自动引用计数

在本页

Swift使用*自动引用计数*（ARC）来跟踪和管理应用程序的内存使用情况。在大多数情况下，这意味着内存管理在Swift中“正常工作”，并且您不需要自己考虑内存管理。当这些实例不再需要时，ARC会自动释放类实例使用的内存。

但是，在少数情况下，ARC需要有关代码各部分之间关系的更多信息才能为您管理内存。本章将介绍这些情况，并说明如何使ARC能够管理所有应用程序的内存。在Swift中使用ARC非常类似于在Objective-C中使用ARC的*过渡到ARC版本注释中*描述的方法。

引用计数仅适用于类的实例。结构和枚举是值类型，而不是引用类型，不会通过引用存储和传递。

ARC如何工作

每次创建类的新实例时，ARC都会分配一块内存来存储有关该实例的信息。该内存保存关于实例类型的信息以及与该实例关联的任何存储属性的值。

此外，当不再需要实例时，ARC将释放该实例使用的内存，以便内存可以用于其他目的。这可以确保类实例在不再需要时不占用内存空间。

但是，如果ARC要释放一个仍在使用的实例，将不再可能访问该实例的属性，或者调用该实例的方法。事实上，如果你试图访问实例，你的应用很可能会崩溃。

为了确保实例在仍然需要时不会消失，ARC会跟踪当前引用每个类实例的属性，常量和变量的数量。只要至少有一个对该实例的活动引用仍然存在，ARC就不会解除分配实例。

为了做到这一点，无论何时将一个类实例分配给一个属性，常量或变量，该属性，常量或变量都会*强制引用*该实例。该参考文献被称为“强有力的”参考文献，因为它在该实例中保持坚定的态度，并且只要强有力的参考文献保留就不允许它被重新分配。

ARC在行动

以下是自动引用计数如何工作的示例。这个例子从一个叫做简单的类开始`Person`，它定义了一个名为`name`：

```
class Person {
    let name: String
    init(name: String) {
        self.name = name
        print("\(name) is being initialized")
    }
    deinit {
        print("\(name) is being deinitialized")
    }
}
```

本`Person`类有设置实例的一个初始化`name`属性并显示一条消息，指示初始化正在进行中。这个`Person`类还有一个deinitializer，它在类的一个实例被释放时打印一条消息。

下一个代码片段定义了三个类型变量`Person?`，这些变量用于`Person`在后续代码片段中设置对新实例的多个引用。由于这些变量是可选类型（`Person?`不是`Person`），因此它们将自动使用值初始化`nil`，并且当前不引用`Person`实例。

```
var reference1: Person?
var reference2: Person?
var reference3: Person?
```

您现在可以创建一个新`Person`实例并将其分配给以下三个变量之一：

```
reference1 = Person(name: "John Appleseed")
// Prints "John Appleseed is being initialized"
```

请注意，该消息"John Appleseed is being initialized"是在您调用Person该类的初始值设定项时打印的。这证实初始化已经发生。

由于新Person实例已分配给reference1变量，因此现在reference1对新Person实例有很强的参考。由于至少有一个强引用，因

如果将同一Person实例分配给两个更多变量，则会建立对该实例更强的两个引用：

```
reference2 = reference1
reference3 = reference1
```

现在有三个关于这个单一Person实例的强有力的参考。

如果通过分配nil给两个变量来破坏这些强引用（包括原始引用）中的两个，Person则会保留单个强引用，并且实例不会被解除分配：

```
reference1 = nil
reference2 = nil
```

Person在第三个也是最后一个强引用被打破之前， ARC不会释放实例，在这一点上很明显你不再使用该Person实例：

```
reference3 = nil
// Prints "John Appleseed is being deinitialized"
```

类实例之间的强参考循环

在上面的示例中，ARC能够跟踪Person您创建的新实例的引用数量，并在Person不再需要时释放该实例。

但是，可以编写一个代码，其中一个类的实例*永远不*会到达其强引用为零的点。如果两个类实例之间存在强关联，那么每个实例都会保持另一个实例的活动。这被称为*强参考周期*。

通过将类之间的某些关系定义为弱引用或无主引用而不是强引用，可以解决强引用循环。在解决类实例之间的强参考循环中描述了此过程。但是，在您学习如何解决强大的参考周期之前，了解如何导致这样的周期很有用。

下面是一个例子，说明如何强制创建一个参考周期。这个例子定义了两个叫做Personand的类Apartment，它们模拟了一组公寓和它的居民：

```
class Person {
    let name: String
    init(name: String) { self.name = name }
    var apartment: Apartment?
    deinit { print("\(name) is being deinitialized") }
}

class Apartment {
    let unit: String
    init(unit: String) { self.unit = unit }
    var tenant: Person?
    deinit { print("Apartment \(unit) is being deinitialized") }
}
```

每个Person实例最初都有一个name类型属性String和一个可选apartment属性nil。该apartment属性是可选的，因为一个人可能并不总是有一个公寓。

同样，每个Apartment实例都有一个unit类型属性，String并且具有tenant最初的可选属性nil。租户属性是可选的，因为公寓可能并不总是有租客。

这两个类都定义了一个deinitializer，它打印出该类的一个实例正在被初始化的事实。这使您能够按预期查看是否正在Person和Apartment正在释放实例。

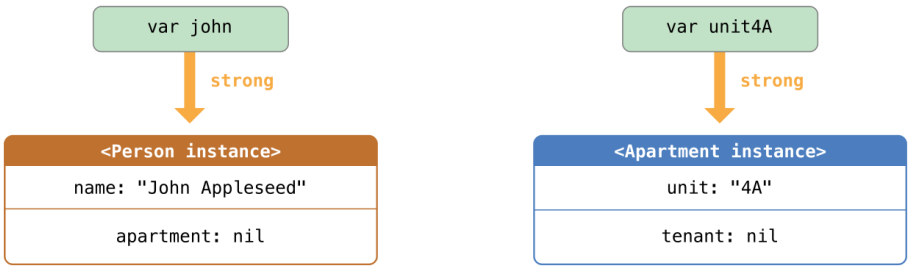
下一个代码片段定义了两个可选类型的变量，称为john和unit4A，它们将被设置为下面的特定Apartment和Person实例。nil由于是可选的，这两个变量都具有初始值：

```
var john: Person?
var unit4A: Apartment?
```

您现在可以创建

```
john = Person(name: "John Appleseed")
unit4A = Apartment(unit: "4A")
```

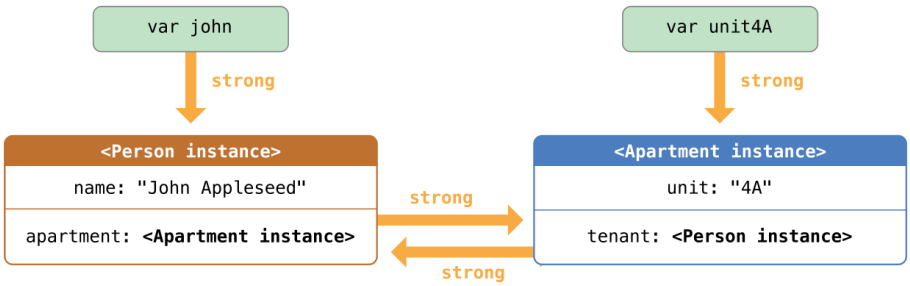
以下是强引用在创建和分配这两个实例后的样子。该john变量现在对新Person实例有很强的参考，并且该unit4A变量对新实例有很强的参考Apartment：



您现在可以将两个实例链接在一起，以便该人拥有一个公寓，而且该公寓有一个租户。注意，感叹号（!）用于解开和访问存储在内部的情况下，john和unit4A任选的变量，以便这些实例的属性可设置为：

```
john!.apartment = unit4A
unit4A!.tenant = john
```

以下是将这两个实例链接在一起后，强引用的效果：

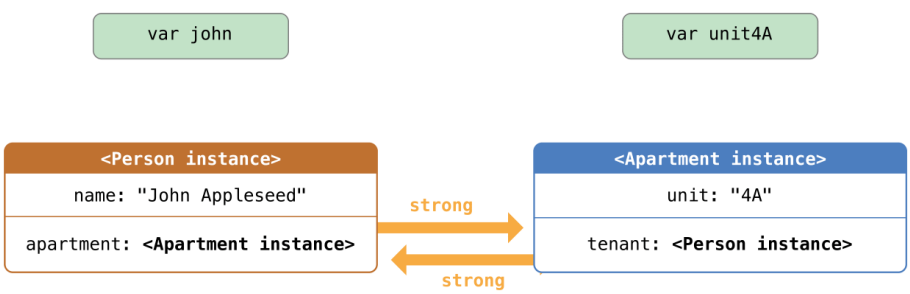


不幸的是，链接这两个实例会在它们之间创建一个强大的参考循环 该Person实例现在具有对该实例的强引用Apartment，并且该Apartment实例具有对该实例的强引用Person。因此，当您打破由变量john和unit4A变量持有的强引用时，引用计数不会降为零，并且实例不会被ARC取消分配：

```
john = nil
unit4A = nil
```

请注意，将这两个变量设置为时，都不会调用deinitializer nil。强引用循环防止Person和Apartment实例从不断被释放，导致在你的应用程序内存泄漏。

以下是将参数john和unit4A变量设置为nil：强参考后的效果：



Person实例和Apartment实例 之间的强烈参考依然存在并且无法被破坏。

解决类实例[illegible]

当您使用类类型的属性时，Swift提供了两种解决强引用周期的方法：弱引用和无主引用。

弱引用和无主引用使参考周期中的一个实例能够引用另一个实例，*而不会对其*保持强有力的保留。然后这些实例可以互相引用，而不会创建强大的参考周期。

当另一个实例的生命周期更短时，使用弱引用 - 也就是说，当其他实例可以先释放时。在Apartment上面的例子中，公寓在其生命周期的某个时间点能够没有租户是合适的，因此在这种情况下，弱参考是一种合适的方式来打破参考周期。相反，当另一个实例具有相同的生命周期或更长的生命周期时，请使用无主引用。

弱引用

一个*弱引用*是一个引用，它不会强制保留它引用的实例，所以不会停止ARC处理引用的实例。此行为可防止参考成为强参考周期的一部分。通过weak在属性或变量声明之前放置关键字来指示弱引用。

因为弱引用不会强引用它所引用的实例，所以可以将该实例解除分配，而弱引用仍然指向它。因此，ARC会自动为nil其所引用的实例解除分配时设置一个弱引用。而且，因为弱引用需要允许它们的值nil在运行时更改，所以它们总是被声明为可选类型的变量而不是常量。

您可以检查弱引用中是否存在值，就像其他任何可选值一样，并且永远不会引用对不再存在的无效实例的引用。

> 注意
> 当ARC设置一个弱引用时，属性观察者不会被调用nil。

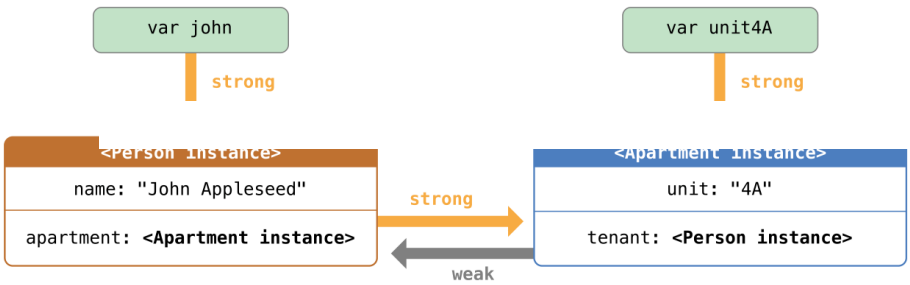
下面的例子是相同的Person，并Apartment例如从上方，有一个重要区别。这一次，该Apartment类型的tenant属性被声明为弱引用：

```
class Person {
    let name: String
    init(name: String) { self.name = name }
    var apartment: Apartment?
    deinit { print("\(name) is being deinitialized") }
}

class Apartment {
    let unit: String
    init(unit: String) { self.unit = unit }
    weak var tenant: Person?
    deinit { print("Apartment \(unit) is being deinitialized") }
}
```

两个变量（john和unit4A）的强引用以及两个实例之间的链接与以前一样创建：

```
var john: Person?
var unit4A: Apartment?

john = Person(name: "John Appleseed")
unit4A = Apartment(unit: "4A")

john!.apartment = unit4A
unit4A!.tenant = john
```

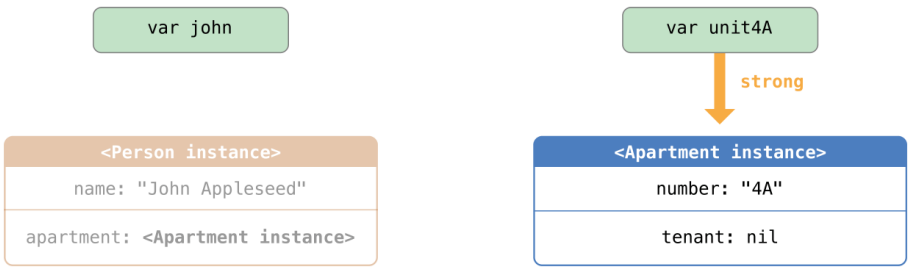
以下是引用看起来如何将这两个实例链接在一起：



该Person实例仍然对该实例有强烈的参考Apartment，但该Apartment实例现在对实例的引用*较弱*Person。这意味着，当您john通过设置来打破变量所nil具有的强引用时，不会再强引用该Person实例：

```
john = nil
// Prints "John Appleseed is being deinitialized"
```

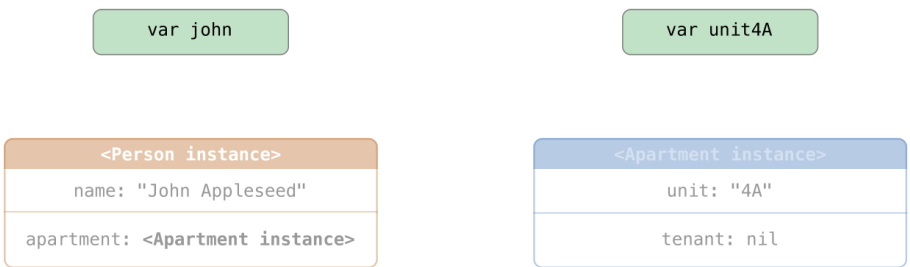
由于没有更强的对Person实例的引用，因此将其解除分配，并将该tenant属性设置为nil：



剩下的唯一强大的参考Apartment来自unit4A变量。如果你打破了*这个*强大的参考，就没有更多的Apartment实例引用了：

```
unit4A = nil
// Prints "Apartment 4A is being deinitialized"
```

因为没有更强的Apartment实例引用，它也被释放：



> 注意
>
> 在使用垃圾收集的系统中，弱指针有时用于实现简单的缓存机制，因为只有当内存压力触发垃圾收集时才会释放没有强引用的对象。但是，使用ARC时，只要删除最后一个强参考值就会释放值，从而导致弱参考不适用于此目的。

无主的参考

就像一个弱引用一样，一个*无主的引用*并不能很好地保持它引用的实例。然而，与弱引用不同，当另一个实例具有相同的生命周期或更长的生命周期时，将使用无主引用。您通过unowned在属性或变量声明之前放置关键字来指示无主的引用。

预计一个无主的参考将始终有一个值。因此，ARC永远不会将无主引用的值设置为nil，这意味着无主引用是使用非选项类型定义的。

> 重要
>
> 只有当您确定引用*始终*引用未被释放的实例时才使用无主引用。
>
> 如果尝试[illegible]

以下示例定义了两个类，Customer以及CreditCard哪个模型是银行客户和该客户可能使用的信用卡。这两个类都将另一个类的实例存储为一个属性。这种关系有可能创造一个强大的参考周期。

之间的关系Customer和CreditCard距离之间的关系略有不同Apartment，并Person在上面的弱参考例所示。在这种数据模型中，客户可能有也可能没有信用卡，但信用卡将*始终*与客户相关联。一个CreditCard实例永远不会超越Customer它所指的。为了表示这个，Customer该类有一个可选card属性，但CreditCard该类具有非拥有（和非可选）customer属性。

此外，新CreditCard实例*只能*通过将number值和customer实例传递给自定义CreditCard初始化程序来创建。这确保实例创建时CreditCard实例始终具有customer与其关联的CreditCard实例。

由于信用卡将始终有客户，因此您将其customer属性定义为无主参考，以避免强大的参考周期：

```
class Customer {
    let name: String
    var card: CreditCard?
    init(name: String) {
        self.name = name
    }
    deinit { print("\(name) is being deinitialized") }
}

class CreditCard {
    let number: UInt64
    unowned let customer: Customer
    init(number: UInt64, customer: Customer) {
        self.number = number
        self.customer = customer
    }
    deinit { print("Card #\(number) is being deinitialized") }
}
```

> 注意
>
> 所述number的的属性CreditCard类与一种类型的定义UInt64，而不是Int，以确保number财产的容量大得足以存储在32位和64位系统的16位卡号。

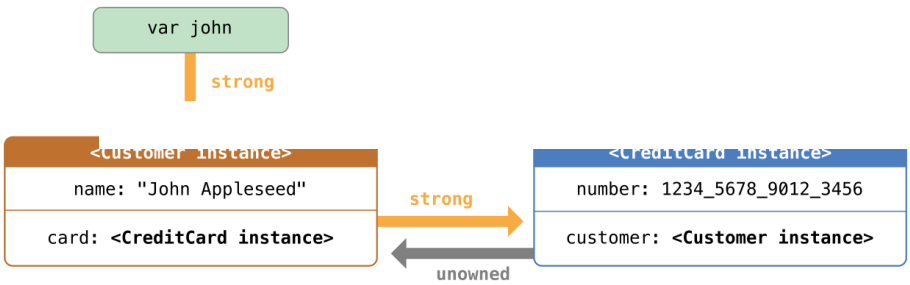
下一个代码片段定义了一个Customer名为的可选变量john，它将用于存储对特定客户的引用。由于可选，此变量的初始值为nil：

```
var john: Customer?
```

您现在可以创建一个Customer实例，并使用它来初始化和分配一个新CreditCard实例作为该客户的card属性：

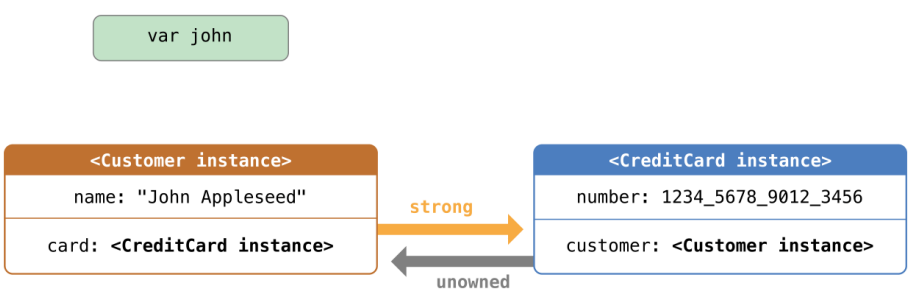
```
john = Customer(name: "John Appleseed")
john!.card = CreditCard(number: 1234_5678_9012_3456, customer: john!)
```

以下是引用的外观，现在已经链接了两个实例：



该Customer实例现在具有对该实例的强引用CreditCard，并且该CreditCard实例拥有对该实例的无主引用Customer。

由于无主的customer引用，当您打破john变量所持有的强引用时，没有更强的对Customer实例的引用：



因为没有更强的Customer实例引用，所以它被释放。发生这种情况后，没有更强的CreditCard实例引用，它也被释放：

```
john = nil
// Prints "John Appleseed is being deinitialized"
// Prints "Card #1234567890123456 is being deinitialized"
```

上面的最后一段代码显示，Customer实例和CreditCard实例的去初始化器在john变量设置为之后都会打印它们的“未初始化”消息nil。

> 注意
>
> 上面的例子展示了如何使用*安全的*无主引用。Swift还为需要禁用运行时安全检查的情况提供*不安全的*无主引用 - 例如，出于性能原因。与所有不安全的操作一样，您需要负责检查该代码的安全性。
>
> 您通过书写表明一个不安全的无主参考文献unowned(unsafe)。如果尝试在其引用的实例解除分配后访问不安全的无主引用，程序将尝试访问实例曾经存在的内存位置，这是一种不安全的操作。

无主的引用和隐式解包的可选属性

上述弱和无主参考的例子涵盖了两个更常见的场景，其中有必要打破强大的参考周期。

在Person和Apartment实施例显示的情况下两个属性，这两者都允许为nil，具有引起强烈的基准周期的潜力。这种情况最好用弱引用解决。

在Customer和CreditCard实施例显示的情况下被允许是一种性质nil不能为与另一个属性nil有可能造成很强的参考周期。这种情况最好用无主的参考来解决。

但是，还有第三种情况，其中*两个*属性都应该始终具有值，并且nil一旦完成初始化，这*两个*属性都不应该有。在这种情况下，将一个类的非拥有属性与另一个类的隐式解包可选属性结合起来很有用。

这使得一旦完成初始化就可以直接访问这两个属性（没有可选的解包），同时仍然避免了参考周期。本节将向您介绍如何建立这种关系。

下面的例子定义了两类，Country并且City，其中的每一个存储所述其他类作为属性的一个实例。在这个数据模型中，每个国家都必须有一个首都，每个城市都必须属于一个国家。为了表示这个，Country该类有一个capitalCity属性，并且City该类有一个country属性：

```
class Country {
```

```
    let name: String
    var capitalCity: City!
    init(name: String, capitalName: String) {

        self.capitalCity = City(name: capitalName, country: self)
    }
}

class City {
    let name: String
    unowned let country: Country
    init(name: String, country: Country) {
        self.name = name
        self.country = country
    }
}
```

为了建立两个类之间的相互依赖关系，初始化器City获取一个Country实例，并将该实例存储在它的country属性中。

初始化程序for在初始化程序中City被调用Country。但是，初始化Country程序无法传递self给City初始化程序，直到新Country实例完全初始化，如“ 两阶段初始化”中所述。

为了处理这个需求，你声明capitalCity属性Country是一个隐式解包的可选属性，在其类型注释（City!）的末尾用感叹号表示。这意味着该capitalCity属性的默认值与nil其他任何可选值一样，但可以在隐式展开选项中所述的情况下访问而无需拆开其值。

由于capitalCity具有默认nil值，因此Country只要Country实例name在其初始化程序中设置了其属性，就会将新实例视为完全初始化。这意味着一旦属性被设置，Country初始化器就可以开始引用并传递隐式self属性name。所述Country因此初始化可以传递self作为参数之一City时初始化Country的初始化中设置其自己的capitalCity属性。

所有这些意味着您可以在单个语句中创建Country和City实例，而无需创建强引用循环，并且capitalCity可以直接访问该属性，而无需使用感叹号来打开其可选值：

```
var country = Country(name: "Canada", capitalName: "Ottawa")
print("\(country.name)'s capital city is called \(country.capitalCity.name)")
// Prints "Canada's capital city is called Ottawa"
```

在上面的示例中，使用隐式解包可选意味着满足所有两阶段类初始化器要求。capitalCity一旦初始化完成，该属性可以像非可选值一样使用和访问，同时还可以避免强大的参考周期。

关闭的强参考周期

您在上面看到，当两个类实例属性彼此强关联时，如何创建强引用循环。您还看到了如何使用弱和无主的引用来打破这些强参考周期。

如果将闭包分配给类实例的属性，并且该闭包的主体捕获实例，则也会发生强引用循环。这种捕获可能发生是因为闭包的主体访问实例的某个属性self.someProperty，或者因为闭包调用实例上的方法，例如self.someMethod()。在任何一种情况下，这些访问都会导致关闭“捕捉” self，从而创建一个强大的参考周期。

这种强大的参考循环发生是因为闭包，就像类一样，是*引用类型*。当您将一个闭包分配给一个属性时，您正在为该闭包分配一个*引用*。实质上，这与上面的问题是一样的 - 两个强有力的引用保持着彼此的活力。但是，这次不是两个类实例，而是一个保持对方活着的类实例和闭包。

Swift为这个问题提供了一个优雅的解决方案，称为*闭包捕获列表*。然而，在你学习如何用封闭捕获列表打破一个强大的引用循环之前，了解如何引起这样一个循环是很有用的。

下面的示例显示了如何使用引用的闭包创建强引用循环self。本示例定义了一个名为的类HTMLElement，它为HTML文档中的单个元素提供了一个简单的模型：

```
class HTMLElement {

    let name: String
    let text: String?

```

```
    lazy var asHTML: () -> String = {
        if let text = self.text {
            return "<\(self.name)>\(text)</\(self.name)>"

            return "<\(self.name) />"
        }
    }

    init(name: String, text: String? = nil) {
        self.name = name
        self.text = text
    }

    deinit {
        print("\(name) is being deinitialized")
    }

}
```

的HTMLElement类定义了一个name属性，该属性指示该元素的名称，诸如"h1"用于一个标题元素，"p"为段落元件，或"br"用于换行元件。HTMLElement还定义了一个可选text属性，您可以将其设置为表示要在该HTML元素中呈现的文本的字符串。

除了这两个简单属性外，HTMLElement该类还定义了一个名为lazy的属性asHTML。此属性引用，结合了封闭name和text成HTML字符串片段。该asHTML属性是类型的() -> String，或者“不带参数的函数，并返回一个String值”。

默认情况下，该asHTML属性被分配一个闭包，该闭包返回HTML标记的字符串表示形式。如果该标记text存在，则该标记包含可选值，否则不包含任何文本内容（如果text不存在）。对于段落元素，封闭将返回" some text "或" "取决于text属性是否等于"some text"或nil。

该asHTML属性的命名和使用有点像实例方法。但是，因为asHTML是闭包属性而不是实例方法，所以asHTML如果要更改特定HTML元素的HTML呈现，可以使用自定义闭包来替换该属性的默认值。

例如，asHTML如果该text属性是nil，则可以将该属性设置为默认为某些文本的闭包，以防止该表示返回空的HTML标记：

```
let heading = HTMLElement(name: "h1")
let defaultText = "some default text"
heading.asHTML = {
    return "<\(heading.name)>\(heading.text ?? defaultText)</\(heading.name)>"
}
print(heading.asHTML())
// Prints "<h1>some default text</h1>"
```

> 注意
>
> 该asHTML属性被声明为一个懒惰属性，因为只有在实际需要将元素呈现为某个HTML输出目标的字符串值时才需要该属性。事实asHTML是一个懒惰的属性意味着你可以self在默认的闭包内引用，因为在初始化完成并self知道存在之前，lazy属性将不会被访问。

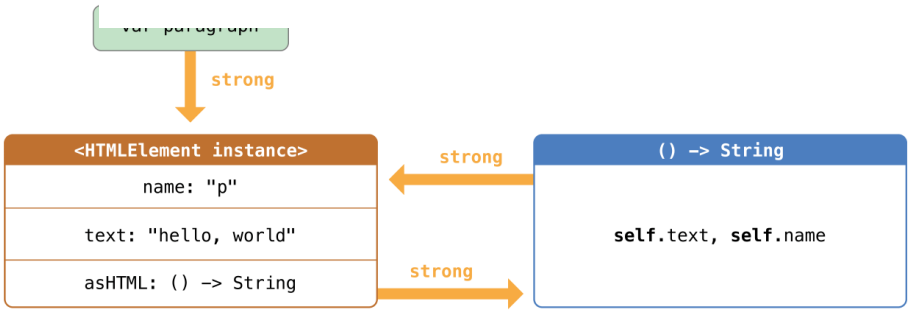
的HTMLElement类提供了一个单一的初始值设定，这需要一个name参数，并（如果需要的话）一个text参数来初始化新的元件。该类还定义了一个deinitializer，它打印一条消息以显示HTMLElement实例何时解除分配。

以下是您如何使用HTMLElement该类创建和打印新实例的方法：

```
var paragraph: HTMLElement? = HTMLElement(name: "p", text: "hello, world")
print(paragraph!.asHTML())
// Prints "<p>hello, world</p>"
```

> 注意
>
> paragraph上面的变量被定义为一个*可选的*HTMLElement，所以它可以被设置为nil下面来展示强参考周期的存在。

不幸的是，HTMLElement如上所述，该类在HTMLElement实例和用于默认asHTML值的闭包之间创建了强大的引用循环。这是周期的外观：



该实例的asHTML属性对其关闭提供了强有力的参考。然而，因为封闭件是指self它的主体内（作为一种方式来引用self.name和self.text），封闭*捕获*自，这意味着它保持很强的参考回HTMLElement实例。两者之间建立了强大的参考循环。（有关在闭包中捕获值的更多信息，请参阅捕获值。）

> 注意
> 尽管闭包涉及self多次，但它仅捕获一个对该HTMLElement实例的强烈参考。

如果您将该paragraph变量设置为nil并断开其对该HTMLElement实例HTMLElement的强引用，则由于强引用周期，实例及其闭包都不会被释放：

```
paragraph = nil
```

请注意，HTMLElementdeinitializer 中的消息未打印，这表明HTMLElement实例未被释放。

解决闭环强参考周期

通过将*捕获列表*定义为闭包定义的一部分，可以解决闭包与类实例之间的强引用循环。捕获列表定义了在封闭体内捕获一个或多个引用类型时使用的规则。与两个类实例之间的强引用周期一样，您将每个捕获的引用声明为弱引用或无主引用，而不是强引用。弱或无主的适当选择取决于代码不同部分之间的关系。

> 注意
> Swift要求你每写一个self.someProperty或者self.someMethod()（而不是只是someProperty或者someMethod()）引用self闭包中的成员。这有助于您记住self偶然捕获的可能性。

定义一个捕获列表

捕获列表中的每个项目都是weakor unowned关键字与对类实例（如self）的引用或使用某个值（例如delegate = self.delegate!）初始化的变量的配对。这些配对写在一对方括号内，用逗号分隔。

如果提供它们，则将捕获列表放置在闭包的参数列表和返回类型之前：

```
lazy var someClosure: (Int, String) -> String = {
    [unowned self, weak delegate = self.delegate!] (index: Int, stringToProcess:
  String) -> String in
    // closure body goes here
}
```

如果闭包没有指定参数列表或返回类型，因为它们可以从上下文中推断出来，请将捕获列表放置在闭包的开头，然后是in关键字：

```
lazy var someClosure: () -> String = {
    [unowned self, weak delegate = self.delegate!] in
    // closure body goes here
```

```
}
```

弱和无主的参

当闭包和它捕获的实例将始终相互引用时，将闭包中的捕获定义为无主引用，并且将始终在同一时间解除分配。

相反，当捕获的参考可能nil在未来的某个点上时，将捕获定义为弱参考。弱引用始终是可选类型，并且nil在它们引用的实例被释放时自动变为。这使您可以检查封闭体内的存在。

> 注意
> 如果捕获的引用永远不会成为nil，则应始终将其捕获为无主引用，而不是弱引用。

无主的参考是用于解决在强基准周期的合适的捕获方法HTMLElement从例如用于瓶盖强参照循环以上。以下是你如何编写HTMLElement课程以避免循环：

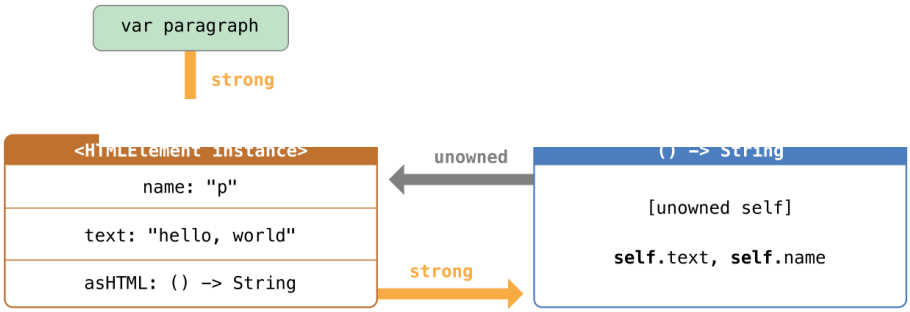
```
class HTMLElement {

    let name: String
    let text: String?

    lazy var asHTML: () -> String = {
        [unowned self] in
        if let text = self.text {
            return "<\(self.name)>\(text)</\(self.name)>"
        } else {
            return "<\(self.name) />"
        }
    }

    init(name: String, text: String? = nil) {
        self.name = name
        self.text = text
    }

    deinit {
        print("\(name) is being deinitialized")
    }

}
```

HTMLElement除了在asHTML闭包中添加捕获列表之外， 此实现与以前的实现完全相同。在这种情况下，捕获列表是[unowned self]指“将自我捕获为无主引用而不是强引用”。

您可以HTMLElement像以前一样创建和打印实例：

```
var paragraph: HTMLElement? = HTMLElement(name: "p", text: "hello, world")
print(paragraph!.asHTML())
// Prints "<p>hello, world</p>"
```

以下是引用在捕获列表中的使用方式：



这一次，self封闭的捕获是一个无主的参考，并且不会HTMLElement对它捕获的实例保持强有力的控制。如果您将paragraph变量的强引用设置为nil，那么该HTMLElement实例将被释放，正如从下面示例中的deinitializer消息的打印中可以看到的那样：

```
paragraph = nil
// Prints "p is being deinitialized"
```

有关捕获列表的更多信息，请参阅捕获列表。