理解引用语义的自定义类型

确定对象的组合用于模拟“has a”的关系

(https://www.boxueio.com/series/understand-ref-types/ebook/178)

继承和多态并不是解决问题的唯一方式

(https://www.boxueio.com/series/understand-ref-types/ebook/181)

永远不要重定义继承而来的默认参数

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

字号
字号
默认主题
金色主题
暗色主题

让我们开门见山，直奔主题，如果派生类中继承而来的方法带有默认参数，不要修改它，通常这不会带来你期望的结果。这是为什么呢？我们先来看个例子。

假设，我们有一个表示形状的基类：

```
class Shape {
    enum Color { case red, yellow, green }

    func draw(color: Color = .red) {
        print("A \(color) shape.")
    }
}
```

它的 draw 方法带有一个有默认值 .red 的参数。为了示意，我们向控制台打印了带有形状颜色的消息。接下来，我们再定义两个 Shape 的派生类，分别表示方形和圆形：

```
class Square: Shape {
    override func draw(color: Color = .yellow) {
        print("A \(color) square.")
    }

}

class Circle: Shape {
    override func draw(color: Color = .green) {
        print("A \(color) circle.")
    }
}
```

在它们各自的实现里，我们给 draw(:Color) 方法指定了不同的默认颜色。这真的可行么？我们用下面的代码来试一下：

```
let s = Square()
let c = Circle()

s.draw() // A yellow square
c.draw() // a green circle
```

可以看到，当 s 和 c 分别是 Square 和 Circle 对象时，修改默认参数可以很好的工作。但通常，我们会利用多态来动态选择调用的方法，像这样：

```
let s: Shape = Square()
let c: Shape = Circle()

s.draw() // A red square.
c.draw() // A red circle.
```

看到注释中的结果了么？我们的确根据不同的对象调用了各自 draw() 方法的实现，但是这些方法的默认参数选择，却统统用了 Shape 的版本。这种混搭的结果一定不是你想要的。

之所以会有这样的结果，是因为在Swift里，继承而来的方法调用是在运行时动态派发的，Swift会在运行时动态选择一个对象真正要调用的方法。但是，方法的参数，出于性能的考虑，却是静态绑定的，编译器会根据调用方法的对象的类型，绑定函数的参数。于是，就造成了之前派生类方法的实现，基类方法的默认参数这样的结果。所以，直接修改继承得来方法的默认参数，并不是个好主意。

但是，如果在派生类的实现中不定义默认参数：

```
class Square: Shape {
    override func draw(color: Color) {
        print("A \(color) square.")
    }
}
```

我们就不能使用type inference创建一个 square 对象时，绘制成默认的颜色了：

```
let s = Square()
s.draw() // Compile time error
```

这显然有悖于在 Shape 中定义 draw 方法的初衷。到底该怎么办呢?

一个笨方法就是我们在 Square.draw 中指定和 Shape.draw 同样的默认值参数。这样，就可以既能默认绘制 Square 对象，又掩盖了实际上选择的是 Shape.draw 方法默认参数的事实：

```
class Square: Shape {
    override func draw(color: Color = .red) {
        print("A \(color) square.")
    }
}
```

但这样真的好么？稍微多往前想一步，你就会否定这个方案。首先，如果你有上百种形状要创建，就要给每一个类中的 draw 方法指定相同的默认参数；其次，如果有一天基类的 draw 方法默认颜色改了，你又将重蹈参数选择错误的覆辙。因此，这并不是一个好方法。

为了能在重定义继承方法的同时，又继承到基类的默认参数，我们还有其它的出路么？如果你知道定义在 extension 中的方法，是不能被重定义的，就看到了一丝曙光。我们可以把绘画的过程抽象在一个 extension 方法里，供外部统一调用，然后把真正的绘制过程定义成一个可以重定义的方法。像这样：

```
class Shape {
    enum Color { case red, yellow, green }

    func doDraw(of color: Color) {
        print("A \(color) shape.")
    }
}

extension Shape {
    func draw(color: Color = .red) {
        doDraw(of: color)
    }
}
```

在上面的代码里，由于 draw(:Color) 定义在 extension 里，它不可以被派生类重写。但我们可以重定义没有默认参数的 doDraw(:Color) 方法：

```
class Square: Shape {
    override func doDraw(of color: Color) {
        print("A \(color) square.")
    }

}

class Circle: Shape {
    override func doDraw(of color: Color) {
        print("A \(color) circle.")
    }
}
```

这样，我们就变相实现了在派生类中重写方法的同时，还保留了基类API默认参数的效果。

```
let s = Square()
let c = Circle()

s.draw() // A red square.
c.draw() // A red circle.
```

What's next?

如你所见，在继承关系中，当默认参数和需要重写的方法发生冲突时，更好的做法，是我们在最后演示的这样，用一个无法被重写的方法在前面充当API。当然，我们更想强调的，其实是永远也不要改写继承而来的方法的默认参数，因为它执行的是静态绑定的语义。

另外，在这一节最后的例子里，我们也暴露了这样一个事实：多态并不是我们完成任务的唯一手段。面对需要解决的多种不同的问题，为了避免让自己陷入面向对象造就的常规设计思维的泥潭，有时，我们需要对着“轮子”猛推一把，在面向对象的世界里，还有诸多道路，值得我们去探索和研究。

⏮ 确定对象的组合用于模拟“has a”的关系
(https://www.boxueio.com/series/understand-ref-types/ebook/178)

继承和多态并不是解决问题的唯一方式 ⏭
(https://www.boxueio.com/series/understand-ref-types/ebook/181)

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/)