方法

4.2

这是一篇社区协同翻译的文章，你可以点击右边区块信息里的『改进』按钮向译者提交改进建议。

方法 是与特定类型相关联的函数。类、结构体和枚举都可以定义实例方法，这些方法会封装特定的任务和功能用于处理给定类型的实例。类、结构体和枚举还可以定义类型方法。类型方法类似于 Objective-C 中的类方法。

可以在结构体和枚举中定义方法，是 Swift 与 C 和 Objective-C 的主要区别。在 Objective-C 中，类是唯一可以定义方法的类型。但在 Swift 中，由于类、结构体和枚举都支持定义方法，所以在选择定义类型时你有更多的选择。

实例方法

实例方法 是属于特定类、结构或枚举实例的函数。它们的功能是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。实例方法与函数具有完全相同的语法，如 函数 中所述。

你在其所属类型的开始和结束括号内编写实例方法。实例方法可以隐式访问该类型的所有其它实例方法和属性。实例方法只能被它所属类型的实例调用。如果没有实例，不能单独调用它。

下面是一个简单 `Counter` 类的示例，可用于计算操作发生的次数：

```
class Counter {
    var count = 0
    func increment() {
        count += 1
    }
    func increment(by amount: Int) {
        count += amount
    }
```

```
    func reset() {
        count = 0
    }
}
```

`Counter` 类中定义了三个实例方法

- `increment()` 将计数器增加 `1` 。
- `increment(by: Int)` 将计数器增加指定的值。
- `reset()` 将计数器重置为 0。

`Counter` 类中还声明了一个可变的属性 `count` ，用来记录计数器当前的值。

和调用属性一样，你可以使用点语法来调用实例方法：

```
let counter = Counter()
// 计数器初始值为 0
counter.increment()
// 计数器的值现在为 1
counter.increment(by: 5)
// 计数器的值现在为 6
counter.reset()
// 计数器的值现在为 0
```

方法的参数可以同时拥有一个参数名称（在方法内使用）和一个参数标签（在调用方法时使用），和 函数的参数标签和参数名 中描述的一样。方法的参数也是如此，因为方法也是函数只不过它和类相关联。

self 属性

类型的每个实例都有一个名为 `self` 的隐式属性，它与实例本身完全等效。你可以使用 `self` 属性来调用自己的实例方法。

上面例子中的 `increment()` 方法也可以这样实现的：

```
func increment() {
    self.count += 1
}
```

实际上，你并不需要经常在代码中书写 `self` 。如果没有显式地书写 `self` ，那么只要在方法中使用了该类的属性或方法名称，Swift 就会假定你调用了当前实例的属性或方法。

Virle 翻译于 5个月前

0 重译

由 Aufree 审阅

DerekCoder 翻译于 5个前

0 重译

由 Aufree 审阅

在 `Counter` 的三个实例方法中使用 `count` （而不是 `self.count` ）证明了这个假设。

当实例方法的参数名称与该实例的属性名称相同时，这时候就会发生命名冲突问题。在这种情况下，会使用参数名称优先原则，这时候你就需要以更严格的方式调用属性。即你需要用书写 `self` 的方式来调用属性以区分参数名称和属性名称。

这里， `self` 消除了一个名为 `x` 的方法参数和一个也被称为 `x` 的实例属性之间名称相同的歧义：

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

如果没有 `self` 前缀，Swift 会假设 `x` 的两个用法都调用了名为 `x` 的方法参数。

在实例方法中修改值类型

结构体和枚举是 *值类型* 。默认情况下，无法在其实例方法中修改值类型的属性。

但是，如果需要在特定方法中修改结构体或枚举的属性，可以选择将这个方法 *异变* 。然后，该方法就可以异变（即更改）其属性，并且当方法结束时，它所做的任何更改都将写回原始的结构体中。该方法还可以为隐式的 `self` 属性分配一个全新的实例，并且该新实例将在方法结束时替换现有实例。

你可以通过在方法的 `func` 关键字前放置 `mutating` 关键字来选择开启此行为：

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

DerekCoder 翻译于 5个前

0 重译

由 Aufree 审阅

```
somePoint.moveBy(x: 2.0, y: 3.0)
print("The point is now at (\(somePoint.x), \(somePoint.y))")
// 打印 "The point is now at (3.0, 4.0)"
```

上面的 `Point` 结构体定义了一个异变方法 `moveBy(x:y:)` ，它会基于特定的数值移动 `Point` 实例。此方法实际上修改了调用它的那个点，而不是返回一个新点。 `mutating` 关键字被添加到其定义中，以使其能够修改其属性。

请注意，你不能在常量结构体类型上调用异变方法，因为它的属性不能更改，即使它们是变量属性，如 常量结构体实例的存储属性 中所述：

```
let fixedPoint = Point(x: 3.0, y: 3.0)
fixedPoint.moveBy(x: 2.0, y: 3.0)
// 编译器会报错
```

在可变方法中给 self 赋值

可变方法能给隐性的 `self` 属性赋以一个全新的实例。上面以 `Point` 为例的代码也可以用下面这种方式来实现：

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

这个版本的可变方法 `moveBy(x:y:)` 创建了一个新的结构体，且该结构体的 `x` 与 `y` 值都被设定为目标位置。调用这个可变版本的方法与上个版本的方法最终结果一样。

枚举的可变方法可以将隐性的 `self` 参数设置成同一枚举类型中的不同成员。

```
enum TriStateSwitch {
    case off, low, high
    mutating func next() {
        switch self {
        case .off:
            self = .low
        case .low:
            self = .high
        case .high:
            self = .off
```

雪之下八幡机长 翻译于
个月前

0 重译

由 RecherJ 审阅

```
        }
    }
}
var ovenLight = TriStateSwitch.low
ovenLight.next()
// ovenLight 现在等于 .high
ovenLight.next()
// ovenLight 现在等于 .off
```

在这个例子中，我们定义了一个枚举，它拥有三种状态的开关。当 `next()` 方法被调用时，这个开关将在三种不同的电源状态（ `off` ， `low` 和 `high` ）之间循环切换。

类型方法

就像上面所描述的，实例方法是在类型的实例上被调用的方法。你也可以直接在类型本身上定义方法，这样的方法被称作 *类型方法* 。为了明确一个方法是类型方法，你可以在这个方法的 `func` 关键词前加上 `static` 关键词。在类中，也可以使用 `class` 关键词来声明一个类型方法。与 `static` 关键词不同的是，用 `class` 关键词声明的类型方法允许它的子类重写其父类对类型方法的实现。

> 注意
>
> 在 Objective-C 中，你只能为 Objective-C 的类定义类型方法。而在 Swift 中，你可以为所有的类、结构体或枚举定义类型方法。每一个类型方法可被调用的作用域都被其所支持的类型明确划分。

就像实例方法那样，类型方法可以通过点语法来调用。不同的是，你将直接在类上调用类型方法，而不是在类型的实例上调用。你可以像下面这样在一个被叫做 `SomeClass` 的类上调用其类型方法：

```
class SomeClass {
    class func someTypeMethod() {
            // 这里是类型方法的实现细节
    }
}
SomeClass.someTypeMethod()
```

在一个方法的函数体中，隐性属性 `self` 指代其类型本身，而非指代类型的实例。就像你对实例属性与实例方法的参数之间做的那样，这意味着你可以使用 `self` 来消除类属性与

雪之下八幡机长 翻译于
个月前

0 重译

由 RecherJ 审阅

类型方法的参数之间的歧义。

一般来说，你在类型方法函数体内使用的任何非完全标准的方法和属性名称将会引用对应的类型级别的方法和属性。类型方法可以直接使用方法的名称来调用另一个类型方法，而无需使用类型名称作为前缀。同样的，结构体和枚举上的类型方法可以通过使用不带类型名称前缀的类型属性名称来访问该类型属性。

下面的示例定义了一个名为 `LevelTracker` 的结构体，它用来追踪玩家在游戏的不同级别或阶段的进度。这是一款单人游戏，但可以在一台设备上存储多个玩家的信息。

首次玩游戏时，所有的游戏等级（除了第一级）都会被锁定。每当玩家完成一个等级时，该等级就会被设备上的所有玩家解锁。 `LevelTracker` 结构体使用类型属性和方法来追踪游戏的哪些级别已被解锁。它还追踪单个玩家的当前级别。

```
struct LevelTracker {
    static var highestUnlockedLevel = 1
    var currentLevel = 1

    static func unlock(_ level: Int) {
        if level > highestUnlockedLevel { highestUnlockedLevel = level }
    }

    static func isUnlocked(_ level: Int) -> Bool {
        return level <= highestUnlockedLevel
    }

    @discardableResult
    mutating func advance(to level: Int) -> Bool {
        if LevelTracker.isUnlocked(level) {
            currentLevel = level
            return true
        } else {
            return false
        }
    }
}
```

`LevelTracker` 结构体追踪任何玩家解锁的最高级别。该值存储在名为 `highestUnlockedLevel` 的类型属性中。

`LevelTracker` 还定义了两个类型函数来处理 `highestUnlockedLevel` 属性。第一个是名为 `unlock(_:)` 的类型函数，每当解锁一个新级别时它会更新 `highestUnlockedLevel` 的值。第二个是一个名为 `isUnlocked(_:)` 的便捷类型函

数，如果某个特定的级别号已被解锁，则返回 `true` 。（请注意，这些类型方法可以直接使用 `highestUnlockedLevel` 类型属性，而无需将其写为 `LevelTracker.highestUnlockedLevel` 。）

除了类型属性和类型方法之外， `LevelTracker` 还追踪单个玩家在游戏中的进度。它使用一个名为 `currentLevel` 的实例属性来跟踪玩家当前的游戏等级。

为了帮助管理 `currentLevel` 属性， `LevelTracker` 定义了一个名为 `advance(to:)` 的实例方法。在更新 `currentLevel` 之前，此方法先检查所请求的新级别是否已解锁。 `advance(to:)` 方法返回一个布尔值，表示它是否实际上能够设置 `currentLevel` 。因为调用 `advance(to:)` 方法时我们有时候可能会需要忽略返回值，所以这个函数用 `@discardableResult` 特性标记。有关此特性的更多信息，请参阅 特性 。

`LevelTracker` 结构体与 `Player` 类一起使用，如下所示，用于追踪和更新单个玩家的进度：

```
class Player {
    var tracker = LevelTracker()
    let playerName: String
    func complete(level: Int) {
        LevelTracker.unlock(level + 1)
        tracker.advance(to: level + 1)
    }
    init(name: String) {
        playerName = name
    }
}
```

`Player` 类创建一个 `LevelTracker` 实例来跟追踪该玩家的进度。它还提供了一个名为 `complete(level:)` 的方法，只要玩家完成特定级别就会调用它。此方法会为所有玩家解锁下一关，并更新当前玩家的进度到下一关。（忽略 `advance(to:)` 的布尔性返回值，是因为已经知道通过调用前一行的 `LevelTracker.unlock(_:)` 来解锁该级别。）

你可以为新玩家创建一个 `Player` 类的实例，看看当玩家完成第一级时会发生什么：

```
var player = Player(name: "Argyrios")
player.complete(level: 1)
print("highest unlocked level is now \(LevelTracker.highestUnlockedLevel)")
// 打印 "highest unlocked level is now 2"
```

如果你创建了第二个玩家，并尝试将其移动到游戏中任何玩家都尚未解锁的等级，则设置玩

DerekCoder 翻译于 5个前

0 重译

由 Aufree 审阅

DerekCoder 翻译于 5个前

0 重译

由 Aufree 审阅

家当前等级的尝试会失败：

```
player = Player(name: "Beto")
if player.tracker.advance(to: 6) {
    print("player is now on level 6")
} else {
    print("level 6 has not yet been unlocked")
}
// 打印 "level 6 has not yet been unlocked"
```


上一篇

下一篇

暂无话题~