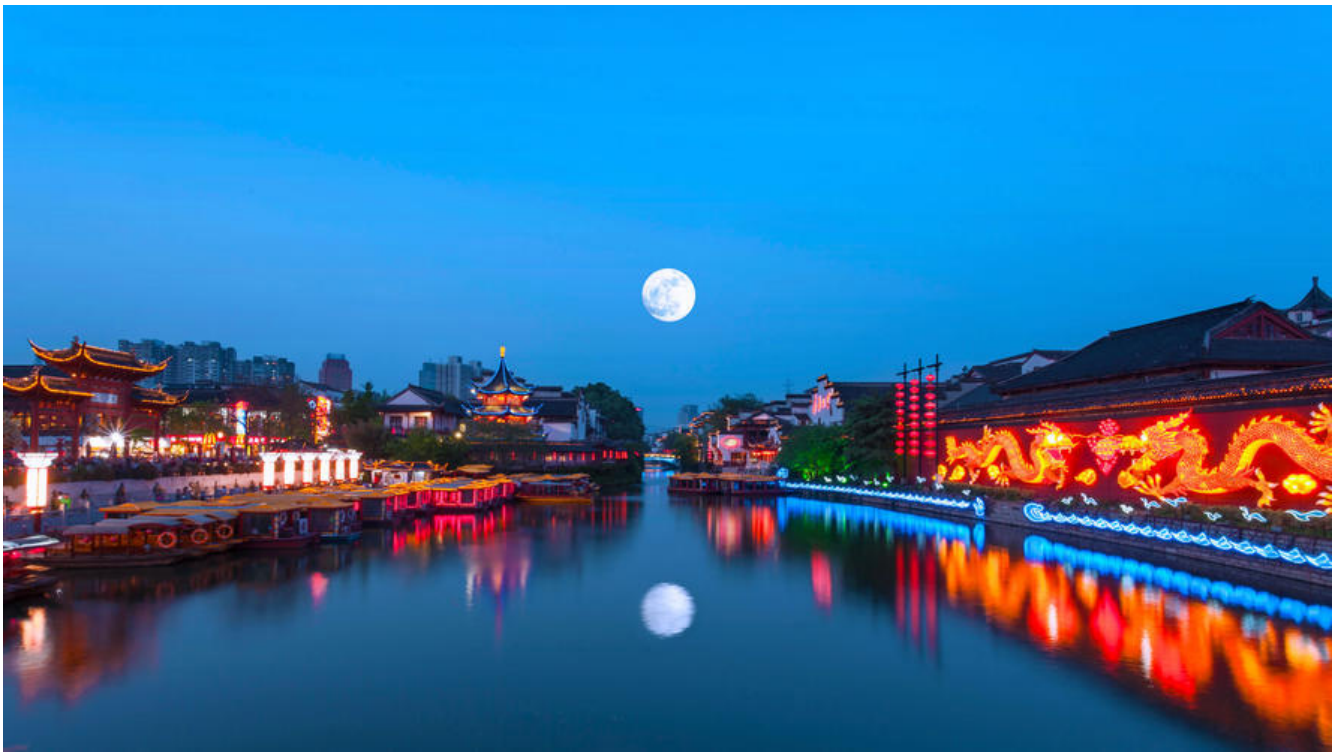
链滴

设计模式学习笔记之代理模式

作者：hjljy

原文链接：https://ld246.com/article/1564412694752

来源网站：链滴



前言

这是一篇学习笔记，内容很多是来源于网上的资料，然后按照自己学习情况进行的总结，有些是自身感受，有些是网上比较好的资料的引用。

<b>如果有人看到我写的笔记有不对的地方欢迎留言指出来，是真的欢迎指出来，因为我可能会错很，然后才发现。学习技术不能闭门造车，要多交流，多讨论，多思考才能成长的快，学的快。

我的个人博客：海加尔金鹰

什么是代理模式

代理模式的定义：

代理模式的定义：为其他对象提供一种代理以控制对这个对象的访问。

根据代理类的创建时机和创建方式的不同，可以将其分为静态代理和动态代理两种形式：在程序运行就已经存在的编译好的代理类是为静态代理，在程序运行期间根据需要动态创建代理类及其实例来完具体的功能是为动态代理。代理模式的目的就是为真实业务对象提供一个代理对象以控制对真实业务象的访问，代理对象的作用有：

代理对象存在的价值主要用于拦截对真实业务对象的访问；

代理对象具有和目标对象(真实业务对象)实现共同的接口或继承于同一个类；

代理对象是对目标对象的增强，以便对消息进行预处理和后处理。[1]

静态代理

说明及主要结构

说明：

静态代理需要先定义一个公共接口，然后被代理对象和代理对象一起实现公共接口。

结构图：

```
graph LR
被代理对象 --> 代理对象
代理对象 -.实现.-> 公共接口
被代理对象 -.实现.->公共接口
```

代码实现

第一步：创建公共接口

```
public interface StaticProxy {
    //具体要做的事情
    void dosomething();
}
```

第二步：创建目标对象

```
public class StaticProxyTarget implements StaticProxy {
    @Override
    public void dosomething() {
        System.out.println("我要打10个");
    }
}
```

第三步：创建代理对象（代理对象需要持有目标对象，并且真正做事的是目标对象）

```
public class StaticProxyImpl implements StaticProxy {

    //代理对象需要持有目标对象
    private StaticProxy staticProxy;

    //通过构造器的方式传入目标对象
    public StaticProxyImpl(StaticProxy staticProxy) {
        this.staticProxy = staticProxy;
    }

    @Override
    public void dosomething() {
        System.out.println("我是代理做的事，可有可无");
        //必须要有，并且真正做事的还是目标对象。
        this.staticProxy.dosomething();
        System.out.println("目标对象做完之后做的事，可有可无");
    }
}
```

第四部：测试验证结果

```
public class StaticProxyTest {
    public static void main(String[] args) {
        System.out.println("--------使用代理模式之前---------");
```

```
        StaticProxyTarget target = new StaticProxyTarget();
        target.dosomething();
        System.out.println("--------使用代理模式之后---------");
        StaticProxy staticProxy = new StaticProxyImpl(target);
        staticProxy.dosomething();
    }
}
```

结果如下：

```
--------使用代理模式之前---------
我要打10个
--------使用代理模式之后---------
我是代理做的事，可有可无
我要打10个
目标对象做完之后做的事，可有可无
```

优缺点及作用

优点： 符合开闭原则，添加代理类就可以在原来的基础上增加功能，不必修改源代码

缺点：一个类一个代理，如果需要代理的类很多怎么办？就会显得不是很方便。

作用：可以进行业务逻辑上的增强，比如日志，消息处理等。

动态代理

什么是动态代理?

对代理模式而言，一般来说，具体主题类与其代理类是一一对应的，这也是静态代理的特点。但是，存在这样的情况：有N个主题类，但是代理类中的“预处理、后处理”都是相同的，仅仅是调用主题同。那么，若采用静态代理，那么必然需要手动创建N个代理类，这显然让人相当不爽。动态代理则以简单地为各个主题类分别生成代理类，共享“预处理，后处理”功能，这样可以大大减小程序规模这也是动态代理的一大亮点。

在动态代理中，代理类是在运行时期生成的。因此，相比静态代理，动态代理可以很方便地对委托类相关方法进行统一增强处理，如添加方法调用次数、添加日志功能等等。动态代理主要分为JDK动态理和cglib动态代理两大类。[1]

JDK动态代理

看名字就知道JDK动态代理是JDK里面自带的一种代理模式实现方式。主要有3个比较重要的接口和类。

1. java.lang.reflect.Proxy ： 该类用于动态生成代理类，只需传入目标接口、目标接口的类加载器及InvocationHandler便可为目标接口生成代理类及代理对象。

2. java.lang.reflect.InvocationHandler ： 该接口包含一个invoke方法，通过该方法实现对委托的代理的访问，是代理类完整逻辑的集中体现，包括要切入的增强逻辑和进行反射执行的真实业务逻。

3. java.lang.ClassLoader ： 加载器类，负责将类的字节码装载到Java虚拟机中并为其定义类对象然后该类才能被使用。Proxy静态方法生成动态代理类同样需要通过类加载器来进行加载才能使用，与普通类的唯一区别就是其字节码是由JVM在运行时动态生成的而非预存在于任何一个.class 文件中

代码实现：

第一步：创建公共接口以及被代理对象

```
/**
*  公共接口
*/
public interface PhoneService {
    //具体要做的事情
    void dosomething2();
}

/**
*被代理的对象
*/
public class PhoneServiceImpl implements PhoneService {
    @Override
    public void dosomething2() {
        System.out.println("用手机看电影");
    }
}
```

第二步：创建动态代理处理器

```
/**
*动态代理处理器，需要实现接口InvocationHandler   在invoke里面进行业务逻辑的书写
*/
public class DynamicProxyHandler implements InvocationHandler {

    //被代理对象
    private Object target;

    //传入被代理对象
    public DynamicProxyHandler(Object target) {
        this.target = target;
    }

    @Override
    public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
        //这里在调用前可以进行逻辑增强操作
        System.out.println("----调用前先吃个饭----");
        //调用被代理对象的方法
        Object invoke = method.invoke(target, args);
        //在掉用后也可以进行逻辑增强操作
        System.out.println("----调用后在洗个澡----");
        //返回调用的结果，这里是根据InvocationHandler接口的invoke方法说明进行的返回
        return invoke;
    }
}
```

第三步：测试以及结果

```
public class DynamicProxyHandlerTest {
    public static void main(String[] args) {
        //获取类加载器，获取到的是AppClassLoader类加载器
```

```
        ClassLoader classLoader = PhoneService.class.getClassLoader();
        //获取接口
        Class<?>[] interfaces = PhoneServiceImpl.class.getInterfaces();
        //创建动态代理处理器
        DynamicProxyHandler dynamicProxyHandler = new DynamicProxyHandler(new PhoneSe
viceImpl());
        //通过Proxy创建代理类
        PhoneService proxyInstance =(PhoneService) Proxy.newProxyInstance(classLoader,interf
ces ,dynamicProxyHandler );
        //执行方法查看结果
        proxyInstance.dosomething2();
    }
}

----调用前先吃个饭----
用手机看电影
----调用后在洗个澡----
```

JDK动态代理原理及源码分析

详见：深入理解代理模式：静态代理与JDK动态代理

JDK动态代理小结

1. 基于反射技术实现。

2. 只能代理接口不能代理类，因为最后生成的代理类都继承了Proxy，在java当中只支持单继承。

3. InvocationHandler中的invoke()方法是代理类完整逻辑的集中体现。

4. JDK动态代理生成的代理类也代理了三个Object类的方法：equals()方法、hashCode()方法和toStrng()方法。

cglib动态代理

在Spring AOP当中还有一个cglib动态代理，它相比jdk动态代理只能代理接口来说，通过字节码技术实现了对类的代理。

Spring AOP 中的代理使用逻辑了：如果目标对象实现了接口，默认情况下会采用 JDK 的动态代理实 AOP；如果目标对象没有实现了接口，则采用 CGLIB 库，Spring 会自动在 JDK 动态代理和 CGLIB 态代理之间转换[2]

SpringAOP源码分析

版本：spring-aop:5.1.2.RELEASE

位置：org.springframework.aop.framework 包下

主要类：

1. AopProxy

```
public interface AopProxy {
    //获取代理类
```

```
    Object getProxy();

    Object getProxy(@Nullable ClassLoader var1);
}
```

2. AopProxyFactory

```
public interface AopProxyFactory {
    //获取AopProxy
    AopProxy createAopProxy(AdvisedSupport var1) throws AopConfigException;
}
```

3. DefaultAopProxyFactory

```
//默认Aop代理工厂类
public class DefaultAopProxyFactory implements AopProxyFactory, Serializable {
    public DefaultAopProxyFactory() {
    }

    public AopProxy createAopProxy(AdvisedSupport config) throws AopConfigException {
        //判断是否是实现了接口
        if (!config.isOptimize() && !config.isProxyTargetClass() && !this.hasNoUserSuppliedProxy
nterfaces(config)) {
            //返回JDK代理类
            return new JdkDynamicAopProxy(config);
        } else {
            Class<?> targetClass = config.getTargetClass();
            if (targetClass == null) {
                throw new AopConfigException("TargetSource cannot determine target class: Either
an interface or a target is required for proxy creation.");
            } else {
                //如果不是接口并且没有代理类，返回cglib代理类
                return (AopProxy)(!targetClass.isInterface() && !Proxy.isProxyClass(targetClass) ? n
w ObjenesisCglibAopProxy(config) : new JdkDynamicAopProxy(config));
            }
        }
    }

    private boolean hasNoUserSuppliedProxyInterfaces(AdvisedSupport config) {
        Class<?>[] ifcs = config.getProxiedInterfaces();
        return ifcs.length == 0 || ifcs.length == 1 && SpringProxy.class.isAssignableFrom(ifcs[0]);
    }
}
```

实现原理及代码

原理说明：

CGLIB动态代理的原理就是用Enhancer生成一个原有类的子类，并且设置好callback到proxy， 则原类的每个方法调用都会转为调用实现了MethodInterceptor接口的proxy的intercept() 函数[3]

代码实现：

```
class MethodInterceptorImpl implements MethodInterceptor {
```

```
    @Override
    public Object intercept(Object obj, Method method, Object[] args, MethodProxy proxy) thr
ws Throwable {
        System.out.println("Before invoke " + method);
        Object result = proxy.invokeSuper(obj, args);
        System.out.println("After invoke" + method);
        return result;
    }
}
```

详情见资料：CGLIB学习笔记

在进行代码实现时需要下载对于的CGLIBjar包，Maven中央库下载很慢，难受哎。

额外知识点

类加载器:ClassLoader

在JDK动态代理当中，我发现在代码获取类加载器这一步当中：

```
        //获取类加载器，获取到的是AppClassLoader类加载器
       ClassLoader classLoader = PhoneService.class.getClassLoader();
```

如果我新建一个接口：PhoneService2 然后将上诉代码转换成：

```
        //获取类加载器，获取到的是AppClassLoader类加载器
       //ClassLoader classLoader = PhoneService.class.getClassLoader();
        ClassLoader classLoader = PhoneService2.class.getClassLoader();
```

程序也是可以正常运行并得出正确结果，但是如果将接口变成JDK自带的例如List就会报错：<font co
or="red"> interface dynamic_proxy.PhoneService is not visible from class loader

```
        //获取类加载器，获取到的是AppClassLoader类加载器
       //ClassLoader classLoader = PhoneService.class.getClassLoader();
       //正常运行
         //ClassLoader classLoader = PhoneService2.class.getClassLoader();
       // 报错 ： interface dynamic_proxy.PhoneService is not visible from class loader
        ClassLoader classLoader = List.class.getClassLoader();
```

在网上查找了一下资料：发现有三种类加载器，

BootStrap ClassLoader：称为启动类加载器，是Java类加载层次中最顶层的类加载器，负责加载JD
中的核心类库，如：rt.jar、resources.jar、charsets.jar等。

Extension ClassLoader：称为扩展类加载器，负责加载Java的扩展类库，默认加载JAVA_HOME/jre/l
b/ext/目下的所有jar。

AppClassLoader：称为系统类加载器，负责加载应用程序classpath目录下的所有jar和class文件。在程序当中获取到的加载器就是这个加载器）

报错原因：<font color="red">猜测可能就是因为使用的类加载器不同导致的。

1. 深入理解代理模式：静态代理与JDK动态代理
2. 面试中关于Spring AOP和代理模式的那些事
3. CGLIB学习笔记