利用反射生成动态代理

> java.lang.reflect包下有Proxy类和InvocationHandler接口，主要是利用这两个来生成动态代理对象。
> Proxy类是**所有动态代理类的父类**，通过它的两个静态方法来创建**动态代理类**和**动态代理类对象。**
>
> - 方法一：Class getProxyClass(ClassLoader loader,Class<?>...interfaces):创建一个**动态代理类的Class对象。**第一个参数为：类加载器；第二个参数为：实现的**多个接口。**
> - 方法二：Object Proxy.newProxyInstance(ClassLoader loader,Class<?>...interfaces,InvocationHandler h):创建**动态代理类对象**，第一二参数同上，第三个参数实际传入**自定义的实现InvocationHandler接口的类对象。** 代理类对象执行接口interfaces的任何方法实际上都是**执行InvocationHandler对象的invoke方法**，所以自定义的重点在于**重写invoke方法。 因为调用第一个方法后，也需要利用Class创建实例，所以一般直接使用第二个方法**，实际上**第二个方法的实现也调用了第一个方法，所以才称之为利用反射生成的动态代理。** 下面代码中**获取参数加载类，使用的是Person.class.getClassLoader()，接口类的构造器。**

```
//使用第一种方法
public class TestProxy {
    public static void main(String[] args){
        InvocationHandler Handler = new MyInvocationHandler();
        //获取代理类的Class对象，注意第二个参数
        Class<?> clazz = Proxy.getProxyClass(Person.class.getClassLoader(),new
Class[]{Person.class});
        try {
            //获取构造器
            Constructor sot = clazz.getConstructor(new Class[]
{InvocationHandler.class});
            //创建对象，注意要转类型
            Person s = (Person)sot.newInstance(new Object[]{Handler});
        } catch (NoSuchMethodException e) {
            e.printStackTrace();
        } catch (IllegalAccessException e) {
            e.printStackTrace();
        } catch (InstantiationException e) {
            e.printStackTrace();
        } catch (InvocationTargetException e) {
            e.printStackTrace();
        }
        /*
        Person s =
(Person)Proxy.newProxyInstance(Person.class.getClassLoader(),new Class[]
{Person.class},Handler);
        //同样可以写成上面的形式
        */
    }
}

//设计一个接口
interface Person{
    void work();
    void say();
}
```

```
class MyInvocationHandler implements InvocationHandler{
    @Override
    public Object invoke(Object proxy, Method method, Object[] args) throws
Throwable {
        return null;
    }
}
```

上面代码中，**生成的代理实例对象调用say或者work方法，都是执行invoke方法。**

上述只是单纯的说明了代理类会使抽象类的实例都实际执行invoke方法，实际上要运用其invoke方法的参数

> invoke参数列表
>
> - proxy:是指代理对象本身，实际上是**一个Proxy的子类的实例，实现了我们输入的接口**（第一句话）；
> - method:代表正在执行的方法。如：s.say()，这时候method代表的是say()方法。
> - args:代表调用目标方法时传入的参数。 主要是使用method这个参数来区分不同方法。

> 动态代理 动态代理主要是为了解决当有一段代码

```
public class TestProxy_2 {
    public static void main(String[] args){
        //被代理的实体类
        Dog target = new GunDog();
        //利用工厂生产代理类
        Dog dog = (Dog) ProxyFactory.getProxy(target);
        dog.info();
        dog.run();
    }
}

//接口
interface Dog{
    void info();
    void run();
}

//实体类
class GunDog implements Dog{

    @Override
    public void info() {
        System.out.println("猎狗");
    }

    @Override
    public void run() {
        System.out.println("极速奔跑");
    }
```

```
}

//增强的功能模块代码
class DogUtil{
    public void method1(){
        System.out.println("---------第一个增强功能--------------");
    }
    public void method2(){
        System.out.println("---------第二个增强功能--------------");
    }
}

class MyInvocationHandler2 implements InvocationHandler {
    //需要被代理（增强）的对象
    //用来作为Invoke方法中调用方法method的对象
    private Object target;

    public void setTarget(Object target) {
        this.target = target;
    }

    @Override
    public Object invoke(Object proxy, Method method, Object[] args) throws
Throwable {
        //创建增强类
        DogUtil d = new DogUtil();
        d.method1();
        //被代理对象target执行其方法
        Object result = method.invoke(target,args);
        d.method2();
        return result;
    }

}

//创建代理类工厂
class ProxyFactory{
    public static Object getProxy(Object target){
        MyInvocationHandler2 handler = new MyInvocationHandler2();
        handler.setTarget(target);
        //注意第一个参数和第二个参数
        //由于target传入是一个Object所以和前面的不一样
        //第一个加载器参数：先使用getClass()来获取Class对象。
        //第二个加载器参数：使用getInterfaces()来获取对象实现的接口组。
        return Proxy.newProxyInstance(target.getClass().getClassLoader(),
                target.getClass().getInterfaces(),handler);
    }
}
```

上述代码实际上是为了**给实体类GunDog增加某些功能（DogUtil），为了解耦，即把增强功能的代码从实体类中提取出来。** 作用：1.**解耦，方便维护。**2.**增强类功能（DogUtil）可作为公共代码，给其他类使用。反过来说如果存在多个类使用一段公共代码，则可以考虑使用代理模式提取公共代码。**