### Java第三阶段—DAY05-JAVA案例

1. 使用静态代理完成手机销售代理

通过案例讲解代理模式的作用：1）间接访问 2) 功能扩展

突出静态代理的问题，引出动态代理

/\*\*  
\* 手机销售  
\*/  
public interface CellphoneSales {  
​  
  /\*\*  
    \* 销售手机  
    \* @param phone  
    \*/  
  void sell(String phone);  
}

/\*\*  
\* 手机工厂  
\*/  
public class CellphoneFactory implements CellphoneSales {  
​  
  public void sell(String phone) {  
      System.out.println("卖了一部" + phone);  
  }  
}

/\*\*  
\* 手机商店  
\*/  
public class CellphoneShop implements CellphoneSales{  
​  
  //手机工厂对象  
  private CellphoneSales factory = null;  
​  
  //通过构造方法传入被代理者  
  public CellphoneShop(CellphoneSales factory) {  
      this.factory = factory;  
  }  
​  
  public void sell(String phone) {  
      //扩展功能  
      System.out.println("打广告！！！新手机，黑科技！！");  
      //调用原来的被代理者方法  
      this.factory.sell(phone);  
      //扩展功能  
      System.out.println("做售后！！！");  
  }  
}

public class TestCellphone {  
​  
  public static void main(String[] args) {  
      //创建手机工厂  
      CellphoneSales factory = new CellphoneFactory();  
      factory.sell("华为P40手机");  
      //创建商店  
      CellphoneSales shop = new CellphoneShop(factory);  
      //调用方法  
      shop.sell("华为P40手机");  
  }  
}

1. 使用JDK动态代理完成手机销售

讲解动态代理的特点：动态生成代理类，剖析AOP的机制

介绍jdk动态代理的局限：实现接口，引出cglib

/\*\*

\* 工厂的JDK动态代理

\*/

public class JDKFactoryProxy implements InvocationHandler {

//被代理对象

private Object target;

/\*\*

\* 创建代理对象

\* @param target 被代理对象

\* @return 代理对象

\*/

public Object createProxy(Object target){

this.target = target;

//创建代理对象 参数1：类加载器, 参数2：代理对象实现的接口，参数3：InvocationHandler的实现对象

return Proxy.newProxyInstance(target.getClass().getClassLoader(),target.getClass().getInterfaces(),this);

}

/\*\*

\* 代理类的方法调用

\*/

public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {

//功能扩展

System.out.println("商店帮忙打广告！！");

//调用被代理者方法

Object invoke = method.invoke(target, args);

System.out.println("商店帮忙做售后！！");

return invoke;

}

}

1. 使用CGLib实现动态代理

讲解cglib动态代理的特点，继承机制。

/\*\*

\* CGLib动态代理

\*/

public class CGlibFactoryProxy implements MethodInterceptor {

//被代理对象

private Object target;

public Object createProxy(Object target){

this.target = target;

Enhancer enhancer = new Enhancer();

//设置父类

enhancer.setSuperclass(this.target.getClass());

//设置方法回调MethodInterceptor实现

enhancer.setCallback(this);

//返回代理对象

return enhancer.create();

}

public Object intercept(Object o, Method method, Object[] objects, MethodProxy methodProxy) throws Throwable {

//扩展

System.out.println("CGLib商店打广告！");

//调用原来方法

Object invoke = method.invoke(target, objects);

System.out.println("CGLib商店做售后！");

return invoke;

}

}