Swift使用自动引用计数(ARC)来跟踪并管理应用使用的内存。大部分情况下，这意味着在Swift语言中，内存管理"仍然工作"，不需要自己去考虑内存管理的事情。当实例不再被使用时，ARC会自动释放这些类的实例所占用的内存。

然而，在少数情况下，为了自动的管理内存空间，ARC需要了解关于你的代码片段之间关系的更多信息。本章描述了这些情况，并向大家展示如何打开ARC来管理应用的所有内存空间。

注意：引用计数只应用在类的实例。结构体(Structure)和枚举类型是值类型，并非引用类型，不是以引用的方式来存储和传递的。

**ARC如何工作**

每次创建一个类的实例，ARC就会分配一个内存块，用来存储这个实例的相关信息。这个内存块保存着实例的类型，以及这个实例相关的属性的值。

当实例不再被使用时，ARC释放这个实例使用的内存，使这块内存可作它用。这保证了类实例不再被使用时，它们不会占用内存空间。

但是，如果ARC释放了仍在使用的实例，那么你就不能再访问这个实例的属性或者调用它的方法。如果你仍然试图访问这个实例，应用极有可能会崩溃。

为了保证不会发生上述的情况，ARC跟踪与类的实例相关的属性、常量以及变量的数量。只要有一个有效的引用，ARC都不会释放这个实例。

为了让这变成现实，只要你将一个类的实例赋值给一个属性或者常量或者变量，这个属性、常量或者变量就是这个实例的强引用(strong reference)。之所以称之为“强”引用，是因为它强持有这个实例，并且只要这个强引用还存在，就不能销毁实例。

**ARC实践**

下面的例子展示了ARC是如何工作的。本例定义了一个简单的类，类名是Person，并定义了一个名为name的常量属性：

1. **class** Person {
2. let name: String
3. init(name: String) {
4. self.name = name
5. println("\(name) is being initialized")
6. }
8. deinit {
9. println("\(name) is being deinitialized")
10. }
11. }

类Person有一个初始化函数（initializer），设置这个实例的name属性，打印一条消息来指示初始化正在进行。类Person还有一个deinitializer方法，当销毁一个类的实例时，会打印一条消息。

接下来的代码片段定义了三个Person?类型的变量，这些变量用来创建多个引用，这些引用都引用紧跟着的代码所创建的Person对象。因为这些变量都是可选类型（Person?，而非Person），因此他们都被自动初始化为nil，并且当前并没有引用一个Person的实例。

1. **var** reference1: Person?
2. **var** reference2: Person?
3. **var** reference3: Person?

现在我们创建一个新的Person实例，并且将它赋值给上述三个变量重的一个：

1. reference1 = Person(name: "John Appleseed")
2. // prints "Jonh Appleseed is being initialized"

注意，消息“John Appleseed is being initialized”在调用Person类的初始化函数时打印。这印证初始化确实发生了。

因为Person的实例赋值给了变量reference1，所以reference1是Person实例的强引用。又因为至少有这一个强引用，ARC就保证这个实例会保存在内存重而不会被销毁。

如果将这个Person实例赋值给另外的两个变量，那么将建立另外两个指向这个实例的强引用：

1. reference2 = reference1
2. reference3 = reference2

现在，这一个Person实例有三个强引用。

如果你通过赋值nil给两个变量来破坏其中的两个强引用（包括原始的引用），只剩下一个强引用，这个Person实例也不会被销毁：

1. reference1 = nil
2. reference2 = nil

直到第三个也是最后一个强引用被破坏，ARC才会销毁Person的实例，这时，有一点非常明确，你无法继续使用Person实例：

1. referenece3 = nil
2. // 打印 “John Appleseed is being deinitialized”

**类实例间的强引用环**

在上面的例子中，ARC可以追踪Person实例的引用数量，并且在它不再被使用时销毁这个实例。

然而，我们有可能会写出这样的代码，一个类的实例永远不会有0个强引用。在两个类实例彼此保持对方的强引用，使得每个实例都使对方保持有效时会发生这种情况。我们称之为强引用环。

通过用弱引用或者无主引用来取代强引用，我们可以解决强引用环问题。在开始学习如何解决这个问题之前，理解它产生的原因会很有帮助。

下面的例子展示了一个强引用环是如何在不经意之间产生的。例子定义了两个类，分别叫Person和Apartment，这两个类建模了一座公寓以及它的居民：

1. **class** Person {
2. let name: String
3. init(name: String) { self.name = name }
4. **var** apartment: Apartment?
5. deinit { println("\(name) is being deinitialized") }
6. }
8. **class** Apartment {
9. let number: Int
10. init(number: Int) { self.number = number }
11. **var** tenant: Person?
12. deinit { println("Apartment #\(number) is being deinitialized") }
13. }

每个Person实例拥有一个String类型的name属性以及一个被初始化为nil的apartment可选属性。apartment属性是可选的，因为一个人并不一定拥有一座公寓。

类似的，每个Apartment实例拥有一个Int类型的number属性以及一个初始化为nil的tenant可选属性。tenant属性是可选的，因为一个公寓并不一定有居民。

这两个类也都定义了初始化函数，打印消息表明这个类的实例正在被初始化。这使你能够看到Person和Apartment的实例是否像预期的那样被销毁了。

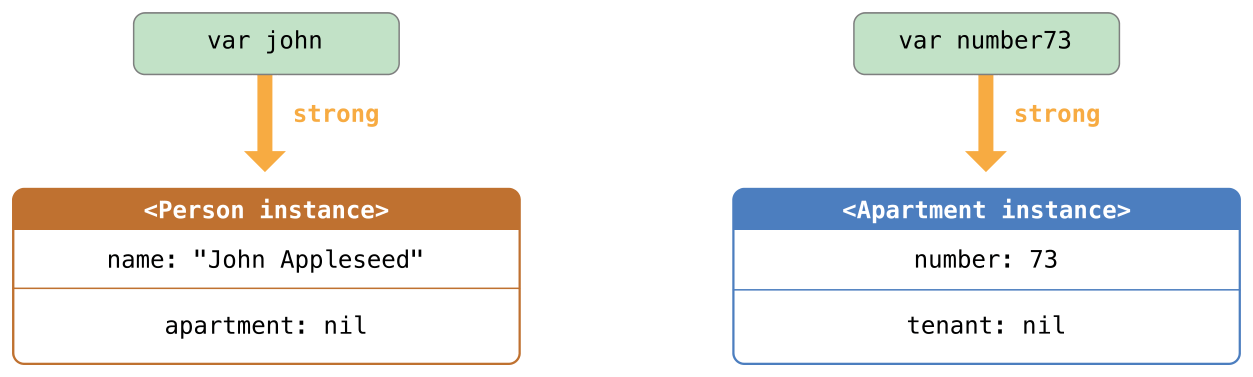
下面的代码片段定义了两个可选类型变量，john和number73，分别被赋值为特定的Apartment和Person的实例。得益于可选类型的优点，这两个变量初始值均为nil：

1. **var** john: Person?
2. **var** number73: Apartment?

现在，你可以创建特定的Person实例以及Apartment实例，并赋值给john和number73：

1. jhon = Person(name: "John Appleseed")
2. number73 = Apartments(number: 73)

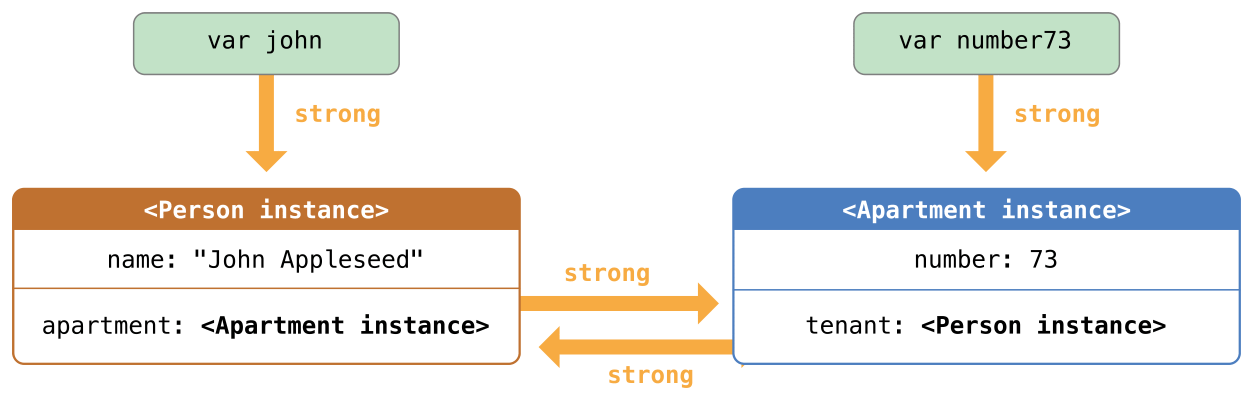
下面的图表明了在创建以及赋值这两个实例后强引用的关系。john拥有一个Person实例的强引用，number73拥有一个Apartment实例的强引用:



现在你可以将两个实例关联起来，一个人拥有一所公寓，一个公寓也拥有一个房客。注意：用感叹号（!）来展开并访问可选类型的变量，只有这样这些变量才能被赋值：

1. john!.apartment = number73
2. number73!.tenant = john

两个实例关联起来后，强引用关系如下图所示：

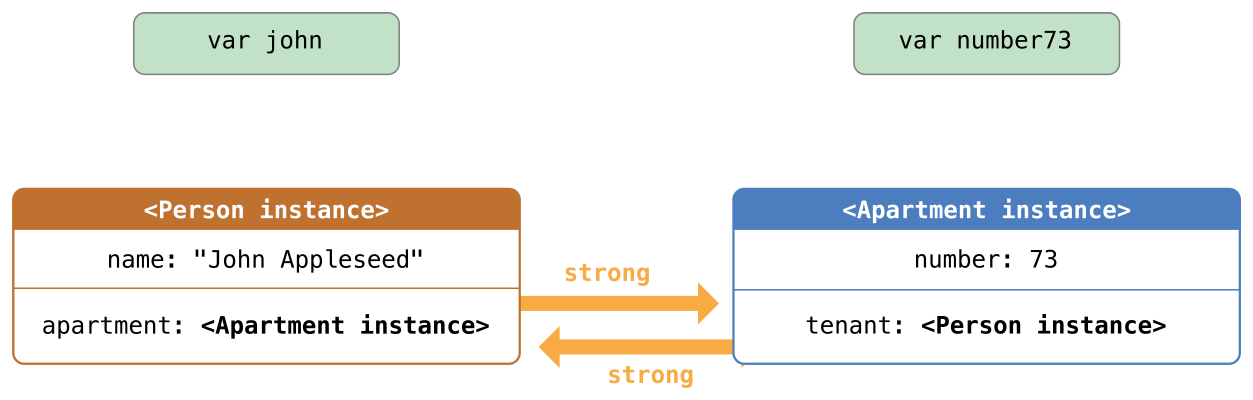


糟糕的是，关联这俩实例生成了一个强引用环，Person实例和Apartment实例各持有一个对方的强引用。因此，即使你破坏john和number73所持有的强引用，引用计数也不会变为0，因此ARC不会销毁这两个实例：

1. john = nil
2. nuber73 = nil

注意，当上面两个变量赋值为nil时，没有调用任何一个deinitializer。强引用环阻止了Person和Apartment实例的销毁，进一步导致内存泄漏。

此时强引用关系如下图所示：



Person和Apartment实例之间的强引用依然存在。

**解决实例间的强引用环**

Swift提供两种方法来解决强引用环：弱引用和无主引用。

弱引用和无主引用允许引用环中的一个实例引用另外一个实例，但不是强引用。因此实例可以互相引用但是不会产生强引用环。

对于生命周期中引用会变为nil的实例，使用弱引用；对于初始化时赋值之后引用再也不会赋值为nil的实例，使用无主引用。

**弱引用**

弱引用不会增加实例的引用计数，因此不会阻止ARC销毁被引用的实例。这种特性使得引用不会变成强引用环。声明属性或者变量的时候，关键字weak表明引用为弱引用。

在实例的生命周期中，如果某些时候引用没有值，那么弱引用可以阻止强引用环。如果整个生命周期内引用都有值，那么相应的用无主引用，在无主引用这一章中有详细描述。在上面的Apartment例子中，有时一个Apartment实例可能没有房客，因此此处应该用弱引用。

注意：弱引用只能声明为变量类型，因为运行时它的值可能改变。弱引用绝对不能声明为常量。

因为弱引用可以没有值，所以声明弱引用的时候必须是可选类型的。在Swift语言中，推荐用可选类型来作为可能没有值的引用的类型。

如前所述，弱引用不会保持实例，因此即使实例的弱引用依然存在，ARC也有可能会销毁实例，并将弱引用赋值为nil。你可以想检查其他的可选值一样检查弱引用是否存在，永远也不会碰到引用了也被销毁的实例的情况。

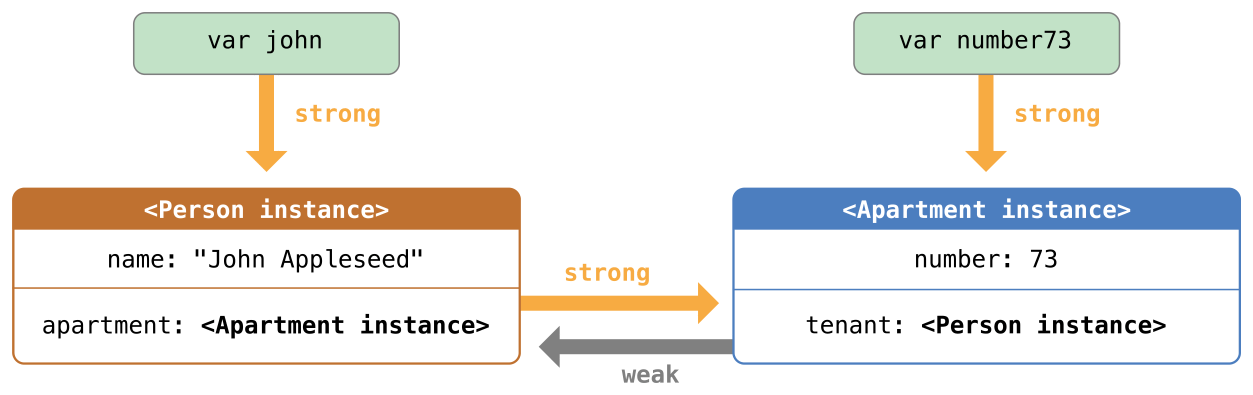
下面的例子和之前的Person和Apartment例子相似，除了一个重要的区别。这一次，我们声明Apartment的tenant属性为弱引用：

1. **class** Person {
2. let name: String
3. init(name: String) { self.name = name }
4. **var** apartment: Apartment?
5. deinit { println("\(name) is being deinitialized") }
6. }
8. **class** Apartment {
9. let number: Int
10. init(number: Int) { self.number = number }
11. weak **var** tenant: Person?
12. deinit { println("Apartment #\(number) is being deinitialized") }
13. }

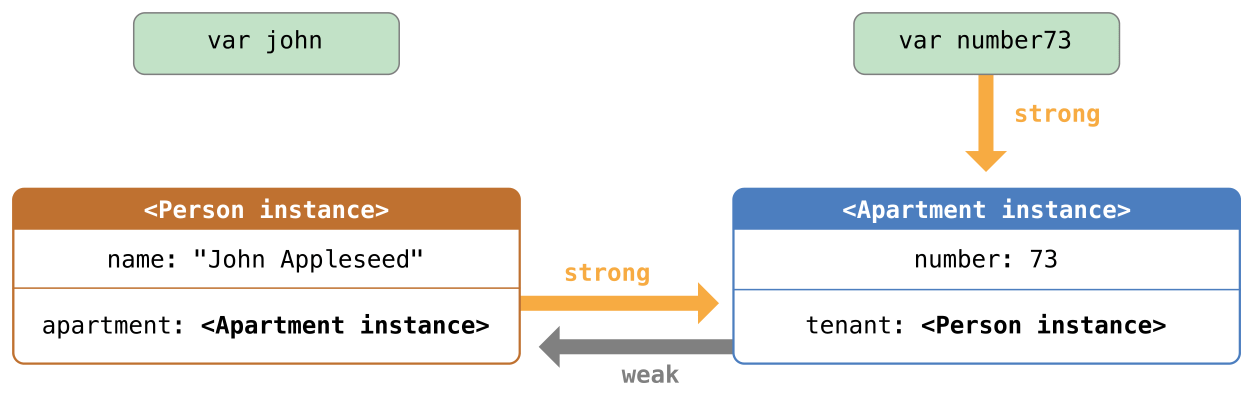
然后创建两个变量(john和number73)的强引用，并关联这两个实例：

1. **var** john: Person?
2. **var** number73: Apartment?
4. john = Person(name: "John Appleseed")
5. number73 = Apartment(nunber: 73)
7. john!.apartment = number73
8. number73!.tenant = john

下面是引用的关系图：



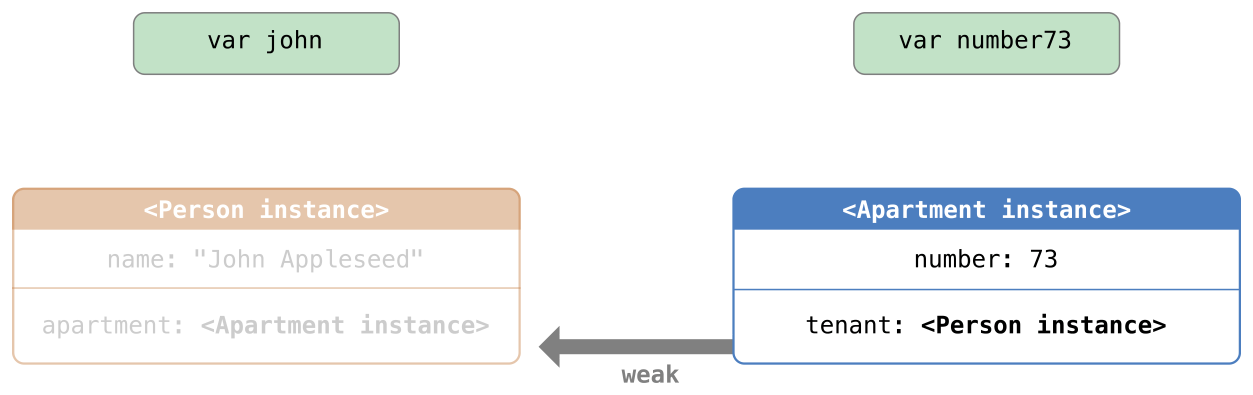
Person的实例仍然是Apartment实例的强引用，但是Apartment实例则是Person实例的弱引用。这意味着当破坏john变量所持有的强引用后，不再存在任何Person实例的强引用：



既然不存在Person实例的强引用，那么该实例就会被销毁：

1. john = nil
2. //  打印"John Appleseed is being deinitialized"

只有number73还持有Apartment实例的强引用。如果你破坏这个强引用，那么也不存在Apartment实例的任何强引用：



这时，Apartment实例也被销毁：

1. number73 = nil
2. // 打印"Apartment #73 is being deinitialized"

上面的两段代码表明在john和number73赋值为nil后，Person和Apartment实例的deinitializer都打印了“销毁”的消息。这证明了引用环已经被打破了。

**无主引用**

和弱引用相似，无主引用也不强持有实例。但是和弱引用不同的是，无主引用默认始终有值。因此，无主引用只能定义为非可选类型（non-optional type）。在属性、变量前添加unowned关键字，可以声明一个无主引用。

因为是非可选类型，因此当使用无主引用的时候，不需要展开，可以直接访问。不过非可选类型变量不能赋值为nil，因此当实例被销毁的时候，ARC无法将引用赋值为nil。

注意:

当实例被销毁后，试图访问该实例的无主引用会触发运行时错误。使用无主引用时请确保引用始终指向一个未销毁的实例。 上面的非法操作会百分百让应用崩溃，不会发生无法预期的行为。因此，你应该避免这种情况。

接下来的例子定义了两个类，Customer和CreditCard，模拟了银行客户和客户的信用卡。每个类都一个属性，存储另外一个类的实例。这样的关系可能会产生强引用环。

Customer、CreditCard的关系和之前弱引用例子中的Apartment、Person的关系截然不同。在这个模型中，消费者不一定有信用卡，但是每张信用卡一定对应一个消费者。鉴于这种关系，Customer类有一个可选类型属性card，而CreditCard类的customer属性则是非可选类型的。

进一步，要创建一个CreditCard实例，只能通过传递number值和customer实例到定制的CreditCard初始化函数来完成。这样可以确保当创建CreditCard实例时总是有一个customer实例与之关联。

因为信用卡总是对应一个消费者，因此定义customer属性为无主引用，这样可以避免强引用环：

1. **class** Customer {
2. let name: String
3. **var** card: CreditCard?
4. init(name: String) {
5. self.name = name
6. }
8. deinit { println("\(name) is being deinitialized")
9. }
11. **class** CreditCard {
12. let number: Int
13. unowned let customer: Customer
14. init(number: Int, customer: Customer) {
15. self.number = number
16. self.customer = customer
17. }
19. deinit { println("Card #\(number) is being deinitialized")
20. }

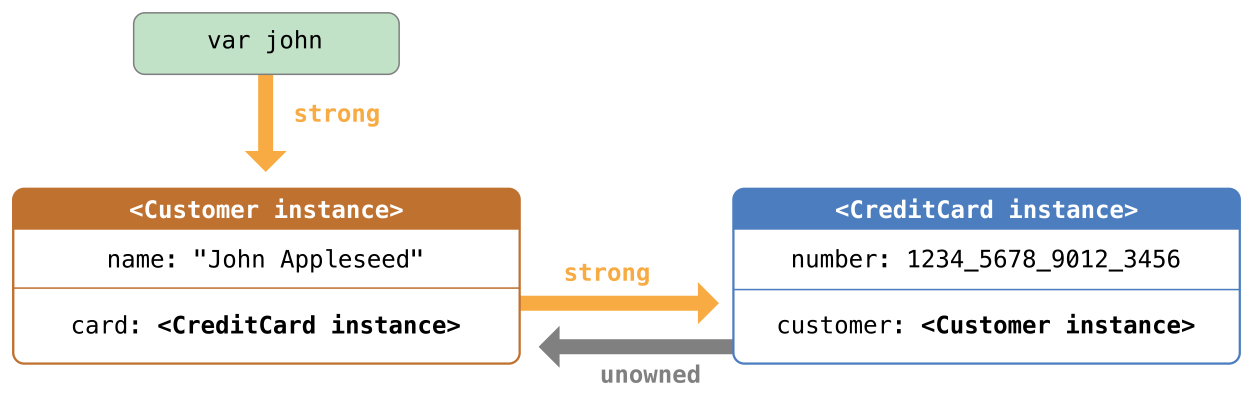
下面的代码定义了一个叫john的可选类型Customer变量，用来保存某个特定消费者的引用。因为是可变类型，该变量的初始值为nil：

1. **var** john: Customer?

现在创建一个Customer实例，然后用它来初始化CreditCard实例，并把刚创建出来的CreditCard实例赋值给Customer的card属性：

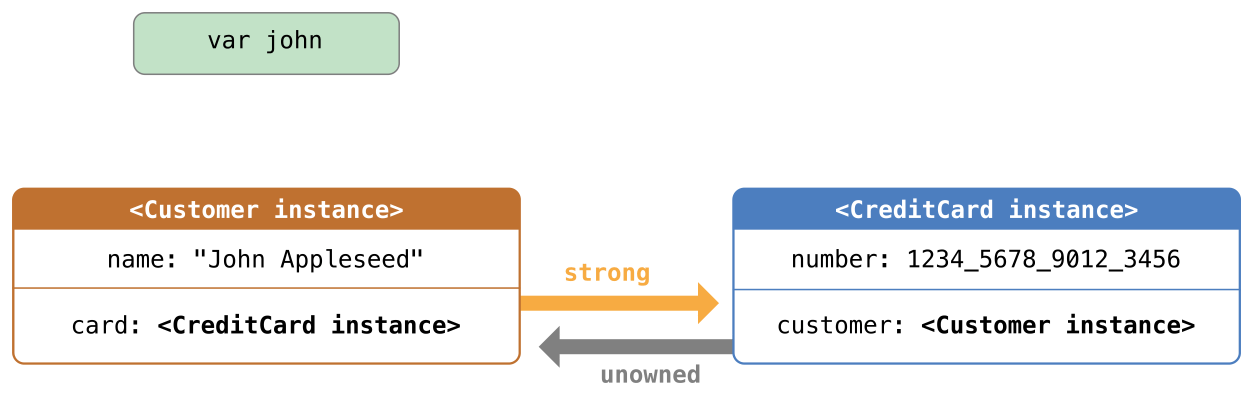
1. john = Customer(name: "John Appleseed")
2. john!.card = CreditCard(number: 1234\_5678\_9012\_3456, customer:john!)

我们来看看此时的引用关系：



Customer实例持有CreditCard实例的强引用，而CreditCard实例则持有Customer实例的无主引用。

因为customer的无主引用，当破坏john变量持有的强引用时，就没有Customer实例的强引用了：



此时Customer实例被销毁。然后，CreditCard实例的强引用也不复存在，因此CreditCard实例也被销毁：

1. john = nil
2. // 打印"John Appleseed is being deinitialized"
3. // 打印"Card #1234567890123456 is being deinitialized"

上面的代码证明，john变量赋值为nil后，Customer实例和CreditCard实例的deinitializer方法都打印了"deinitialized"消息。

**无主引用以及隐式展开的可选属性**

上述的弱引用和无主引用的例子覆盖了两种常用的需要打破强引用环的应用场景。

Person和Apartment的例子说明了下面的场景：两个属性的值都可能是nil,并有可能产生强引用环。这种场景下适合使用弱引用。

Customer和CreditCard的例子则说明了另外的场景：一个属性可以是nil，另外一个属性不允许是nil，并有可能产生强引用环。这种场景下适合使用无主引用。

但是，存在第三种场景：两个属性都必须有值，且初始化完成后不能为nil。这种场景下，则要一个类用无主引用属性，另一个类用隐式展开的可选属性。

这样，在初始化完成后我们可以立即访问这两个变量（而不需要可选展开），同时又避免了引用环。本节将告诉你应该如何配置这样的关系。

下面的例子顶一个了两个类，Country和City，都有一个属性用来保存另外的类的实例。在这个模型里，每个国家都有首都，每个城市都隶属于一个国家。所以，类Country有一个capitalCity属性，类City有一个country属性：

1. **class** Country {
2. let name: String
3. let capitalCity: City!
4. init(name: String, capitalName: String) {
5. self.name = name
6. self.capitalCity = City(name: capitalName, country: self)
7. }
8. }
10. **class** City {
11. let name: String
12. unowned let country: Country
13. init(name: String, country: Country) {
14. self.name = name
15. self.country = country
16. }
17. }

City的初始化函数有一个Country实例参数，并且用country属性来存储这个实例。这样就实现了上面说的关系。

Country的初始化函数调用了City的初始化函数。但是，只有Country的实例完全初始化完后（在Two-Phase Initialization），Country的初始化函数才能把self传给City的初始化函数。

为满足这种需求，通过在类型结尾处加感叹号(City!)，我们声明Country的capitalCity属性为隐式展开的可选类型属性。就是说，capitalCity属性的默认值是nil，不需要展开它的值（在Implicity Unwrapped Optionals中描述）就可以直接访问。

因为capitalCity默认值是nil，一旦Country的实例在初始化时给name属性赋值后，整个初始化过程就完成了。这代表只要赋值name属性后，Country的初始化函数就能引用并传递隐式的self。所以，当Country的初始化函数在赋值capitalCity时，它也可以将self作为参数传递给City的初始化函数。

综上所述，你可以在一条语句中同时创建Country和City的实例，却不会产生强引用环，并且不需要使用感叹号来展开它的可选值就可以直接访问capitalCity：

1. **var** country = Country(name: "Canada", capitalName: "Ottawa")
2. println("\(country.name)'s captial city is called \(country.capitalCity.name)")
3. // 打印"Canada's capital city is called Ottawa"

在上面的例子中，使用隐式展开的可选值满足了两个类的初始化函数的要求。初始化完成后，capitalCity属性就可以做为非可选值类型使用，却不会产生强引用环。

**闭包产生的强引用环**

前面我们看到了强引用环是如何产生的，还知道了如何引入弱引用和无主引用来打破引用环。

将一个闭包赋值给类实例的某个属性，并且这个闭包使用了实例，这样也会产生强引用环。这个闭包可能访问了实例的某个属性，例如self.someProperty，或者调用了实例的某个方法，例如self.someMethod。这两种情况都导致了闭包使用self，从而产生了抢引用环。

因为诸如类这样的闭包是引用类型，导致了强引用环。当你把一个闭包赋值给某个属性时，你也把一个引用赋值给了这个闭包。实质上，这个之前描述的问题是一样的－两个强引用让彼此一直有效。但是，和两个类实例不同，这次一个是类实例，另一个是闭包。

Swift提供了一种优雅的方法来解决这个问题，我们称之为闭包占用列表(closuer capture list)。同样的，在学习如何避免因闭包占用列表产生强引用环之前，先来看看这个抢引用环是如何产生的。

下面的例子将会告诉你当一个闭包引用了self后是如何产生一个抢引用环的。本例顶一个一个名为HTMLElement的类，来建模HTML中的一个单独的元素：

1. **class** HTMLElement {
3. let name: String
4. let text: String?
6. @lazy **var** asHTML: () -> String = {
7. **if** let text = self.text {
8. **return** "<\(self.name)>\(text)</\(self.name)>"
9. } **else** {
10. **return** "<\(self.name) />"
11. }
12. }
14. init(name: String, text: String? = nil) {
15. self.name = name
16. self.text = text
17. }
19. deinit {
20. println("\(name) is being deinitialized")
21. }
23. }

类HTMLElement定义了一个name属性来表示这个元素的名称，例如代表段落的"p"，或者代表换行的"br"；以及一个可选属性text，用来设置HTML元素的文本。

除了上面的两个属性，HTMLElement还定义了一个lazy属性asHTML。这个属性引用了一个闭包，将name和text组合成HTML字符串片段。该属性是() -> String类型，就是“没有参数，返回String的函数”。

默认将闭包赋值给了asHTML属性，这个闭包返回一个代表HTML标签的字符串。如果text值存在，该标签就包含可选值text；或者不包含文本。对于段落，根据text是"some text"还是nil，闭包会返回"<p>some text</p>"或者"<p />"。

可以像实例方法那样去命名、使用asHTML。然而，因为asHTML终究是闭包而不是实例方法，如果你像改变特定元素的HTML处理的话，可以用定制的闭包来取代默认值。

注意：asHTML声明为lazy属性，因为只有当元素确实需要处理为HTML输出的字符串时，才需要使用asHTML。也就是说，在默认的闭包中可以使用self，因为只有当初始化完成以及self确实存在后，才能访问lazy属性。

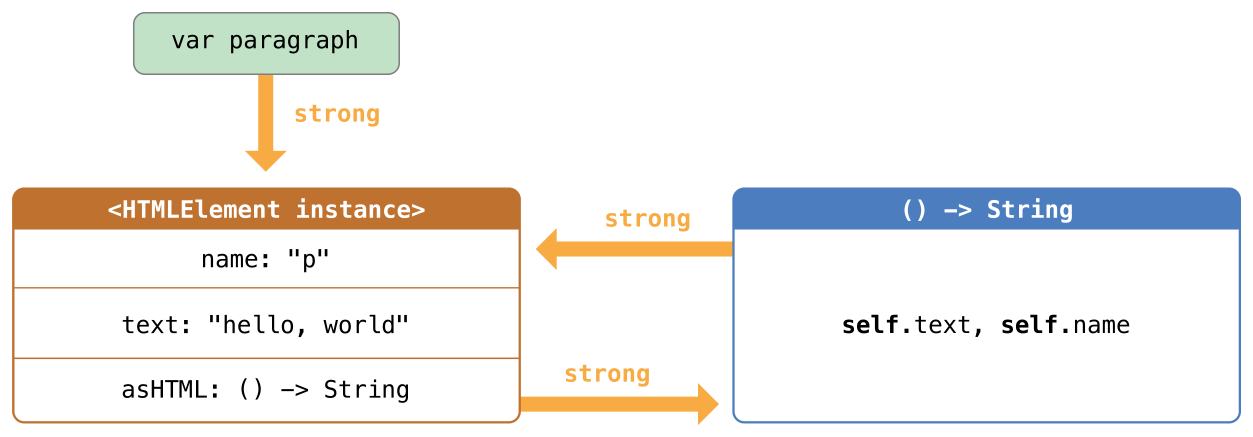
HTMLElement只有一个初始化函数，根据name和text(如果有的话)参数来初始化一个元素。该类也定义了一个deinitializer，当HTMLElement实例被销毁时，打印一条消息。

下面的代码创建一个HTMLElement实例并打印消息。

1. **var** paragraph: HTMLElement? = HTMLElement(name: "p", text: "hello, world")
2. println(paragraph!.asHTML())
3. // 打印"<p>hello, world</p>"

注意：上面的paragraph变量定义为可选HTMLElement，因此我们可以赋值nil给它来演示强引用环。

不幸的是，HTMLElement类产生了类实例和asHTML默认值的闭包之间的强引用环。如下图所示：



实例的asHTML属性持有闭包的强引用。

但是，闭包使用了self（引用了self.name和self.text），因此闭包占有了self，这意味着闭包又反过来持有了HTMLElement实例的强引用。这样就产生了强引用环。（更多闭包哪占有值的信息，请参考Capturing Values）。

注意：虽然闭包多次使用了self，它只占有HTMLElement实例的一个强引用。

如果设置paragraph为nil，打破它持有的HTMLElement实例的强引用，HTMLElement实例和它的闭包都不会被销毁，就因为强引用环：

1. paragraph = nil

注意，HTMLElementdeinitializer中的消息并没有别打印，印证了HTMLElement实例并没有被销毁。

**解决闭包产生的强引用环**

在定义闭包时同时定义占有列表作为闭包的一部分，可以解决闭包和类实例之间的强引用环。占有列表定义了闭包内占有一个或者多个引用类型的规则。和解决两个类实例间的强引用环一样，声明每个占有的引用为弱引用或无主引用，而不是强引用。根据代码关系来决定使用弱引用还是无主引用。

注意：Swift有如下约束：只要在闭包内使用self的成员，就要用self.someProperty或者self.someMethod（而非只是someProperty或someMethod）。这可以提醒你可能会不小心就占有了self。

**定义占有列表**

占有列表中的每个元素都是由weak或者unowned关键字和实例的引用(如self或someInstance)组成。每一对都在花括号中，通过逗号分开。

占有列表放置在闭包参数列表和返回类型之前：

1. @lazy **var** someClosure: (Int, String) -> String = {
2. [unowned self] (index: Int, stringToProcess: String) -> String **in**
3. // closure body goes here
4. }

如果闭包没有指定参数列表或者返回类型（可以通过上下文推断），那么占有列表放在闭包开始的地方，跟着是关键字in：

1. @lazy **var** someClosure: () -> String = {
2. [unowned self] **in**
3. // closure body goes here
5. }

**弱引用和无主引用**

当闭包和占有的实例总是互相引用时并且总是同时销毁时，将闭包内的占有定义为无主引用。

相反的，当占有引用有时可能会是nil时，将闭包内的占有定义为弱引用。弱引用总是可选类型，并且当引用的实例被销毁后，弱引用的值会自动置为nil。利用这个特性，我们可以在闭包内检查他们是否存在。

注意：如果占有的引用绝对不会置为nil，应该用无主引用，而不是弱引用。

前面提到的HTMLElement例子中，无主引用是正确的解决强引用的方法。这样编码HTMLElement类来避免强引用环：

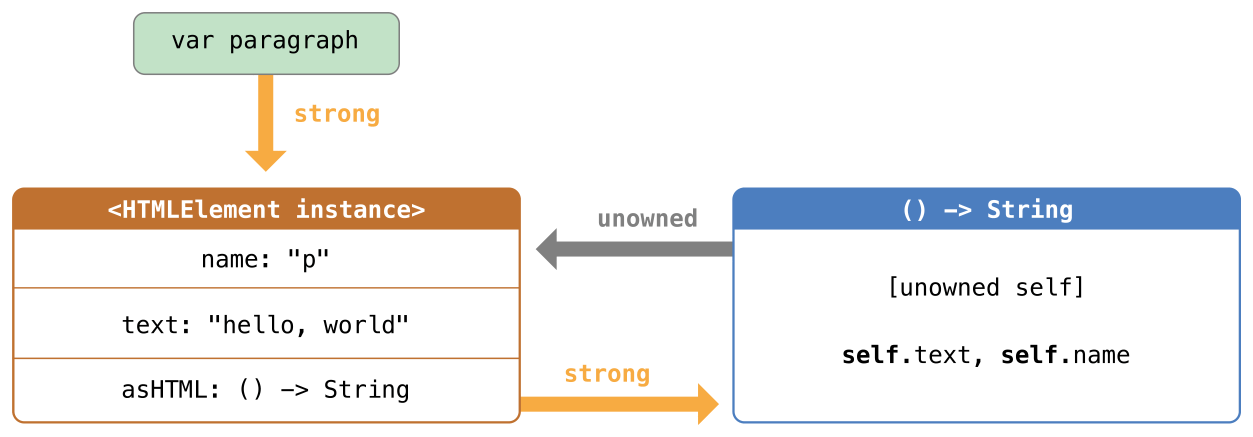
1. **class** HTMLElement {
3. let name: String
4. let text: String?
6. @lazy **var** asHTML: () -> String = {
7. [unowned self] **in**
8. **if** let text = self.text {
9. **return** "<\(self.name)>\(text)</\(self.name)>"
10. } **else** {
11. **return** "<\(self.name) />"
12. }
13. }
15. init(name: String, text: String? = nil) {
16. self.name = name
17. self.text = text
18. }
20. deinit {
21. println("\(name) is being deinitialized")
22. }
24. }

上面的HTMLElement实现和之前的实现相同，只是多了占有列表。这里，占有列表是[unowned self]，代表“用无主引用而不是强引用来占有self”。

和之前一样，我们可以创建并打印HTMLElement实例：

1. **var** paragraph: HTMLElement? = HTMLElement(name: "p", text: "hello, world")
2. println(paragraph!.asTHML())
3. // 打印"<p>hello, world</p>"

使用占有列表后引用关系如下图所示：



这一次，闭包以无主引用的形式占有self，并不会持有HTMLElement实例的强引用。如果赋值paragraph为nil，HTMLElement实例将会被销毁，并能看到它的deinitializer打印的消息。

1. paragraph = nil
2. // 打印"p is being deinitialized"