImporter 是一个将外部文件中的数据导入的类。
    这个类的初始化会消耗不少时间。
    */
    var fileName = "data.txt"
    // 这是提供数据导入功能
}

class DataManager {
    @lazy var importer = DataImporter()
    var data = String[]()
    // 这是提供数据管理功能
}

let manager = DataManager()
manager.data += "Some data"
```

```
manager.data += "Some more data"
// DataImporter 实例的 importer 属性还没有被创建
```

DataManager类包含一个名为data的存储属性，初始值是一个空的字符串（String）数组。虽然没有写出全部代码，DataManager类的目的是管理和提供对这个字符串数组的访问。

DataManager的一个功能是从文件导入数据，该功能由DataImporter类提供，DataImporter需要消耗不少时间完成初始化：因为它的实例在初始化时可能要打开文件，还要读取文件内容到内存。

DataManager也可能不从文件中导入数据。所以当DataManager的实例被创建时，没必要创建一个DataImporter的实例，更明智的是当用到DataImporter的时候才去创建它。

由于使用了@lazy，importer属性只有在第一次被访问的时候才被创建。比如访问它的属性fileName时：

```
println(manager.importer.fileName)
// DataImporter 实例的 importer 属性现在被创建了
// 输出 "data.txt”
```

存储属性和实例变量

如果您有过 Objective-C 经验，应该知道有两种方式在类实例存储值和引用。对于属性来说，也可以使用实例变量作为属性值的后端存储。

Swift 编程语言中把这些理论统一用属性来实现。Swift 中的属性没有对应的实例变量，属性的后端存储也无法直接访问。这就避免了不同场景下访问方式的困扰，同时也将属性的定义简化成一个语句。 一个类型中属性的全部信息——包括命名、类型和内存管理特征——都在唯一一个地方（类型定义中）定义。

计算属性

除存储属性外，类、结构体和枚举可以定义计算属性，计算属性不直接存储值，而是提供一个 getter 来获取值，一个可选的 setter 来间接设置其他属性或变量的值。

```
struct Point {
    var x = 0.0, y = 0.0
}
struct Size {
    var width = 0.0, height = 0.0
}
struct Rect {
    var origin = Point()
    var size = Size()
    var center: Point {
    get {
        let centerX = origin.x + (size.width / 2)
        let centerY = origin.y + (size.height / 2)
        return Point(x: centerX, y: centerY)
    }
    set(newCenter) {
        origin.x = newCenter.x - (size.width / 2)
        origin.y = newCenter.y - (size.height / 2)
    }
    }
}
var square = Rect(origin: Point(x: 0.0, y: 0.0),
    size: Size(width: 10.0, height: 10.0))
let initialSquareCenter = square.center
square.center = Point(x: 15.0, y: 15.0)
println("square.origin is now at (\(square.origin.x),
\(square.origin.y))")
// 输出 "square.origin is now at (10.0, 10.0)”
```

这个例子定义了 3 个几何形状的结构体：

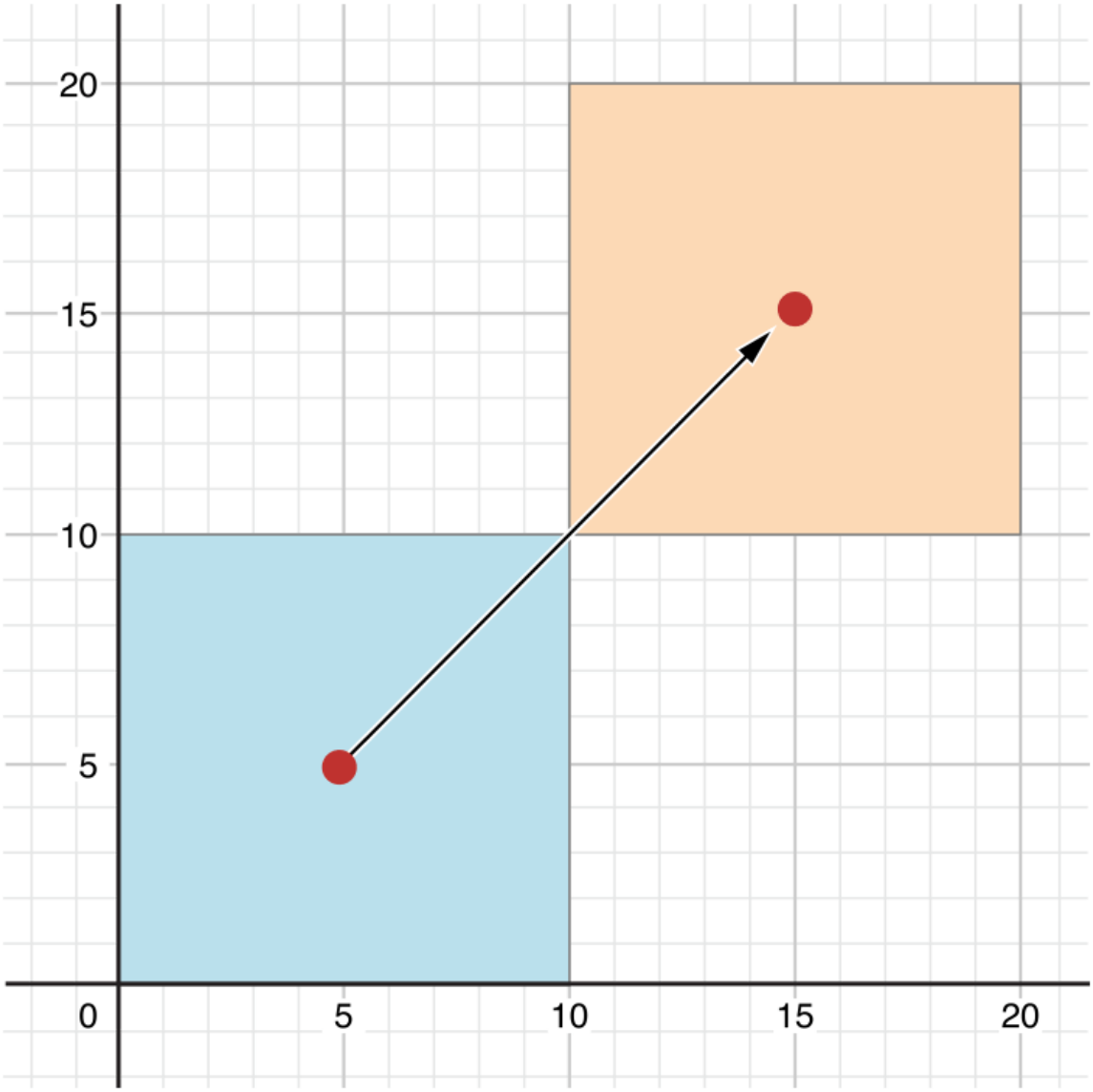
- `Point`封装了一个`(x, y)`的坐标
- `Size`封装了一个`width`和`height`
- `Rect`表示一个有原点和尺寸的矩形

`Rect`也提供了一个名为`center`的计算属性。一个矩形的中心点可以从原点和尺寸来算出，所以不需要将它以显式声明的`Point`来保存。`Rect`的计算属性`center`提供了自定义的 getter 和 setter 来获取和设置矩形的中心点，就像它有一个存储属性一样。

例子中接下来创建了一个名为`square`的`Rect`实例，初始值原点是`(0, 0)`，宽度高度都是`10`。如图所示蓝色正方形。

`square`的`center`属性可以通过点运算符（`square.center`）来访问，这会调用 getter 来获取属性的值。跟直接返回已经存在的值不同，getter 实际上通过计算然后返回一个新的`Point`来表示`square`的中心点。如代码所示，它正确返回了中心点`(5, 5)`。

`center`属性之后被设置了一个新的值`(15, 15)`，表示向右上方移动正方形到如图所示橙色正方形的位置。设置属性`center`的值会调用 setter 来修改属性`origin`的`x`和`y`的值，从而实现移动正方形到新的位置。



便捷 setter 声明

如果计算属性的 setter 没有定义表示新值的参数名，则可以使用默认名称`newValue`。下面是使用了便捷 setter 声明的`Rect`结构体代码：

```
struct AlternativeRect {
    var origin = Point()
    var size = Size()
    var center: Point {
    get {
        let centerX = origin.x + (size.width / 2)
        let centerY = origin.y + (size.height / 2)
```

```
        return Point(x: centerX, y: centerY)
    }
    set {
        origin.x = newValue.x - (size.width / 2)
        origin.y = newValue.y - (size.height / 2)
    }
    }
}
```

只读计算属性

只有 getter 没有 setter 的计算属性就是只读计算属性。只读计算属性总是返回一个值，可以通过点运算符访问，但不能设置新的值。

注意：

必须使用`var`关键字定义计算属性，包括只读计算属性，因为他们的值不是固定的。`let`关键字只用来声明常量属性，表示初始化后再也无法修改的值。

只读计算属性的声明可以去掉`get`关键字和花括号：

```
struct Cuboid {
    var width = 0.0, height = 0.0, depth = 0.0
    var volume: Double {
    return width * height * depth
    }
}
let fourByFiveByTwo = Cuboid(width: 4.0, height: 5.0,
depth: 2.0)
println("the volume of fourByFiveByTwo is \
(fourByFiveByTwo.volume)")
// 输出 "the volume of fourByFiveByTwo is 40.0"
```

这个例子定义了一个名为`Cuboid`的结构体，表示三维空间的立方体，包含`width`、`height`和`depth`属性，还有一个名为`volume`的只读计算属性用来返回立方体的体积。设置`volume`的值毫无意义，因为通过`width`、`height`和`depth`就能算出`volume`。然而，`Cuboid`提供一个只读计算属性

来让外部用户直接获取体积是很有用的。

属性监视器

属性监视器监控和响应属性值的变化，每次属性被设置值的时候都会调用属性监视器，甚至新的值和现在的值相同的时候也不例外。

可以为除了延迟存储属性之外的其他存储属性添加属性监视器，也可以通过重载属性的方式为继承的属性（包括存储属性和计算属性）添加属性监视器。属性重载请参考继承一章的重载。

注意：

不需要为无法重载的计算属性添加属性监视器，因为可以通过 setter 直接监控和响应值的变化。
可以为属性添加如下的一个或全部监视器：

- `willSet`在设置新的值之前调用
- `didSet`在新的值被设置之后立即调用

`willSet`监视器会将新的属性值作为固定参数传入，在`willSet`的实现代码中可以为这个参数指定一个名称，如果不指定则参数仍然可用，这时使用默认名称`newValue`表示。

类似地，`didSet`监视器会将旧的属性值作为参数传入，可以为该参数命名或者使用默认参数名`oldValue`。

注意：

`willSet`和`didSet`监视器在属性初始化过程中不会被调用，他们只会当属性的值在初始化之外的地方被设置时被调用。
这里是一个`willSet`和`didSet`的实际例子，其中定义了一个名为`StepCounter`的类，用来统计当人步行时的总步数，可以跟计步器或其他日常锻炼的统计装置的输入数据配合使用。

```
class StepCounter {
```

```
    var totalSteps: Int = 0 {
    willSet(newTotalSteps) {
        println("About to set totalSteps to \
(newTotalSteps)")
    }
    didSet {
        if totalSteps > oldValue  {
            println("Added \(totalSteps - oldValue)
steps")
        }
    }
    }
}
let stepCounter = StepCounter()
stepCounter.totalSteps = 200
// About to set totalSteps to 200
// Added 200 steps
stepCounter.totalSteps = 360
// About to set totalSteps to 360
// Added 160 steps
stepCounter.totalSteps = 896
// About to set totalSteps to 896
// Added 536 steps
```

`StepCounter`类定义了一个`Int`类型的属性`totalSteps`，它是一个存储属性，包含`willSet`和`didSet`监视器。

当`totalSteps`设置新值的时候，它的`willSet`和`didSet`监视器都会被调用，甚至当新的值和现在的值完全相同也会调用。

例子中的`willSet`监视器将表示新值的参数自定义为`newTotalSteps`，这个监视器只是简单的将新的值输出。

`didSet`监视器在`totalSteps`的值改变后被调用，它把新的值和旧的值进行对比，如果总的步数增加了，就输出一个消息表示增加了多少步。`didSet`没有提供自定义名称，所以默认值`oldValue`表示旧值的参数名。

注意：

如果在`didSet`监视器里为属性赋值，这个值会替换监视器之前设置的值。

全局变量和局部变量

计算属性和属性监视器所描述的模式也可以用于全局变量和局部变量，全局变量是在函数、方法、闭包或任何类型之外定义的变量，局部变量是在函数、方法或闭包内部定义的变量。

前面章节提到的全局或局部变量都属于存储型变量，跟存储属性类似，它提供特定类型的存储空间，并允许读取和写入。

另外，在全局或局部范围都可以定义计算型变量和为存储型变量定义监视器，计算型变量跟计算属性一样，返回一个计算的值而不是存储值，声明格式也完全一样。

注意：

全局的常量或变量都是延迟计算的，跟延迟存储属性相似，不同的地方在于，全局的常量或变量不需要标记`@lazy`特性。

局部范围的常量或变量不会延迟计算。

类型属性

实例的属性属于一个特定类型实例，每次类型实例化后都拥有自己的一套属性值，实例之间的属性相互独立。

也可以为类型本身定义属性，不管类型有多少个实例，这些属性都只有唯一一份。这种属性就是类型属性。

类型属性用于定义特定类型所有实例共享的数据，比如所有实例都能用的一个常量（就像 C 语言中的静态常量），或者所有实例都能访问的一个变量（就像 C 语言中的静态变量）。

对于值类型（指结构体和枚举）可以定义存储型和计算型类型属性，对于类（class）则只能定义计算型类型属性。

值类型的存储型类型属性可以是变量或常量，计算型类型属性跟实例的计算属性一样定义成变量属性。

注意：

跟实例的存储属性不同，必须给存储型类型属性指定默认值，因为类型本身无法在初始化过程中使用构造器给类型属性赋值。

类型属性语法

在 C 或 Objective-C 中，静态常量和静态变量的定义是通过特定类型加上`global`关键字。在 Swift 编程语言中，类型属性是作为类型定义的一部分写在类型最外层的花括号内，因此它的作用范围也就在类型支持的范围内。

使用关键字`static`来定义值类型的类型属性，关键字`class`来为类（class）定义类型属性。下面的例子演示了存储型和计算型类型属性的语法：

```
struct SomeStructure {
    static var storedTypeProperty = "Some value."
    static var computedTypeProperty: Int {
    // 这里返回一个 Int 值
    }
}
enum SomeEnumeration {
    static var storedTypeProperty = "Some value."
    static var computedTypeProperty: Int {
    // 这里返回一个 Int 值
    }
}
class SomeClass {
    class var computedTypeProperty: Int {
    // 这里返回一个 Int 值
```

```
    }
}
```
注意：

例子中的计算型类型属性是只读的，但也可以定义可读可写的计算型类型属性，跟实例计算属性的语法类似。

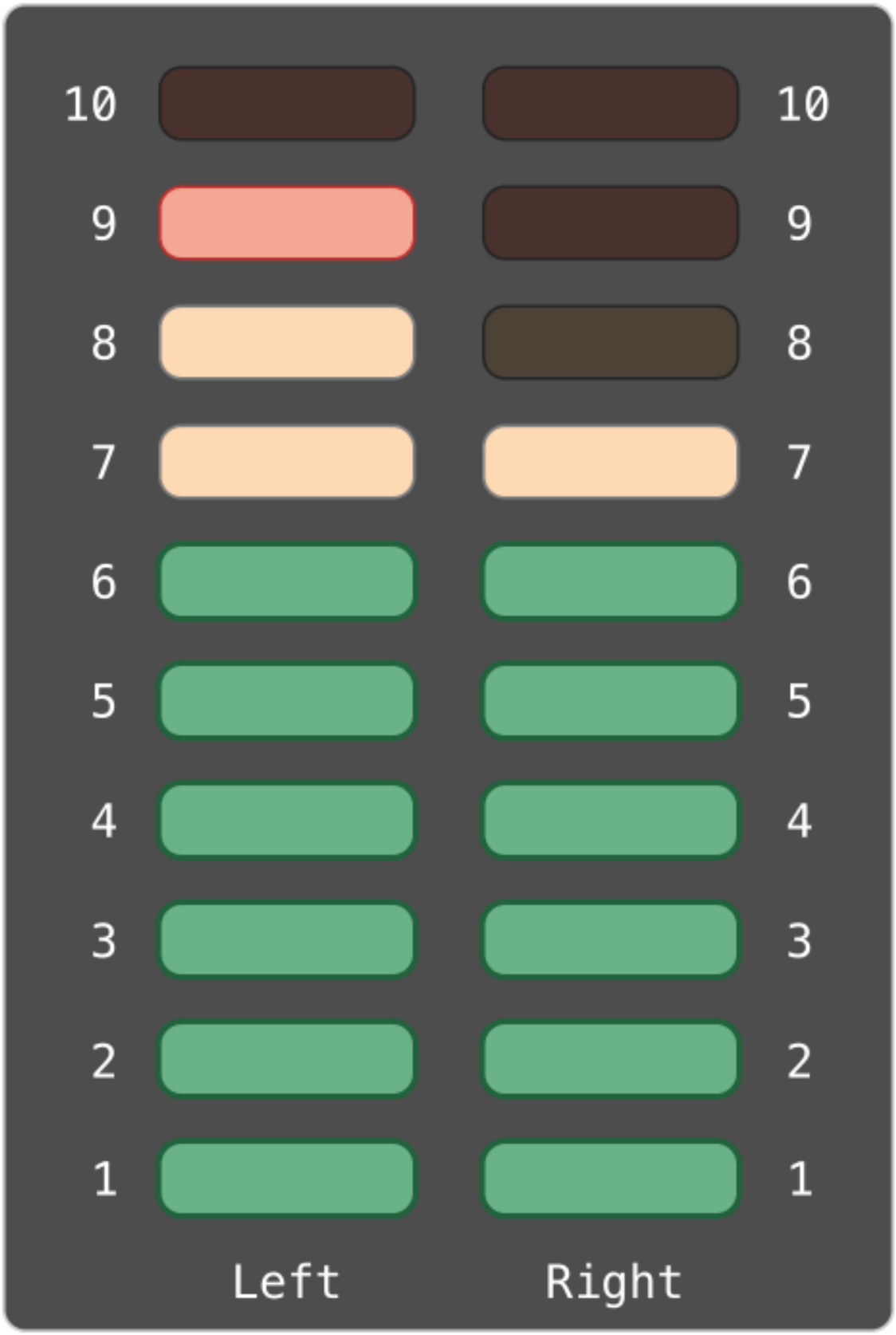
获取和设置类型属性的值

跟实例的属性一样，类型属性的访问也是通过点运算符来进行，但是，类型属性是通过类型本身来获取和设置，而不是通过实例。比如：

```
println(SomeClass.computedTypeProperty)
// 输出 "42"

println(SomeStructure.storedTypeProperty)
// 输出 "Some value."
SomeStructure.storedTypeProperty = "Another value."
println(SomeStructure.storedTypeProperty)
// 输出 "Another value.”
```
下面的例子定义了一个结构体，使用两个存储型类型属性来表示多个声道的声音电平值，每个声道有一个 0 到 10 之间的整数表示声音电平值。

后面的图表展示了如何联合使用两个声道来表示一个立体声的声音电平值。当声道的电平值是 0，没有一个灯会亮；当声道的电平值是 10，所有灯点亮。本图中，左声道的电平是 9，右声道的电平是 7。



上面所描述的声道模型使用`AudioChannel`结构体来表示：

```
struct AudioChannel {
    static let thresholdLevel = 10
    static var maxInputLevelForAllChannels = 0
    var currentLevel: Int = 0 {
    didSet {
        if currentLevel > AudioChannel.thresholdLevel
{
            // 将新电平值设置为阀值
            currentLevel =
AudioChannel.thresholdLevel
        }
        if currentLevel >
AudioChannel.maxInputLevelForAllChannels {
            // 存储当前电平值作为新的最大输入电平
            AudioChannel.maxInputLevelForAllChannels
= currentLevel
        }
    }
    }
}
```

结构`AudioChannel`定义了 2 个存储型类型属性来实现上述功能。第一个是`thresholdLevel`，表示声音电平的最大上限阈值，它是一个取值为 10 的常量，对所有实例都可见，如果声音电平高于 10，则取最大上限值 10（见后面描述）。

第二个类型属性是变量存储型属性`maxInputLevelForAllChannels`，它用来表示所有`AudioChannel`实例的电平值的最大值，初始值是 0。

`AudioChannel`也定义了一个名为`currentLevel`的实例存储属性，表示当前声道现在的电平值，取值为 0 到 10。

属性`currentLevel`包含`didSet`属性监视器来检查每次新设置后的属性值，有如下两个检查：

- 如果`currentLevel`的新值大于允许的阈值`thresholdLevel`，属性监视器将`currentLevel`的值限定为阈值`thresholdLevel`。
- 如果修正后的`currentLevel`值大于任何之前任意`AudioChannel`实

例中的值，属性监视器将新值保存在静态属性`maxInputLevelForAllChannels`中。

注意：

在第一个检查过程中，`didSet`属性监视器将`currentLevel`设置成了不同的值，但这时不会再次调用属性监视器。

可以使用结构体`AudioChannel`来创建表示立体声系统的两个声道`leftChannel`和`rightChannel`：

```
var leftChannel = AudioChannel()
var rightChannel = AudioChannel()
```

如果将左声道的电平设置成 7，类型属性`maxInputLevelForAllChannels`也会更新成 7：

```
leftChannel.currentLevel = 7
println(leftChannel.currentLevel)
// 输出 "7"
println(AudioChannel.maxInputLevelForAllChannels)
// 输出 "7"
```

如果试图将右声道的电平设置成 11，则会将右声道的`currentLevel`修正到最大值 10，同时`maxInputLevelForAllChannels`的值也会更新到 10：

```
rightChannel.currentLevel = 11
println(rightChannel.currentLevel)
// 输出 "10"
println(AudioChannel.maxInputLevelForAllChannels)
// 输出 "10"
```