属性

*属性将*值与特定的类，结构或枚举关联。存储属性存储常量和变量值作为实例的一部分，而计算属性计算（而不是存储）值。计算属性由类，结构和枚举提供。存储的属性仅由类和结构提供。

存储和计算的属性通常与特定类型的实例相关联。但是，属性也可以与类型本身相关联。这些属性称为类型属性。

另外，您可以定义属性观察器来监视属性值的更改，您可以使用自定义操作进行响应。属性观察者可以添加到你自己定义的存储属性中，也可以添加到子类从其超类继承的属性中。

存储属性

以最简单的形式，存储属性是一个常量或变量，作为特定类或结构的实例的一部分存储。存储属性可以是*变量存储属性*（由var关键字引入）或*常量存储属性*（由let关键字引入）。

您可以为存储属性提供默认值作为其定义的一部分，如“默认属性值”中所述。您还可以在初始化期间设置和修改存储属性的初始值。即使对于常量存储的属性也是如此，如在初始化期间分配常量属性中所述。

下面的例子定义了一个叫做的结构FixedLengthRange，它描述了一个范围长度在创建后不能被改变的整数范围：

```
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
```

实例FixedLengthRange有一个被调用的变量存储属性firstValue和一个被称为的常量存储属性length。在上面的例子中，length当新的范围被创建并且之后不能被改变时被初始化，因为它是一个常量属性。

常量结构实例的存储属性

如果您创建了一个结构实例并将该实例分配给一个常量，那么即使它们被声明为可变属性，也不能修改该实例的属性：

```
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 though firstValue is a variable property
```

因为rangeOfFourItems被声明为常量（使用let关键字），所以无法更改其firstValue属性，即使它firstValue是一个变量属性。

这种行为是由于结构是*值类型*。当一个值类型的实例被标记为常量时，它的所有属性也是如此。

对于*引用类型的*类也是如此。如果将引用类型的实例分配给常量，则仍然可以更改该实例的变量属性。

懒惰存储的属性

一个*懒惰的存储属性*是一个属性，其初始值直到第一次使用时才被计算。通过lazy在声明前写入修饰符来指示惰性存储属性。

> 注意
>
> 您必须始终将lazy属性声明为变量（使用var关键字），因为它的初始值可能在实例初始化完成之后才能检索到。*在初始化完成之前*，常量属性必须始终有一个值，因此不能声明为惰性。

当属性的初始值取决于外部因素，只有在实例的初始化完成后才能知道其值，外部因素可用于延迟属性。当属性的初始值需要复杂的或计算上昂贵的设置时，惰性属性也很有用，除非需要或不需要执行，否则不应执行该设置。

下面的例子使用
DataImporter而且这两个类DataManager都没有完整显示：

```
class DataImporter {
    /*
     DataImporter is a class to import data from an external file.
     The class is assumed to take a nontrivial amount of time to initialize.
     */
    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.append("Some data")
manager.data.append("Some more data")
// the DataImporter instance for the importer property has not yet been created
```

的DataManager类有一个存储属性调用data，这是与一个新的，空数组初始化String的值。尽管没有显示其余的功能，但DataManager该类的目的是管理和提供对这组String数据的访问。

DataManager该类的 部分功能是从文件导入数据的功能。该功能由DataImporter该类提供，该类假定需要花费很多时间进行初始化。这可能是因为实例初始化DataImporter时，DataImporter实例需要打开文件并将其内容读入内存。

一个DataManager实例可以管理其数据，而无需从文件中导入数据，因此创建本身DataImporter时不需要创建新实例DataManager。相反，DataImporter如果在第一次使用实例时创建该实例会更有意义。

因为它是用lazy修饰符标记的，所以只有当该属性第一次被访问时才会创建该属性的DataImporter实例，例如查询属性时： importerimporterfilename

```
print(manager.importer.filename)
// the DataImporter instance for the importer property has now been created
// Prints "data.txt"
```

> 注意
>
> 如果一个标有lazy修饰符的属性被多个线程同时访问，并且该属性尚未初始化，则不能保证该属性仅被初始化一次。

存储的属性和实例变量

如果您有Objective-C的经验，您可能知道它提供了*两种*方法将值和引用存储为类实例的一部分。除了属性之外，您还可以使用实例变量作为存储在属性中的值的后备存储。

Swift将这些概念统一为一个属性声明。Swift属性没有相应的实例变量，并且不直接访问属性的后备存储。这种方法避免了在不同情况下如何访问价值以及将财产的声明简化为单一的明确声明的混淆。有关该属性的所有信息（包括其名称，类型和内存管理特性）都在单个位置中定义为该类型定义的一部分。

计算属性

除了存储的属性外，类，结构和枚举还可以定义*计算的属性*，这些*属性*实际上并不存储值。相反，他们提供了一个getter和一个可选的setter来间接检索和设置其他属性和值。

```
struct Point {
    var x = 0.0, y = 0.0
}
struct
    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)
print("square.origin is now at (\(square.origin.x), \(square.origin.y))")
// Prints "square.origin is now at (10.0, 10.0)"
```

这个例子定义了三种用于处理几何形状的结构：

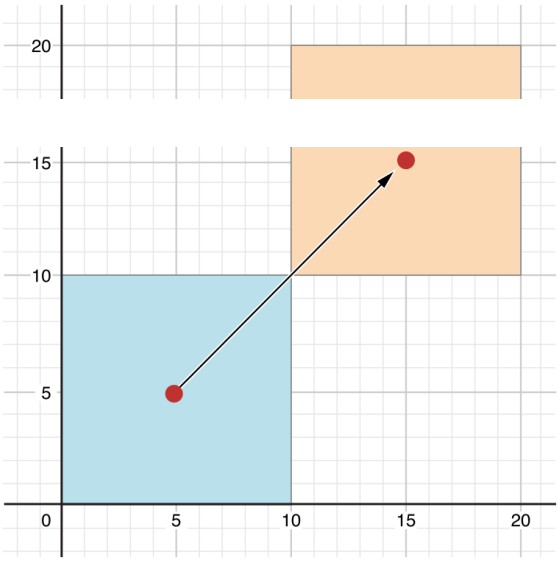
- `Point` 封装点的x坐标和y坐标。
- `Size`封装a `width`和a `height`。
- `Rect` 通过原点和大小定义一个矩形。

该`Rect`结构还提供了一个计算属性`center`。的当前中心位置`Rect`总是可以从它的确定`origin`和`size`，所以你不需要中心点存储为一个明确的`Point`价值。相反，`Rect`为一个名为的计算变量定义一个自定义getter和setter `center`，以使您能够使用该矩形`center`，就好像它是一个真正的存储属性。

上面的例子创建了一个`Rect`名为的新变量`square`。该`square`变量初始化的起点为`(0, 0)`，宽度和高度为`10`。这个正方形由下图中的蓝色方块表示。

然后通过点语法（）来访问 该`square`变量的`center`属性`square.center`，这会导致`center`调用getter 来检索当前属性值。getter不是返回一个现有的值，而是实际计算并返回一个新值`Point`来表示平方中心。从上面可以看出，吸气剂正确地返回了一个中心点`(5, 5)`。

`center`然后 将该属性设置为一个新的值`(15, 15)`，该值将向上和向右移动到下图中橙色方块所示的新位置。设置`center`属性将调用setter `center`，它将修改存储属性的值x和y值`origin`，并将该方块移动到新的位置。



速记制定者声明

如果计算属性的设置者没有为要设置的新值定义名称，`newValue`则使用默认名称。这是一个替代版本的`Rect`结构，它利用了这个简写符号：

```
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

具有getter但没有setter的*计算属性*被称为*只读计算属性*。只读计算属性总是返回一个值，可以通过点语法访问，但不能设置为不同的值。

> 注意
>
> 您必须将计算属性（包括只读计算属性）声明为具有`var`关键字的变量属性，因为它们的值不是固定的。该`let`关键字仅用于常量属性，以表明它们的值一旦被设置为实例初始化的一部分就无法更改。

您可以通过删除`get`关键字及其大括号来简化只读计算属性的声明：

```
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)
print("the volume of fourByFiveByTwo is \(fourByFiveByTwo.volume)")
```

```
// Prints "the volume of fourByFiveByTwo is 40.0"
```

这个例子定义了一个新的结构叫做Cuboid，其表示与3D矩形框width，height和depth特性。这个结构也有一个只读的计算属性
含糊不清哪个值
使外部用户能够发现其当前计算量是非常有用的。

财产观察员

物业观察员观察并回应物业价值的变化。每次设置属性值时都会调用属性观察器，即使新值与属性的当前值相同。

您可以将属性观察器添加到您定义的任何存储属性中，但惰性存储属性除外。您也可以通过重写子类中的属性来将属性观察器添加到任何继承的属性（无论是存储还是计算）。您不需要为非重载计算属性定义属性观察器，因为您可以观察并响应计算属性设置器中对其值的更改。物业压倒一切的描述重写。

您可以选择在属性上定义这些观察者中的一个或两个：

- willSet 在值被存储之前被调用。
- didSet 在新值被存储后立即被调用。

如果你实现了一个willSet观察者，它将传递新的属性值作为常量参数。您可以为此参数指定一个名称作为willSet实现的一部分。如果您未在实现中编写参数名称和括号，则该参数可用默认参数名称newValue。

同样，如果你实现了一个didSet观察者，它会传递一个包含旧属性值的常量参数。您可以命名参数或使用默认参数名称oldValue。如果您为其自己的didSet观察者中的属性赋值，则分配的新值将替换刚刚设置的值。

> 注意
>
> 在调用超类初始化程序之后，如果在子类初始化程序中设置了属性，则将调用超类属性的观察者willSet和didSet观察者。在超类初始化程序被调用之前，它们不会在类正在设置自己的属性时被调用。
>
> 有关初始化程序委托的更多信息，请参阅初始化程序委派的值类型和初始化程序委派的类类型。

这里是一个例子willSet和didSet行动。下面的例子定义了一个叫做的新类StepCounter，它跟踪了一个人在步行时所需的步数总数。该课程可与来自计步器或其他步数计数器的输入数据一起使用，以在日常工作中跟踪个人的锻炼。

```
class StepCounter {
    var totalSteps: Int = 0 {
        willSet(newTotalSteps) {
            print("About to set totalSteps to \(newTotalSteps)")
        }
        didSet {
            if totalSteps > oldValue  {
                print("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
```

这个StepCounter类声明了一个totalSteps类型的属性Int。这是一个存储的财产willSet和didSet观察员。

在willSet和didSet观察员totalSteps每当属性分配一个新的值被调用。即使新值与当前值相同，情况也是如此。

这个例子的willSet观察者newTotalSteps为即将到来的新值使用自定义参数名称。在这个例子中，它只是打印出即将设置的值

该didSet观测器的值后调用totalSteps被更新。它将新价值totalSteps与旧价值进行比较。如果总步数增加，则会打印一条消息以指示已采取了多少新步骤。该didSet观察者不提供旧值自定义参数名称，默认的名称oldValue来代替。

> 注意
>
> 如果您通过令观察家属性的功能，作为在输出参数中，willSet和didSet观察员始终调用。这是因为输入参数的copy-in copy-out内存模型：该值总是写回到函数结尾的属性。有关输入输出参数的详细讨论，请参阅输入输出参数。

全局和局部变量

上面描述的用于计算和观察属性的功能也可用于*全局变量*和*局部变量*。全局变量是在任何函数，方法，闭包或类型上下文之外定义的变量。局部变量是在函数，方法或闭包上下文中定义的变量。

前面章节中遇到的全局变量和局部变量都是*存储变量*。存储变量（如存储属性）为特定类型的值提供存储，并允许设置和检索该值。

但是，您也可以在全局或本地范围内定义*计算变量*并为存储的变量定义观察器。计算变量计算它们的值，而不是存储它们，它们以与计算属性相同的方式编写。

> 注意
>
> 全局常量和变量总是以懒惰方式计算，与Lazy Stored Properties类似。与惰性存储的属性不同，全局常量和变量不需要使用lazy修饰符进行标记。
>
> 局部常量和变量永远不会被延迟计算。

类型属性

实例属性是属于特定类型实例的属性。每次创建该类型的新实例时，它都有自己的一组属性值，与其他实例分开。

您也可以定义属于类型本身的属性，而不是该类型的任何一个实例。无论您创建多少个该类型的实例，只会有这些属性的一个副本。这些属性称为*类型属性*。

Type属性可用于定义对特定类型的*所有*实例通用的值，例如所有实例可以使用的常量属性（如C中的静态常量）或存储全局全部值的变量属性该类型的实例（如C中的静态变量）。

存储的类型属性可以是变量或常量。计算类型属性总是以变量属性的形式声明为变量属性。

> 注意
>
> 与存储的实例属性不同，您必须始终将存储的类型属性设置为默认值。这是因为类型本身没有初始化程序，可以在初始化时将值赋给存储的类型属性。
>
> 在第一次访问时，存储类型属性会被懒惰地初始化。它们保证只被初始化一次，即使是在多个线程同时访问的情况下，也不需要使用lazy修饰符进行标记。

类型属性语法

在C和Objective-C中，将静态常量和与某个类型关联的变量定义为全局静态变量。然而，在Swift中，类型属性作为类型定义的一部分写入类型的外部花括号中，并且每个类型属性都被明确限定为它所支持的类型。

在本页

您可以使用static关键字定义类型属性。对于类类型的计算类型属性，可以使用class关键字来代替，以允许子类重写超类的实现。下面的例子显示了存储和计算的类型属性的语法：

```
struct SomeStructure {
    static var storedTypeProperty = "Some value."
    static var computedTypeProperty: Int {

    }
}
enum SomeEnumeration {
    static var storedTypeProperty = "Some value."
    static var computedTypeProperty: Int {
        return 6
    }
}
class SomeClass {
    static var storedTypeProperty = "Some value."
    static var computedTypeProperty: Int {
        return 27
    }
    class var overrideableComputedTypeProperty: Int {
        return 107
    }
}
```

> 注意
>
> 上面计算的类型属性示例适用于只读计算类型属性，但您也可以使用与计算的实例属性相同的语法定义读写计算类型属性。

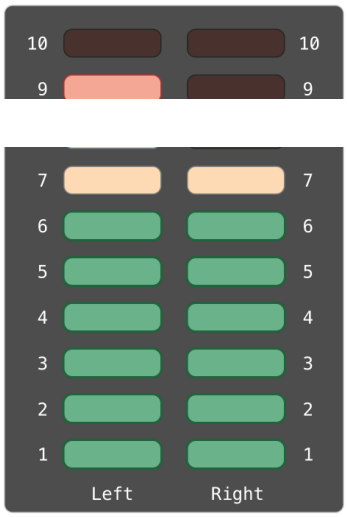
查询和设置类型属性

类型属性被查询并使用点语法设置，就像实例属性一样。但是，类型属性被查询并设置在*类型上*，而不是该*类型*的实例。例如：

```
print(SomeStructure.storedTypeProperty)
// Prints "Some value."
SomeStructure.storedTypeProperty = "Another value."
print(SomeStructure.storedTypeProperty)
// Prints "Another value."
print(SomeEnumeration.computedTypeProperty)
// Prints "6"
print(SomeClass.computedTypeProperty)
// Prints "27"
```

下面的示例使用两个存储的类型属性作为为多个音频通道建模音频电平表的结构的一部分。每个通道都有一个整数音频级别`0`，`10`包括两者之间。

下图说明了这些音频通道中的两个可如何组合来建立立体声音频电平表。当某个频道的音频电平`0`为时，该频道的所有灯都不亮。当音频电平时`10`，该通道的所有灯都点亮。在此图中，左声道的当前级别为`9`，右声道的当前级别为`7`：



上述音频通道由AudioChannel结构的实例表示：

```
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
            }
        }
    }
}
```

该AudioChannel结构定义了两个存储的类型属性以支持其功能。第一个，thresholdLevel定义音频电平可以采用的最大阈值。这是10所有AudioChannel实例的常数值。如果音频信号的值高于10此值，则会将其限制为此阈值（如下所述）。

第二个类型属性是一个名为变量的存储属性maxInputLevelForAllChannels。这会跟踪*任何*AudioChannel实例接收到的最大输入值。它从一个初始值开始0。

该AudioChannel结构还定义了一个名为的存储实例属性currentLevel，它表示通道当前的音频级别，大小为0to 10。

该currentLevel属性有一个didSet属性观察者来检查currentLevel设置时的值。这位观察员执行两次检查：

- 如果新的值currentLevel大于允许的thresholdLevel，物业观察者帽currentLevel来thresholdLevel。
- 如果currentLevel（任何上限之后）的新值高于*任何*AudioChannel实例先前收到的值，则属性观察器将新currentLevel值存储在maxInputLevelForAllChannelstype属性中。

> 注意
>
> 在这两项检查的第一项中，didSet观察者设置currentLevel为不同的值。但是，这并不会导致观察者再次被调用。

您可以使用AudioChannel结构来创建两个新的音频通道叫leftChannel和rightChannel，代表立体声音响系统的音频电平：

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

如果currentLevel将左声道设置为，则7可以看到maxInputLevelForAllChannelstype属性更新为等于7：

```
leftChannel.currentLevel = 7
print(l
// Prints 7
print(AudioChannel.maxInputLevelForAllChannels)
// Prints "7"
```

如果您尝试currentLevel将右声道设置为，则11可以看到右声道的currentLevel属性被限制为最大值10，并且maxInputLevelForAllChannelstype属性更新为等于10：

```
rightChannel.currentLevel = 11
print(rightChannel.currentLevel)
// Prints "10"
print(AudioChannel.maxInputLevelForAllChannels)
// Prints "10"
```