翻译：pp-prog　校对：地

# 方法（Methods）

本页包含内容：



方法是与某些特定类型相关联的函数。类、结构体、枚举都可以定义实例方法；实例方法为给定类型的实例封装了具体的任务与功能。类、结构体、枚举也可以定义类型方法；类型方法与类型本身相关联。类型方法与 Objective-C 中的类方法（class methods）相似。

结构体和枚举能够定义方法是 Swift 与 C/Objective-C 的主要区别之一。在 Objective-C 中，类是唯一能定义方法的类型。但在 Swift 中，你不仅能选择是否要定义一个类/结构体/枚举，还能灵活的在你创建的类型（类/结构体/枚举）上定义方法。

# 实例方法(Instance Methods)

实例方法是属于某个特定类、结构体或者枚举类型实例的方法。实例方法提供访问和修改实例属性的方法或提供与实例目的相关的功能，并以此来支撑实例的功能。实例方法的语法与函数完全一致，详情参见函数。

实例方法要写在它所属的类型的前后大括号之间。实例方法能够隐式访问它所属类型的所有的其他实例方法和属性。实例方法只能被它所属的类的某个特定实例调用。实例方法不能脱离于现存的实例而被调用。

下面的例子，定义一个很简单的类`Counter`，`Counter`能被用来对一个动作发生的次数进行计数：

```
class Counter {
```

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

`Counter`类定义了三个实例方法：

- `increment`让计数器按一递增；
- `incrementBy(amount: Int)`让计数器按一个指定的整数值递增；
- `reset`将计数器重置为0。

`Counter`这个类还声明了一个可变属性`count`，用它来保持对当前计数器值的追踪。

和调用属性一样，用点语法（dot syntax）调用实例方法：

```
let counter = Counter()
// 初始计数值是0
counter.increment()
// 计数值现在是1
counter.incrementBy(5)
// 计数值现在是6
counter.reset()
// 计数值现在是0
```

# 方法的局部参数名称和外部参数名称(Local and External Parameter Names for Methods)

函数参数可以同时有一个局部名称（在函数体内部使用）和一个外部名称（在调用函数时使用），详情参见函数的外部参数名。方法参数也一样（因为方法就是函数，只是这个函数与某个类型相关联了）。但是，方法

和函数的局部名称和外部名称的默认行为是不一样的。

Swift 中的方法和 Objective-C 中的方法极其相似。像在 Objective-C 中一样，Swift 中方法的名称通常用一个介词指向方法的第一个参数，比如：`with`，`for`，`by`等等。前面的`Counter`类的例子中`incrementBy`方法就是这样的。介词的使用让方法在被调用时能像一个句子一样被解读。和函数参数不同，对于方法的参数，Swift 使用不同的默认处理方式，这可以让方法命名规范更容易写。

具体来说，Swift 默认仅给方法的第一个参数名称一个局部参数名称;默认同时给第二个和后续的参数名称局部参数名称和外部参数名称。这个约定与典型的命名和调用约定相适应，与你在写 Objective-C 的方法时很相似。这个约定还让表达式方法在调用时不需要再限定参数名称。

看看下面这个`Counter`的另一个版本（它定义了一个更复杂的`incrementBy`方法）：

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

`incrementBy`方法有两个参数：`amount`和`numberOfTimes`。默认情况下，Swift 只把`amount`当作一个局部名称，但是把`numberOfTimes`即看作局部名称又看作外部名称。下面调用这个方法：

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

你不必为第一个参数值再定义一个外部变量名：因为从函数名`incrementBy`已经能很清楚地看出它的作用。但是第二个参数，就要被一个外部参数名称所限定，以便在方法被调用时明确它的作用。

这种默认的行为能够有效的处理方法（method）,类似于在参数`numberOfTimes`前写一个井号（`#`）：

```
func incrementBy(amount: Int, #numberOfTimes: Int) {
 count += amount * numberOfTimes
}
```
这种默认行为使上面代码意味着：在 Swift 中定义方法使用了与 Objective-C 同样的语法风格，并且方法将以自然表达式的方式被调用。

## 修改方法的外部参数名称(Modifying External Parameter Name Behavior for Methods)

有时为方法的第一个参数提供一个外部参数名称是非常有用的，尽管这不是默认的行为。你可以自己添加一个显式的外部名称或者用一个井号（`#`）作为第一个参数的前缀来把这个局部名称当作外部名称使用。

相反，如果你不想为方法的第二个及后续的参数提供一个外部名称，可以通过使用下划线（`_`）作为该参数的显式外部名称，这样做将覆盖默认行为。

## `self`属性(The self Property)

类型的每一个实例都有一个隐含属性叫做`self`，`self`完全等同于该实例本身。你可以在一个实例的实例方法中使用这个隐含的`self`属性来引用当前实例。

上面例子中的`increment`方法还可以这样写：

```
func increment() {
  self.count++
}
```
实际上，你不必在你的代码里面经常写`self`。不论何时，只要在一个方法中使用一个已知的属性或者方法名称，如果你没有明确的写`self`，Swift 假定你是指当前实例的属性或者方法。这种假定在上面的`Counter`中已经示范了：`Counter`中的三个实例方法中都使用的是`count`（而不是

self.count）。

使用这条规则的主要场景是实例方法的某个参数名称与实例的某个属性名称相同的时候。在这种情况下，参数名称享有优先权，并且在引用属性时必须使用一种更严格的方式。这时你可以使用self属性来区分参数名称和属性名称。

下面的例子中，self消除方法参数x和实例属性x之间的歧义：

```
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")
}
// 输出 "This point is to the right of the line where
x == 1.0"（这个点在x等于1.0这条线的右边）
```

如果不使用self前缀，Swift 就认为两次使用的x都指的是名称为x的函数参数。

# 在实例方法中修改值类型(Modifying Value Types from Within Instance Methods)

结构体和枚举是值类型。一般情况下，值类型的属性不能在它的实例方法中被修改。

但是，如果你确实需要在某个具体的方法中修改结构体或者枚举的属性，你可以选择变异(mutating)这个方法，然后方法就可以从方法内部改变它的属性；并且它做的任何改变在方法结束时还会保留在原始结构中。方法还可以给它隐含的self属性赋值一个全新的实例，这个新实例在方法

结束后将替换原来的实例。

要使用变异方法， 将关键字mutating 放到方法的func关键字之前就可以了：

```
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))")
// 输出 "The point is now at (3.0, 4.0)"
```

上面的Point结构体定义了一个变异方法（mutating method）moveByX，moveByX用来移动点。moveByX方法在被调用时修改了这个点，而不是返回一个新的点。方法定义时加上mutating关键字,这才让方法可以修改值类型的属性。

注意：不能在结构体类型常量上调用变异方法，因为常量的属性不能被改变，即使想改变的是常量的变量属性也不行，详情参见存储属性和实例变量

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

## 在变异方法中给self赋值(Assigning to self Within a Mutating Method)

变异方法能够赋给隐含属性self一个全新的实例。上面Point的例子可以

用下面的方式改写：

```
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)
  }
}
```

新版的变异方法`moveByX`创建了一个新的结构（它的 x 和 y 的值都被设定为目标值）。调用这个版本的方法和调用上个版本的最终结果是一样的。

枚举的变异方法可以把`self`设置为相同的枚举类型中不同的成员：

```
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 现在等于 .High
ovenLight.next()
// ovenLight 现在等于 .Off
```

上面的例子中定义了一个三态开关的枚举。每次调用`next`方法时，开关在不同的电源状态（`Off`，`Low`，`High`）之前循环切换。

# 类型方法(Type Methods)

实例方法是被类型的某个实例调用的方法。你也可以定义类型本身调用的方法，这种方法就叫做类型方法。声明类的类型方法，在方法的`func`关键字之前加上关键字`class`；声明结构体和枚举的类型方法，在方法的`func`关键字之前加上关键字`static`。

注意：

在 Objective-C 里面，你只能为 Objective-C 的类定义类型方法（type-level methods）。在 Swift 中，你可以为所有的类、结构体和枚举定义类型方法：每一个类型方法都被它所支持的类型显式包含。

类型方法和实例方法一样用点语法调用。但是，你是在类型层面上调用这个方法，而不是在实例层面上调用。下面是如何在`SomeClass`类上调用类型方法的例子：

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

在类型方法的方法体（body）中，`self`指向这个类型本身，而不是类型的某个实例。对于结构体和枚举来说，这意味着你可以用`self`来消除静态属性和静态方法参数之间的歧义（类似于我们在前面处理实例属性和实例方法参数时做的那样）。

一般来说，任何未限定的方法和属性名称，将会来自于本类中另外的类型级别的方法和属性。一个类型方法可以调用本类中另一个类型方法的名称，而无需在方法名称前面加上类型名称的前缀。同样，结构体和枚举的类型方法也能够直接通过静态属性的名称访问静态属性，而不需要类型名称前缀。

下面的例子定义了一个名为`LevelTracker`结构体。它监测玩家的游戏发展情况（游戏的不同层次或阶段）。这是一个单人游戏，但也可以存储多

个玩家在同一设备上的游戏信息。

游戏初始时，所有的游戏等级（除了等级 1）都被锁定。每次有玩家完成一个等级，这个等级就对这个设备上的所有玩家解锁。`LevelTracker`结构体用静态属性和方法监测游戏的哪个等级已经被解锁。它还监测每个玩家的当前等级。

```
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
    }
  }
}
```

`LevelTracker`监测玩家的已解锁的最高等级。这个值被存储在静态属性`highestUnlockedLevel`中。

`LevelTracker`还定义了两个类型方法与`highestUnlockedLevel`配合工作。第一个类型方法是`unlockLevel`：一旦新等级被解锁，它会更新`highestUnlockedLevel`的值。第二个类型方法是`levelIsUnlocked`：如果某个给定的等级已经被解锁，它将返回`true`。（注意：尽管我们没有使用类似`LevelTracker.highestUnlockedLevel`的写法，这个类型方法还是能够访问静态属性`highestUnlockedLevel`）

除了静态属性和类型方法，`LevelTracker`还监测每个玩家的进度。它用

实例属性currentLevel来监测玩家当前的等级。

为了便于管理currentLevel属性，LevelTracker定义了实例方法advanceToLevel。这个方法会在更新currentLevel之前检查所请求的新等级是否已经解锁。advanceToLevel方法返回布尔值以指示是否能够设置currentLevel。

下面，Player类使用LevelTracker来监测和更新每个玩家的发展进度：

```
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类创建一个新的LevelTracker实例来监测这个用户的发展进度。他提供了completedLevel方法：一旦玩家完成某个指定等级就调用它。这个方法为所有玩家解锁下一等级，并且将当前玩家的进度更新为下一等级。（我们忽略了advanceToLevel返回的布尔值，因为之前调用LevelTracker.unlockLevel时就知道了这个等级已经被解锁了）。

你还可以为一个新的玩家创建一个Player的实例，然后看这个玩家完成等级一时发生了什么：

```
var player = Player(name: "Argyrios")
player.completedLevel(1)
println("highest unlocked level is now \
(LevelTracker.highestUnlockedLevel)")
// 输出 "highest unlocked level is now 2"（最高等级现在是
2
）
```

如果你创建了第二个玩家，并尝试让他开始一个没有被任何玩家解锁的等

级，那么这次设置玩家当前等级的尝试将会失败：

```
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")
}
// 输出 "level 6 has not yet been unlocked"（等级6还没被解锁）
```