跳出面向对象思想(二) 多态 (http://casatwy.com/tiao-chu-mian-xiang-dui-xiang-si-xiang-er-duo-tai.html)

Date Tue 16 December 2014 Tags Object Oriented Programming (http://casatwy.com/tag/object-oriented-programming.html) / Experience (http://casatwy.com/tag/experience.html) / jooo (http://casatwy.com/tag/jooo.html)

简述

多态一般都要跟继承结合起来说，其本质是子类通过覆盖或重载（在下文里我会多次用到 覆盖或重载 ，我打算把它简化成 覆重 ，意思到就好，不要太纠结这种名词。）父类的方法，来使得对**同一类对象同一方法**的调用产生不同的结果。这里需要辨析的地方在：同一类对象指的是继承层级再上一层的对象，更加泛化。

举个例子：

```
Animal -> Cat
Animal -> Dog

Animal.speak()  // I'm an Animal
Cat.speak()     // I'm a Cat
Dog.speak()     // I'm a Dog
```

此处 Cat 和 Dog 虽然不是同一种对象，但它们算是同一类对象，因为他们的父类都是 Animal 。 种 和 类 的表达可能不是很对，其实我也不知道谁更大一点，在文章中我打算用这样的符号来表示两者区别： ^ 和 ^^

```
^ 表示他们是同一类
^^ 表示他们同种同类

Animal -> Cat
Animal -> Dog

Cat kitty, kate
Dog lucky, lucy

我们可以这么说：

    kitty ^^ kate       同种同类，他们都是猫
    kitty ^ lucy        同类不同种，他们都是Animal
    kitty !^^ lucy      因为kitty是猫，lucy是狗
    kitty ^ kate        他们当然同种啦，都是Animal
```

应该算是能够描述清楚了吧？嗯，我们开始了。

多态

一般来说我们采用多态的场景还是很多的，有些在设计的时候就是用于继承的父类，希望子类覆盖自己的某些方法，然后才能够使程序正常运行下去。比如：

```
BaseController需要它的子类去覆盖loadView等方法来执行view的显示逻辑
BaseApiManager需要它的子类去覆盖methodName等方法来执行具体的API请求
```

以上是我列举的应用多态的几个场景，在基于上面提到的需求，以及站在代码观感的立场，我们在实际采用多态的时候会有下面四种情况：

1. 父类有部分public的方法是不需要，也不允许子类覆重
2. 父类有一些特别的方法是必须要子类去覆重的，在父类的方法其实是个空方法
3. 父类有一些方法是可选覆重的，一旦覆重，则以子类为准
4. 父类有一些方法即便被覆重，父类原方法还是要执行的

这四种情况在大多数支持多态的语言里面都没有做很好的原生限制，在程序规模逐渐变大的时候，会给维护代码的程序员带来各种各样的坑。

父类有部分public的方法是不需要，也不允许子类覆重

对于客户程序员来说，他们是有动机去覆重那些不需要覆重的方法的，比如需要在某个方法调用的时候做UserTrack，或者希望在方法调用之前做一些额外的事情，但是又找不到外面应该在哪儿做，于是就索性覆重一个了。这样做的缺点在于 使得一个对象引入了原本不属于它的业务逻辑 。如果在引入的这些额外逻辑中又对其他模块产生依赖，那么这个对象在将来的代码复用中就会面临一个艰难的选择：

- 是把这些不必要的逻辑删干净然后移过去？
- 还是所以把依赖连带着这个对象一起copy过去？

前者太累，后者太蠢。

如果是要针对原来的对象进行功能拓展，但拓展的时候发现是需要针对 原本不允许覆重 的函数进行操作，那么这时候就有理由怀疑父类当初是不是没有设计好了。

> 父类有一些特别的方法是必须要子类去覆重的，在父类的方法其实是个空方法

这非常常见，由于逻辑的主要代码在父类中，若要跑完整个逻辑，则需要调用一些特定的方法来基于不同的子类获得不同的数据，这个特定的方法最终交由子类通过覆重来实现。如果不在父类里面写好这个方法吧，父类中的代码在执行逻辑的时候就调用不到。如果写了吧，一个空函数放在那儿十分难看。

也有的时候客户程序员会不知道在派生之后需要覆重某个方法才能完成完整逻辑，因为空函数在那儿不会导致warning或error，只有在发现程序运行结果不对的时候，才会感觉哪儿有错。如果这时候程序员发现原来是有个方法没覆重，一定会拍桌子骂娘。

总结一下，其实就是代码不好看，以及有可能忘记覆重。

> 父类有一些方法是可选覆重的，一旦覆重，则以子类为准

这是大多数面向对象语言默认的行为。设计可选覆重的动机其中有一个就是可能要做拦截器，在每个父类方法调用时，先调一个willDoSomething()，然后调用完了再调一个didFinishedSomething()，由子类根据具体情况进行覆重。

一般来说这类情况如果正常应用的话，不会有什么问题，就算有问题，也是前面提到的 容易使得一个对象引入原本不属于它的业务逻辑 。

> 父类有一些方法即便被覆重，父类原方法还是要执行的

这个是经典的坑，尤其是交付给客户程序员的时候是以链接库的模式交付的。父类的方法是放在覆重函数的第一句调用呢还是放在最后一句调用？这是个值得深思的问题。更有甚者索性就直接忘记调用了，各种傻傻分不清楚。

解决方案

面向接口编程（Interface Oriented Programming, IOP）是解决这类问题比较好的一种思路。下面我给大家看看应该如何使用IOP来解决上面四种情况的困境:

(示例里面有些表达的约定，可以在这里 (http://casatwy.com/tiao-chu-mian-xiang-dui-xiang-si-xiang-yi-ji-cheng.html)看完整的上下文规范。)

```
<ManagerInterface> : APIName()                  我们先定义一个ManagerIn
<Interceptor> : willRun(), didRun()          我们再定义一个Interceptor


BaseManager.child<ManagerInterface>          在BaseController里面添加


BaseManager.init() {

    ...

    self.child = self                        在init的时候把child设置成

    # 如果语言支持反射，那么我们可以这么写：
    if self.child implemented <ManagerInterface> {
        self.child = self
    }
    # 如上的写法就能够保证我们的子类能够基于这些接口有对应的实现

    self.interceptor = self                  # interceptor可以是自己,

    ...

}


BaseManager.run() {

    self.interceptor.willRun()

    ...

    apiName = self.child.APIName()           # 原本是self.APIName(),
    request with apiName

    ...

    self.interceptor.didRun()

}
```

通过引入这样面向接口编程的做法，就能相对好地解决上面提到的困境，下面我来解释一下是如何解决困境的：

- 父类有部分public的方法是不需要，也不允许子类覆重

由于子类必须要遵从 <ManagerInterface> ，架构师可以跟客户程序员约定 所有的public方法在一般情况下都是不需要覆重的 。除非特殊需要，则可以覆重，其他情况都通过实现接口中定义的方法解决。由于这是接口方法，所以 即便引入了原本不需要的逻辑，也能很容易将其剥离 。

- 父类有一些特别的方法是必须要子类去覆重的，在父类的方法其实是个空方法

因为引入了 child，父类不再需要摆一个空方法在那儿了，直接从 child 调用即可，因为 child 是实现了对应接口的，所以可以放心调用。空方法就消灭了。

- 父类有一些方法是可选覆重的，一旦覆重，则以子类为准

我们可以通过在接口中设置哪些方法是必须要实现，哪些方法是可选实现的来处理对应的问题。这本身倒不是缺陷，正是多态希望的样子。

- 父类有一些方法即便被覆重，父类原方法还是要执行的

由于我们通过接口规避了多态，那么这些其实是可以通过在接口中定义 可选方法 来实现的，由父类方法调用 child 的 可选方法，调用时机就可以由父类决定。这两个方法不必重名，因此也不存在多态时，不能分辨调用时机或是否需要调用父类方法的情况。

总结一下，通过IOP，我们做好了两件事：

1. 将子类与可能被子类引入的不相关逻辑剥离开来，提高了子类的可重用性，降低了迁移时可能的耦合。
2. 接口实际上是子类头上的金箍，规范了子类哪些必须实现，哪些可选实现。那些不在接口定义的方法列表里的父类方法，事实上就是不建议覆重的方法。

什么时候用多态

由于多态和继承紧密地结合在了一起，我们假设父类是架构师去设计，子类由客户程序员去实现，那么这个问题实际上是这样的两个问题：

1. 作为架构师，我何时要为多态提供接入点?
2. 作为客户程序员，我何时要去覆重父类方法?

这本质上需要程序员针对对象建立一个 角色 的概念。

举个例子：当一个对象的主要业务功能是搜索，那么它在整个程序里面扮演的角色是搜索者的角色。在基于搜索派生出的业务中，会做一些跟搜索无关的事情，比如搜索后进行人工加权重排列表，搜索前进行关键词分词（假设分词方案根据不同的派生类而不同）。那么这时候如果采用多态的方案，就是由子类覆重父类关于重排列表的方法，覆重分词方法。如果在编写子类的程序员忘记这些必要的覆重或者覆重了不应该覆重的方法，就会进入上面提到的四个困境。所以这时候需要提供一套接口，规范子类去做覆重，从而避免之前提到的四种困境：

```
Search : { search(), split(), resort()}

采用多态的方案:
Search -> ClothSearch : { [ Search ], @split(), @resort() }

function search() {

    ...

    self.split()    # 如果子类没有覆重这个方法而父类提供的只是空方法，这里就

    ...

    self.resort()

    ...

}


采用IOP的方案:
<SearchManager> : {split(), resort()}
Search<SearchManager> : { search(), assistant<SearchManager> }

function search() {

    ...

    self.assistant.split()  # self.assistant可以就是self，也可以由初始

    ...

    self.assistant.resort()

    ...

}

Search -> ClothSearch<SearchManager> : { [ Search ], split(), resor

外面使用对象时：ClothSearch.search()
```

如果示例中不同的子类对于search()方法有不同的实现，那么这个时候就适用多态。

```
Search : { search() }

ClothSearch : { [Search], @search() }

此时适用多态，外面使用对象时：ClothSearch.search()
```

总结是否决定应当使用多态的两个要素：

- 如果引入多态之后导致对象角色不够单纯，那就不应当引入多态，如果引入多态之后依旧是单纯角色，那就可以引入多态

- 如果要覆重的方法是角色业务的其中一个组成部分，例如split()和resort()，那么就最好不要用多态的方案，用IOP，因为在外界调用的时候其实并不需要通过多态来满足定制化的需求。

其实这是一个 角色 问题，越单纯的角色就越容易维护。还有一个就是区分被覆重的方法是否需要被外界调用的问题。好了，现在我们回到这一节前面提出的两个问题：何时引入接入点和何时采用覆重。针对第一个问题架构师一定要分清楚 角色 ，在保证 角色 单纯的情况下可以引入多态。另外一点要考虑 被覆重的方法是否需要被外界使用 ，还是只是父类运行时需要子类通过覆重提供中间数据的。如果是 只要子类通过覆重提供中间数据的，一律应当采用IOP而不是多态 。

针对第二个问题，在必须要覆重的场合下就采取覆重的方案好了，主要是可覆重可不覆重的情况下，客户程序员主要还是要遵守：

- 覆重的方法本身是跟逻辑密切相关的，不要在覆重方法里做跟这个方法本意不相关的事情
- 如果要覆重一系列的方法，那么就要考虑角色问题和外界是否需要调用的问题，这些方法是不是这个对象的角色应当承担的任务

比如说不要在一个原本要跑步的函数里面去做吃饭的事情，如果真的要吃饭，父类又没有，实在不行的时候， 就需要在覆重的方法里面启用IOP，在子类里面弥补架构师的设计缺陷 。把这个不属于跑步的事情IOP出去，负责实现对应接口的可以是self，也可以是别人。只要不是强耦合地去覆重，这样在代码迁移的时候，由于IOP的存在，使得代码接收方也可以接受并实现对应的interface，从而不影响整体功能，又能提供迁移的灵活性。

总结

多态在面向对象程序中的应用相当广泛，只要有继承的地方，或多或少都会用到多态。然而多态比起继承来，更容易被不明不白地使用，一切看起来都那么顺其自然。在客户程序员这边，一般是只要多态是可行方案的一种，到最后大部分都会采用多态的方案来解决问题。

然而多态正如它名字中所暗示的，它有非常大的潜在可能引入不属于对象初衷的逻辑，巨大的灵活性也导致客户程序员在面对问题的时候不太愿意采用其他相对更优的方案，比如IOP。在决定是否采用多态时，我们要有一个清晰的 角色 概念，做好角色细分，不要角色混乱。该是拦截器的，就给他制定一个拦截器接口，由另一个对象（逻辑上的另一个对象，当然也可以是自己）去实现接口里的方法集。不要让一个对象在逻辑上既是拦截器又是业务模块。这样才方便未来的维护。另外也要注意被覆重方法的作用，如果只是单纯为了提供父类所需要的中间数据的， 一律都用IOP ，这是比直接采用多态更优的方案。

IOP能够带来的好处当然不止文中写到的这些，它在其他场合也有非常好的应用，它最主要的好处就在于分离了定义和实现，并且能够带来更高的灵活性，灵活到既可以对语言过高的自由度有一个限制，也可以灵活到允许同一接口的不同实现能够合理地组合。

在架构设计方面是个非常重要的思想。

评论系统我用的是Disqus，不定期被墙。所以如果你看到文章下面没有加载出评论列表，翻个墙就有了。

我的博客拒绝挂任何广告，如果您觉得文章有价值，可以通过支付宝扫描下面的二维码捐助我。

Comments

39 条评论 Casa Taloyum 登录

Recommend 3 分享 按从新到旧排序

Join the discussion...

sire · 2个月前
接口对象，自身实现写法存在循环强引用问题
· 回复 · 分享 ›

CasaTaloyum 管理员 sire · 2个月前
你要么就是没看懂文章，要么就是不懂什么叫循环强引用。
· 回复 · 分享 ›

Horex Chen · 2个月前
大神，有个问题。我想问下。就是这里多态的解决方式确实是很巧妙的。但是，和我心中学习到的设计模式中，有一条不符合。因为我认为，应该是具体依赖抽象。子类依赖父类才对。但是这里父类拥有了子类。然后调用子类的方法，虽然只是一个属性，耦合也算不上吧，但是我总感觉有点别捏。或者我还不太理解吧。请大神指点下哈~
· 回复 · 分享 ›

CasaTaloyum 管理员 Horex Chen · 2个月前
父类和子类直观上的感觉是依赖关系，但在多态语境下，他们本质上还是同一个对象，父和子之间的关系其实应该要具备约束关系。

父给子下定义，是目前大多数语言在面向对象的实现上的缺陷。就类似凡是我的儿子，那就都能花我的钱（继承），但同时并没有做到必须都姓我的姓（本文的方案）。

所以文章中的父子关系强调的并不是依赖关系，而是约束关系，父子之间的约束关系在目前大部分面向对象语言中都是不具备的，然而父对子有约束是很正常的场景。有了这层约束，就能规避多态的缺陷，从而设计出更优良的架构。
· 回复 · 分享 ›

Horex Chen CasaTaloyum · 2个月前
有点明白了。从依赖到约束，这里的协议是为了约束子类。我之前从依赖层面上来理解。错了哈，感谢~PS：这个比喻很贴切O(∩_∩)O哈哈~
· 回复 · 分享 ›

peixinchen · 4个月前
这种“技巧”其实也是通过运行时异常来保证子类必须实现特定的方法的，那这样的话，和直接基类实现空方法，或者直接在空方法中throw NotImplementation想比较有什么区别呢?

还有就是“父类有一些方法即便被覆重，父类原方法还是要执行的”，这一点，没有明白是怎么实现的，self和self.child都已经是子类的instance了，直接[self method]和[self.child method]有什么区别么？都是调用的子类的method方法啊，如果method不希望被覆盖，那这么做是不起作

↑ Back to top