文章已收录我的仓库：Java学习笔记与免费书籍分享

代理模式

引言

代理模式是非常常见的模式，在生活中的例子也非常多，例如你不好意思向你关系不太好朋友帮个忙，这时需要找一个和它关系好的应一个朋友帮忙转达，这个中间朋友就是代理对象。例如购买火车票不一定要去火车站买，可以通过12306网站或者去火车票代售点买。又如找女朋友、找保姆、找工作等都可以通过找中介完成。

代理模式的定义与特点

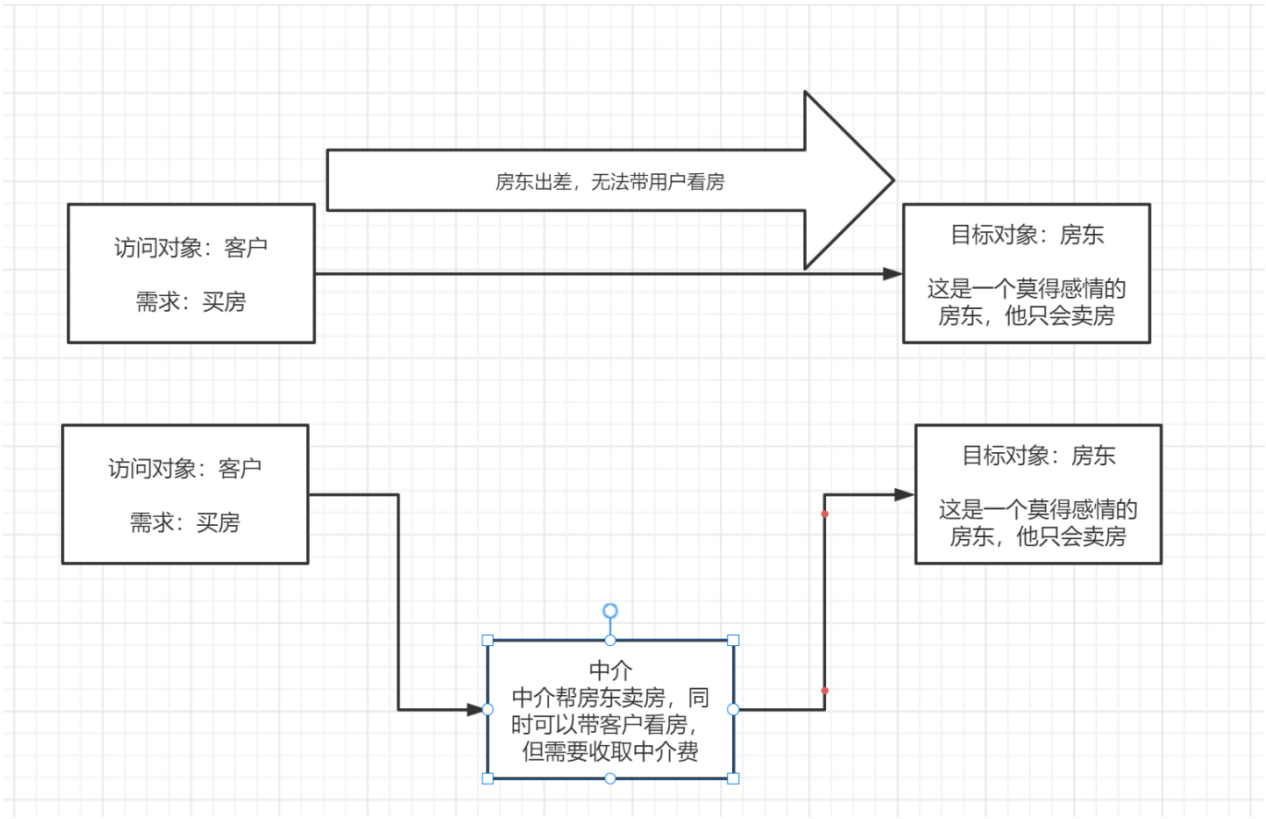
代理模式的定义：由于某些原因需要给某对象**提供一个代理以控制对该对象的访问**。这时，访问对象不适合或者不能直接引用目标对象，代理对象作为访问对象和目标对象之间的中介。

考虑生活中一个常见的例子，客户想买房，房东有很多房，提供卖房服务，但房东不会带客户看房，于是客户通过中介买房。

你可能无法理解这里中介是代替客户买房还是代替房东卖房，其实这是很好理解的。我们程序编写代码是为客户服务的，中介是代替一名服务商处理业务，这种服务可能被定义为`卖房`，也可能被定义为`帮助客户买房`，但中介唯独不可能去实现买房的功能，在代码中，我们定义的是服务于客户的业务接口，而不是客户的需求接口，如果让客户和中介都去实现买房接口，那么这里的买房就是一种业务，服务于卖房的客户，这样房东就是客户端，买房的一方就是服务端。

但在生活中，买房的一方往往是客户端，卖房的才是服务端，因此这里中介和房东都要实现卖房的接口方法，换句话说，中介是代替房东卖房而不是代替客户买房。

客户将中介抽象看成房东，直接从中介手中买房(中介==房东，提供卖房服务)。这里中介就是代理对象，客户是访问对象，房东是目标对象，实际由代理完全操控与目标对象的访问，访问对象客户仅与代理对象交流。



，

代理模式的结构

代理模式的结构比较简单，主要是通过定义一个继承抽象主题的代理来包含真实主题，从而实现对真实主题的访问，下面来分析其基本结构。

代理模式的主要角色如下。

1. 抽象主题（Subject）类(业务接口类)：通过接口或抽象类声明真实主题和代理对象实现的业务方法，服务端需要实现该方法。
2. 真实主题（Real Subject）类(业务实现类)：实现了抽象主题中的具体业务，是代理对象所代表的真实对象，是最终要引用的对象。
3. 代理（Proxy）类：提供了与真实主题相同的接口，其内部含有对真实主题的引用，它可以访问、控制或扩展真实主题的功能。

其结构图如图 1 所示。

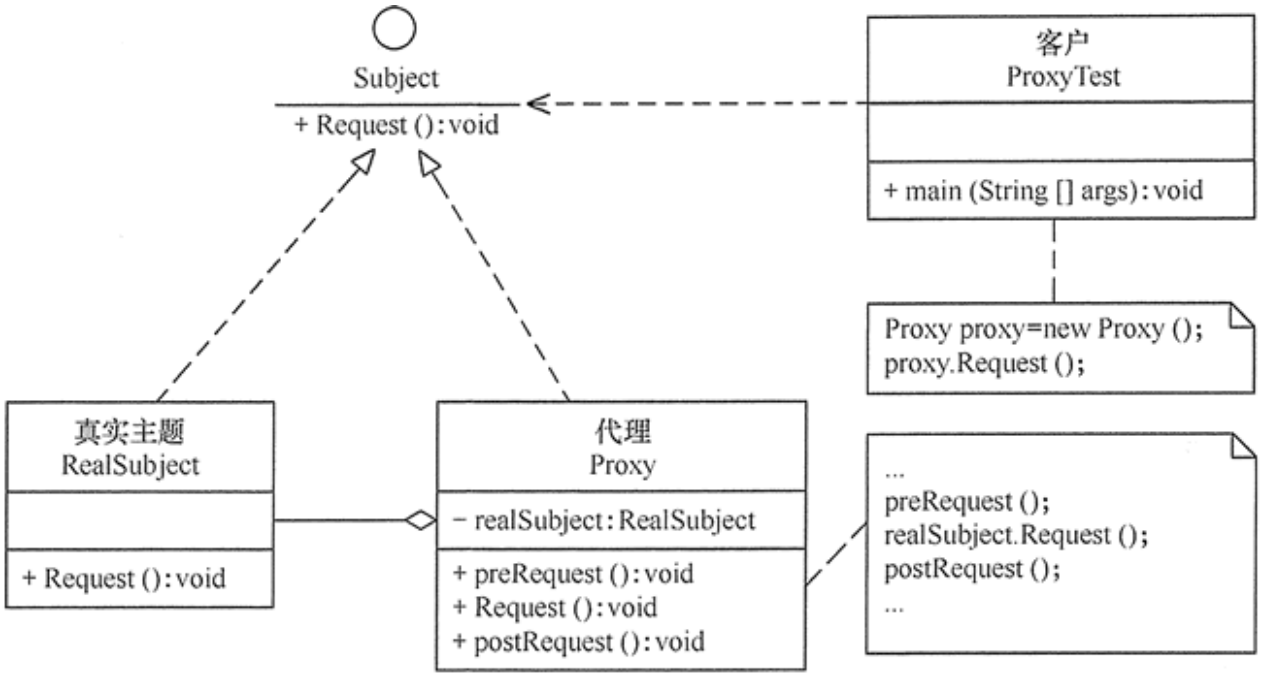


图1 代理模式的结构图

在代码中，一般代理会被理解为代码增强，实际上就是在原代码逻辑前后增加一些代码逻辑，而**使调用者无感知**。

模式实现

根据代理的创建时期，代理模式分为静态代理和动态代理。

- 静态：由程序员创建代理类或特定工具自动生成源代码再对其编译，在程序运行前代理类的 .class 文件就已经存在了。
- 动态：在程序运行时，运用反射机制动态创建而成。

静态代理

静态代理服务于单个接口，我们来考虑实际工程中的一个例子，现在已经有业务代码实现一个增删功能，原有的业务代码由于仍有大量程序无法改变，现在新增需求，即以后每执行一个方法输出一个日志。

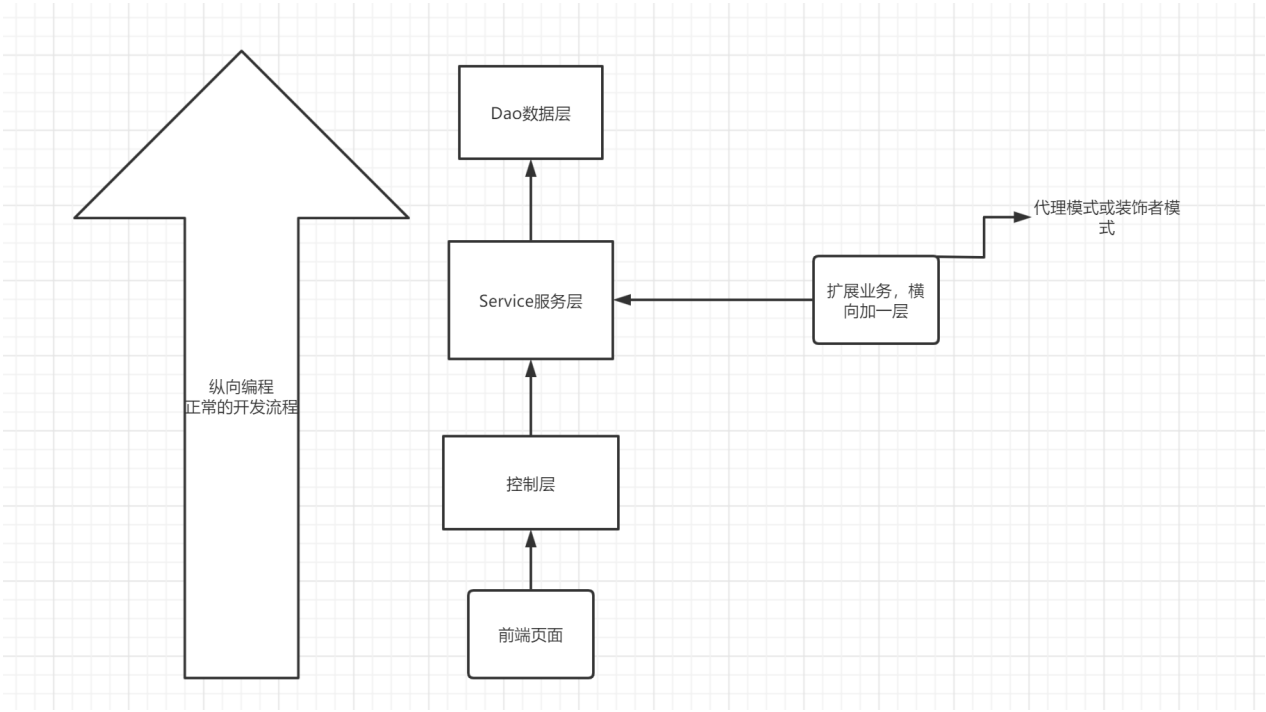
我们不改变原有代码而添加一个代理来实现：

```
//业务接口
interface DateService {
    void add();
    void del();
}

class DateServiceImplA implements DateService {
    @Override
    public void add() {
        System.out.println("成功添加！");
    }
```

```
    @Override
    public void del() {
        System.out.println("成功删除！");
    }
}

class DateServiceProxy implements DateService {
    DateServiceImplA server = new DateServiceImplA();

    @Override
    public void add() {
        server.add();
        System.out.println("程序执行add方法，记录日志.");
    }
    @Override
    public void del() {
        server.del();
        System.out.println("程序执行del方法，记录日志.");
    }
}

//客户端
public class Test {
    public static void main(String[] args) {
        DateService service = new DateServiceProxy();
        service.add();
        service.del();
    }
}
```

现在，我么成功的在不改变程序原有代码的情况下，扩展了一些功能！

我们来思考一下这种情况，当原有的业务处理由于某种原因无法改变，而目前又需要扩展一些功能，此时可以通过代理模式实现：



如上图所示，我们原有的业务十分庞大，牵一发而动全身，难以修改，而现在需要扩展一些功能，这里就需要代理模式实现，在纵向代码之间，横向扩展一些功能，这也是所谓的面向切面编程。

如果你设计思想比较良好的话，你很快就能发现上面代码的不足：一个代理只能服务于一个特定的业务实现类，假设我们又另外一个类也实现了业务接口，即`class DateServiceImplB implements DateService`，发现想要扩展该类必须要为其也编写一个代理，扩展性极低。想要解决这个问题也是很简单的，我们面向接口编程而不是面向实现，我们给代理类持有接口而不是持有具体的类：

```
class DateServiceProxy implements DateService {
    DateService server;

    public DateServiceProxy(DateService server) {
        this.server = server;
    }
}
```

这样一个代理就可以同时代理多个实现了同一个业务接口的业务，但这种方式必须要求客户端传入一个具体的实现类，**这样客户就必须要获得具体目标对象实例，目标对象就直接暴露在访问对象面前了**，对于某些情况这是不可接受的，例如你想获得某资源，但需要一定的权限，这时由代理控制你对目标资源对象的访问，不能由你直接区去访问，这是代理就必须将目标资源对象牢牢的控制在自己手中，**后面会讲到这其实就是保护代理。**但在这里，这种方法是可以接受的，并且带给程序较高的灵活性。

动态代理

我们为什么需要动态代理？要理解这一点，我们必须要知道静态代理有什么不好，要实现静态代理，我们必须要提前将代理类硬编码在程序中，这是固定死的，上面也提到过，有

一些代理一个代理就必须要负责一个类，这种情况下代理类的数量可能是非常多的，但我们真的每个代理都会用上吗？例如，在普通的项目中，可能99%的时间都仅仅只是简单的查询，而不会设计到增删功能，此时是不需要我们的增删代理类的，但在静态代理中，我们仍然必须硬编码代理类，这就造成了不必要的资源浪费并且增加了代码量。

动态代理可以帮助我们仅仅在需要的时候再创建代理类，减少资源浪费，此外由于动态代理是一个**模板的形式**，也可以减少程序的代码量，例如在静态代码示例中，我们在每个方法中加入`System.out.println("程序执行***方法，记录日志.");`，当业务方法非常多时，我们也得为每个业务方法加上记录日志的语句，而动态代理中将方法统一管理，无论几个业务方法都只需要一条记录语句即可实现，具体请看代码。

动态代理采用**反射**的机制，在运行时创建一个接口类的实例。在JDK的实现中，我们需要借助Proxy类和InvocationHandler接口类。

在运行期动态创建一个`interface`实例的方法如下：

1. 定义一个类去实现`InvocationHandler`接口，这个接口下有一个`invoke(Object proxy, Method method, Object[] args)`方法，它负责调用对应接口的接口方法；

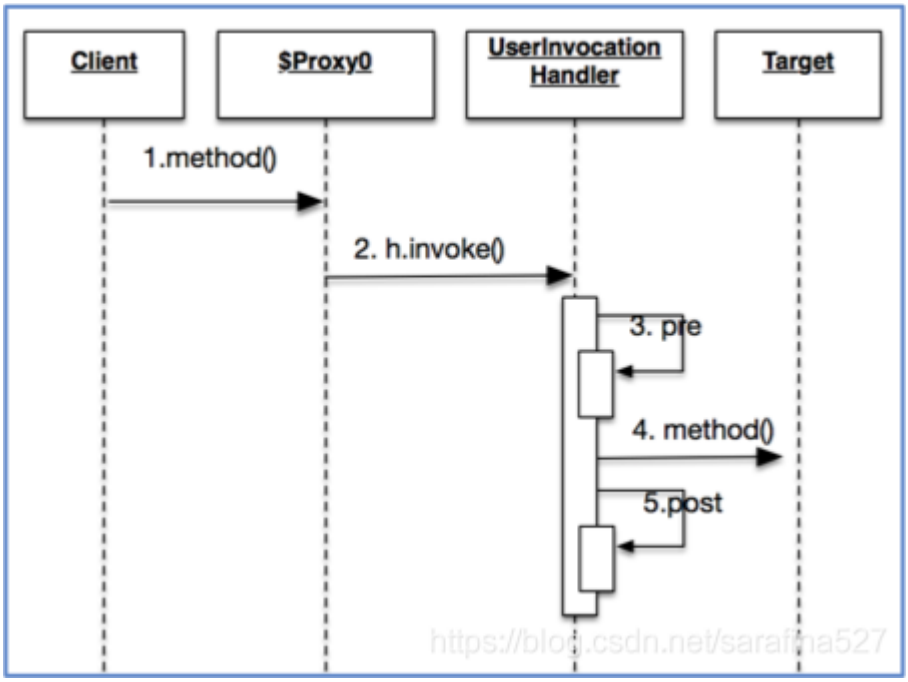
   > 调用代理类的方法时，处理程序会利用反射，将代理类、代理类的方法、要调用代理类的参数传入这个函数，并运行这个函数，这个函数是实际运行的，我们在这里编写代理的核心代码。

2. 通过`Proxy.newProxyInstance()`创建某个`interface`实例，它需要3个参数：

   i. 使用的`ClassLoader`，通常就是接口类的`ClassLoader`；
   ii. 需要实现的接口数组，至少需要传入一个接口进去；
   iii. 一个处理程序的接口。

   > 这个方法返回一个代理类$Proxy0，它有三个参数，第一个通常是类本身的ClassLoader，第二个是该类要实现的接口，例如这里我们要实现增删接口，第三个是一个处理程序接口，即调用这个类的方法时，这个类的方法会被委托给该处理程序，该处理程序做一些处理，这里对应了上面这个方法，通常设置为this。

3. 将返回的`Object`强制转型为接口。



来看一下具体实现：

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

//业务接口
interface DateService {
    void add();
    void del();
}

class DateServiceImplA implements DateService {
    @Override
    public void add() {
        System.out.println("成功添加！");
    }

    @Override
    public void del() {
        System.out.println("成功删除！");
    }
}

class ProxyInvocationHandler implements InvocationHandler {
    private DateService service;

    public ProxyInvocationHandler(DateService service) {
        this.service = service;
    }

    public Object getDateServiceProxy() {
        return Proxy.newProxyInstance(this.getClass().getClassLoader(),
service.getClass().getInterfaces(), this);
    }
```

```
    @Override
    public Object invoke(Object proxy, Method method, Object[] args)
throws Throwable {
        var result = method.invoke(service, args); // 让service调用方法，方
法返回值
        System.out.println(proxy.getClass().getName() + "代理类执行" +
method.getName() + "方法，返回" + result +  "，记录日志！");
        return result;
    }
}

//客户端
public class Test {
    public static void main(String[] args) {
        DateService serviceA = new DateServiceImplA();
        DateService serviceProxy = (DateService) new
ProxyInvocationHandler(serviceA).getDateServiceProxy();
        serviceProxy.add();
        serviceProxy.del();
    }
}
/*
成功添加！
$Proxy0代理类执行add方法，返回null，记录日志！
成功删除！
$Proxy0代理类执行del方法，返回null，记录日志！
*/
```

我们代理类是通过`Proxy.newProxyInstance(this.getClass().getClassLoader(),service.getClass().getInterfaces(), this);`方法得到的，这个方法中，第二个参数我们传入了类service的接口部分，即DateService，在底层通过该接口的字节码帮我们创建一个新类$Proxy0，该类具有接口的全部方法。第三个参数是一个处理程序接口，此处传入this即表明将方法交给ProxyInvocationHandler 的接口即InvocationHandler的invoke方法执行。

$Proxy并不具备真正处理的能力，当我们调用$$Proxy0.add()时，会陷入invoke处理程序，这是我们编写核心代码的地方，在这里`var result = method.invoke(service, args);`调用目标对象的方法，我们可以编写代理的核心代码。

我们还可以编写一个更加万能的接口，让其能扩展不同的业务接口，在静态代理中，如果要扩展两个接口我们最少要编写两个代理类，尽管这两个代理类的代码是一样的，通过一个向上转型，动态代理可以更好的实现这一功能，能够极大的较少代码量。

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

//业务接口
interface DateService {
```

```
    void add();
    void del();
}

class DateServiceImplA implements DateService {
    @Override
    public void add() {
        System.out.println("成功添加！");
    }

    @Override
    public void del() {
        System.out.println("成功删除！");
    }
}


interface OperateService {
    void plus();
    void subtract();
}

class OperateServiceImplA implements OperateService {
    @Override
    public void plus() {
        System.out.println("+ 操作");
    }

    @Override
    public void subtract() {
        System.out.println("- 操作");
    }
}

//w
class ProxyInvocationHandler implements InvocationHandler {
    private Object service;

    public ProxyInvocationHandler(Object service) {
        this.service = service;
    }

    public Object getDateServiceProxy() {
        return Proxy.newProxyInstance(this.getClass().getClassLoader(),
service.getClass().getInterfaces(), this);
    }

    @Override
    public Object invoke(Object proxy, Method method, Object[] args)
throws Throwable {
        var result = method.invoke(service, args); // 方法返回值
        System.out.println(proxy.getClass().getName() + "代理类执行" +
method.getName() + "方法，返回" + result +  "，记录日志！");
        return result;
    }
```

```
}

//客户端
public class Test {
    public static void main(String[] args) {
        DateService dateServiceA = new DateServiceImplA();
        DateService dateServiceProxy = (DateService) new
ProxyInvocationHandler(dateServiceA).getDateServiceProxy();
        dateServiceProxy.add();
        dateServiceProxy.del();

        OperateService operateServiceA = new OperateServiceImplA();
        OperateService operateServiceProxy = (OperateService) new
ProxyInvocationHandler(operateServiceA).getDateServiceProxy();
        operateServiceProxy.plus();
        operateServiceProxy.subtract();
    }
}
/*
成功添加！
$Proxy0代理类执行add方法，返回null，记录日志！
成功删除！
$Proxy0代理类执行del方法，返回null，记录日志！
+ 操作
$Proxy1代理类执行plus方法，返回null，记录日志！
- 操作
$Proxy1代理类执行subtract方法，返回null，记录日志！
*/
```

总结

代理模式通常有如下几种用途：

- 远程代理，这种方式通常是为了隐藏目标对象存在于不同地址空间的事实，方便客户端访问。例如，用户申请某些网盘空间时，会在用户的文件系统中建立一个虚拟的硬盘，用户访问虚拟硬盘时实际访问的是网盘空间。
- 虚拟代理，这种方式通常用于要创建的目标对象开销很大时。例如，下载一幅很大的图像需要很长时间，因某种计算比较复杂而短时间无法完成，这时可以先用小比例的虚拟代理替换真实的对象，消除用户对服务器慢的感觉。
- 保护代理，当对目标对象访问需要某种权限时，保护代理提供对目标对象的受控保护，例如，它可以拒绝服务权限不够的客户。
- 智能指引，主要用于调用目标对象时，代理附加一些额外的处理功能。例如，增加计算真实对象的引用次数的功能，这样当该对象没有被引用时，就可以自动释放它(C++智能指针)；例如上面的房产中介代理就是一种智能指引代理，代理附加了一些额外的功能，例如带看房等。

代理模式的主要优点有：

- 代理模式在客户端与目标对象之间起到一个中介作用和保护目标对象的作用；
- 代理对象可以扩展目标对象的功能；
- 代理模式能将客户端与目标对象分离，在一定程度上降低了系统的耦合度，增加了程序的可扩展性;

其主要缺点是：

- 静态代理模式会造成系统设计中类的数量增加，但动态代理可以解决这个问题；
- 在客户端和目标对象之间增加一个代理对象，会造成请求处理速度变慢；
- 增加了系统的复杂度；

与装饰者模式

我们实现的代理模式和装饰者模式十分相像，但他们的目的不同。在上面我们提到过，某些代理会严格将访问对象和受控对象分离开来，一个代理仅仅只负责一个类，这与装饰器模式是不同的，对于装饰器模式来说，目标对象就是访问对象所持有的。此外虚拟代理的实现与装饰者模式实现是不同的，虚拟代理一开始并不持有远程服务器的资源对象，而是对域名和文件名进行解析才得到该对象，这与我们上面的代码都是不同的，在我们的代码中我们要么传入一个实例，要么让代理持有一个实例，但在虚拟代理中，我么传入一个虚拟的文件资源，虚拟代理对远程服务器进行解析才会获得真实的对象实例，这一点也是不同的。