The Swift Programming Language in Chinese

https://github.com/letsswift

http://letsswift.com/

June 15, 2014

目录

1 Swift 中文教程（十）属性

属性是描述特定类、结构或者枚举的值。存储属性作为实例的一部分存储常量与变量的值，而计算属性计算他们的值（不只是存储)。计算属性存在于类、结构与枚举中。存储属性仅仅只在类与结构中。

属性通常与特定类型实例联系在一起。但属性也可以与类型本身联系在一起，这样的属性称之为类型属性。

另外，可以定义属性观察者来处理属性值发生改变的情况，这样你就可以对用户操作做出反应。属性观察者可以被加在自己定义的存储属性之上，也可以在从父类继承的子类属性之上。

1.1 存储属性

最简单的情形，作为特定类或结构实例的一部分，存储属性存储着常量或者变量的值。存储属性可分为变量存储属性（关键字 var 描述）和常量存储属性(关键字 let 描述)。

当定义存储属性时，你可以提供一个默认值，这些在 “默认属性值” 描述。在初始化过程中你也可以设置或改变存储属性的初值。这个准则对常量存储属性也同样适用（在 “初始化过程中改变常量属性” 描述)

下面的例子定义了一个叫 FixedLengthRange 的结构，它描述了一个一定范围内的整数值，当创建这个结构时，范围长度是不可以被改变的：

```
struct FixedLengthRange {
var firstValue: Int
let length: Int
}
var rangeOfThreeItems = FixedLengthRange(firstValue: 0, length: 3)
// the range represents integer values 0, 1, and 2
rangeOfThreeItems.firstValue = 6
// the range now represents integer values 6, 7, and 8
```

FixedLengthRange 的实例包含一个名为 firstValue 的变量存储属性和名为 length 的常量存储属性。以上的例子中，当范围确定，length 被初始化之后它的值是不可以被改变的

1.1.1 常量结构实例的存储属性

如果你创建一个结构实例，并将其赋给一个常量，这个实例中的属性将不可以被改变，即使他们被声明为变量属性

```
let rangeOfFourItems = FixedLengthRange(firstValue: 0, length: 4)
// this range represents integer values 0, 1, 2, and 3
rangeOfFourItems.firstValue = 6
// this will report an error, even thought firstValue is a variable property
```

因为 rangeOfFourItems 是一个常量（let），即便 firstValue 是一个变量属性，它的值也是不可以被改变的

这样的特性是因为结构是值类型。当一个值类型实例作为常量而存在，它的所有属性也作为常量而存在。

而这个特性对类并不适用，因为类是引用类型。如果你将引用类型的实例赋值给常量，依然能够改变实例的变量属性。

1.1.2 Lazy Stored Properties（懒惰存储属性?）

懒惰存储属性是当它第一次被使用时才进行初值计算。通过在属性声明前加上 @lazy 来标识一个懒惰存储属性。

> 注意必须声明懒惰存储属性为变量属性（通过 var），因为它的初始值直到实例初始化完成之后才被检索。常量属性在实例初始化完成之前就应该被赋值，因此常量属性不能够被声明为懒惰存储属性。

当属性初始值因为外部原因，在实例初始化完成之前不能够确定时，就要定义成懒惰存储属性。当属性初始值需要复杂或高代价的设置，在它需要时才被赋值时，懒惰存储属性就派上用场了。

下面的例子使用懒惰存储属性来防止类中不必要的初始化操作。它定义了类 DataImporter 和类 DataManager：

```
class DataImporter {
   /*DataImporter is a class to import data from an external file.    The class is assu
    var fileName = "data.txt"
   // the DataImporter class would provide data importing functionality here
}
```

```
class DataManager {
    @lazy var importer = DataImporter()
    var data = String[]()
   // the DataManager class would provide data management functionality here
}
let manager = DataManager()
manager.data += "Some data"
manager.data += "Some more data"
// the DataImporter instance for the importer property has not yet been created
```

类 DataManager 有一个称为 data 的存储属性，它被初始化为一个空的 String 数组。虽然 DataManager 定义的其它部分并没有写出来，但可以看出 DataManager 的目的是管理 String 数据并为其提供访问接口。

DataManager 类的部分功能是从文件中引用数据。这个功能是由 DataImporter 类提供的，这个类需要一定的时间来初始化，因为它的实例需要打开文件并见内容读到内存中。

因为 DataManager 实例可能并不需要立即管理从文件中引用的数据，所以在 DataManager 实例被创建时，并不需要马上就创建一个新的 DataImporter 实例。这就使得当 DataImporter 实例在需要时才被创建理所当然起来。

因为被声明为 @lazy 属性，DataImporter 的实例 importer 只有在当它在第一次被访问时才被创建。例如它的 fileName 属性需要被访问时：

```
println(manager.importer.fileName)
// the DataImporter instance for the importer property has now been created
// prints "data.txt
```

1.1.3 存储属性与实例变量

如果你使用过 Objective − C，你应该知道它提供两种方式来存储作为类实例一部分的值与引用。除了属性，你可以使用实例变量作为属性值的后备存储

Swift 使用一个单一属性声明来统一这些概念。一个 Swift 属性没有与之相符的实例变量，并且属性的后备存储也不能直接访问。这防止了在不通上下文中访问值的混淆，并且简化属性声明成为一个单一的、最终的语句。关于属性的所有信息 – 包含名称、类型和内存管理等 – 作为类型定义的一部分而定义。

1.2 计算属性

除了存储属性，类、结构和枚举能够定义计算属性。计算属性并不存储值，它提供 getter 和可选的 setter 来间接地获取和设置其它的属性和值。

```
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))")
// prints "square.origin is now at (10.0, 10.0)"
```

这个例子定义了三个处理几何图形的结构:

- Point 包含一个（x，y）坐标
- Size 包含宽度 width 和高度 height
- Rect 定义了一个长方形，包含原点和大小 size

Rect 结构包含一个称之为 center 的计算属性。Rect 当前中心点的坐标可以通过 origin 和 size 属性得来，所以并不需要显式地存储中心点的值。取而代之

的是，Rect 定义一个称为 center 的计算属性，它包含一个 get 和一个 set 方法，通过它们来操作长方形的中心点，就像它是一个真正的存储属性一样。

例子中定义了一个名为 square 的 Rect 变量，它的中心点初始化为（0，0），高度和宽度初始化为 10，由以下图形中的蓝色正方形部分。

变量 square 的 center 属性通过点操作符访问，它会调用 center 的 getter 方法。不同于直接返回一个存在的值，getter 方法要通过计算才能返回长方形的中心点的值（point）。以上的例子中，getter 方法返回中心点（5，5）。

然后 center 属性被设置成新的值（15，15），这样就把这个正方形向右向上移动到了途中黄色部分所表示的新的位置。通过调用 setter 方法来设置 center，改变 origin 中坐标 x 和 y 的值，将正方形移动到新的位置。

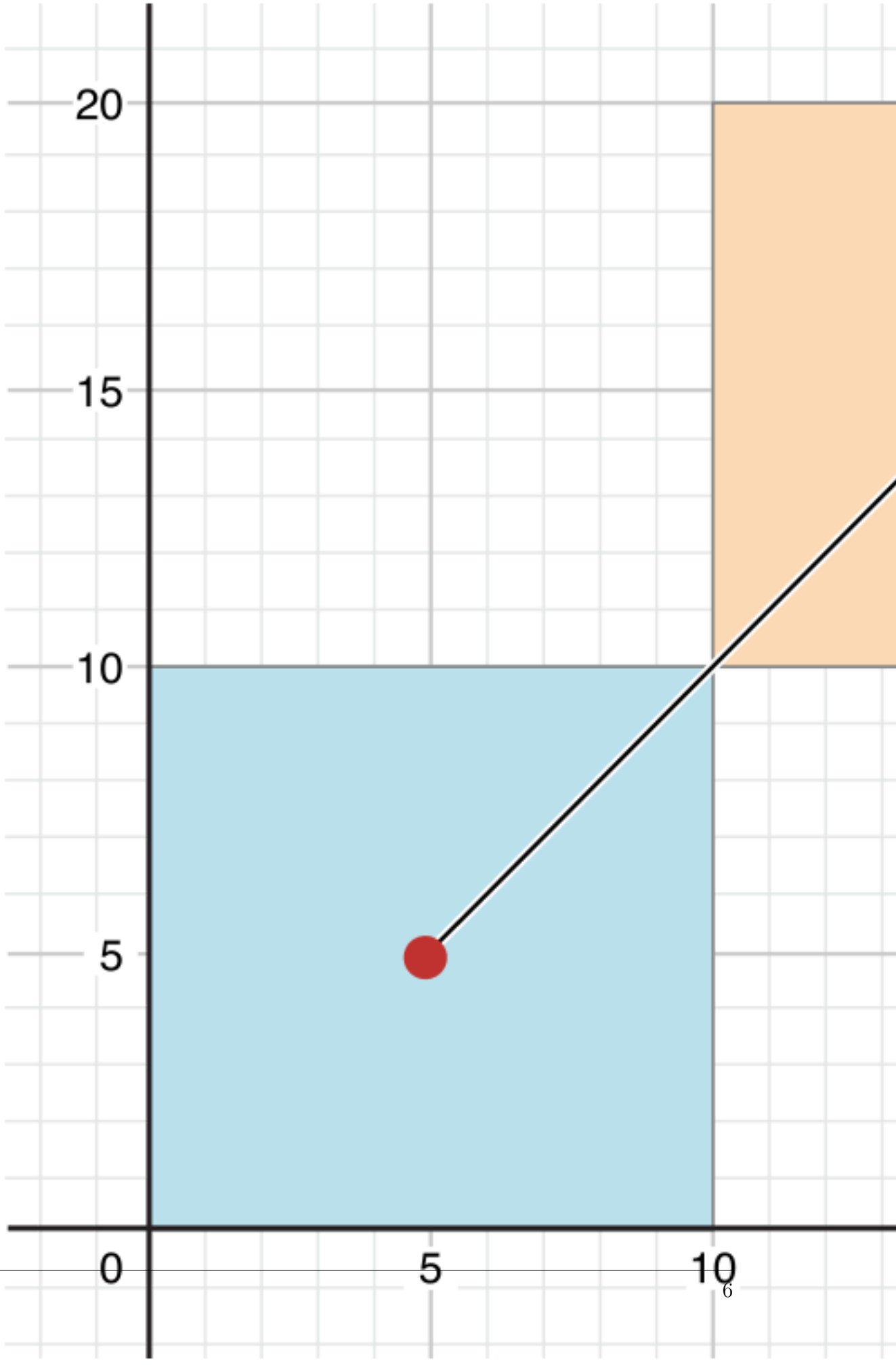
1.2.1 setter 声明的简略写法

如果计算属性的 setter 方法没有将被设置的值定义一个名称，将会默认地使用 newValue 这个名称来代替。下面的例子采用了这样一种特性，定义了 Rect 结构的新版本：

```
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)
        }
    }
}
```

1.2.2 只读计算属性

只读计算属性只带有一个 getter 方法，通过点操作符，可以放回属性值，但是不能修改它的值。


20
15
10
5
0
5
10

> 注意应该使用 var 关键字将计算属性－包含只读计算属性－定义成变量属性，因为它们的值并不是固定的。let 关键字只被常量属性说使用，以表明一旦被设置它们的值就是不可改变的了

通过移除 get 关键字和它的大括号，可以简化只读计算属性的定义：

```
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)")
// prints "the volume of fourByFiveByTwo is 40.0
```

这个例子定义了一个三维长方体结构 Cuboid，包含了长宽高三个属性，和一个表示长方体容积的只读计算属性 volume。volume 值是不可被设置的，因为它直接由长宽高三个属性计算而来。通过提供这样一个只读计算属性，Cuboid 使外部用户能够访问到其当前的容积值。

1.3 属性观察者

属性观察者观察属性值的改变并对此做出响应。当设置属性的值时，属性观察者就被调用，即使当新值同原值相同时也会被调用。

除了懒惰存储属性，你可以为任何存储属性加上属性观察者定义。另外，通过重写子类属性，也可以继承属性（存储或计算）加上属性观察者定义。属性重写在“重写”章节定义。

> 注意不必为未重写的计算属性定义属性观察者，因为可以通过它的 setter 方法直接对值的改变做出响应

定义属性的观察者时，你可以单独或同时使用下面的方法：

- willSet：设置值前被调用
- didSet：设置值后立刻被调用

当实现 willSet 观察者时，新的属性值作为常量参数被传递。你可以为这个参数起一个名字，如果不的话，这个参数就默认地被命名成 newValue。

在实现 didSet 观察者时也是一样，只不过传递的产量参数表示的是旧的属性值。

> 注意：属性初始化时，willset 和 didSet 并不会被调用。只有在初始化上下文之外，当设置属性值时才被调用

下面是一个 willSet 和 didSet 用法的实例。定义了一个类 StepCounter，用来统计人走路时的步数。它可以从计步器或其它计数器上获取输入数据，对日常联系锻炼的步数进行追踪。

```
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
```

类 StepCounter 声明了一个 Int 类型的、含有 willSet 和 didSet 观察者的存储属性 totalSteps。当这个属性被赋予新值时，willSet 和 didSet 将会被调用，即使新值和旧值是相同的。

例子中的 willSet 观察者为参数起了个新的名字 newTotalSteps，它简单地打印了即将被设置的值。

当 totalSteps 值被更新时，didSet 观察者被调用，它比较 totalSteps 的新值和旧值，如果新值比旧值大，就打印所增加的步数。didSet 并没有为旧值参数命名，在本例中，将会使用默认的名字 oldValue 来表示旧的值。

> 注意如果通过 didSet 来设置属性的值，即使属性值刚刚被设置过，起作用的也将会是 didSet，即新值是 didSet 设置的值

1.4 全局和局部变量

以上所写的关于计算与观察属性值的特性同样适用于全局和局部变量。全局变量是在任何函数、方法、闭包、类型上下文外部定义的变量，而局部变量是在函数、方法、闭包中定义的变量。

前面章节所遇到过的全局、局部变量都是存储变量。和存储属性一样，存储变量为特定类型提供存储空间并且可以被访问

但是，你可以在全局或局部范围定义计算变量和存储变量观察者。计算变量并不存储值，只用来计算特定值，它的定义方式与计算属性一样。

> 注意全局常量和变量通常是延迟计算的，跟懒惰存储属性一样，但是不需要加上 @lazy。而局部常量与变量不是延迟计算的。

1.5 类型属性

实例属性是特定类型实例的属性。当创建一个类型的实例时，这个实例有自己的属性值的集合，这将它与其它实例区分开来。

也可以定义属于类型本身的属性，即使创建再多的这个类的实例，这个属性也不属于任何一个，它只属于类型本身，这样的属性就称为类型属性。

类型属性适用于定义那些特定类型实例所通用的属性，例如一个可以被所有实例使用的常量属性（就像 c 中的静态常量），或者变量属性（c 中的静态变量）。

可以为值类型（结构、枚举）定义存储类型属性和计算类型属性。对类而言，只能够定义计算类型属性。

值类型的存储类型属性可以是常量也可以是变量。而计算类型属性通常声明成变量属性，类似于计算实例属性

> 注意不想存储实例属性，你需要给存储类型属性一个初始值。因为类型本身在初始化时不能为存储类型属性设置值

1.5.1 类型属性句法

在 C 和 Objective − C 中，定义静态常量、变量和全局静态变量一样。但是在 swift 中，类型属性的定义要放在类型定义中进行，在类型定义的大括号中，显示地声明它在类型中的作用域。

对值类型而言，定义类型属性使用 static 关键字，而定义类类型的类型属性使用 class 关键字。下面的例子展示了存储和计算类型属性的用法：

```
struct SomeStructure {
    static var storedTypeProperty = "Some value."
    static var computedTypeProperty: Int {
        // return an Int value here
    }
}
enum SomeEnumeration {
    static var storedTypeProperty = "Some value."
   static var computedTypeProperty: Int {    // return an Int value here
    }
}
class SomeClass {
    class var computedTypeProperty: Int {
        // return an Int value here
    }
}
```

> 注意上面的例子是针对只读计算类型属性而言的，不过你也可以像计算实例属性一样定义可读可写的计算类型属性

1.5.2 查询与设置类型属性

像实例属性一样，类型属性通过点操作符来查询与设置。但是类型属性的查询与设置是针对类型而言的，并不是针对类型的实例。例如：

```
println(SomeClass.computedTypeProperty)
// prints "42"
println(SomeStructure.storedTypeProperty)
// prints "Some value."
```

```
SomeStructure.storedTypeProperty = "Another value."
println(SomeStructure.storedTypeProperty)
// prints "Another value.
```

下面的例子在一个结构中使用两个存储类型属性来展示一组声音通道的音频等级表。每个通道使用 0 到 10 来表示声音的等级。

从下面的图表中可以看出，使用了两组声音通道来表示一个立体声音频等级表。当一个通道的等级为 0 时，所有的灯都不会亮，当等级为 10 时，所有的灯都会亮。下面的图中，左边的通道表示声音等级为 9，右边的为 7

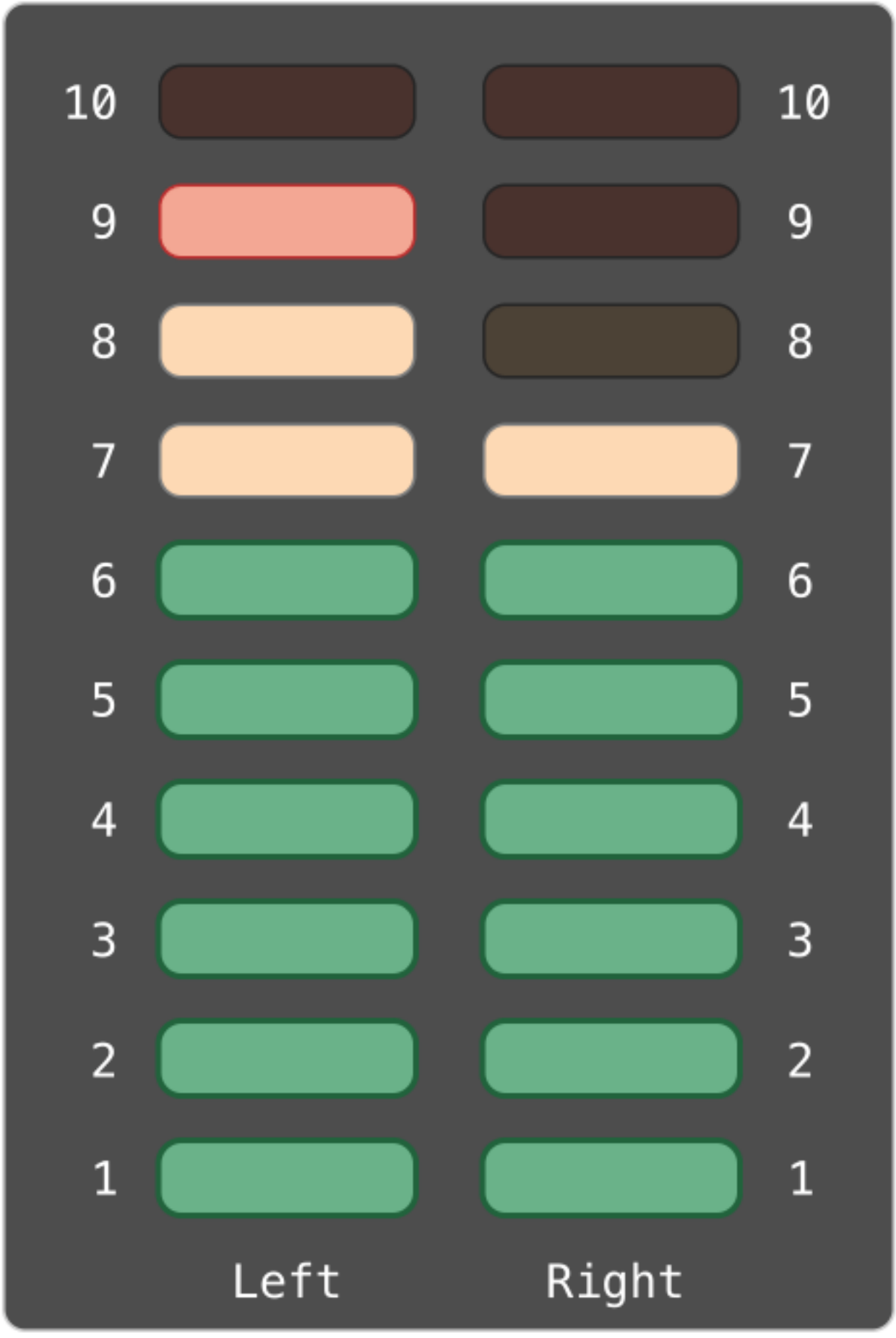
上述的声音通道由以下的 AudioChannel 结构实例来表示：

```
struct AudioChannel {
    static let thresholdLevel = 10
    static var maxInputLevelForAllChannels = 0
    var currentLevel: Int = 0 {
        didSet {
            if currentLevel > AudioChannel.thresholdLevel {
                //cap the new audio level to the threshold level
                currentLevel = AudioChannel.thresholdLevel
            }
         if currentLevel > AudioChannel.maxInputLevelForAllChannels {
                // store this as the new overall maximum input level
             AudioChannel.maxInputLevelForAllChannels = currentLevel
            }
        }
    }
}
```

AudioChannel 结构定义了两个存储类型属性。thresholdLevel 定义了音频所能达到的最高等级，对所有的 AudoChannel 实例而言，是个值为 10 的常量。当一个声音信号的值超过 10 时，会被截断为其阈值 10。

第二个类型属性是一个变量存储属性 maxInputLevelForAllChannels。它保存了当前所有 AudioChannel 实例中所接受到声音的最高等级，它被初始化为 0。

结构还定义了一个存储实例属性 currentLevel，表示当前的通道声音等级。这个属性使用 didSet 属性观察者来检测 currentLevel 的改变。这个观察者执行两道检查：


10
10
9
9
8
8
7
7
6
6
5
5
4
4
3
3
2
2
1
1
Left
Right

- 如果 currentlevel 的新值比阈值 thresholdLevel 大，currentLevel 将被设置成 thresholdLevel
- 如果 currentLevel 的新值比所有 AudioChannel 实例之前接受到的最大声音等级还要大，那么 maxInputLevelForAllChannles 将会被设置成 cueentLevel 大值。

  注意第一道检查中，didSet 为 currentLevel 设置了新值。这并不会造成观察者再次被调用

可以创建两个 AudioChannel 实例，leftChannel 和 rightChannel，来表示一个立体声系统：

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

如果设置左通道的 currentLevel 为 7，它的类型属性 maxInputLevelForAllChannels 将更新成为 7：

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

如果像设置右通道的 currentlevel 为 11，它的值将被截短成为 10，而且 maxInputLevelForAllChannels 的值也将更新为 10：

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

感谢翻译小组成员：李起攀 (微博)、若晨 (微博)、YAO、粽子、山有木兮木有枝、渺 -Bessie、墨离、Tiger 大顾 (微博)，校对：CXH_ME(微博),Joshua 孟思拓 (微博)