# 设计模式报告

**——代理模式**

**1 简介**

代理模式为其他对象提供一种代理以控制对这个对象的访问。在代理模式（Proxy Pattern）中，一个类代表另一个类的功能。这种类型的设计模式属于结构型模式。在代理模式中，我们创建具有现有对象的对象，以便向外界提供功能接口。代理模式为其他对象提供一种代理以控制对这个对象的访问。主要解决在直接访问对象时带来的问题，比如说：要访问的对象在远程的机器上。在面向对象系统中，有些对象由于某些原因（比如对象创建开销很大，或者某些操作需要安全控制，或者需要进程外的访问），直接访问会给使用者或者系统结构带来很多麻烦，我们可以在访问此对象时加上一个对此对象的访问层。

**2.应用场景**

代理类主要负责为委托类预处理消息、过滤消息、把消息转发给委托类，以及事后对返回结果的处理等。代理类本身并不真正实现服务，而是通过调用委托类的相关方法，来提供特定的服务。真正的业务功能还是由委托类来实现，但是可以在业务功能执行的前后加入一些公共的服务。例如我们想给项目加入缓存、日志这些功能，我们就可以使用代理类来完成，而没必要打开已经封装好的委托类，比如SpringAop。

**3．代理模式的具体实现**

**3.1 代理模式类图**

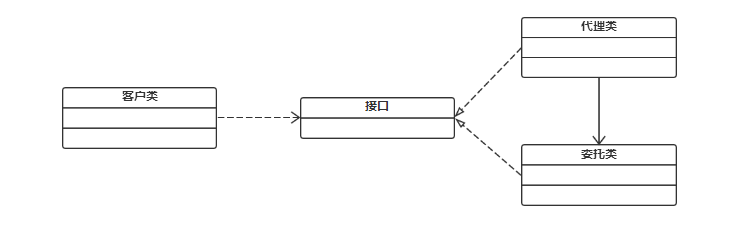


图 1代理模式类图示例

如图1，代理模式其特征是代理类和委托类实现相同的接口。

如果按照代理创建的时期来进行分类的话， 可以分为两种：静态代理、动态代理。我们以买房子为例，看代理模式的具体实现。

**3.2 静态代理**

静态代理类图如下：

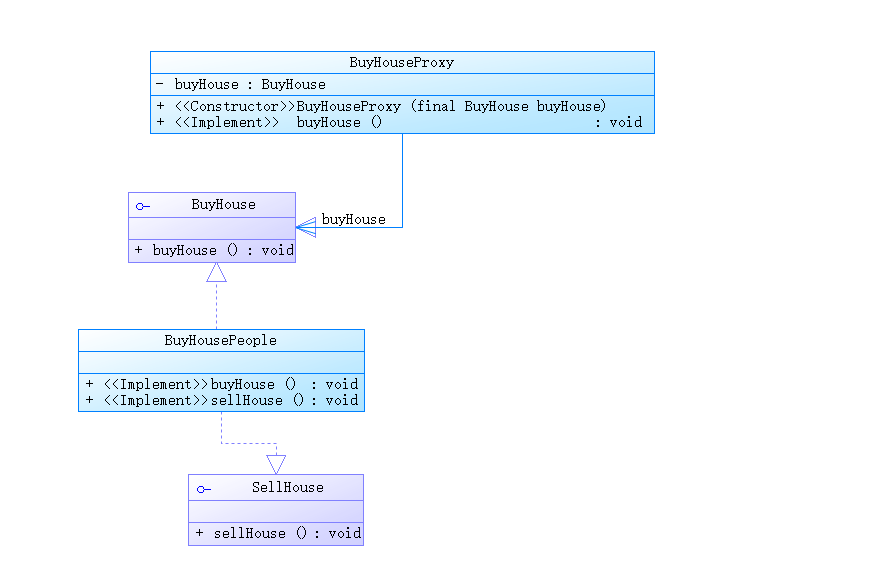


图 2 静态代理类图

代码实现如下：

（1）定义“买房”接口，

public interface BuyHouse {  
 void buyHouse();  
}

（2）定义“卖房”接口，

public interface SellHouse {  
 void sellHouse();  
}

（3）新建一个委托类“买房子的人”，并且实现“买房”的服务，

public class BuyHousePeople implements BuyHouse,SellHouse{  
 @Override  
 public void buyHouse() {  
 System.*out*.println("我要买新房子");  
 }  
 @Override  
 public void sellHouse() {  
 System.*out*.println("我要卖掉老房子");  
 }  
}

（4）新建一个代理类“买房代理”，同时也实现“买房”接口，

public class BuyHouseProxy implements BuyHouse {  
  
 private BuyHouse buyHouse; //需要代理的服务  
  
 public BuyHouseProxy(final BuyHouse buyHouse) {  
 this.buyHouse = buyHouse;  
 }  
 @Override  
 public void buyHouse() {  
 System.*out*.println("提前准备");  
 buyHouse.buyHouse();  
 System.*out*.println("签订合同");  
 }  
 //为什么要叫静态代理，因为还没有人调用我的时候 我就知道我能代理什么服务。  
 //如果现在买房的人也实现了买房的接口，那么也需要为卖房这个服务类写一个代理类。动态代理的话，就不需要了，可以直接调用。  
}

（5）新建客户类（测试）

public class ProxyTest {  
 public static void main(String[] args) {  
 BuyHouse buyHousePeople = new BuyHousePeople();  
 buyHousePeople.buyHouse();  
 System.*out*.println("=====================");  
 BuyHouseProxy buyHouseProxy = new BuyHouseProxy(buyHousePeople);  
 buyHouseProxy.buyHouse();  
 }  
}

**3.3 动态代理**

动态代理类图如下:

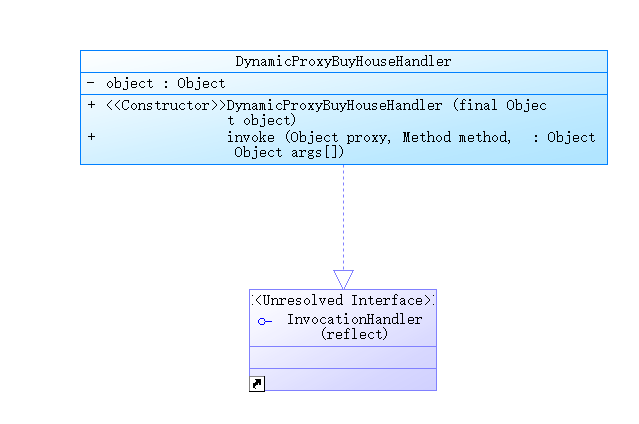


图 3动态代理类图

代码实现如下：

（1）定义动态代理类，实现InvocationHandler接口，

public class DynamicProxyBuyHouseHandler implements InvocationHandler {  
  
 private Object object;  
  
 public DynamicProxyBuyHouseHandler(final Object object) {  
 this.object = object;  
 }  
  
 @Override  
 public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {  
 System.*out*.println("提前准备");  
 Object result = method.invoke(object, args);  
 System.*out*.println("签订合同");  
 return result;  
 }  
 //为什么要叫动态代理呢，因为我这个中介具体要代理什么服务，我自己都不知道。只有在运行的时候，我才知道，哦原来客户想让我帮他买房子。  
}

（2）新建动态代理测试类。

public class DynamicProxyTest {  
 public static void main(String[] args) {  
 BuyHouse buyHousePeople = new BuyHousePeople();  
 BuyHouse proxyBuyHouse = (BuyHouse) Proxy.*newProxyInstance*(BuyHouse.class.getClassLoader(), new  
 Class[]{BuyHouse.class}, new DynamicProxyBuyHouseHandler(buyHousePeople));  
 proxyBuyHouse.buyHouse();  
 System.*out*.println("=========================");  
 SellHouse proxySellHouse= (SellHouse) Proxy.*newProxyInstance*(SellHouse.class.getClassLoader(), new  
 Class[]{SellHouse.class}, new DynamicProxyBuyHouseHandler(buyHousePeople));  
 proxySellHouse.sellHouse();  
 }  
// ClassLoader loader:指定当前目标对象使用的类加载器,获取加载器的方法是固定的  
// Class<?>[] interfaces:指定目标对象实现的接口的类型,使用泛型方式确认类型  
// InvocationHandler:指定动态处理器，执行目标对象的方法时,会触发事件处理器的方法  
}

**3.4静态代理和动态代理的比较**

静态代理优点：

（1）可以做到在符合开闭原则的情况下对目标对象进行功能控制，或者功能增强。

静态代理缺点：

（1）如果接口增加一个方法，除了所有实现类需要实现这个方法外，所有代理类也需要实现此方法。显而易见，增加了代码维护的复杂度。

（2）我们得为每一个服务都得创建代理类，如果现在买房的人也实现了买房的接口，那么也需要为卖房这个服务类写一个代理类。动态代理的话，就不要了，可以直接调用。

动态代理优点

（1）动态代理实现了只需要将被代理对象作为参数传入代理类就可以获取代理类对象，从而实现类代理，具有较强的灵活性。

（2）动态代理的服务内容不需要像静态代理一样写在每个代码块中，只需要写在invoke()方法中即可，降低了代码的冗余度。

动态代理缺点

（1）动态代理类仍然需要实现接口。

**4.代理模式的优缺点分析**

**4.1 代理模式优缺点**

代理模式具有如下优点：

（1）职责清晰。

（2）高扩展性。

（3）智能化。

代理模式缺点如下：

（1）由于在客户端和真实主题之间增加了代理对象，因此有些类型的代理模式可能会造成请求的处理速度变慢。

（2）实现代理模式需要额外的工作，有些代理模式的实现非常复杂。

**4.2 代理模式与装饰器模式的区别**

代理模式和装饰器模式的区别：装饰器模式为了增强功能，而代理模式是为了加以控制。如表1所示

表1 代理模式与装饰器模式比较



综上，对于代理类，如何调用对象的某一功能是思考重点，而不需要兼顾对象的所有功能； 对于装饰类，如何扩展对象的某一功能是思考重点，同时也需要兼顾对象的其它功能，因为再怎么装饰，本质也是对象本身，要担负起对象应有的职责。尽管他们在写法上会有很多相似的地方，但所表达的意思缺失完全不同的。大家只要记住一句话：代理模式是控制对象的访问，装饰器是增强对象的功能。代理模式重点在于方法（功能）的增强（拓展）处理，装饰器模式重点在于功能的增加。