Swift开发必备技巧：内存管理、weak和unowned

时间 2015-03-02 10:15:32 极客头条 (/sites/e2aA32)
原文 http://swifter.tips/retain-cycle/ (http://swifter.tips/retain-cycle/?utm_source=tuicool&utm_medium=referral)
主题 Swift (/topics/11080178)

因为 Playground 本身会持有所有声明在其中的东西，因此本节中的示例代码需要在 Xcode 项目环境中运行。在 Playground 中可能无法得到正确的结果。

不管在什么语言里，内存管理的内容都很重要，所以我打算花上比其他 tip 长一些的篇幅仔细地说说这块内容。

Swift 是自动管理内存的，这也就是说，我们不再需要操心内存的申请和分配。当我们通过初始化创建一个对象时，Swift 会替我们管理和分配内存。而释放的原则遵循了自动引用计数 (ARC) 的规则：当一个对象没有引用的时候，其内存将会被自动回收。这套机制从很大程度上简化了我们的编码，我们只需要保证在合适的时候将引用置空 (比如超过作用域，或者手动设为 `nil` 等)，就可以确保内存使用不出现问题。

但是，所有的自动引用计数机制都有一个从理论上无法绕过的限制，那就是循环引用 (retain cycle) 的情况。

什么是循环引用

虽然我觉得循环引用这样的概念介绍不太应该出现在这本书中，但是为了更清晰地解释 Swift 中的循环引用的一般情况，这里还是简单进行说明。假设我们有两个类 `A` 和 `B`， 它们之中分别有一个存储属性持有对方：

```
class A {
        let b: B
        init() {
                b = B()
                b.a = self
        }
        deinit {
                println("A deinit")
        }
}
class B {
        var a: A? = nil
        deinit {
                println("B deinit")
        }
}
```

在 `A` 的初始化方法中，我们生成了一个 `B` 的实例并将其存储在属性中。然后我们又将 `A` 的实例赋值给了 `b.a` 。这样 `a.b` 和 `b.a` 将在初始化的时候形成一个引用循环。现在当有第三方的调用初始化了 `A` ，然后即使立即将其释放， `A` 和 `B` 两个类实例的 `deinit` 方法也不会被调用，说明它们并没有被释放。

```
func application(application: UIApplication!,
                                 didFinishLaunchingWithOptions launchOptions: NSD
ictionary!)
                                 -> Bool
{
        // Override point for customization after application launch.
        var obj: A? = A()
        obj = nil
        // 内存没有释放
        return true
}
```

因为即使 `obj` 不再持有 `A` 的这个对象，b 中的 `b.a` 依然引用着这个对象，导致它无法释放。而进一步，a 中也持有着 b，导致 b 也无法释放。在将 `obj` 设为 `nil` 之后，我们在代码里再也拿不到对于这个对象的引用了，所以除非是杀掉整个进程，我们已经 **永远** 也无法将它释放了。多么悲伤的故事啊..

在 Swift 里防止循环引用

为了防止这种人神共愤的悲剧的发生，我们必须给编译器一点提示，表明我们不希望它们互相持有。一般来说我们习惯希望 "被动" 的一方不要去持有 "主动" 的一方。在这里 b.a 里对 A 的实例的持有是由 A 的方法设定的，我们在之后直接使用的也是 A 的实例，因此认为 b 是被动的一方。可以将上面的 `class B` 的声明改为：

```
class B {
    weak var a: A? = nil
    deinit {
        println("B deinit")
    }
}
```

在 `var a` 前面加上了 `weak` ，向编译器说明我们不希望持有 a。这时，当 `obj` 指向 `nil` 时，整个环境中就没有对 `A` 的这个实例的持有了，于是这个实例可以得到释放。接着，这个被释放的实例上对 b 的引用 `a.b` 也随着这次释放结束了作用域，所以 `b` 的引用也将归零，得到释放。添加 `weak` 后的输出：

```
A deinit
B deinit
```

可能有心的朋友已经注意到，在 Swift 中除了 `weak` 以外，还有另一个冲着编译器叫喊着类似的 "不要引用我" 的标识符，那就是 `unowned` 。它们的区别在哪里呢？如果您是一直写 Objective-C 过来的，那么从表面的行为上来说 `unowned` 更像以前的 `unsafe_unretained` ，而 `weak` 就是以前的 `weak` 。用通俗的话说，就是 `unowned` 设置以后即使它原来引用的内容已经被释放了，它仍然会保持对被已经释放了的对象的一个 "无效的" 引用，它不能是 Optional 值，也不会被指向 `nil` 。如果你尝试调用这个引用的方法或者访问成员属性的话，程序就会崩溃。而 `weak` 则友好一些，在引用的内容被释放后，标记为 `weak` 的成员将会自动地变成 `nil` (因此被标记为 @ `weak` 的变量一定需要是 Optional 值)。关于两者使用的选择，Apple 给我们的建议是如果能够确定在访问时不会已被释放的话，尽量使用 `unowned` ，如果存在被释放的可能，那就选择用 `weak` 。

我们结合实际编码中的使用来看看选择吧。日常工作中一般使用弱引用的最常见的场景有两个：

1. 设置 `delegate` 时
2. 在 `self` 属性存储为闭包时，其中拥有对 `self` 引用时

前者是 Cocoa 框架的常见设计模式，比如我们有一个负责网络请求的类，它实现了发送请求以及接收请求结果的任务，其中这个结果是通过实现请求类的 protocol 的方式来实现的，这种时候我们一般设置 `delegate` 为 `weak` ：

```
// RequestManager.swift
class RequestManager: RequestHandler {
        func requestFinished() {
                println("请求完成")
        }
        func sendRequest() {
                let req = Request()
                req.delegate = self
                req.send()
        }
}
// Request.swift
@objc protocol RequestHandler {
        optional func requestFinished()
}
class Request {
        weak var delegate: RequestHandler!;
        func send() {
                // 发送请求
                // 一般来说会将 req 的引用传递给网络框架
        }
        func gotResponse() {
                // 请求返回
                delegate?.requestFinished?()
        }
}
```

`req` 中以 `weak` 的方式持有了 delegate，因为网络请求是一个异步过程，很可能会遇到用户不愿意等待而选择放弃的情况。这种情况下一般都会将 `RequestManager` 进行清理，所以我们其实是无法保证在拿到返回时作为 `delegate` 的 `RequestManager` 对象是一定存在的。因此我们使用了 `weak` 而非 `unowned` ，并在调用前进行了判断。

闭包和循环引用

另一种闭包的情况稍微复杂一些：我们首先要知道，闭包中对任何其他元素的引用都是会被闭包自动持有的。如果我们在闭包中写了 `self` 这样的东西的话，那我们其实也就在闭包内持有了当前的对象。这里就出现了一个在实际开发中比较隐蔽的陷阱：如果当前的实例直接或者间接地对这个闭包又有引用的话，就形成了一个 self -> 闭

包 -> self 的循环引用。最简单的例子是，我们声明了一个闭包用来以特定的形式打印 `self` 中的一个字符串：

```
class Person {
        let name: String
        lazy var printName: ()->() = {
                println("The name is \(self.name)")
        }
        init(personName: String) {
                name = personName
        }
        deinit {
                println("Person deinit \(self.name)")
        }
}
func application(application: UIApplication!,
                didFinishLaunchingWithOptions launchOptions: NSDictionary!)
                -> Bool
{
        // Override point for customization after application launch.
        var xiaoMing: Person = Person(personName: "XiaoMing")
        xiaoMing.printName()
        return true
}
```

```
// 输出:
// The name is XiaoMing
```

`printName` 是 `self` 的属性，会被 `self` 持有，而它本身又在闭包内持有 `self` ，这导致了 `xiaoMing` 的 `deinit` 在自身超过作用域后还是没有被调用，也就是没有被释放。为了解决这种闭包内的循环引用，我们需要在闭包开始的时候添加一个标注，来表示这个闭包内的某些要素应该以何种特定的方式来使用。可以将 `printName` 修改为这样：

```
lazy var printName: ()->() = {
    [weak self] in
    if let strongSelf = self {
        println("The name is \(strongSelf.name)")
    }
}
```

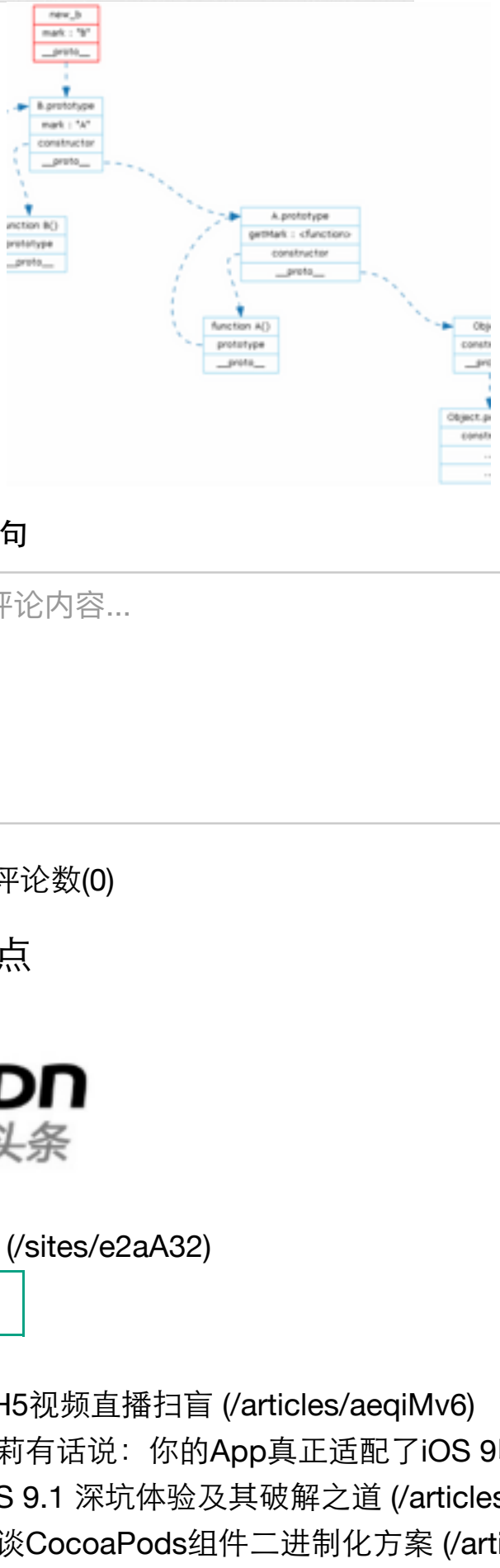
现在内存释放就正确了：

```
// 输出:
// The name is XiaoMing
// Person deinit XiaoMing
```

如果我们可以确定在整个过程中 `self` 不会被释放的话，我们可以将上面的 `weak` 改为 `unowned` ，这样就不再需要 `strongSelf` 的判断。但是如果在过程中 `self` 被释放了而 `printName` 这个闭包没有被释放的话(比如 生成 `Person` 后，某个外部变量持有了 `printName` ，随后这个 `Person` 对象被释放了，但是 `printName` 已然存在并可能被调用)，使用 `unowned` 将造成崩溃。在这里我们需要根据实际的需求来决定是使用 `weak` 还是 `unowned` 。

这种在闭包参数的位置进行标注的语法结构是将要标注的内容放在原来参数的前面，并使用中括号括起来。如果有多个需要标注的元素的话，在同一个中括号内用逗号隔开，举个例子:

```
// 标注前
{ (number: Int) -> Bool in
        //...
        return true
}
// 标注后
{ [unowned self, weak someObject] (number: Int) -> Bool in
        //...
        return true
}
```

- 
  《匿名收藏》 82
- by jj8j 
  (/kans/781870171) 《默认推刊》 (/kans/781870171) *118*

我来评几句

请输入评论内容...

登录后评论

已发表评论数(0)

相关站点

极客头条 (/sites/e2aA32)

+ 订阅

热门文章

- 1. 3.H5视频直播扫盲 (/articles/aeqiMv6)
- 2. 萝莉有话说：你的App真正适配了iOS 9吗？ (/articles/AjMR7v6)
- 3. iOS 9.1 深坑体验及其破解之道 (/articles/RNnEZfe)
- 4. 谈谈CocoaPods组件二进制化方案 (/articles/q6Vj6rr)
- 5. iOS 懒加载的使用 (/articles/uMbYV3b)

(http://click.aliyun.com/m/17039/)

(https://activity.ksyun.com/1703/index.html?ch=00033.00018&hmsr=%E6%8E%A8%E9%85%B7&hmpl=1703&hmcu=&hmkw=&hmci=)

(http://ml-summit.org/?hmsr=tuicool&hmpl=banner&hmcu=&hmkw=&hmci=)

(https://www.mysubmail.com/sms?s=tuicool)

(https://mos.meituan.com/firework/newcustomer?site=tuicool&campaign=20170401sales)

(http://www.bagevent.com/event/268776?bag_track=tuicool)

(https://sspaas.com/)

关于我们 (http://www.tuicool.com/about) 移动应用 (http://www.tuicool.com/mobile) 意见反馈 (http://www.tuicool.com/bbs/go/issues) 官方微博 (http://e.weibo.com/tuicool2012)