学习代理设计模式

proxy

> A dynamic proxy class (simply referred to as a proxy class below) is a class that implements a list of interfaces specified at runtime when the class is created, with behavior as described below. A proxy interface is such an interface that is implemented by a proxy class. A proxy instance is an instance of a proxy class. Each proxy instance has an associated invocation handler object, which implements the interface InvocationHandler. A method invocation on a proxy instance through one of its proxy interfaces will be dispatched to the invoke method of the instance's invocation handler, passing the proxy instance, a java.lang.reflect.Method object identifying the method that was invoked, and an array of type Object containing the arguments. The invocation handler processes the encoded method invocation as appropriate and the result that it returns will be returned as the result of the method invocation on the proxy instance.

翻译(本人人工翻译，在个人理解的基础上稍有修改,请重点看加粗的内容，并绕过这些非加粗的内容，非加粗的内容是对翻译内容的注释)

proxy类是什么？是一个在创建(即实例化)后在运行时会实现很多特殊接口的类

proxy类的表现？先明确一个概念: ==代理接口指的是被proxy类实现的接口==。==代理实例指的是proxy类的实例化对象==。每一个代理实例都会关联(*所谓关联，即和什么东西有关系，比如和狗有关联，即和狗有某关系，仅此而已*)**一个被称作 `invocation handler object` (译作调用帮手？)的object。这个object实现了一个叫 `InvocationHandler` 的接口。一个代理实例的方法调用通过代理接口被发送到 `invocation handler object` 的调用方法**(the invoke method),**通过代理实例， a java.lang.reflect.Method object找到被调用的方法**(the invoke method)**和包含数据的数组。返回的结果将作为代理实例上方法调用的结果返回**

InvocationHandler

`InvocationHandler` 是一个接口，通过实现这个接口变成 `invocation handler object` 。有一个方法叫invoke

```
/*
    代理逻辑编写的方法：代理对象调用的所有方法都会触发该方法执行
        参数:
            1. proxy:代理对象
            2. method: 代理对象调用的方法，被封装为的对象
            3. args:代理对象调用的方法时，传递的实际参数
 */
@Override
public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
    System.out.println(method.getName());
    System.out.println("before");
    Object invoke = method.invoke(ss, args);
    System.out.println("after");
    return invoke;
}
```

一个实例

先定义一个接口

```
package tk.suyuesheng.proxy;

/**
 * 一个sale接口
 * @author 苏月星
 */
public interface ISale {
    //定义两个方法，一个有返回值，一个没有返回值

    /**
     * 进货
     * @param goodsName 商品名称
     */
    void get(String goodsName);//进货

    /**
     * 数钱 注：成本一律6块2
     * @param money 收的钱
     * @return
     */
    double count(double money);//数钱
}
```

再定义一个实现接口的**真实类**

```
package tk.suyuesheng.proxy;
//代理设计模式的真实对象
public class Saleimpl implements ISale {
    @Override
    public void get(String goodsName) {
        System.out.println("商品名称是："+goodsName);
    }

    @Override
    public double count(double money) {
        System.out.println("开始数钱");
        double count = money - 6.2;
        return count;//返回真正的利润
    }
}
```

至关重要的一步，定义一个 `invocation handler object` ,即中间类，或者调用处理器，这个要实现InvocationHandler接口。中间类的作用是在原有的真实类上定义**要增加的东西**，比如说要在真实类的所有方法执行前都打印一句before。

```
package tk.suyuesheng.proxy;

import java.lang.reflect.InvocationHandler;
import java.lang.reflect.Method;

/**
 * 中介类，或称作调用处理器，或乘坐调用帮手
 */
public class SaleProxy implements InvocationHandler {
    ISale ss;//真实对象
    public SaleProxy(){

    }
    public SaleProxy(ISale ss){
        //调用处理器要指定真实对象,这是必要的
        this.ss=ss;
    }

    /*
        代理逻辑编写的方法：代理对象调用的所有方法都会触发该方法执行
            参数:
                1. proxy:代理对象
                2. method: 代理对象调用的方法，被封装为的对象
                3. args:代理对象调用的方法时，传递的实际参数
     */
    @Override
    public Object invoke(Object proxy, Method method, Object[] args) thro
        System.out.println(method.getName());
        System.out.println("before");
        Object invoke = method.invoke(ss, args);//执行真实类的方法，返回真实类
        System.out.println("after");
        return invoke;
    }
}
```

获得代理对象，并执行代理对象的方法

```
package tk.suyuesheng.proxy;

import java.lang.reflect.Proxy;

public class Main {
    public static void main(String[] args) {
        System.out.println(ISale.class.getInterfaces());
        System.out.println(ISale.class);
        SaleProxy saleProxy = new SaleProxy(new Saleimpl());
        ISale o = (ISale)Proxy.newProxyInstance(ISale.class.getClassLoade
        o.get("家电");
        double count = o.count(12.5);
        System.out.println(count);
    }
}
```

`ISale o = (ISale)Proxy.newProxyInstance(ISale.class.getClassLoader(), new Class[]{ISale.class}, saleProxy);` 的解释：

此举是为获取代理类 `ISale.class.getClassLoader()` 指定了类的加载器，也就是说指定了代理类代理的是谁？返回的是谁的类。 `new Class[]{ISale.class}` 是接口列表，指定了被代理的类实现了哪些接口 `saleProxy` 是中间类，定义了在真实类基础上被修改的内容。

即返回了一个实现了ISale接口的ISale实例化对象，且ISale实例化对象的方法已经按照 `saleProxy` 里定义的内容所修改增加。