## 静态代理(static Proxy)

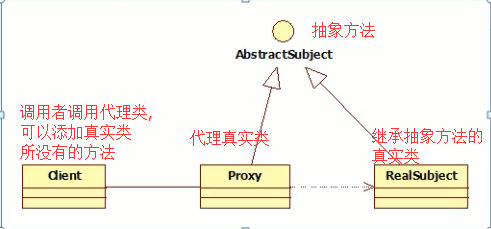
1. 静态代理的角色分析:

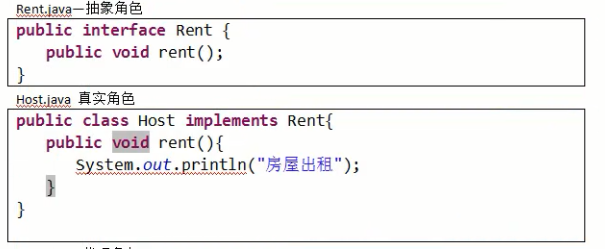
抽象角色 -- 一般使用抽象或者接口来实现

真实角色 -- 被代理的角色

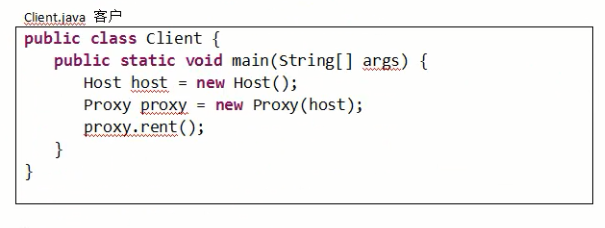
代理角色 -- 代理真实角色 代理真实角色后一般会做一些附属操作

客户 -- 使用代理角色来进行一些操作





Proxy代理角色:



1. 使用静态代理的好处:

使得真实角色处理的业务更加纯粹,不再去关注一些公共的事情.

公共的业务由代理完成----实现业务的分工

公共业务发生扩展时变得更加集中和方便

缺点:

类多了----多了代理类.工作量变大了,开发效率降低了

## 动态代理

1. 动态代理和静态代理的角色是一样的
2. 动态代理的代理类是动态生成的
3. 分为两类,一类是基于接口动态代理和基于类的动态代理
   1. 基于接口的动态代理---jdk动态代理
   2. 基于类的动态代理---cglib

现在用javasist来生成动态代理

4. jdk动态代理---proxy类和invocationhandler接口

InvocationHandler是代理实例的 调用处理程序 实现的接口.

每个代理实例都具有一个关联的调用处理程序.对代理实例调用方法时, 将

对方法调用进行编码并将其指派到它的调用处理程序的invoke方法.

在java的动态代理机制中，有两个重要的类或接口，一个是 InvocationHandler(Interface)、另一个则是 Proxy(Class)，这一个类和接口是实现我们动态代理所必须用到的。首先我们先来看看java的API帮助文档是怎么样对这两个类进行描述的：

每一个动态代理类都必须要实现InvocationHandler这个接口，并且每个代理类的实例都关联到了一个handler，当我们通过代理对象调用一个方法的时候，这个方法的调用就会被转发为由InvocationHandler这个接口的 invoke 方法来进行调用。我们来看看InvocationHandler这个接口的唯一一个方法 invoke 方法：

**Object invoke(Object proxy, Method method, Object[] args) throws Throwable**

我们看到这个方法一共接受三个参数，那么这三个参数分别代表什么呢？

Object invoke(Object proxy, Method method, Object[] args) throws Throwable

**proxy**:　　**指代我们所代理的那个真实对象**

**method**:　　**指代的是我们所要调用真实对象的某个方法的Method对象**

**args**:　　**指代的是调用真实对象某个方法时接受的参数**

**在invoke方法编码指定返回的代理对象干的工作**

**proxy : 把代理对象自己传递进来**

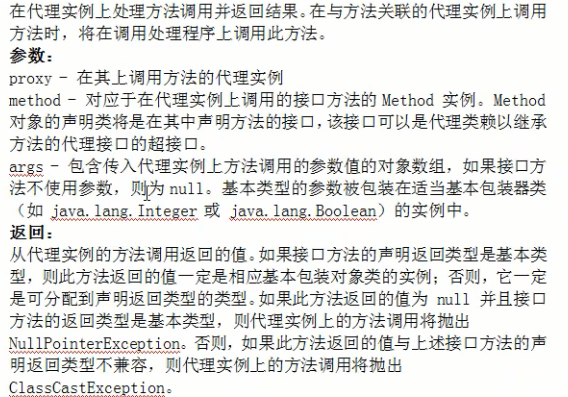
**method：把代理对象当前调用的方法传递进来**

**args:把方法参数传递进来**

**当调用代理对象的person.sing("冰雨");或者 person.dance("江南style");方法时，**

**实际上执行的都是invoke方法里面的代码，**

**因此我们可以在invoke方法中使用method.getName()就可以知道当前调用的是代理对象的哪个方法**



Proxy 提供用于创建动态代理类和实例的静态方法,他还是由这些方法创建的所有动态代理类的超类

static Object newProxyInstance(ClassLoader loader, Class<?>[] interface, InvcationHandler h)

返回一个指定接口的代理类实例, 该接口可以将方法调用指派到指定的调用处理程序

**现在要生成某一个对象的代理对象，这个代理对象通常也要编写一个类来生成**，所以首先要编写用于生成代理对象的类。在java中如何用程序去生成一个对象的代理对象呢，java在JDK1.5之后提供了一个"**java.lang.reflect.Proxy**"类，通过"**Proxy**"类提供的一个**newPro**

1. static Object **newProxyInstance**(ClassLoader loader, Class<?>[] interfaces, InvocationHandler h)
2. **proxyInstance**方法用来创建一个对象的代理对象，如下所示：

**newProxyInstance**方法用来返回一个代理对象，这个方法总共有3个参数

ClassLoader loader 用来指明生成的代理对象使用哪个类装载器，

Class<?>[] interfaces 用来指明生成哪个对象的代理对象，通过接口指定， InvocationHandler h 用来指明产生的这个代理对象要做什么事情。

所以我们只需要调用**newProxyInstance**方法就可以得到某一个对象的代理对象了。

|  |
| --- |
| @Repository("invocationhandlerProxy")  **public** **class** InvocationhandlerProxy **implements** InvocationHandler {  @Resource(name="UserDaoImpl")  **private** Object obj;    /\*\*  \* 第一个参数 由哪个ClassLoader(类装载器)对象来对生成的代理对象进行加载至虚拟机  \* 第二个参数 用来指明生成哪个对象的代理对象, 通过接口指定  \* 第三个参数 用来指明产生的代理对象要做什么事情  \*/  **public** Object getProxy(){  **return** Proxy.*newProxyInstance*(**this**.getClass().getClassLoader(),  obj.getClass().getInterfaces(), **this**);  }    /\*\*  \* 第一个参数 指要代理的那个真实对象  \* 第二个参数 指我们要调用真实对象的某个方法的对象  \* 第三个参数 指调用真实对象的某个参数所接收的参数数组  \*/  **public** Object invoke(Object proxy, Method method, Object[] args) **throws** Throwable {  System.***out***.println("111111");  method.invoke(obj, args);  System.***out***.println("222222");  **return** **null**;  }  } |
| **//测试类**  **public** **static** **void** main(String[] args) {  InvocationhandlerProxy cxcp = (InvocationhandlerProxy) ApplicationContextUtil.*getInstance*().getBean("invocationhandlerProxy");  UserDao asd = (UserDao) cxcp.getProxy();  asd.addUser();  } |

|  |
| --- |
| public class Client  {  public static void main(String[] args)  {  // 我们要代理的真实对象  Subject realSubject = new RealSubject();  // 我们要代理哪个真实对象，就将该对象传进去，最后是通过该真实对象来调用其方法的  InvocationHandler handler = new DynamicProxy(realSubject);  /\*  \* 通过Proxy的newProxyInstance方法来创建我们的代理对象，我们来看看其三个参数  \* 第一个参数 handler.getClass().getClassLoader() ，我们这里使用handler这个类的ClassLoader对象来加载我们的代理对象  \* 第二个参数realSubject.getClass().getInterfaces()，我们这里为代理对象提供的接口是真实对象所实行的接口，表示我要代理的是该真实对象，这样我就能调用这组接口中的方法了  \* 第三个参数handler， 我们这里将这个代理对象关联到了上方的 InvocationHandler 这个对象上  \*/  Subject subject = (Subject)Proxy.newProxyInstance(handler.getClass().getClassLoader(), realSubject  .getClass().getInterfaces(), handler);    System.out.println(subject.getClass().getName());  subject.rent();  subject.hello("world");  }  } |