

☰ 理解引用语义的自定义类型

◀ 容易让人犯错的closure内存管理 - I

使用访问控制管理代码 ▶

(<https://www.boxueio.com/series/understand-ref-types/ebook/184>)

(<https://www.boxueio.com/series/understand-ref-types/ebook/186>)

容易让人犯错的closure内存管理 - II

⌕ Back to series (</series/understand-ref-types>)

为了演示通过capture list解决引用循环的问题，我们用之前通过外部closure造成引用循环的代码举例。
首先，我们可以通过capture list，让 fn 不要捕获 boss 变量，而是捕获 boss 变量引用的对象：

```
if true {
  var boss = Role("boss")
  let fn = { [boss] in
    print("\(boss) takes action.")
  }

  boss.action = fn
}
```

🔍 字号

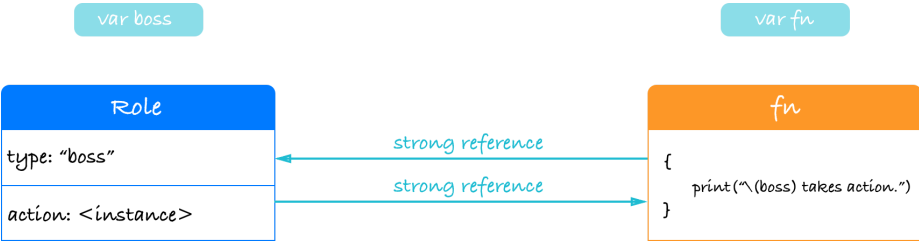
● 字号

✍ 默认主题

✍ 金色主题

✍ 暗色主题

但是，由于 boss 变量引用的对象仍就是一个引用类型，离开 if 循环之后，这样依旧会存在下面这样的引用循环：



但解决这样的问题我们就比较熟悉了，可以让循环中的一方变成 unowned 或 weak 就好了。

使用unowned处理closure和类对象同生共死的引用循环

```
if true {
  var boss = Role("boss")
  let fn = { [unowned boss] in
    print("\(boss) takes action.")
  }

  boss.action = fn
}
```

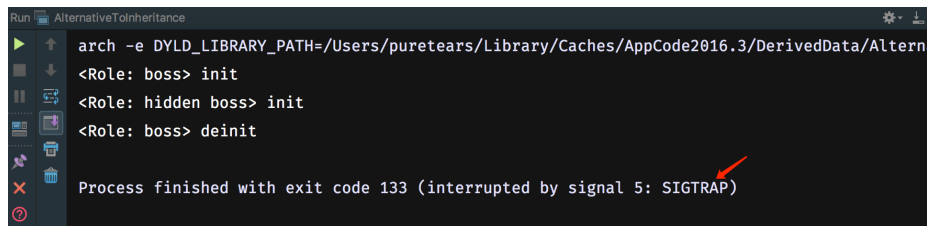
这样，离开作用域之后， boss 引用的对象就不再有strong reference了，它会被回收。进而closure对象也就没有了strong reference，它也就被回收掉了。但是，**只有在closure对象和它捕获的类对象“同生共死”的时候，使用 unowned 才是安全的。**如果不是这样，使用 unowned 的结果就绝对超出你想象，呃……，没错，是超出你想象的糟糕。来看下面两个例子：

```
if true {
  var boss = Role("boss")
  let fn = { [unowned boss] in
    print("\(boss) takes action.")
  }

  boss = Role("hidden boss")
  boss.action = fn

  boss.action()
}
```

在给 action 赋值之前，我们让 boss 等于了一个新的 Role 对象，执行一下这段代码就会得到这样的结果：



```

Run AlternativeToInheritance
arch -e DYLD_LIBRARY_PATH=/Users/puretears/Library/Caches/AppCode2016.3/DerivedData/Altern
<Role: boss> init
<Role: hidden boss> init
<Role: boss> deinit
Process finished with exit code 133 (interrupted by signal 5: SIGTRAP)

```

从图中可以看到，还没等 action 执行，程序已经异常退出了。这是为什么呢？我们来分析一下：

1. fn 通过capture list捕获了 name 等于"boss"的对象；
2. 当 boss 等于新创建的"hidden boss"对象之后，由于 fn 是按照 unowned 方式捕获的，因此 closure内的 boss 引用的对象实际上就不存在了；
3. 新创建的 boss 对象的 action 引用了 fn ；
4. 调用 boss.action 的时候，closure对象之前捕获的boss对象已经不存在了，于是就发生了异常；

因此，我们既不会看到closure对象打印的消息，也不会看到hidden boss的 deinit 调用。那么，如果我们把 boss.action = fn 放到 boss = Role("hidden boss") 前面呢？

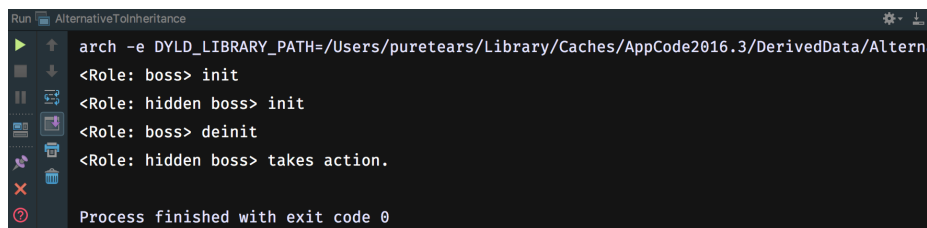
```

if true {
    var boss = Role("boss")
    let fn = { [unowned boss] in
        print("\(boss) takes action.")
    }
    boss.action = fn

    boss = Role("hidden boss")
    boss.action()
}

```

执行结果就会变成这样：



```

Run AlternativeToInheritance
arch -e DYLD_LIBRARY_PATH=/Users/puretears/Library/Caches/AppCode2016.3/DerivedData/Altern
<Role: boss> init
<Role: hidden boss> init
<Role: boss> deinit
<Role: hidden boss> takes action.
Process finished with exit code 0

```

可以看到，"boss"对象正常创建和销毁了，我们调用了"hidden boss"的 action 方法，但是"hidden boss"却没有被销毁，显然我们创建了引用循环，这又是为什么呢？

其实，这次和closure捕获变量没关系，当 boss = Role("hidden boss") 执行后，boss 对象的内容就被新对象“刷新”了，这次造成循环引用的，是 boss 对象内的 action 默认实现捕获了 self 导致的。

无论如何，你已经看到了，如果closure对象和类对象的生命周期不一致，使用 unowned 就不会带来你想要的结果。对于这种情况，我们至多可以做的，就是告诉closure对象：“当你捕获到的对象已经不存在的时候，就不要再访问它了。”

为此，我们可以在capture list中使用 weak 关键字。

使用weak处理closure和类对象生命周期不同的引用循环

为了解决刚才我们提到的引用循环的问题，我们可以把 action 的默认实现改成这样：

```

class Role {
    var name: String
    lazy var action: () -> Void = { [weak self] in
        if let role = self {
            print("\(role) takes action.")
        }
    }

    // ...
}

```

然后，重新执行我们的测试代码，就可以看到"hidden boss"可以正常执行并销毁了：

```

Run AlternativeToInheritance
arch -e DYLD_LIBRARY_PATH=/Users/puretears/Library/Caches/AppCode2016.3/DerivedData/Altern
<Role: boss> init
<Role: hidden boss> init
<Role: boss> deinit
<Role: hidden boss> takes action.
<Role: hidden boss> deinit
Process finished with exit code 0

```

在capture list里，一旦我们把捕获的内容变成 weak 之后，就意味着它有可能为 nil，所以，在closure的实现里，捕获到的内容就变成了一个optional。但对于 self 来说，这样做有一个小缺陷，就是我们无法编写这样的代码：

```

class Role {
    // ...
    lazy var action: () -> Void = { [weak self] in
        if let self = self { // <-- Error here
            print("\(self) takes action.")
        }
    }
}

```

Swift不允许我们用 self 作为value binding的变量，要么，我们自己手动在closure里unwrap到它的值，要么，我们只能使用其它名字的变量来绑定它的值。

实际上，Swift 3允许我们使用 let `self` = self 这样的形式在代码中使用关键字作为变量，但不要过度使用这种方法。

一个不会造成引用循环却容易造成困惑的情况

以上就是closure和类对象之间引用循环有关的所有内容，实际上判断是否会造成引用循环，一个根本的出发点就是，你要了解一个类对象是否真的拥有你正在使用的closure。如果答案是不，那你大可不必考虑这个问题。

但在有些场景里，closure和类对象的关系并不那么明显。最典型的一类情况，就是类对象和closure之间还隔了一层API调用，像这样：

```

class Role {
    // ...
    func levelUp() {
        let globalQueue = DispatchQueue.global()

        globalQueue.async {
            print("Before: \(self) level up")
            usleep(1000)
            print("After: \(self) level up")
        }
    }
}

```

这里，我们使用 usleep(1000) 来模拟角色升级时需要进行的设置，并在升级前后打印了对应的消息。我们需要在 async 的closure中使用capture list么？

实际上并不需要，因为 async 方法中使用的closure并不归 Role 对象所有，只是closure会捕获 Role 对象，它们之间不会发生引用循环。我们可以用下面的代码试一下：

```

var player: Role? = Role("P1")
player?.levelUp()

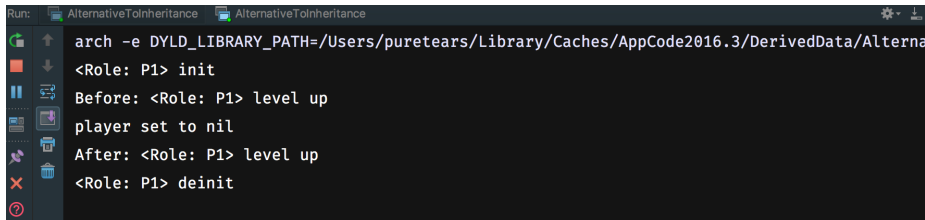
usleep(500)

print("Player set to nil")
player = nil

dispatchMain()

```

在这里，我们调用 levelUp 之后，故意拖延了500毫秒后，把 player 设置为了 nil。虽然这做法并不科学，但足以让我们观察到 async 使用的closure捕获 self 对象的效果：



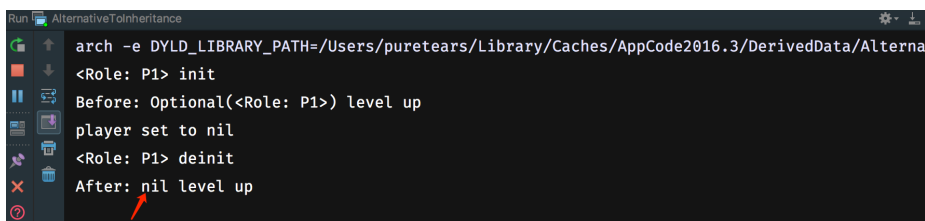
```
arch -e DYLD_LIBRARY_PATH=/Users/puretears/Library/Caches/AppCode2016.3/DerivedData/AlternativeToInheritance
<Role: P1> init
Before: <Role: P1> level up
player set to nil
After: <Role: P1> level up
<Role: P1> deinit
```

可以看到，尽管我们把 `player` 设置为了 `nil`，`player` 引用的对象也是在 `async` 执行之后，才会被回收掉。

但是，有时我们还是习惯在 `capture list` 中使用 `weak`，以避免在不知情的情况下引入引用循环。当你理解了 `capture list` 以及 `weak` 真正的工作方式之后，就会知道，这样做并不是避免问题的万金油，有时，还会给你带来诡异的麻烦。如果我们在 `async closure` 的 `capture list` 中加入 `weak`：

```
globalQueue.async { [weak self] in
    print("Before: \(self) level up")
    usleep(1000)
    print("After: \(self) level up")
}
```

执行一下，就会看到下面的结果：



```
arch -e DYLD_LIBRARY_PATH=/Users/puretears/Library/Caches/AppCode2016.3/DerivedData/AlternativeToInheritance
<Role: P1> init
Before: Optional(<Role: P1>) level up
player set to nil
<Role: P1> deinit
After: nil level up
```

看到红色箭头指的内容了么？`async` 线程恢复执行之后，由于 `self` 已经被主线程设置成 `nil`，此时打印的结果就已经不正确了。

你看，我们不仅让 `self` 变成了一个 `optional`，用起来不太方便，连执行的语义都错了。难道就没有既不会造成引用循环，又可以保证 `closure` 执行过程内对象一定存在的办法么？

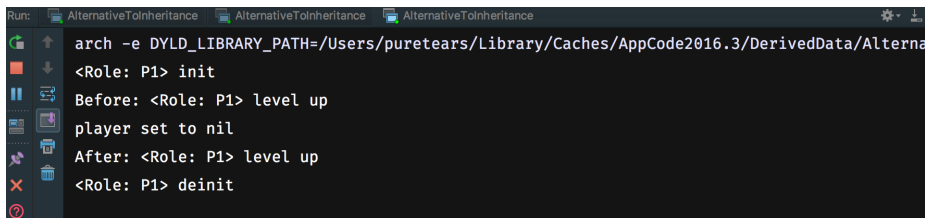
有，当然有。

一个可以稍稍改进开发体验的语法糖

Swift 标准库中，有一个叫做 `withExtendedLifetime` 的函数，它有两个参数：第一个参数是它要“延长寿命”的对象；第二个参数是一个 `closure`，在这个 `closure` 返回之前，第一个参数会一直“存活”在内存里。于是，我们的 `async closure` 就可以改成这样：

```
globalQueue.async { [weak self] in
    withExtendedLifetime(self) {
        print("Before: \(self!) level up")
        usleep(1000)
        print("After: \(self!) level up")
    }
}
```

由于在 `withExtendedLifetime` 的 `closure` 参数里，`self` 对象一直存在，我们可以安全的使用 `force unwrapping` 来读取对象的值。于是，重新执行，我们就会看到下面的结果：



```
arch -e DYLD_LIBRARY_PATH=/Users/puretears/Library/Caches/AppCode2016.3/DerivedData/AlternativeToInheritance
<Role: P1> init
Before: <Role: P1> level up
player set to nil
After: <Role: P1> level up
<Role: P1> deinit
```

看到了吧，这次尽管我们在 `capture list` 中使用了 `weak self`，但 `Role` 对象还是在 `async` 执行完之后才被 ARC 回收的。

这下两全其美了么？还差一点点。

之前我们说过，`self` 不能作为 `value binding` 的变量，因此，在 `withExtendedLifetime` 的实现里，我们只能使用 `self!` 来读取对象，如果有很多地方都要访问对象这就很麻烦。但如果我们一定要使用一个和 `self` 不同的名字来表示对象又会降低代码的可读性和一致性。

于是，作为一个折中的办法，我们只要把给 `self` 改名这个行为，单独封装起来。像这样，给 `Optional` 添加一个 `extension`：

```
extension Optional {
    func withExtendedLifetime(_ body: (Wrapped) -> Void) {
        if let value = self {
            body(value)
        }
    }
}
```

然后，我们之前的 `async closure` 就可以写成这样：

```
globalQueue.async { [weak self] in
    self.withExtendedLifetime { // <- Out optional extension
        print("Before: \"($0) level up")
        usleep(1000)
        print("After: \"($0) level up")
    }
}
```

相比之前的原生API，我们自己实现的 `withExtendedLifetime` 不仅有更好的表意，并且，也通过 `$0` 解决了给 `self` 重命名的麻烦。而这，应该是我们当前可以想到的最安全和理想的方案了。

What's next?

以上，就是closure和类对象内存管理相关的话题。由于closure自身是一个引用语义的对象，加上它的各种capture的用法，让引用循环以及如何closure中正确使用对象这个问题显得比单纯的类对象之间复杂一些。

但通过这一节的内容，你应该知道了，用好closure的关键无非就三点内容：

- 确定类对象是否真正拥有一个closure；
- 确保自己理解closure按引用捕获对象的含义；
- 理解capture list的行为；

一旦你做到以上三点，再面对closure内存管理相关问题的时候，就可以做到胸有成竹了。在连续几节讨论了和内存管理相关的话题之后，接下来，我们来讨论如何实现代码访问权限控制。如果你熟悉其它编程语言中类似的特性，呃.....，一个不好的消息就是，Swift中的实现会和你想象的，有点儿不同。

◀ 容易让人犯错的closure内存管理 - I

(<https://www.boxueio.com/series/understand-ref-types/ebook/184>)

使用访问控制管理代码 ▶

(<https://www.boxueio.com/series/understand-ref-types/ebook/186>)



职场漂泊的你，每天多学一点。

从开发、测试到运维，让技术不再成为你成长的绊脚石。我们用打磨产品的精神去传播知识，把最新的移动开发技术，通过简单的图表，清晰的视频，简明的文字和切实可行的例子一一向你呈现。让学习不仅是一种需求，也是一种享受。

泊学动态

一个工作十年PM终创业的故事（二）(<https://www.boxueio.com/after-the-full-upgrade-to-swift3>)
Mar 4, 2017

人生中第一次创业的"10有"(<https://www.boxueio.com/founder-chat>)
Jan 9, 2016

猎云网采访报道泊学(<http://www.lieyunwang.com/archives/144329>)
Dec 31, 2015

What most schools do not teach(<https://www.boxueio.com/what-most-schools-do-not-teach>)
Dec 21, 2015

一个工作十年PM终创业的故事（一）(<https://www.boxueio.com/founder-story>)
May 8, 2015

泊学相关

关于泊学	>
加入泊学	>
泊学用户隐私及服务条款 (HTTPS://WWW.BOXUEIO.COM/TERMS-OF-SERVICE)	
版权声明 (HTTPS://WWW.BOXUEIO.COM/COPYRIGHT-STATEMENT)	

联系泊学

Email: 10[AT]boxue.io (mailto:10@boxue.io)
QQ: 2085489246

2017 © Boxue, All Rights Reserved. 京ICP备15057653号-1 (http://www.miibeian.gov.cn/) 京公网安备 11010802020752号 (http://www.beian.gov.cn/portal/registerSystemInfo?recordcode=11010802020752)

友情链接 SwiftV (http://www.swiftv.cn) | Seay信息安全博客 (http://www.cnseay.com) | Swift.gg (http://swift.gg/) | Laravist (http://laravist.com/) | SegmentFault (https://segmentfault.com) | 骛青K的博客 (http://blog.dianqk.org/)