**菜鸟版JAVA设计模式—适配器模式，装饰模式，代理模式异同**

原创 2014年10月14日 10:55:01

* 标签：
* [JAVA](http://so.csdn.net/so/search/s.do?q=JAVA&t=blog) /
* [设计模式](http://so.csdn.net/so/search/s.do?q=%E8%AE%BE%E8%AE%A1%E6%A8%A1%E5%BC%8F&t=blog) /
* [适配器模式](http://so.csdn.net/so/search/s.do?q=%E9%80%82%E9%85%8D%E5%99%A8%E6%A8%A1%E5%BC%8F&t=blog) /
* [装饰模式](http://so.csdn.net/so/search/s.do?q=%E8%A3%85%E9%A5%B0%E6%A8%A1%E5%BC%8F&t=blog) /
* [代理模式](http://so.csdn.net/so/search/s.do?q=%E4%BB%A3%E7%90%86%E6%A8%A1%E5%BC%8F&t=blog)
* 10576

节前把3个设计模式的学习心得分享了一下，分别是适配器模式，装饰模式，代理模式。

但是似乎越写越有些迷糊，这三种模式都有着一些大同小异的地方，总是有些搞不清。

今天再重新学习下，把学习心得分享给大家一下。这次是结合的我工作中的实际项目。

设计模式，其实是一种解决方法，JAVA作为一种面向对象的语言，对于所有的设计模式在实现上，总是少不了对接口的实现，对抽象类的继承，有时候似乎是为了去实现接口而去写接口，所以在这三种模式的说明代码中，我们都定义了接口这些东西，所以才有了现在的混淆。

先不厌其烦的介绍下这三种的设计模式的概念吧。

适配器模式，一个适配允许通常因为接口不兼容而不能在一起工作的类工作在一起，做法是将类自己的接口包裹在一个已存在的类中。

装饰器模式，原有的不能满足现有的需求，对原有的进行增强。

代理模式，同一个类而去调用另一个类的方法，不对这个方法进行直接操作。

适配器的特点在于兼容，从代码上的特点来说，适配类与原有的类具有相同的接口，并且持有新的目标对象。

就如同一个三孔转2孔的适配器一样，他有三孔的插头，可以插到三孔插座里，又有两孔的插座可以被2孔插头插入。

适配器模式是在于对原有3孔的改造。

在使用适配器模式的时候，我们必须同时持有原对象，适配对象，目标对象。。。。

装饰器模式特点在于增强，他的特点是被装饰类和所有的装饰类必须实现同一个接口，而且必须持有被装饰的对象，可以无限装饰。

代理模式的特点在于隔离，隔离调用类和被调用类的关系，通过一个代理类去调用。

总的来说就是如下三句话：

 1 适配器模式是将一个类(a)通过某种方式转换成另一个类(b).  
 2 装饰模式是在一个原有类(a)的基础之上增加了某些新的功能变成另一个类(b).  
 3 代理模式是将一个类(a)转换成具体的操作类(b).

又是罗里吧嗦一大段话，然后我们开心的来到了举例时间，这次举的是工作中实际的例子。

先简单介绍一下，公司有一个ORDER系统，专门用于提供订单管理的接口，提供给O2O商城，WAP商城，手机类APP，微信等客户端调用。

ORDER系统在上线之时，已经包含了非常完整的数据操作。

现手机类APP需要升级，老的接口可能不能满足需求，如原O2O订单提货延长有效期接口需要提供截至时间和订单号，新APP可能不提供截至时间，只需要 提供订单号即可延长时间。

而老的接口又不能修改，因为老的order接口是针对所有平台的，那么必须将新接口要求与原接口进行适配。。。

解决方案如下：新增AppAdepter类，用来适配老的接口，同时开发给新的接口。。。。

先给出原接口和实现类

原接口：

**[java]** [view plain](http://blog.csdn.net/lulei9876/article/details/39994825) [copy](http://blog.csdn.net/lulei9876/article/details/39994825)

1. /\*\*
2. \* 原接口，需要传入orderId，时间
3. \*
4. \*/
5. **public** **interface** SourceOrderApi {
6. **public** **void** updateDate(String orderId,String date,String client);
7. }

实现类：

**[java]** [view plain](http://blog.csdn.net/lulei9876/article/details/39994825) [copy](http://blog.csdn.net/lulei9876/article/details/39994825)

1. **public** **class** SourceOrderApiImpl **implements** SourceOrderApi{

4. @Override
5. **public** **void** updateDate(String orderId, String date, String client) {
6. System.out.println(client+"已将订单"+orderId+"的有效期延长至"+date);
7. }
8. }

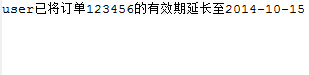
在这套代码中，对于客户端来说需要传入orderId  和  有效期date 和客户端名字才能延长。。。

Main方法：

**[java]** [view plain](http://blog.csdn.net/lulei9876/article/details/39994825) [copy](http://blog.csdn.net/lulei9876/article/details/39994825)

1. **public** **class** Test {
3. **public** **static** **void** main(String[] args) {
4. SourceOrderApi sourceOrderApi = **new** SourceOrderApiImpl();
5. sourceOrderApi.updateDate("123456", "2014-10-15", "user");
6. }
8. }

运行结果：



而对于新的接口，客户端则不必去传入date 参数

**[java]** [view plain](http://blog.csdn.net/lulei9876/article/details/39994825) [copy](http://blog.csdn.net/lulei9876/article/details/39994825)

1. **public** **interface** AppOrderApi {
2. //只需要传入订单Id即可
3. **public** **void** updateDate(String orderId,String client);
4. }

新的实现类：

**[java]** [view plain](http://blog.csdn.net/lulei9876/article/details/39994825) [copy](http://blog.csdn.net/lulei9876/article/details/39994825)

1. **public** **class** AppOrderApiImpl **implements** AppOrderApi{
3. SourceOrderApi sourceOrderApi;
5. **public** AppOrderApiImpl(){
6. sourceOrderApi = **new** SourceOrderApiImpl();
7. }
9. @Override
10. **public** **void** updateDate(String orderId,String client) {
11. //这里适配的方式随意，但是保证是要完全兼容原有的，就是保证调用原有的接口
12. sourceOrderApi.updateDate(orderId, "9999-12-31",client);
13. }
15. }

main方法：

**[java]** [view plain](http://blog.csdn.net/lulei9876/article/details/39994825) [copy](http://blog.csdn.net/lulei9876/article/details/39994825)

1. **public** **class** Test {
3. **public** **static** **void** main(String[] args) {
4. AppOrderApi appOrderApi = **new** AppOrderApiImpl();
5. appOrderApi.updateDate("123456", "user");
6. }
8. }

运行结果：

http://img.blog.csdn.net/20141014103038985?watermark/2/text/aHR0cDovL2Jsb2cuY3Nkbi5uZXQvbHVsZWk5ODc2/font/5a6L5L2T/fontsize/400/fill/I0JBQkFCMA==/dissolve/70/gravity/Center

在这套代码中，新的实现类持有了老接口的对象，就是把这个对象new 出来。。。

然后在新的方法里，进行适配操作。

而这里所谓的适配就是兼容老接口和兼容新接口。

兼容老接口非常简单，就是直接调用老的方法即可。

而兼容新接口就是你所要去做的业务逻辑。。。

比如我这里做了比较简单的操作，app类的统一传入“9999-12-31”，这些都是根据具体的项目需求去实现你的业务代码。

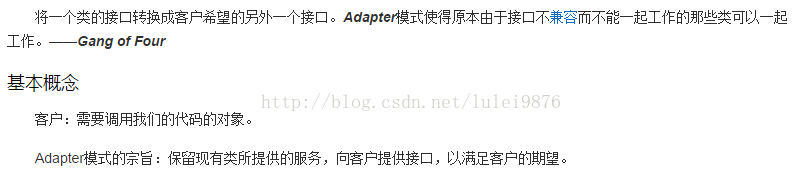
看到这，很多人会觉得 这样的代码是不是很常见。。。。

我自己都特意去翻了一下我之前的代码，很多类似的写法。。。

其实设计模式很多时候都融入到我们的代码中，只是我们没有去理解这种模式，所以就算你自己写出来了，都不知道这是一种适配器模式。。。

设计模式是一种思想，他没有固定的实现代码，我大量查阅了很多大神的的适配器模式解释，大家都不约而同的提到了类的适配和对象的适配，代码中有实现继承这样的写法。当然了，这些大神的写法都是比较标准的。。。

但是仔细再回看下GOF的适配器模式解释。



只需要将原接口转化为客户希望的另一个接口，就是适配器模式！

转化无非就是1.继承原类或者实现原接口  2.持有原接口的对象

再实现目标接口。。。。

那么第一种就是类的适配，第二种就是对象的适配！仅此而已。。。。

那么再回头来说代理模式，在代码上，和适配器模式有着相似的地方！

再比如，现在我们公司的系统又要进行升级，我们原先的接口没有加入安全机制，导致了任何人都可以随意调用这个接口，现在公司需要对这个接口进行改造，只其只能被admin这个客户端调用，其他用户一律要输入账号密码才能调用

那么上面原接口和类不需要改动，我们只需要新增代理器即可。。。

这里就会显出代理模式和适配器模式最大的区别，代理模式是与原对象实现同一个接口，而适配器类则是匹配新接口，说白了，实现一个新的接口。

代理类：

**[java]** [view plain](http://blog.csdn.net/lulei9876/article/details/39994825) [copy](http://blog.csdn.net/lulei9876/article/details/39994825)

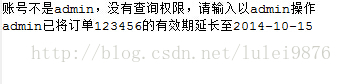
1. **public** **class** ProxySourceOrderApiImpl **implements** SourceOrderApi {
2. SourceOrderApi sourceOrderApiImpl;
3. **public** ProxySourceOrderApiImpl(){
4. sourceOrderApiImpl = **new** SourceOrderApiImpl();
5. }
7. @Override
8. **public** **void** updateDate(String orderId, String date, String client) {
9. //进行判断，如果是admin则更新否则让其输入账号密码
10. **if**("admin".equals(client)){
11. sourceOrderApiImpl.updateDate(orderId, date, client);
12. }**else**{
13. System.out.println("账号不是admin，没有查询权限，请输入以admin操作");
14. }
15. }
16. }

main方法：

**[java]** [view plain](http://blog.csdn.net/lulei9876/article/details/39994825) [copy](http://blog.csdn.net/lulei9876/article/details/39994825)

1. **public** **class** Test {
3. **public** **static** **void** main(String[] args) {
4. SourceOrderApi proxySourceOrderApiImpl = **new** ProxySourceOrderApiImpl();
5. proxySourceOrderApiImpl.updateDate("123456", "2014-10-15", "user");
6. proxySourceOrderApiImpl.updateDate("123456", "2014-10-15", "admin");
7. }
9. }

运行结果：



这里的代理类必须要持要实现原接口和持有原接口的对象，才能称之为代理类。

这样，我们不需要修改原先的实现类，用一个代理类来进行过滤。有些人可能有这样的疑惑，为什么不直接在原实现类中修改，记住JAVA设计模式的基本原则，对内关闭修改。

假如你并没有对方的类，别人只是提供给你这样一个可以操作的JAR包你如何去做？把JAR包反编译，再修改？又假如某些地方可以直接调用原有接口，你如果修改原有的实现类，岂不是对其他的地方也造成了影响？不能修改原有代码这是基本原则。

最后，就是装饰器模式，装饰器模式与相对于上面2种模式更好理解，差别也最大。

假如说，现在这个延长订单，不光可以延长订单提货有效期，可以延长订单的退货有效期。

这就是一个典型的装饰器，你需要做的是丰富原接口的功能，并且不改动原先的接口。

**[java]** [view plain](http://blog.csdn.net/lulei9876/article/details/39994825) [copy](http://blog.csdn.net/lulei9876/article/details/39994825)

1. **public** **class** NewSourceOrderApiImpl **implements** SourceOrderApi {
3. SourceOrderApi sourceOrderApi;
4. **public** NewSourceOrderApiImpl(SourceOrderApi sourceOrderApi){
5. **this**.sourceOrderApi = sourceOrderApi;
6. }
7. @Override
8. **public** **void** updateDate(String orderId, String date, String client) {
9. sourceOrderApi.updateDate(orderId, date, client);
10. System.out.println(client+"已将订单"+orderId+"的退款期延长至"+date);
12. }
14. }

main方法：

**[java]** [view plain](http://blog.csdn.net/lulei9876/article/details/39994825) [copy](http://blog.csdn.net/lulei9876/article/details/39994825)

1. **public** **class** Test {
3. **public** **static** **void** main(String[] args) {
4. SourceOrderApi sourceOrderApi = **new** NewSourceOrderApiImpl(**new** SourceOrderApiImpl());
5. sourceOrderApi.updateDate("123456", "2014-10-15", "user");
6. }
8. }

运行结果：

http://img.blog.csdn.net/20141014104235292?watermark/2/text/aHR0cDovL2Jsb2cuY3Nkbi5uZXQvbHVsZWk5ODc2/font/5a6L5L2T/fontsize/400/fill/I0JBQkFCMA==/dissolve/70/gravity/Center

在装饰器模式中，必须要有被装饰的类和装饰的类。。在这套代码中，原先SourceOrderApi的对象就是被装饰的类，而新建NewSourceOrderApiImpl 就是装饰类，装饰类必须把被装饰的对象当作参数传入。

这就是和代理模式的代码不同之处，代理模式一定是自身持有这个对象，不需要从外部传入。而装饰模式的一定是从外部传入，并且可以没有顺序，按照代码的实际需求随意挑换顺序，就如你吃火锅先放白菜还是先放丸子都可以。

再从使用上来看，代理模式注重的是隔离限制，让外部不能访问你实际的调用对象，比如权限控制，装饰模式注重的是功能的拓展，在同一个方法下实现更多的功能。

啰啰嗦嗦一大篇文章，大概是把这三种模式好好的整理一下，个人本身也是彻底理解了这三种模式。

现在可以安心的进入下一个环节学习了。。。

匆匆拙笔，如有疏漏错误，敬请指出