代理模式是我们比较常用的设计模式之一。其中新思想是为了提供**额外的处理**或者**不同的操作**而在实际对象与调用者之间插入一个代理对象。这些额外的操作通常需要与实际对象进行通信，代理模式一般涉及到的角色有： 抽象角色：声明真实对象和代理对象的共同接口；

　　代理模式是我们比较常用的设计模式之一。**其中新思想是为了提供额外的处理或者不同的操作而在实际对象与调用者之间插入一个代理对象。**这些额外的操作通常需要与实际对象进行通信，代理模式一般涉及到的角色有：

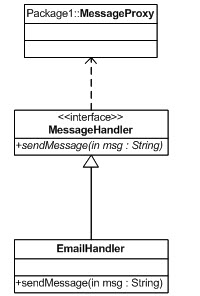
**抽象角色：声明真实对象和代理对象的共同接口；**

**代理角色：代理对象角色内部含有对真实对象的引用，从而可以操作真实对象**，**同时代理对象提供与真实对象相同的接口以便在任何时刻都能代替真实对象。**同时，代理对象可以在执行真实对象操作时，附加其他的操作，相当于对真实对象进行封装。

**真实角色：**代理角色所代表的真实对象，是我们最终要引用的对象。

　　以下以发送消息为例来说明一个简单的代理模式的基本实现：

　　首先明确目的：有一条消息，需要把这个消息发送出去，根据这个目的定义对应接口MessageHandler。需要的附加操作：假设需要验证消息的长度不能超过指定长度并且不能为空，并且我们需要统计相关信息发送到次数，超过指定的次数我们需要输出警报。我们通过代理模式来实现这个附加的操作。下面为对应的类关系图及示例代码。



|  |
| --- |
| //接口定义 public interface **MessageHandler** {  public void sendMessage(String msg); }  　　// 通过Email方式发送消息的实现类 public class **EmailMessage implements MessageHandler** {  @Override  public void sendMessage(String msg) {   // TODO Auto-generated method stub   System.out.println(msg + " send!!");  } }  　　// 消息处理的代理类 public class **MessageProxy implements MessageHandler** {  private static int count;  **private MessageHandler emailMsg;**  　　@Override  public void sendMessage(String msg) {   // TODO Auto-generated method stub   if (checkMessage(msg)) {    if (emailMsg == null)     emailMsg = new EmailMessage();    count++;    emailMsg.sendMessage(msg);    System.out.println("Message sent:" + count);   }  } |

|  |
| --- |
| private boolean checkMessage(String msg) {   return msg != null && msg.length() > 10;  } }  　　// 调用类 public class MainClass { **private static void runProxy(MessageHandler handler) {   handler.sendMessage("message for test");  }**  　　/\*\*   \* @param args   \*/  public static void main(String[] args) {   // TODO Auto-generated method stub **runProxy(new EmailMessage());**   System.out.println("++++++++++++++++Pjroxy++++++++++++++++++");   **runProxy(new MessageProxy());**  } } |

　　输出

　　message for test send!!  
　　++++++++++++++++Pjroxy++++++++++++++++++  
　　message for test send!!  
　　Message sent:1

　　在例子中我们可以方便的在消息发送过程中添加各种需要的附加处理方式，也能方便的替换消息的处理方式，如将通过Email发送消息替换为通过短信发送消息，而调用方不会有丝毫察觉！在任何你想要将一些额外操作分离到具体对象之外，特别是希望能够很容易做出修改，或者想在具体对象的方法执行前插入一些额外操作的时候，代理就显得十分有用！

　　动态代理

**java中动态代理机制**的引入使得代理模式的思想更加完善与进步，它允许动态的创建代理并支持对动态的对所代理的方法进行调用。Java动态代理类位于Java.lang.reflect包下，一般主要涉及到以下两个类：

　　（1）Interface InvocationHandler：该接口中仅定义了一个方法Object：invoke(Object obj，Method method， Object[] args)。在实际使用时，**第一个参数obj一般是指代理类**，**method是被代理的方法**，如上例中的request()，**args为该方法的参数数组。**这个抽象方法在代理类中动态实现。

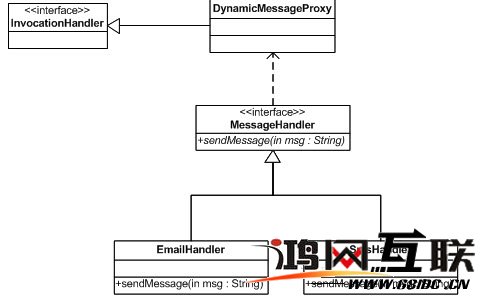
　　（2）**Proxy：该类即为动态代理类**，作用类似于上例中的ProxySubject，其中主要包含以下内容：

　　Protected Proxy(InvocationHandler h)：构造函数，估计用于给内部的h赋值。

　　Static Class getProxyClass (ClassLoader loader， Class[] interfaces)：获得一个代理类，其中loader是类装载器，interfaces是真实类所拥有的全部接口的数组。

　　Static Object newProxyInstance(ClassLoader loader， Class[] interfaces， InvocationHandler h)：返回代理类的一个实例，返回后的代理类可以当作被代理类使用(可使用被代理类的在Subject接口中声明过的方法)。

　　所谓Dynamic Proxy是这样一种class：它是在运行时生成的class，在生成它时你必须提供一组interface给它，然后该class就宣称它实现了这些interface。你当然可以把该class的实例当作这些interface中的任何一个来用。当然啦，这个Dynamic Proxy其实就是一个Proxy，它不会替你作实质性的工作，在生成它的实例时你必须提供一个handler，由它接管实际的工作。下面我们通过动态代理来重新实现上面发送信息的例子！



　　在上面的例子基础上，我们先添加一个通过短信来发送消息的处理类：

|  |
| --- |
| public class **SmsMessage implements MessageHandler** { @Override public void **sendMessage(**String msg) { // TODO Auto-generated method stub System.out.println("SMS Message :" + msg+" sent !"); } } //动态代理类 import java.lang.reflect.InvocationHandler; import java.lang.reflect.Method; public class **DynamicMessageProxy implements InvocationHandler** { private static int count; **private MessageHandler msgHandler; public DynamicMessageProxy(MessageHandler handler) { msgHandler = handler; }** @Override **public Object invoke(Object proxy, Method method, Object[] args)** throws Throwable { // TODO Auto-generated method stub System.out.println("++++++++=============+++++++++"); System.out.println("proxy:" + proxy.getClass()); System.out.println("method:" + method); System.out.println("++++++++=============+++++++++"); if (args != null && args.length == 1 && checkMessage((String) args[0])) { count++; System.out.println("Message sent:" + count); **return method.invoke(msgHandler, args);** } return null; } **private boolean checkMessage(String msg) { return msg != null && msg.length() > 10; }** } //下面是调用 import java.lang.reflect.Proxy; public class MainClass { private static void **runProxy(MessageHandler handler)** { handler.sendMessage("message for test"); } /\*\*  \* @param args  \*/ public static void main(String[] args) { // TODO Auto-generated method stub // runProxy(new EmailMessage()); // System.out.println("++++++++++++++++Proxy++++++++++++++++++"); // runProxy(new MessageProxy()); MessageHandler **handler** = new EmailMessage(); runProxy(handler); **MessageHandler proxy = (MessageHandler) Proxy.newProxyInstance( MessageHandler.class.getClassLoader(), new Class[] { MessageHandler.class }, new DynamicMessageProxy( handler));** runProxy(proxy); System.out.println("++++++++++++++++++++++++++++++++++"); // 短信方式 handler = new SmsMessage(); runProxy(handler); proxy = (MessageHandler) Proxy.newProxyInstance(MessageHandler.class .getClassLoader(), new Class[] { MessageHandler.class }, new DynamicMessageProxy(handler)); runProxy(proxy); } } |

　　下面为以上方法的输出：

　　message for test send!!  
　　++++++++=============+++++++++  
　　proxy:class $Proxy0  
　　method:public abstract void MessageHandler.sendMessage(java.lang.String)  
　　++++++++=============+++++++++  
　　Message sent:1  
　　message for test send!!  
　　++++++++++++++++++++++++++++++++++  
　　SMS Message :message for test sent !  
　　++++++++=============+++++++++  
　　proxy:class $Proxy0  
　　method:public abstract void MessageHandler.sendMessage(java.lang.String)  
　　++++++++=============+++++++++  
　　Message sent:2  
　　SMS Message :message for test sent !

　　以上例子中，通过调用Proxy.newProxyInstance方法创建动态代理对象，该方法需要传入一个类加载器、一组希望代理实现的接口列表、InvocationHandler 接口的一个具体实现。动态代理可以将所有调用重定向到调用处理器，通常我们会向该处理器传递一个时间对象的引用。invoke()方法中传递进来了代理对象，当你需要区分请求来源时这是非常有用的，例如你可以通过判断传入的方法名屏蔽掉某些方法的执行！动态代理机制并不是会很频繁使用的方法，它通常用来解决一些特定情况下的问题，因此不要盲目的为了使用而使用，要根据自己的实际需求来决定！