链滴

理解设计模式之代理模式

作者：michael
原文链接：https://ld246.com/article/1538892447991
来源网站：链滴

代理模式 Proxy Pattern

什么是代理模式？首先理解代理这个词，从概念上讲有代理人与被代理人之分，代理人替被代理办事。那么从程序的角度上讲，就是代理类，对真实的类，进行逻辑功能增强。之前讲过装饰模式，饰模式也是对一个类进行增强，那么装饰模式和代理模式到底有什么不同？首先，两者的业务出发点同，装饰模式主要是理解被装饰对象方法的业务逻辑，并对其进行扩展增强，而代理模式并不关心你个对象内部方法的业务逻辑是什么，而只关心你调用了什么方法，对你调用的方法进行拦截控制，进额外的业务处理。其次两者的实现方式不一样，装饰模式实现难度小，代理模式实现难度大，不过目已经有很多开源的工具供我们使用，我们只需要实现具体的代理逻辑即可。

代理模式的实现

代理模式的实现我看网上资料分为静态代理和动态代理，说实话，我觉得静态代理不太合理，实方式跟装饰模式没有太大不同，有些人说构造函数中直接分配代理对象，就是静态代理，我觉得这种式有点牵强。首先违背开闭原则，代理类还要实现与被代理类的接口？这相当于跟业务进行了绑定，务接口改变，代理类会跟着改变，这既不符合开闭原则，又与代理模式的设计理念相违背，代理根本关心你这个类有多少个方法，我只关心你调用了何种方法，除非你新增的方法跟代理业务有关，否则本不用管，但是静态代理就打破了这种模式，所以我觉得静态代理不合理。

我觉得动态代理才是代理模式的精髓，有第三方工具来辅助动态代理的实现，首先是Java的标准现：

1、接口定义

```
public interface Subject {
    public String getName();

    public void setName(String name);

    public int getAge();

    public void setAge(int age);
}
```

2、接口的实现对象

```
public class RealSubject implements Subject{
    private String name;

    private int age;

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public int getAge() {
        return age;
    }

    public void setAge(int age) {
```

```
        this.age = age;
    }
}
```

3、代理拦截器实现

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

public class DynamicProxy implements InvocationHandler {

    private Object subject;

    public DynamicProxy(Object subject) {
        this.subject = subject;
    }

    @Override
    public Object invoke(Object object, Method method, Object[] args) throws Throwable {
        String methodName = method.getName();
        System.out.println("调用的方法:" + methodName);
        if ("setName".equals(methodName)) {
            args[0] = args[0] + "-Proxy";
        }
        Object obj = method.invoke(subject, args);
        return obj;
    }

}
```

4、测试类

```
import java.lang.reflect.InvocationHandler;
import java.lang.reflect.Proxy;

public class DynamicProxyTest {
    public static void main(String[] args) {
        RealSubject person = new RealSubject();
        InvocationHandler handler = new DynamicProxy(person);
        Subject proxy = (Subject) Proxy.newProxyInstance(person.getClass().getClassLoader(), pe
son.getClass().getInterfaces(), handler);
        System.out.println(proxy.getClass());
        proxy.setName("michael");
        System.out.println(proxy.getName());
    }
}
```

从上面可以看出，具体的代理拦截器中，代理了具体对象的方法执行，对其进行拦截处理，这是是跟我们常见的一个词很相似？这个词就是AOP，面向切面的编程，没错，就是这个玩意。面向切面编程就是根据方法拦截器来实现的，当然有其他第三方工具包来辅助，例如cglib。下面是基于cglib方式来实现动态代理：

1、方法拦截器

```
import java.lang.reflect.Method;
```

```
import org.springframework.cglib.proxy.MethodInterceptor;
import org.springframework.cglib.proxy.MethodProxy;

public class CglibMethodInterceptor implements MethodInterceptor {

    @Override
    public Object intercept(Object obj, Method method, Object[] args, MethodProxy proxy) thr
ws Throwable {
        String methodName = method.getName();
        System.out.println("before method " + methodName);
        Object result = proxy.invokeSuper(obj, args);
        System.out.println("after method " + methodName);
        return result;
    }

}
```

2、测试类

```
import org.springframework.cglib.proxy.Enhancer;

public class CglibTest {
    public static void main(String[] args) {
        Enhancer enhancer = new Enhancer();
        enhancer.setSuperclass(RealSubject.class);
        enhancer.setCallback(new CglibMethodInterceptor());
        RealSubject obj = (RealSubject) enhancer.create();
        System.out.println(obj.getClass());
        obj.setName("michael");
        System.out.println(obj.getName());
    }
}
```

这里可以看到，基于cglib的方式要比java原生的方便很多，spring的AOP就是基于这种方式来实的。