The Swift Programming Language in Chinese

https://github.com/letsswift

http://letsswift.com/

June 15, 2014

目录

1 Swift 中文教程（十一）方法

方法是关联到一个特定类型的函数。类、结构、枚举所有可以定义实例方法, 封装特定任务和功能处理给定类型的一个实例。类、结构、枚举类型还可以定义方法, 相关的类型本身。类型方法类似于 objective – c 类方法。

结构和枚举可以定义方法 swift 与 C 和 objective – C 是一个重大的区别。在 objective – c 中, 类是唯一类型可以定义方法。在 swift, 你可以选择是否要定义一个类, 结构, 或枚举, 还有你定义方法类型的灵活性创造。

1.1 实例方法

实例方法是属于一个特定的类，结构或枚举实例的功能。他们支持这些实例的功能，无论是通过提供方法来访问和修改实例属性，或提供的功能与实例的目的。实例方法具有完全相同的语法功能，如功能描述你所属的类型的打开和关闭括号内写一个实例方法。一个实例方法具有隐式访问所有其他实例方法和该类型的属性。一个实例方法只能在它所属的类的特定实例调用，它不能访问一个不存在的实例。这里，定义了一个简单的计数器类，它可以用来计数一个动作发生的次数的示例:

```
class Counter {
    var count = 0
    func increment() {
        count++
    }
    func incrementBy(amount: Int) {
        count += amount
    }
    func reset() {
        count = 0
    }
}
```

counter 类可以定义三个实例方法:- 增量递增计数器 1。- incrementBy(amount:Int) 由指定的整数金额递增计数器。- 重置将计数器的值重置为零。

计数类也声明了一个变量属性, 统计, 跟踪当前的计数器值。

你调用实例方法具有相同点语法的属性

```
let counter = Counter()
// the initial counter value is 0
counter.increment()
// the counter's value is now 1
counter.incrementBy(5)
// the counter's value is now 6
counter.reset()
// the counter's value is now 0
```

1.1.1 本地和外部参数名称的方法

函数参数可以有一个本地名称 (在函数体内使用) 和外部名称 (在调用函数时使用), 所述外部参数名称。方法参数也是如此, 因为方法与类型相关的函数。然而, 本地名称和外部名称的默认行为是不同的函数和方法。

方法在 Swift 非常类似于 objective – c 的同行。在 objective – c 中, 一个方法的名称在 Swift 通常是指使用 preposition 等方法的第一个参数„ 或者, 就像在 incrementBy 方法从前面的 counter 类的例子。使用可以被解读为一个判断的方法叫做 preposition。Swift 使这个方法建立命名约定易于编写通过使用一个不同的默认方法。

具体来说,Swift 给第一个参数名称方法默认本地参数名称, 并给出第二和后续的参数名称默认本地和外部参数名称。这个约定可以在熟悉的 objective – c 中调用到, 并使得表达方法调用而不需要符合你的参数名称。

考虑这个替代版本的 counter 类, 它定义了一个更复杂的形式的 incrementBy 方法:

```
class Counter {
    var count: Int = 0
        func incrementBy(amount: Int, numberOfTimes: Int) {
            count += amount * numberOfTimes
        }
}
```

这有两个 parameters-amount 和 numberOfTimes incrementBy 方法。默认情况下,Swift 将 amount 视为本地名称, 但将 numberOfTimes 视为本地和外部名称。您调用的方法如下:

```
let counter = Counter()
```

```
counter.incrementBy(5, numberOfTimes: 3)
// counter value is now 15
```

你不需要定义一个外部参数名称为第一个参数值, 因为它是明确的函数名 incrementBy。然而, 第二个参数是由外部参数名称进行限定。

这种默认行为有效的外部方法, 如果你有 numberOfTimes 参数之前写了一个 hash 符号（#）:

```
func incrementBy(amount: Int, #numberOfTimes: Int) {
    count += amount * numberOfTimes
}
```

上面描述的默认行为在 Swift 写入相同的方法定义，语法类似于 objective – c，可以更方便地被调用。

1.1.2 修改外部参数名称的行为方法

有时是有用的提供一个外部方法的第一个参数的参数名称, 即使这不是默认行为。你自己可以添加一个显式的外部名称, 或者你可以用一个散列前缀的名字的第一个参数标志使用本地名称作为外部的名字。相反, 如果你不想为第二个提供外部名称或后续参数的方法, 覆盖默认行为通过使用下划线字符 (_) 作为一个明确的外部参数名称参数。

1.1.3 Self 属性

一个类型的每个实例都有所谓的一个隐含 self 属性，它是完全等同于该实例本身。您可以使用这个隐含的 self 属性来引用当前实例中它自己的实例方法。

在上面的例子中，增量方法也可以写成这样:

```
func increment() {
    self.count++
}
```

在实践中，你不需要写 self，这在你的代码会非常频繁。如果你没有明确写 self，Swift 假设你是指当前实例的属性或方法，每当你使用一个方法中一个已知的属性或方法名。这个假设是证明了里边三个实例方法的计数器使用 count (rather than self.count) 的。

主要的例外发生在一个实例方法的参数名称相同的名称作为该实例的属性。在这种情况下，参数名称的优先，有必要参考属性更多合格的方式。您可以使用隐式的自我属性的参数名和属性名来区分。

如果一个方法参数叫 x，还有一个实例属性也叫 x，在 Swift 中可以自动对两个 x 消除歧义，不会混淆。

```
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"
```

如果没有 self 前缀，Swift 将假定 x 的两种用法称为 X 的方法参数

1.1.4 修改值类型的实例方法

结构和枚举值类型。默认情况下, 一个值类型的属性不能修改它的实例方法

然而, 如果您需要修改的属性结构或枚举在一个特定的方法, 你可以选择该方法的变化行为。但任何更改都会使它得编写的方法结束时回到原来的结构。当该方法结束时还可以分配一个完全新的实例对其隐含的 self 属性, 而这个新的实例将取代现有的。

你可以选择这个行为之前将变异的关键字嵌入函数关键字的方法:

```
struct Point {
    var x = 0.0, y = 0.0
    mutating func moveByX(deltaX: Double, y deltaY: Double) {
        x += deltaX
        y += deltaY
    }
}
var somePoint = Point(x: 1.0, y: 1.0)
```

```
somePoint.moveByX(2.0, y: 3.0)
println("The point is now at (\(somePoint.x), \(somePoint.y))")
// prints "The point is now at (3.0, 4.0)"
```

Point 结构上面定义了一个变异 moveByX 方法，它通过一定量移动一个 Point 实例。而不是返回一个新的起点，这种方法实际上会修改在其上调用点。该变异包含被添加到它的定义，使其能够修改其属性。请注意, 您不能调用变异方法结构类型的常数，因为它的属性不能改变，即使它们是可变的特性，如在固定结构实例存储的属性描述：

```
let fixedPoint = Point(x: 3.0, y: 3.0)
fixedPoint.moveByX(2.0, y: 3.0)
// this will report an error
```

1.1.5 分配中的 **self** 变异方法

变异的方法可以分配一个全新的实例隐含的 self 属性。上面所示的点的例子也可以写成下面的方式来代替：

```
struct Point {
    var x = 0.0, y = 0.0
    mutating func moveByX(deltaX: Double, y deltaY: Double) {
        self = Point(x: x + deltaX, y: y + deltaY)
    }
}
```

此版本的突变 moveByX 方法创建一个全新的结构，它的 x 和 y 值被设置到目标位置。调用该方法的结果和早期版本是完全一样的

变异的方法枚举可以设置 self 参数是从同一个枚举不同的成员

```
enum TriStateSwitch {
    case Off, Low, High
    mutating func next() {
        switch self {
            case Off:
                self = Low
            case Low:
```

```
                self = High
            case High:
                self = Off
        }
    }
}
var ovenLight = TriStateSwitch.Low
ovenLight.next()
// ovenLight is now equal to .High
ovenLight.next()
// ovenLight is now equal to .Off
```

这个例子定义了一个三态开关枚举。三种不同的功率状态之间的切换周期(关, 低, 高)

1.2 类型方法

如上所述, 实例方法的方法要求一个特定类型的实例。您还可以定义该类型自身的方法，这种方法被称为 type 方法，您显示的 type 方法直接在类结构体里面用 class func 开头，对于枚举和结构来说，类型方法是用 static func 开头。

> 请注意; 在 objective – c 中, 您可以定义 type-level 方法仅为 objective – c 类。在 Swift 可以为所有类定义 type-level 方法, 结构, 和枚举。每种方法的显示局限于它所支持的类型。

类型方法调用 dot syntax，就像实例方法。但是，您调用的是类型的方法，而不是该类型的一个实例。这里是你如何调用一个类调用 SomeClass 的一个类型的方法:

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

在类型方法的主体，隐含的 self 属性是指类型本身，而不是该类型的一个实例。对于结构体和枚举，这意味着你可以使用自助静态属性和静态方法的参数消除歧义，就像你做的实例属性和实例方法的参数。

更普遍的是，你一个类型的方法体中使用任何不合格的方法和属性名称会参考其他 type-level 方法和属性。一种方法可以调用另一个类的方法与其他方法的名称，而不需要与类型名称前缀了。同样，结构和枚举类型的方法可以使用静态属性的名称，没有类型名称前缀访问静态属性。

下面的例子定义了一个名为 LevelTracker 结构，它通过游戏的不同层次或阶段跟踪球员的进步。这是一个单人游戏，但可以存储的信息为一个单一的设备上的多个玩家。

所有的游戏的水平 (除了一级) 当游戏第一次玩。每当玩家完成一个级别，该级别解锁设备上的所有玩家。LevelTracker 结构使用静态属性和方法来跟踪哪些级别的比赛已经解锁。它还跟踪当前个别球员水平

```
struct LevelTracker {
    static var highestUnlockedLevel = 1
    static func unlockLevel(level: Int) {
        if level > highestUnlockedLevel {
            highestUnlockedLevel = level
        }
    }
    static func levelIsUnlocked(level: Int) -> Bool {
        return level <= highestUnlockedLevel
    }
    var currentLevel = 1
    mutating func advanceToLevel(level: Int) -> Bool {
        if LevelTracker.levelIsUnlocked(level) {
            currentLevel = level
            return true
        } else {
            return false
        }
    }
}
```

该 LevelTracker 结构跟踪任何玩家解锁的最高水平。这个值是存储在一个名为 highestUnlockedLevel 的静态属性。

LevelTracker 还定义了两种类型的功能与 highestUnlockedLevel，首先是一种叫做 unlockLevel 功能，每当一个新的水平解锁都会用来更新 highestUnlockedLevel，第二个是 levelIsUnlocked 功能，如果一个特定的水平数已经解锁，

就会返回 ture。注意, 这些类型的方法可以访问 highestUnlockedLevel 静态属性但是你需要把它写成 LevelTracker.highestUnlockedLevel)。

除了它的静态属性和类型的方法，LevelTracker 通过游戏追踪每个玩家的进度。它使用被称为 currentLevel 实例属性来跟踪玩家级别。

为了帮助管理 urrentLevel 属性，advanceToLevel LevelTracker 定义一个实例方法。这种方法更新 currentLevel 之前，用来检查是否要求新的水平已经解除锁定。该 advanceToLevel 方法返回一个布尔值来指示它是否能够设置 currentLevel。

该 LevelTracker 结构使用 Player 类, 如下所示, 跟踪和更新单个球员的进步:

```
class Player {
    var tracker = LevelTracker()
    let playerName: String
    func completedLevel(level: Int) {
        LevelTracker.unlockLevel(level + 1)
        tracker.advanceToLevel(level + 1)
    }
    init(name: String) {
        playerName = name
    }
}
```

Player 类创建 LevelTracker 的一个新实例来跟踪球员的进步。它也提供了一个名为 completedLevel 方法，每当玩家到达一个特定的级别，这种方法就会解锁一个新的级别和进度并把玩家移到下一个级别。(advanceToLevel 返回的布尔值将被忽略, 因为已知被调用 LevelTracker.unlockLevel。)

您可以创建一个新球员 Player 的实例, 看看当玩家完成一个级别会发生什么:

```
var player = Player(name: "Argyrios")
player.completedLevel(1)
println("highest unlocked level is now \(LevelTracker.highestUnlockedLevel)")
// prints "highest unlocked level is now 2"
```

如果你创建第二个球员, 你想尝试移动到尚未被游戏解锁的级别，就会出现当前级别失败

```
player = Player(name: "Beto")
if player.tracker.advanceToLevel(6) {
    println("player is now on level 6")
} else {
    println("level 6 has not yet been unlocked")
}
// prints "level 6 has not yet been unlocked"
```

参考文献