Swift新特性介绍（三） 属性、方法、下标

存储属性和计算属性

类、结构和枚举都能够定义存储属性和计算属性。其中存储属性就是常见的形式，又分为变量属性和常量属性，如:

```
struct Point {
    var x = 0.0, y = 0.0
}
struct Size {
    var width = 0.0, height = 0.0
}
```

计算属性本身不是一个值，但是它提供`getter`和`setter`来间接地使用和设置存储属性的值:

```
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)"
```

上面例子中的`center`就是一个计算属性

如果只有`getter`没有`setter`，那这个计算属性就是只读计算属性。

属性观察者

属性观察者可以根据属性值的变化做出响应，当代码尝试修改属性的值时就会被条用，主要有两种：`willSet`：设置属性前被调用；`didSet`：设置属性后被调用：

```
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
```

类型属性

跟实例属性一样，类、结构和枚举类型还可以定义类型属性，在Swift中需要放在类型定义中进行，结构和枚举用关键词`static`定义，类用`class`关键词定义类型属性：

```
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
    }
}
```

在使用的时候，可以直接用`类名.属性名`的形式来调用：

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

方法

Swift中除了类可以定义方法以外，结构和枚举也可以定义方法：

```
struct Point {
    var x = 0.0, y = 0.0
    func isToTheRightOfX(x: Double) -> Bool {
        return self.x > x
    }
}
let somePoint = Point(x: 4.0, y: 5.0)
if somePoint.isToTheRightOfX(1.0) {
    println("This point is to the right of the line where x == 1.0")
}
// prints "This point is to the right of the line where x == 1.0"
```

类型方法

和类型属性类似，类、结构和枚举还可以定义类型方法，也是分别使用`static`和`class`关键词定义：

```
class SomeClass {
    class func someTypeMethod() {
        // type method implementation goes here
    }
}
SomeClass.someTypeMethod()
```

下标

Swift中的下标可以定义更加复杂的功能，用下面这个例子来说明：

```
struct Matrix {
    let rows: Int, columns: Int
    var grid: Double[]
    init(rows: Int, columns: Int) {
        self.rows = rows
        self.columns = columns
        grid = Array(count: rows * columns, repeatedValue: 0.0)
    }
    func indexIsValidForRow(row: Int, column: Int) -> Bool {
        return row >= 0 && row < rows && column >= 0 && column < columns
    }
    subscript(row: Int, column: Int) -> Double {
        get {
            assert(indexIsValidForRow(row, column: column), "Index out of
range")
            return grid[(row * columns) + column]
        }
        set {
            assert(indexIsValidForRow(row, column: column), "Index out of
range")
            grid[(row * columns) + column] = newValue
        }
    }
}
```

这个矩阵定义了一个二维的下标，分别接受两个`Int`类型的参数，返回一个`Double`值，其中`get`和`set`分别是取值和赋值时会做的操作。