**1 什么是代理模式**

代理模式(Proxy Pattern)：代理模式是 Java 常见的设计模式之一。

所谓代理模式是指客户端并不直接调用实际的对象，而是通过调用代理对象，来间接的调用实际的对象。

通俗的来讲代理模式就是我们生活中常见的娱乐圈的经纪人或租房圈的中介。

**2 为什么使用代理模式**

隔离作用：在某些情况下，一个客户类不想或者不能直接引用一个委托对象，而代理类

对象可以在客户类和委托对象之间起到中介的作用，其特征是代理类和委托类实现相同的接口。

开闭原则：代理类除了是客户类和委托类的中介之外，我们还可以通过给代理类增加额 外的功能来扩展委托类的功能，这样做我们只需要修改代理类而不需要再修改委托类，符合 代码设计的开闭原则。

代理模式的优点：

 代理模式能将代理对象与真实对象被调用的目标对象分离。

 一定程度上降低了系统的耦合度，扩展性好。

 保护目标对象。

 增强目标对象。

代理模式的缺点：

 代理模式会造成系统设计中类的数目的增加。

 在客户端和目标对象增加一个代理对象，会造成请求处理速度变慢

 增加了系统的复杂度。

**3 代理模式的实现**

代理模式可以分为两种：静态代理、动态代理。

**3.1静态代理**

|  |
| --- |
| **未命名文件(6)** |

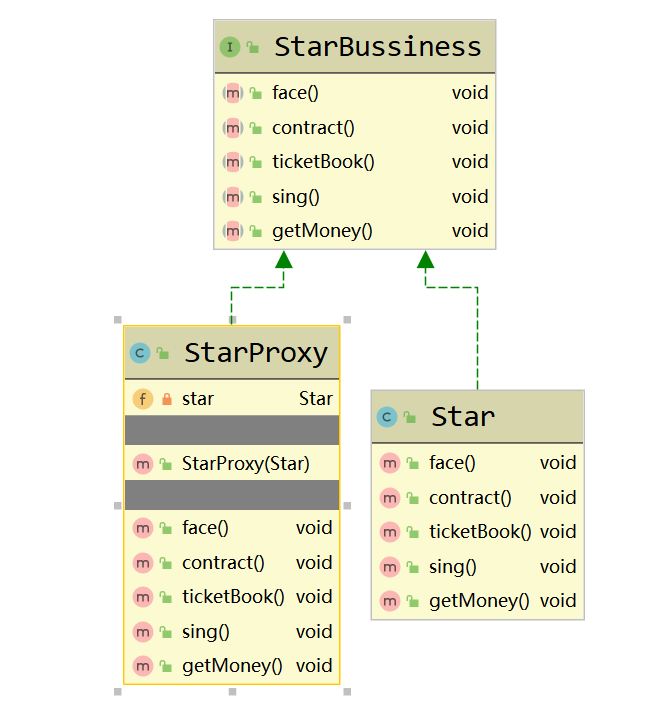
静态代理模式由三个部分构成：

 一个公共的接口

 一个被代理角色

 一个代理角色

#### 图解:

`

**3.1.1 创建公共的接口**

|  |
| --- |
| **package** com.xiangxue;  */\*\*  \** ***@Description*** *明星演出的功能接口与  \** ***@Author*** *SaiLing  \*/* **public interface** StarBussiness {  **public void** face();*//面谈* **public void** contract();*//签约* **public void** ticketBook();*//订行程机票* **public void** sing();*//表演* **public void** getMoney();*//收款* } |

**3.1.2 创建被代理角色**

|  |
| --- |
| **package** com.xiangxue;  */\*\*  \** ***@Description*** *明星类  \** ***@Author*** *SaiLing  \*/* **public class** Star **implements** StarBussiness {  @Override  **public void** face() {   }   @Override  **public void** contract() {   }   @Override  **public void** ticketBook() {   }   @Override  **public void** sing() {  System.***out***.println(**"明星本人表演,代言等等"**);  }   @Override  **public void** getMoney() {   } } |

**3.1.3 创建代理角色**

|  |
| --- |
| **package** com.xiangxue;  */\*\*  \** ***@Description*** *媒体公司(杨幂)  \** ***@Author*** *SaiLing  \*/* **public class** StarProxy **implements** StarBussiness {  *//经纪人召唤,安排明星的行程列表* **private** Star **star**;  **public** StarProxy(Star star){  **this**.**star** = star;  }  @Override  **public void** face() {  System.***out***.println(**"经纪人面签商业活动"**);  }   @Override  **public void** contract() {  System.***out***.println(**"经纪人去签约"**);  }   @Override  **public void** ticketBook() {  System.***out***.println(**"经纪人给订行程机票"**);  }   @Override  **public void** sing() {  **star**.sing();  }   @Override  **public void** getMoney() {  System.***out***.println(**"经纪人,收款"**);  } } |

**3.1.4 创建测试类**

|  |
| --- |
| **package** com.xiangxue;  */\*\*  \** ***@Description*** *\** ***@Author*** *SaiLing  \*/* **public class** StarDemo {  **public static void** main(String[] args) {  Star s= **new** Star();  StarProxy proxy = **new** StarProxy(s);  proxy.face();  proxy.contract();  proxy.sing();  proxy.getMoney();  } } |

**3.2动态代理的实现**

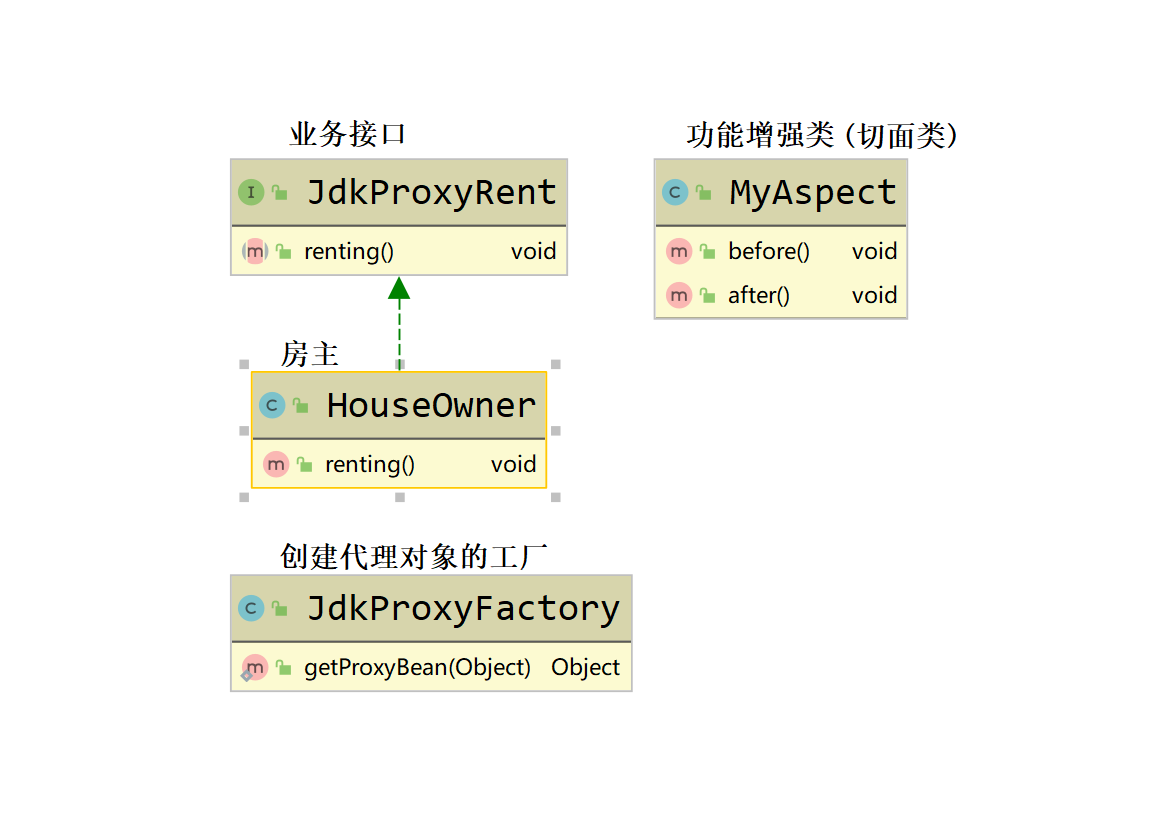
在动态代理中分为两种实现方式：

 使用 JDK 的 Proxy 类实现动态代理

 使用 CGLIB 实现动态代理

**3.2.1 使用 JDK 的 Proxy 类实现动态代理(出租房的案例)**

**图解:**

****

**3.2.1.1 创建业务接口**

|  |
| --- |
| **package** com.jdkproxy;  */\*\*  \** ***@Description*** *业务功能接口  \** ***@Author*** *SaiLing  \*/* **public interface** JdkProxyRent {  *//出租业务* **public void** renting(); } |

**3.2.1.2 创建接口实现类**

|  |
| --- |
| **package** com.jdkproxy;  */\*\*  \** ***@Description*** *被代理对象  \** ***@Author*** *SaiLing  \*/* **public class** HouseOwner **implements** JdkProxyRent{   @Override  **public void** renting() {  System.***out***.println(**"谭女士的房子出租"**);  } } |

**3.2.1.3 创建生成代理对象的工厂**

|  |
| --- |
| **package** com.jdkproxy;  **import** java.lang.reflect.InvocationHandler; **import** java.lang.reflect.Method; **import** java.lang.reflect.Proxy;  */\*\*  \** ***@Description*** *\** ***@Author*** *SaiLing  \*/* **public class** JdkProxyFactory {  */\*\*  \* //创建代理对象的方法  \** ***@param target*** *被代理的对象  \** ***@return*** *\*/* **public static** Object getProxyBean(Object target){  *//创建功能增强类的对象* MyAspect as = **new** MyAspect();  *//获取要被代理的对象的字节码对象* Class c = target.getClass();  **return** Proxy.*newProxyInstance*(c.getClassLoader(), c.getInterfaces(), **new** InvocationHandler() {  */\*\*  \* 这是动态生成代理对象的类  \** ***@param proxy*** *动态生成代理的代理对象  \** ***@param method*** *目标方法的方法对象  \** ***@param args*** *方法中的参数  \** ***@return*** *\** ***@throws*** *Throwable  \*/* @Override  **public** Object invoke(Object proxy, Method method, Object[] args) **throws** Throwable {  *//之前增强* as.before();  *//由代理对象执行出租业务* Object obj = method.invoke(target, args);  *//之后的增强* as.after();  **return** obj; *//代理对象* }  });  } } |

**3.2.1.4 创建切面对象**

|  |
| --- |
| **package** com.jdkproxy;  */\*\*  \** ***@Description*** *这是一个功能增强类(切面类)  \** ***@Author*** *SaiLing  \*/* **public class** MyAspect {  **public void** before(){  System.***out***.println(**"链家的小陈带看房子"**);  }  **public void** after(){  System.***out***.println(**"链家的小陈完成售后服务"**);  } } |

**3.2.1.5 创建测试类**

|  |
| --- |
| **package** com.jdkproxy;  */\*\*  \** ***@Description*** *\** ***@Author*** *SaiLing  \*/* **public class** Demo {  **public static void** main(String[] args) {  JdkProxyRent misTan = **new** HouseOwner();  *//创建代理对象* JdkProxyRent lianJia=(JdkProxyRent)JdkProxyFactory.*getProxyBean*(misTan);  lianJia.renting();   } } |

**3.2.2 使用 CGLIB 实现动态代理**

CGLIB（Code Generation Library）是一个高性能开源的代码生成包，它被许多框架所使用，其底层是通过使用一个小而快的字节码处理框架 ASM（Java 字节码操控框架）转换字 节码并生成新的类。因此 CGLIB 要依赖于 ASM 的包。

JDK 的动态代理机制只能代理实现了接口的类，而对于没有实现接口的类就不能使用 JDK 的 Proxy 类生成代理对象

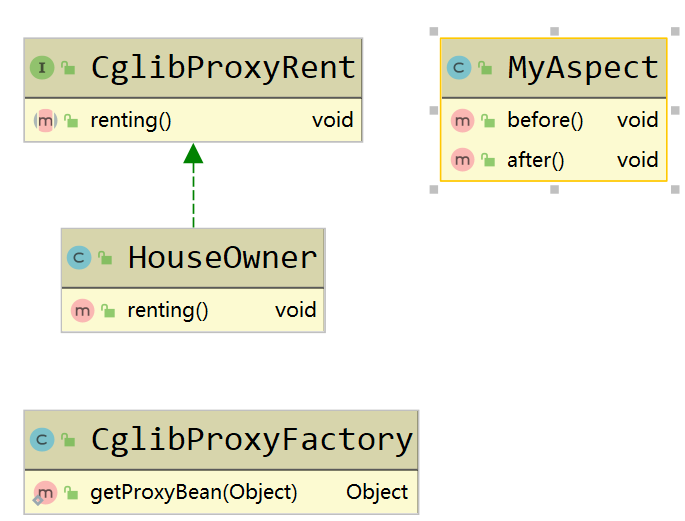
cglib 是针对类来实现代理的，他的原理是对指定的目标类生成一个子类并通过回调的方式来实现增强，但因为采用的是继承，所以不能对 final 修饰的类进行代理。 (创建了一个代理类的子类对象,把被代理的类看作父类)

**3.2.2.1 添加 jar 包**

cglib.jar

asm.jar (在MyBatis框架中有这两个包)

**图解:**



**3.2.2.2 创建业务接口**

|  |
| --- |
| **package** com.cglibproxy;  */\*\*  \** ***@Description*** *业务功能接口  \** ***@Author*** *SaiLing  \*/* **public interface** CglibProxyRent {  *//出租业务* **public void** renting(); } |

**3.2.2.3 创建接口实现类**

|  |
| --- |
| **package** com.cglibproxy;  **import** com.jdkproxy.JdkProxyRent;  */\*\*  \** ***@Description*** *被代理对象  \** ***@Author*** *SaiLing  \*/* **public class** HouseOwner **implements** CglibProxyRent{   @Override  **public void** renting() {  System.***out***.println(**"段女士的别墅出租"**);  } } |

**3.2.2.4 创建生成代理对象的工厂**

|  |
| --- |
| **package** com.cglibproxy;  **import** net.sf.cglib.proxy.Enhancer; **import** net.sf.cglib.proxy.MethodInterceptor; **import** net.sf.cglib.proxy.MethodProxy;  **import** java.lang.reflect.InvocationHandler; **import** java.lang.reflect.Method; **import** java.lang.reflect.Proxy;  */\*\*  \** ***@Description*** *\** ***@Author*** *SaiLing  \*/* **public class** CglibProxyFactory {  */\*\*  \* //创建代理对象的方法  \** ***@param target*** *被代理的对象  \** ***@return*** *\*/* **public static** Object getProxyBean(Object target){  MyAspect as = **new** MyAspect();  Class c = target.getClass();  *//Cglib提供的创建代理对象的模板对象* Enhancer enhancer = **new** Enhancer();  *//被代理对象作为父类,设置父类* enhancer.setSuperclass(c);  *//生成动态代理对象的模板类对象* enhancer.setCallback(**new** MethodInterceptor() {  */\*\*  \*  \** ***@param o*** *代理对象的应用  \** ***@param method*** *目标对象的方法对象  \** ***@param objects*** *目标方法的参数列表  \** ***@param methodProxy*** *目标对象的方法对象的代理对象  \** ***@return*** *\** ***@throws*** *Throwable  \*/* @Override  **public** Object intercept(Object o, Method method, Object[] objects, MethodProxy methodProxy) **throws** Throwable {  *//之前增强* as.before();  Object obj = method.invoke(target, objects);  *//之后增强* as.after();  **return** obj;  }  });  **return** enhancer.create();*//由模板对象创建代理对象* } } |

**3.2.2.5 创建切面**

|  |
| --- |
| **package** com.cglibproxy;  */\*\*  \** ***@Description*** *这是一个功能增强类(切面类)  \** ***@Author*** *SaiLing  \*/* **public class** MyAspect {  **public void** before(){  System.***out***.println(**"链家的小刘带看房子"**);  }  **public void** after(){  System.***out***.println(**"链家的小刘完成售后服务"**);  } } |

**3.2.2.5 测试类**

|  |
| --- |
| **package** com.cglibproxy;  */\*\*  \** ***@Description*** *\** ***@Author*** *SaiLing  \*/* **public class** Demo {  **public static void** main(String[] args) {  CglibProxyRent msDuan = **new** HouseOwner();  *//Cglib的代理对象* CglibProxyRent lianjia = (CglibProxyRent)CglibProxyFactory.*getProxyBean*(msDuan);  lianjia.renting();   } } |

**两种动态代理的区别**

Cglib动态代理的性能高于JDK的动态代理

JDK的动态代理是采用反射实现

Cglib动态代理是直接修改字节码实现