方法

cnswift.org/methods

方法 是关联了特定类型的函数。类，结构体以及枚举都能定义实例方法，方法封装了给定类型特定的任务和功能。类，结构体和枚举同样可以定义类型方法，这是与类型本身关联的方法。类型方法与 Objective-C 中的类方法相似。

事实上在 结构体和枚举中定义方法是 Swift 语言与 C 语言和 Objective-C 的主要区别。在 Objective-C 中，类是唯一能定义方法的类型。但是在 Swift ，你可以选择无论类，结构体还是方法，它们都拥有强大的灵活性来在你创建的类型中定义方法。

实例方法

实例方法 是属于特定类实例、结构体实例或者枚举实例的函数。他们为这些实例提供功能性，要么通过提供访问和修改实例属性的方法，要么通过提供与实例目的相关的功能。实例方法与函数的语法完全相同，就如同函数章节里描述的那样。

要写一个实例方法，你需要把它放在对应类的花括号之间。实例方法默认可以访问同类下所有其他实例方法和属性。实例方法只能在类型的具体实例里被调用。它不能在独立于实例而被调用。

这里有个定义
Counter类的栗子，它可以用来计算动作发生的次数：

```
1   class Counter {
2       var count = 0
3       func increment() {
4           count += 1
5       }
6       func increment(by amount: Int) {
7           count += amount
8       }
9       func reset() {
10          count = 0
11      }
12  }
```

Counter 类定义了三个实例方法：

- increment每次给计数器增加 `1;`
- increment(by: Int)按照特定的整型数量来增加计数器；
- reset会把计数器重置为零。

Counter类同样声明了一个变量属性
count来追踪当前计数器的值。

调用实例方法与属性一样都是用点语法：

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

如同函数实际参数标签和形式参数名中描述的那样，函数的形式参数可以同时拥有一个局部名称（用于函数体）和一个实际参数标签（用于调用函数的时候）。同样的，对于方法的形式参数来说也可以，因为方法就是与类型关联的函数。

self 属性

每一个类的实例都隐含一个叫做
self的属性，它完完全全与实例本身相等。你可以使用
self属性来在当前实例当中调用它自身的方法。

在上边的例子中，
increment()方法可以写成这样：

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

实际上，你不需要经常在代码中写
self。如果你没有显式地写出
self，Swift会在你于方法中使用已知属性或者方法的时候假定你是调用了当前实例中的属性或者方法。这个假定通过在
Counter的三个实例中使用
count（而不是
self.count）来做了示范。

对于这个规则的一个重要例外就是当一个实例方法的形式参数名与实例中某个属性拥有相同的名字的时候。在这种情况下，形式参数名具有优先权，并且调用属性的时候使用更加严谨的方式就很有必要了。你可以使用
self属性来区分形式参数名和属性名。
这时，
self就避免了叫做 x 的方法形式参数还是同样叫做 x 的实例属性这样的歧义。

```
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")
}
// Prints "This point is to the right of the line where x == 1.0"
```

除去 self 前缀，Swift 将会假定两个 x 都是叫做 x 的方法形式参数。

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，值类型属性不能被自身的实例方法修改。

总之，如果你需要在特定的方法中修改结构体或者枚举的属性，你可以选择将这个方法*异变*。然后这个方法就可以在方法中异变（嗯，改变）它的属性了，并且任何改变在方法结束的时候都会写入到原始的结构体中。方法同样可以指定一个全新的实例给它隐含的self属性，并且这个新的实例将会在方法结束的时候替换掉现存的这个实例。

你可以选择在
func关键字前放一个
mutating关键字来使用这个行为：

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

上文中的
Point 结构体定义了一个异变方法
moveBy(x:y:)，它以特定的数值移动一个
Point实例。相比于返回一个新的点，这个方法实际上修改了调用它的点。 被添加到定义中的
mutating关键字允许它修改自身的属性。

注意，如同 常量结构体实例的存储属性 里描述的那样，你不能在常量结构体类型里调用异变方法，因为自身属性不能被改变，就算它们是变量属性：

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

在异变方法里指定自身

异变方法可以指定整个实例给隐含的
self属性。上文中那个
Point的栗子可以用下边的代码代替：

```
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)
    }
}
```

这次的异变方法
moveBy(x:y:)创建了一个
x和
y设置在目的坐标的全新的结构体。调用这个版本的方法和的结果会和之前那个完全一样。
枚举的异变方法可以设置隐含的
self属性为相同枚举里的不同成员：

```
enum TriStateSwitch {
    case off, low, high
    mutating func next() {
        switch self {
        case .off:
            self = .low
        case .low:
            self = .high
        case .high:
            self = .off
        }
    }
}
var ovenLight = TriStateSwitch.low
ovenLight.next()
// ovenLight is now equal to .high
ovenLight.next()
// ovenLight is now equal to .off
```

这个栗子定义了一个三种开关状态的枚举。每次调用
next()方法时，这个开关就会在三种不同的电力状态（
Off ,
low和
high）下切换。

类型方法

如上文描述的那样，实例方法是特定类型实例中调用的方法。你同样可以定义在类型本身调用的方法。这类方法被称作*类型方法*。你可以通过在
func关键字之前使用
static关键字来明确一个类型方法。类同样可以使用
class关键字来允许子类重写父类对类型方法的实现。

> 注意
>
> 在 Objective-C 中，你只能在 Objective-C 的类中定义类级别的方法。但是在 Swift 里，你可以在所有的类里定义类级别的方法，还有结构体和枚举。每一个类方法都能够对它自身的类范围显式生效。

类型方法和实例方法一样使用点语法调用。不过，你得在类上调用类型方法，而不是这个类的实例。接下来是一个在
SomeClass类里调用类型方法的栗子：

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

在类型方法的函数体中，隐含的
self属性指向了类本身而不是这个类的实例。对于结构体和枚举，这意味着你可以使用
self来消除类型属性和类型方法形式参数之间的歧义，用法和实例属性与实例方法形式参数之间的用法完全相同。

一般来说，你在类型方法函数体内书写的任何非完全标准的方法和属性名 都将会指向另一个类级别的方法和属性。一个类型方法可以使用方法名调用另一个类型方法，并不需要使用类型名字作为前缀。同样的，结构体和枚举中的类型方法也可以通过直接使用类型属性名而不需要写类型名称前缀来访问类型属性。
下边的栗子定义了一个叫做
LevelTracker的结构体，它通过不同的等级或者阶层来追踪玩家的游戏进度。这是一个单人游戏，但是可以在一个设备上储存多个玩家的信息。

当游戏第一次开始的时候所有的的游戏等级（除了第一级）都是锁定的。每当一个玩家完成一个等级，那这个等级就对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解锁的游戏等级。它同样追踪每一个独立玩家的当前等级。

```
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结构体持续追踪任意玩家解锁的最高等级。这个值被储存在叫做highestUnlockedLevel的类型属性里边。

LevelTracker同时还定义了两个类型函数来操作highestUnlockedLevel属性。第一个类型函数叫做unlock(_:)，它在新等级解锁的时候更新highestUnlockedLevel。第二个是叫做isUnlocked(_:)的便捷类型方法，如果特定等级已经解锁，则返回true。（注意这些类型方法可以访问highestUnlockedLevel类型属性而并不需要写作LevelTracker.highestUnlockedLevel。）

除了其自身的类型属性和类型方法，LevelTracker还追踪每一个玩家在游戏中的进度。它使用一个实例属性currentLevel来追踪当前游戏中的游戏等级。

为了帮助管理currentLevel属性，LevelTracker定义了一个叫做advance(to:)的实例方法。在更新currentLevel之前，这个方法会检查请求的新等级是否解锁。advance(to:)方法返回一个布尔值来明确它是否真的可以设置currentLevel。由于调用时忽略advance(to:) 的返回值并不是什么大不了的问题，这个函数用@discardableResult 特性。更多关于这个特性的信息，见特性。

看下边的栗子，
LevelTracker结构体与
Player类共同使用来追踪和更新每一个玩家的进度：

```
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类创建了一个新的
LevelTracker实例 来追踪玩家的进度。它同时提供了一个叫做
complete(level:)的方法，这个方法在玩家完成一个特定等级的时候会被调用。这个方法会为所有的玩家解锁下一个等级并且更新玩家的进度到下一个等级。（
advance(to:)返回的布尔值被忽略掉了，因为等级已经在先前调用
LevelTracker.unlock(_:)时已知解锁。）
你可以通过为
Player类创建一个实例来新建一个玩家，然后看看当玩家达成等级1时会发生什么：

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

如果你创建了第二个玩家，当你让他尝试进入尚未被任何玩家在游戏中解锁的等级时， 设置玩家当前等级的尝试将会失败：

```
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")
}
// Prints "level 6 has not yet been
unlocked"
```