理解引用语义的自定义类型

什么是two-phase initialization
(https://www.boxueio.com/series/understand-ref-types/ebook/176)

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)

确定继承关系用于模拟“is a”的关系

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

字号
字号
默认主题
金色主题
暗色主题

在面向对象的世界里，一条最重要的行为准则，就是确保你的继承关系表达derived is a base这样的关系。遵从这条准则，你就可以写出语义自然，简单易懂的面向对象代码。这看似简单，但有时候，我们会被自己的直觉欺骗，不知不觉陷入逻辑的泥潭。在这一节中，我们就来看一些具体的例子。

理解类继承的语义

如果两个类存在继承关系，我们相当于告诉编译器，派生类对象一定是一个基类对象，只不过，派生类有更多具像化的属性。可以接受一个基类对象的地方，一定可以接受一个派生类，但反之则不然。例如：

```
class Person {
    var name: String

    init(name: String) {
        self.name = name
    }
}

class Employee: Person {
    var staffNumber: Int

    init(name: String, staffNumber: Int) {
        self.staffNumber = staffNumber

        super.init(name: name)
    }
}
```

在这个例子里，一个 Employee 对象一定是一个 Person 对象，但反之则不一定。也就是说， Person 表达的概念比 Employee 更泛化，而 Employee 表达的概念比 Person 更具象。

如果函数有一个 Person 类型的参数，我们一定可以给它传递一个 Employee 对象：

```
func printName(of person: Person) {
    print(person.name)
}

let mars = Person(name: "Mars")
let jeff = Employee(name: "Jeff", staffNumber: 23)

printName(of: mars) // Mars
printName(of: jeff) // Jeff
```

在这个例子里，尽管 printName 接受一个 Person 参数，但就像你看到的，我们可以给它传递 mars 和 jeff 都是可以的。但反过来，就会很直观从语义上发现错误：

```
func printNumber(of employee: Employee) {
    print(employee.staffNumber)
}

printNumber(of: jeff) // 23
printNumber(of: mars) // compile time error
```

我们无法确保所有的 Person 都是 Employee ，因此打印 mars 的工号，从道理上，就是无法理解的。这看似很简单对不对？但很多时候，当我们把直觉里正确的事情用在面向对象设计里，就会不知不觉陷入逻辑的陷阱。

不信？我们来看下面这两个例子。

自然语言的不确定性带来的困惑

假设，我们在开发一个动物百科类的App。当设计鸟类数据模型的时候，我们可能会不假思索的写下这样的基类：

```
class Bird {
    func fly() {}
}
```

然后，我们就着手开始为各种不同类目的鸟创建派生类了。但是，呃……，当我们处理到企鹅的时候（没错，企鹅也是一种鸟类），你发现下面这段代码的问题么？

```
class Penguin: Bird {
    // ...
}
```

如果你还没转过神来，来看下面这段代码：

```
let qq = Penguin()
qq.fly() // Ah?...
```

我们都知道，企鹅是飞不起来的。但是，在我们刚刚设计的继承关系里，这样的语法却完全没问题。到底哪个环节出了问题呢？

问题的本质在于，我们日常的自然语言是模糊的。当我们表达鸟可以的飞的时候，表达的是绝大多数鸟的行为以及我们对这一类生物的印象。但计算机语言是精确的，所有从 `Bird` 派生的鸟都可以 `fly` ，这里没有意外。因此，实际上，在编程语言里，我们需要的，是这样的派生关系：

```
class Bird {

}

class FlyingBird: Bird {
    func fly() {}
}

class Penguin: Bird {

}
```

这样看上去就好多了。但是，你可能还会想，难道我不能在 `Penguin` 中重载 `fly()` 让它直接返回一个运行时错误么？

```
class Penguin: Bird {
    override func fly() {
        fatalError("Penguin cannot fly.")
    }
}
```

这样，就不用再多引入一个 `FlyingBird` 类了。从语法上说，这样当然没问题。但我们一直以来强调的，都是基于语义的正确性带来的良好设计。从这个角度上来说，这样就和我们之前的设计截然不同了：

- `FlyingBird` 的引入，要表达的概念是企鹅不会飞。而这个概念最终会得到编译器的支持。`qq.fly()` 这样的代码根本无法通过编译。
- 如果直接让 `Penguin.fly` 返回运行时错误，要表达的概念则是企鹅可以飞，但那样做的后果很严重。并且，这个严重的后果只能在让企鹅飞起来的时候才能发现。

现在，你能感受到它们在表现力上的差别了么？一直以来我们都在说好的API应该更容易用对，而更不容易用错。即便是站在这个角度上考虑，引入 `FlyingBird` 也要比重载 `fly` 的方案好的多。

至此，虽然逻辑问题解决了，但有个问题仍旧值得思考：我们真的一定要如此么？其实也不尽然。如果我们要创建的类型系统里，不会涉及到鸟会飞这个动作（例如：只是研究鸟的解剖结构），那么我们一开始的方案就完全没问题，甚至更合理。因此，这也说明了一个事实，在软件工程领域里，并没有从一而终的完美解决方案，你只能根据你要解决问题的集合，找到最适合的设计。

当然，如果你对这些“鸟”问题并不感兴趣，我们可以再来看一个更容易让人迷惑的问题：在面向对象的世界里，一个正方形是一个矩形么？这次，你应该毫不犹豫的说“是”了吧。于是，我们就可以这样来描述正方形和矩形的关系：

```
class Rectangle {
    var w: Double
    var h: Double

    init(w: Double, h: Double) {
        self.w = w
        self.h = h
    }
}

class Square: Rectangle {
    init(edge: Double) {
        super.init(w: edge, h: edge)
    }
}
```

这看似很正常，对么？但是，当我们添加一个增加矩形宽度的方法时，就会让你感到有些意外：

```
func scaleWidth(of rect: Rectangle) {
    let oldHeight = rect.h
    rect.w *= 1.1

    assert(oldHeight == rect.h)
}
```

在 `scaleWidth(:Rectangle)` 的实现里，我们在最后添加一个了确保矩形高度不会被修改的断言。然后，看下下面的代码，它是你想要的么？

```
var s11 = Square(edge: 11)
scaleWidth(of: s11)
```

虽然编译器可以对上面的代码放行，但它却执行了完全错误的语义。对于一个正方形来说，扩展宽度的同时应该扩展高度，因此 `scaleWidth(:Rectangle)` 的实现，是完全不能接受一个 `Square` 参数的。而造成这个问题的原因，和之前 `Penguin` 的例子是完全一样的。在我们要解决的问题集合里，一些适用于 `Rectangle` 的方法，并不适用于 `Square` ，但是，一旦我们让它们之间存在了继承关系，却是意味着**所有**适用于 `Rectangle` 的方法，都**一定**适用于 `Square` 。

怎么样？现在你是不是感觉很神奇，原来直觉上毫无毛病的逻辑，放在面向对象的世界里，居然漏洞百出。

What's the next?

当然，你也不用对此过于谨慎，大部分时候，直觉和常识都是没错的。只不过，面向对象设计里最大的挑战，就是我们要在直觉中引入新的洞察力，去预见两个类的继承关系，是否可以在要解决的问题集合里，完整表达**派生类是一个基类**这样的事实。但凡有一点差池，我们就应该调整设计，以最终达到我们的设计意图。

并且，在面向对象的世界里，继承并不是类之间关系的全部。除了 `IS A` 之外，另一类常见的关系叫做 `HAS A` ，它用于表达一个对象是由其它对象构成的。就像你已经看到的一样，这看似简单，但在OOP的世界里，总有一些藏匿于直觉之外的东西。

◀ 什么是two-phase initialization
(https://www.boxueio.com/series/understand-ref-types/ebook/176)

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)

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/)