理解引用语义的自定义类型

永远不要重定义继承而来的默认参数
(https://www.boxueio.com/series/understand-ref-types/ebook/180)

Reference cycle是如何发生的?
(https://www.boxueio.com/series/understand-ref-types/ebook/182)

继承和多态并不是解决问题的唯一方式

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

字号
字号
默认主题
金色主题
暗色主题

当我们需要在派生类体系中自定义基类的某些行为时，除了重写基类方法之外，还有很多其它的方式。它们往往比重写方法更灵活，表意也更丰富。在这一节中，我们就来了解其中的两种设计模式。

为了演示这些模式的实现，我们假设一个场景。假设我们正在开发一款游戏，其中的每个角色，都有自己的攻击力（当然，你也可以假设它们还有不同的生命值、攻击范围等等，但那并不是我们要讨论的重点）。于是，你很自然的想到了，要为所有的角色抽象出来一个公共的基类，并提供一个获取攻击值的方法：

```
class Role {
    func power() -> Int { return 0 }
}
```

然后，无论是NPC也好，玩家也好甚至是Boss，它们都可以从 Role 派生出来，在它们各自的派生类里，我们可以定义不同的攻击力。这看似是个再正常不过的想法了，以至于我们都不会给自己额外的时间思考是否存在其它的解决方案。事实上，在面向对象的世界里，的确还存在着一些更灵活有趣的方法。

Template method

第一个要介绍的方法，来源于这样一个假设：所有可以被重写的方法都应该只被类型自身使用，而对外的API都应该是不可被重写的方法。如果你还记得我们上一节中 Shape.draw 的实现，就会发现，它们的思路如出一辙。

首先，我们让定义在 Role 里的方法变成只允许类内部使用：

```
class Role {
    fileprivate func doPower() -> Int {
        return 0
    }
}
```

然后，我们通过 extension Role 添加一个可以公共访问，但是不能改写的方法：

```
extension Role {
    public func power() -> Int {
        // pre settings here
        let value = doPower()
        // post settings here
        return value
    }
}
```

这样，所有 Role 的派生类只需要做两件事情：

1. 从 Role 派生；
2. 重定义 doPower() 方法；

就可以定义它们自己的攻击力了。在这种模式里， extension 中的 power() 方法，为读取所有类型角色的攻击力提供了一份模板实现，因此这种模式被称作template method。它有一个额外的好处，就像注释中说明的那样，我们可以在读取攻击力之前和之后，做一些辅助或修正工作。

这种方式的一个问题在于，对于熟悉了面向对象编程的我们来说，要接受在派生类中改写的居然是不会被公众调用的私人方法这个观点，着实有些挑战，它似乎有悖于我们对面向对象的理解：

```
class Player: Role {
    fileprivate func doPower() -> Int {
        return 100
    }
}
```

但实际上，这种只重写私有方法的做法并没有那么诡异。我们在派生类中重写一个方法的目的只是为了表达“某些事情”如何完成（例如我们例子中的获取攻击力）。而调用一个被继承的方法，则是为了表达“某些事情”在何时完成。所以，是否在派生类中改写方法，与一个方法的访问权限，并不是有很大的关联。

基于函数的Strategy模式

除了让一个不可被重写的方法提供模板外，还有一种更“激进”的做法。为什么我们一定要让获取攻击力这个行为和角色类型相关呢？这个关系难道不是我们可以动态指定的么？于是，就有了下面这种用一个函数属性保存计算攻击力的方法。

首先，定义一个接受 Role 参数，并返回 Int 攻击力的方法：

```
func defaultPower(role: Role) -> Int {
    return 0
}
```

其次，我们把 Role 改成这样：

```
class Role {
    typealias PowerFn = (Role) -> Int

    var powerFn: PowerFn

    init(powerFn: @escaping PowerFn = defaultPower) {
        self.powerFn = powerFn
    }
}
```

此时， Role 就有了一个函数属性，它用于计算某类角色的攻击力。最后，我们把 Role 的模板方法改成这样：

```
extension Role {
    func power() -> Int {
        return powerFn(self)
    }
}
```

相比于之前的“模板”方案，基于函数属性的实现多了更多的灵活性，例如：

- 同一类型的对象现在也可以有不同的计算攻击力的方法：

```
let p1 = Player(powerFn: { _ in 100 })
let p2 = Player(powerFn: { _ in 200 })
```

- 我们还可以在运行时，动态修改某个对象计算攻击力的方法：

```
p1.powerFn = { _ in 50 }
```

- 甚至，我们还可以为攻击力的计算添加任意的附加条件。例如，我们添加一个表示游戏难度的 enum ，它带有一个根据游戏难度返回攻击力的方法：

```
enum Level {
    case simple, normal, hard

    func rolePower(role: Role) -> Int {
        switch self {
            case .simple:
                return 300
            case .normal:
                return 200
            case .hard:
                return 100
        }
    }
}
```

我们就可以这样来设置一个玩家的攻击力：

```
let level = Level.simple
p1.powerFn = Level.rolePower(level)
```

这里，我们要稍微解释一下 Level.rolePower(level) 的用法。首先， Level.rolePower 的类型是这样的： (Level)->(Role)->Int ，其中第一个参数表示调用 rolePower 的 Level 对象。但是，我们的 PowerFn 是一个 (Role)->Int 类型的函数，因此，我们要用

`Level.rolePower(level)` 这样的形式，绑定 `rolePower` 的第一个参数，然后，它就可以作为 `PowerFn` 使用了。

理解了基于函数实现的策略方式之后，在最后，我们来看一种更为传统的策略类的实现。思路很简单，让 `PowerFn` 的类型从函数变成一个可以计算攻击力的类。

基于class的strategy模式

```
class Power {
    func calc(role: Role) -> Int {
        return 100
    }
}
```

然后，让 `Role` 保存一个 `Power` 类型的属性：

```
class Role {
    var powerFn: Power

    init(powerFn: Power) {
        self.powerFn = powerFn
    }
}
```

最后，让 `Role.power` 的计算通过 `Power.calc` 完成：

```
extension Role {
    func power() -> Int {
        return powerFn.calc(role: self)
    }
}
```

这样，我们就可以通过给角色安插 `Power` 继承体系中的类对象，来调整计算攻击力的方法了。整体上的思路和函数版本是非常类似的，只不过这种方式更为传统，如果你熟悉设计模式，而不熟悉Swift，这种实现方式更容易识别。

What's next?

以上，就是我们这一节的内容。当然，对于重写继承方法的替代方案，远不止我们在这里提到这两种。但它们应该已经足够让你相信，面向对象的世界里，除了一味的 `override` ，还有很多毫不逊色的替代方案。而我们要做的，是不断比对这些方案的优劣，进而找到问题的最优解。

了解过面向对象世界里关于设计的主流话题之后，在接下来的几个小节中，我们将看到一些和内存管理有关的问题。在Swift里， `class` 作为一个引用类型，在composition关系中很容易造成引用循环，进而导致资源泄漏。因此，在实现上避免循环引用，是和设计上保持语义正确同等重要的事情。

而作为这个话题的开始，下一节，我们就先来看看，循环引用到底是如何发生的。

永远不要重定义继承而来的默认参数
(https://www.boxueio.com/series/understand-ref-types/ebook/180)

Reference cycle是如何发生的?
(https://www.boxueio.com/series/understand-ref-types/ebook/182)

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/)