理解引用语义的自定义类型

使用unowned和weak处理reference cycle

(https://www.boxueio.com/series/understand-ref-types/ebook/183)

容易让人犯错的closure内存管理 - II

(https://www.boxueio.com/series/understand-ref-types/ebook/185)

容易让人犯错的closure内存管理 - I

Back to series (/series/understand-ref-types)

字号
字号
默认主题
金色主题
暗色主题

相比于对象之间造成的引用循环，Closure意外造成引用循环的概率要大的多。以至于每当我们把closure传递给类方法，或者在closure中使用类对象的时候，经常就情不自禁的想一想：这样用会造成引用循环么?

为了彻底搞清楚这个问题，我们得从closure捕获变量的语义开始。

Closure捕获的究竟是什么?

先来看一段简单的代码：

```
var i = 10
var captureI = { print(i) }
i = 11

// What will this print out?
captureI() // 11
```

当调用 `captureI()` 的时候，会得到什么结果呢？答案是11。再来看下面的例子：

```
class Demo { var value = "" }

var c = Demo()
var captureC = { print(c.value) }
c.value = "updated"

captureC() // updated
```

最后， `captureC()` 会打印updated。通过这两个例子我们会发现两点：

- **无论是值类型 `i` 还是引用类型 `c`，closure捕获到的都是它们的引用**，这也是为数不多的值类型变量有引用语义的地方；
- **Closure内表达式的值，是在closure被调用的时候才评估的**，而不是在closure定义的时候评估的；

接下来，我们再来看一个例子，加深closure按引用语义捕获变量的理解：

```
var c = C()
var captureC = { print(c.value) }
c.value = "updated"
c = C() // <-- A new object

captureC() // ""
```

在调用 `captureC()` 之前，我们让 `c` 等于了一个新对象。这时 `captureC()` 就会打印一个空字符串。这说明closure捕获的是它访问的变量，也就是 `c` 的引用，而不是 `c` 引用的对象。

理解了closure捕获变量的语义之后，我们来看closure和类对象之间的引用循环是如何发生的?

Closure为什么会带来引用循环

为了后面的演示，仍旧用我们之前使用过的表示游戏角色的类 `Role` 举例：

```
class Role {
    var name: String
    var action: () -> Void = { }

    init(_ name: String = "Foo") {
        self.name = name
        print("\(self) init")
    }

    deinit {
        print("\(self) deinit")
    }
}

extension Role: CustomStringConvertible {
    var description: String {
        return "<Role: \(name)>"
    }
}
```

它有两个属性：

- name 表示角色的名称；
- action 表示角色执行的动作；

为了方便观察 Role 的行为，我们还让它实现了 CustomStringConvertible ，相信这一切，你都已经很熟悉了。

接下来，我们人为制造一些会引起引用循环的场景。

通过外部closure捕获对象

当我们通过外部closure定制 Role.action 的时候，很容易不经意间就创建引用循环：

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

    boss.action = fn
}
```

如果我们执行上面这段代码：就会在控制台看到类似这样的结果：

可以看到， boss 初始化之后，并没有被回收。这是为什么呢？我们来看下面的示意图。在执行过 boss.action = fn 之后，它们的关系是这样的：

在这里，由于 class 和 closure 都是引用类型，因此 boss 和 fn 都是指向各自对象的strong reference。在 fn 里，我们捕获了 boss ，因此，closure对象就有了一个指向 boss 变量的strong reference。最后， boss.action = fn 让 action 也成了closure对象的strong reference。此时， Role 的引用计数是1，closure对象的引用计数是2。

接下来，当程序离开 if 作用域之后，它们的关系就变成了这样：

这时，只有closure对象的引用计数变成了1。于是，closure继续引用了 boss ， boss 继续引用了它的对象，而这个对象，继续引用着closure。最终，我们就创建了一个引用循环。

但是，这并不是招致引用循环的唯一方式，**当我们在类内部实现closure属性的时候，只要它访问了 `self`，就一定会发生引用循环。**

通过closure属性的内部实现捕获self

我们给 `Role.action` 的默认实现修改成这样：

```
class Role {
    // ...
    lazy var action: () -> Void = {
        print("\(self) takes action.")
    }
}
```

在这里，由于我们在 `action` 的实现中使用了 `self`，因此要把它定义成一个 `lazy` property，以保证它只能在 `Role` 正常初始化完成之后才可以使用。然后，把我们之前的测试代码改成这样：

```
if true {
    var boss = Role("boss")
    boss.action()
}
```

我们就能看到和下图类似的结果：

可以看到， `boss` 同样没有被正确回收。而导致引用循环的原理，和我们之前通过外部closure的例子是一样的，我们把 `self` 理解为前一个例子里的 `boss` 变量就好了。

实际上，这两个例子，通过不同的形式，表达了同一个事实：**只有当类对象拥有一个closure对象时，它们之间才有可能造成循环引用。**

了解了循环引用发生的原理之后，我们该怎么解决呢?

理解closure capture list

在了解解决方案之前，我们再来看下这节开始我们使用的例子，把捕获整数变量的代码改成这样：

```
var i = 10
var captureI = { [i] in print(i) }
i = 11

captureI() // 10
```

这次，我们在 `captureI` 的定义中使用了 `[i]`，这叫做closure的capture list，它的作用就是让closure按值语义捕获变量。因此，当我们执行 `captureI()` 时，打印的结果就变成了10，这是 `captureI` 在定义时变量 `i` 的值。

而对于之前捕获类对象的例子，当我们使用了capture list之后：

```
var c = C()
var captureC = { [c] in print(c.value) }
c.value = "updated"
c = C()

captureC() // updated
```

打印结果就会变成updated。这次， `captureC` 捕获的也不再是变量 `c` 的引用，而是变量 `c` 引用的对象本身。因此，当我们此时执行赋值语句 `c = C()` 的时候，赋值之前的 `C` 对象就会被 `captureC` 保持在内存里，而不会被回收。

What's next?

理解了capture list的含义之后，我们就可以用它来解决之前提到的closure和类对象之间的引用循环问题了。根据closure对象和 `class` 对象生命周期的不同，解决的方案也不同。在下一节，我们就来讨论具体的实现细节。

(https://www.boxueio.com/series/understand-ref-types/ebook/183) (https://www.boxueio.com/series/understand-ref-types/ebook/185)

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 ›

泊学用户隐私以及服务条款 (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

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) | 靛青K的博客 (http://blog.dianqk.org/)