抽象角色:通过接口或抽象类声明真实角色实现的业务方法。

代理角色:实现抽象角色，是真实角色的代理，通过真实角色的业务逻辑方法来实现抽象方法，并可以附加自己的操作。

真实角色:实现抽象角色，定义真实角色所要实现的业务逻辑，供代理角色调用。

**package** cn.subject;

**public** **interface** Subject {

**void** getHouse();

}

**package** cn.subject;

**import** cn.proxysubject.Subject;

**public** **class** RealSubject **implements** Subject {

@Override

**public** **void** getHouse() {

// **TODO** Auto-generated method stub

System.*out*.println("自己得到房子");

}

}

**package** cn.subject;

**import** cn.proxysubject.Subject;

**public** **class** ProxySubject **implements** Subject {

**private** RealSubject rs;

**public** ProxySubject(RealSubject rs){

**this**.rs = rs;

}

@Override

**public** **void** getHouse() {

// **TODO** Auto-generated method stub

System.*out*.println("中介得到房子");

rs.getHouse();

}

}

**package** cn.subject;

**public** **class** Test {

/\*\*

\* **@param** args

\*/

**public** **static** **void** main(String[] args) {

// **TODO** Auto-generated method stub

RealSubject rs = **new** RealSubject();

ProxySubject ps = **new** ProxySubject(rs);

ps.getHouse();

}

}

Dynamic subject

**package** cn.proxysubject;

**import** java.lang.reflect.InvocationHandler;

**import** java.lang.reflect.Method;

**public** **class** DynamicSubject **implements** InvocationHandler {

Object sub;

**public** DynamicSubject(Object sub){

**this**.sub = sub;

}

@Override

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

**throws** Throwable {

// **TODO** Auto-generated method stub

System.*out*.println(method);

System.*out*.println("中介找房子");

System.*out*.println("中介得到房子");

method.invoke(sub, args);

System.*out*.println(method);

**return** **null**;

}

}

**package** cn.proxysubject;

**import** java.lang.reflect.InvocationHandler;

**import** java.lang.reflect.Proxy;

**public** **class** Client {

/\*\*

\* **@param** args

\*/

**public** **static** **void** main(String[] args) {

// **TODO** Auto-generated method stub

RealSubject rs = **new** RealSubject();

InvocationHandler handler = **new** DynamicSubject(rs);

Class c = handler.getClass();

Subject subject = (Subject) Proxy.*newProxyInstance*(c.getClassLoader(), rs.getClass().getInterfaces(), handler);

subject.getHouse();

}

}

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

参数method：指的是被代理对象的方法。

Proxy:指的是代理实例

Args：被代理对象的方法的参数。

返回值：如果被代理对象的方法无返回值，可以return null，也可以return非null

如果方法有参必须返回 return 非null。该方法的返回值为被代理方法的方法的返回值

Method.invoke(被代理的对象，方法参数)。拦截操作，代理对象增强的逻辑（外部逻辑）可以放在这个操作前后。对于某个被代理方法内部的逻辑增强，可以通过method.getName()得到要增强的方法，在进行增强操作。 该方法体的内容为代理对象的方法体的内容。

**目的：**主要用来做方法的增强，让你可以在不修改源码（不用改变这个方法的签名，原来调用这个方法的类依然能正常工作）的情况下，增强一些方法。在方法执行前后做任何你想做的事情（甚至根本不去执行这个方法），因为在InvocationHandler的invoke方法中，你可以直接获取正在调用方法对应的Method对象，具体应用的话，比如可以添加调用日志，做事务控制等。  
  
还有一个有趣的作用是可以用作远程调用，比如现在有Java接口，这个接口的实现部署在其它服务器上，在编写客户端代码的时候，没办法直接调用接口方法，因为接口是不能直接生成对象的，这个时候就可以考虑代理模式（动态代理）了，通过Proxy.newProxyInstance代理一个该接口对应的InvocationHandler对象，然后在InvocationHandler的invoke方法内封装通讯细节就可以了。具体的应用，最经典的当然是Java标准库的RMI，其它比如hessian，各种webservice框架中的远程调用，大致都是这么实现的。  
而像 AspectJ 这种 AOP 刚不同，它直接把人家的 class 代码修改了，它就不需要使用代理。  
这些在新的 JDK 6 中都可以通过 Instrument 来做到，不过也是个通用的方法，还得通过规则来定制什么情况下处理，什么时候不处理。

## **二、jdk的动态代理**

　　目前Java开发包中包含了对动态代理的支持，但是其实现只支持对**接口**的的实现。 其实现主要通过java.lang.reflect.Proxy类和java.lang.reflect.InvocationHandler接口。

　　Proxy类主要用来获取动态代理对象，InvocationHandler接口用来约束调用者实现。

　　动态代理是很多框架和技术的基础, spring 的AOP实现就是基于动态代理实现的。了解动态代理的机制对于理解AOP的底层实现是很有帮助的。

### 

### **2.1、Proxy类**

Porxy类也是在java.lang.reflect，Proxy 提供用于创建动态代理类和实例的静态方法，它还是由这些方法创建的所有动态代理类的超类。

代理类具用以下属性：

* 代理类是公共的、最终的，而不是抽象的。
* 未指定代理类的非限定名称。但是，以字符串 "$Proxy" 开头的类名空间应该为代理类保留。
* 代理类扩展 java.lang.reflect.Proxy。
* 代理类会按同一顺序准确地实现其创建时指定的接口。
* 如果代理类实现了非公共接口，那么它将在与该接口相同的包中定义。否则，代理类的包也是未指定的。注意，包密封将不阻止代理类在运行时在特定包中的成功定义，也不会阻止相同类加载器和带有特定签名的包所定义的类。
* 由于代理类将实现所有在其创建时指定的接口，所以对其 Class 对象调用 getInterfaces 将返回一个包含相同接口列表的数组（按其创建时指定的顺序），对其 Class 对象调用 getMethods将返回一个包括这些接口中所有方法的 Method 对象的数组，并且调用 getMethod 将会在代理接口中找到期望的一些方法。
* 如果 [Proxy.isProxyClass](http://www.cnblogs.com/) 方法传递代理类（由 Proxy.getProxyClass 返回的类，或由 Proxy.newProxyInstance 返回的对象的类），则该方法返回 true，否则返回 false。
* 代理类的 java.security.ProtectionDomain 与由引导类加载器（如 java.lang.Object）加载的系统类相同，原因是代理类的代码由受信任的系统代码生成。此保护域通常被授予java.security.AllPermission。
* 每个代理类都有一个可以带一个参数（接口 [InvocationHandler](http://www.cnblogs.com/" \o "java.lang.reflect 中的接口) 的实现）的公共构造方法，用于设置代理实例的调用处理程序。并非必须使用反射 API 才能访问公共构造方法，通过调用[Proxy.newInstance](http://www.cnblogs.com/) 方法（将调用 [Proxy.getProxyClass](http://www.cnblogs.com/) 的操作和调用带有调用处理程序的构造方法结合在一起）也可以创建代理实例。

### **方法**

[IMG_256](http://www.cnblogs.com/duanxz/archive/2012/12/03/javascript:void(0);" \o "复制代码)

protected Proxy(InvocationHandler h) //使用其调用处理程序的指定值从子类（通常为动态代理类）构建新的 Proxy 实例。   
  
static InvocationHandler getInvocationHandler(Object proxy) //返回指定代理实例的调用处理程序。   
  
static Class<?> getProxyClass(ClassLoader loader, Class<?>... interfaces) //返回代理类的 java.lang.Class 对象，并向其提供类加载器和接口数组。   
  
static boolean isProxyClass(Class<?> cl) //当且仅当指定的类通过 getProxyClass 方法或 newProxyInstance 方法动态生成为代理类时，返回 true。   
  
static Object newProxyInstance(ClassLoader loader, Class<?>[] interfaces, InvocationHandler 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。

[IMG_257](http://www.cnblogs.com/duanxz/archive/2012/12/03/javascript:void(0);" \o "复制代码)

### **2.1、InvocationHandler接口**

InvocationHandler接口也是在java.lang.reflect，唯一的一个方法是invoke如下：

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

 这个方法有三个参数，其中第二和第三个参数都比较好理解，一个是被拦截的方法，一个是该方法的参数列表。关键是第一个参数。按照doc文档的解析，proxy - the proxy instance that the method was invoked on也就是说，proxy应该是一个代理实例（动态代理类）。

## **三、示例**

好了，在介绍完这两个接口(类)以后，我们来通过一个实例来看看我们的动态代理模式是什么样的：

首先我们定义了一个Subject类型的接口，为其声明了两个方法：

public interface Subject

{

public void rent();

public void hello(String str);

}

接着，定义了一个类来实现这个接口，这个类就是我们的真实对象，RealSubject类：

[IMG_258](http://www.cnblogs.com/duanxz/archive/2012/12/03/javascript:void(0);" \o "复制代码)

IMG_259

public class RealSubject implements Subject

{

@Override

public void rent()

{

System.out.println("I want to rent my house");

}

@Override

public void hello(String str)

{

System.out.println("hello: " + str);

}

}

IMG_260

[IMG_261](http://www.cnblogs.com/duanxz/archive/2012/12/03/javascript:void(0);" \o "复制代码)

下一步，我们就要定义一个动态代理类了，前面说个，每一个动态代理类都必须要实现 InvocationHandler 这个接口，因此我们这个动态代理类也不例外：

[IMG_262](http://www.cnblogs.com/duanxz/archive/2012/12/03/javascript:void(0);" \o "复制代码)

IMG_263

public class DynamicProxy implements InvocationHandler

{

//　这个就是我们要代理的真实对象

private Object subject;

// 构造方法，给我们要代理的真实对象赋初值

public DynamicProxy(Object subject)

{

this.subject = subject;

}

@Override

public Object invoke(Object object, Method method, Object[] args)

throws Throwable

{

//　　在代理真实对象前我们可以添加一些自己的操作

System.out.println("before rent house");

System.out.println("Method:" + method);

// 当代理对象调用真实对象的方法时，其会自动的跳转到代理对象关联的handler对象的invoke方法来进行调用

method.invoke(subject, args);

//　　在代理真实对象后我们也可以添加一些自己的操作

System.out.println("after rent house");

return null;

}

}

IMG_264

[IMG_265](http://www.cnblogs.com/duanxz/archive/2012/12/03/javascript:void(0);" \o "复制代码)

最后，来看看我们的Client类：

[IMG_266](http://www.cnblogs.com/duanxz/archive/2012/12/03/javascript:void(0);" \o "复制代码)

IMG_267

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");

}

}

IMG_268

[IMG_269](http://www.cnblogs.com/duanxz/archive/2012/12/03/javascript:void(0);" \o "复制代码)

我们先来看看控制台的输出：

[IMG_270](http://www.cnblogs.com/duanxz/archive/2012/12/03/javascript:void(0);" \o "复制代码)

IMG_271

**$Proxy0**

before rent house

Method:**public abstract void com.xiaoluo.dynamicproxy.Subject.rent()**

I want to rent my house

after rent house

before rent house

Method:**public abstract void com.xiaoluo.dynamicproxy.Subject.hello(java.lang.String)**

hello: world

after rent house

IMG_272

[IMG_273](http://www.cnblogs.com/duanxz/archive/2012/12/03/javascript:void(0);" \o "复制代码)

我们首先来看看 $Proxy0 这东西，我们看到，这个东西是由 System.out.println(subject.getClass().getName()); 这条语句打印出来的，那么为什么我们返回的这个代理对象的类名是这样的呢？

Subject subject = (Subject)Proxy.newProxyInstance(handler.getClass().getClassLoader(), realSubject

.getClass().getInterfaces(), handler);

可能我以为返回的这个代理对象会是Subject类型的对象，或者是InvocationHandler的对象，结果却不是，首先我们解释一下**为什么我们这里可以将其转化为Subject类型的对象？**原因就是在newProxyInstance这个方法的第二个参数上，我们给这个代理对象提供了一组什么接口，那么我这个代理对象就会实现了这组接口，这个时候我们当然可以将这个代理对象强制类型转化为这组接口中的任意一个，因为这里的接口是Subject类型，所以就可以将其转化为Subject类型了。

**同时我们一定要记住，通过 Proxy.newProxyInstance 创建的代理对象是在jvm运行时动态生成的一个对象，它并不是我们的InvocationHandler类型，也不是我们定义的那组接口的类型，而是在运行是动态生成的一个对象，并且命名方式都是这样的形式，以$开头，proxy为中，最后一个数字表示对象的标号**。

接着我们来看看这两句

subject.rent();  
subject.hello("world");

这里是通过代理对象来调用实现的那种接口中的方法，这个时候程序就会跳转到由这个代理对象关联到的 handler 中的invoke方法去执行，而我们的这个 handler 对象又接受了一个 RealSubject类型的参数，表示我要代理的就是这个真实对象，所以此时就会调用 handler 中的invoke方法去执行：

[IMG_274](http://www.cnblogs.com/duanxz/archive/2012/12/03/javascript:void(0);" \o "复制代码)

IMG_275

public Object invoke(Object object, Method method, Object[] args)

throws Throwable

{

//　　在代理真实对象前我们可以添加一些自己的操作

System.out.println("before rent house");

System.out.println("Method:" + method);

// 当代理对象调用真实对象的方法时，其会自动的跳转到代理对象关联的handler对象的invoke方法来进行调用

method.invoke(subject, args);

//　　在代理真实对象后我们也可以添加一些自己的操作

System.out.println("after rent house");

return null;

}

IMG_276

[IMG_277](http://www.cnblogs.com/duanxz/archive/2012/12/03/javascript:void(0);" \o "复制代码)

我们看到，在真正通过代理对象来调用真实对象的方法的时候，我们可以在该方法前后添加自己的一些操作，同时我们看到我们的这个 method 对象是这样的：

public abstract void com.xiaoluo.dynamicproxy.Subject.rent()

public abstract void com.xiaoluo.dynamicproxy.Subject.hello(java.lang.String)

正好就是我们的Subject接口中的两个方法，这也就证明了当我通过代理对象来调用方法的时候，起实际就是委托由其关联到的 handler 对象的invoke方法中来调用，并不是自己来真实调用，而是通过代理的方式来调用的。

## **动态代理内部实现**

首先来看看类**Proxy**的代码实现 **Proxy**的主要静态变量

[IMG_278](http://www.cnblogs.com/duanxz/archive/2012/12/03/javascript:void(0);" \o "复制代码)

IMG_279

// 映射表：用于维护类装载器对象到其对应的代理类缓存

private static Map loaderToCache = new WeakHashMap();

// 标记：用于标记一个动态代理类正在被创建中

private static Object pendingGenerationMarker = new Object();

// 同步表：记录已经被创建的动态代理类类型，主要被方法 isProxyClass 进行相关的判断

private static Map proxyClasses = Collections.synchronizedMap(new WeakHashMap());

// 关联的调用处理器引用

protected InvocationHandler h;

IMG_280

[IMG_281](http://www.cnblogs.com/duanxz/archive/2012/12/03/javascript:void(0);" \o "复制代码)

**Proxy**的构造方法

// 由于 Proxy 内部从不直接调用构造函数，所以 private 类型意味着禁止任何调用

private Proxy() {}

// 由于 Proxy 内部从不直接调用构造函数，所以 protected 意味着只有子类可以调用

protected Proxy(InvocationHandler h) {this.h = h;}

**Proxy**静态方法**newProxyInstance**

[IMG_282](http://www.cnblogs.com/duanxz/archive/2012/12/03/javascript:void(0);" \o "复制代码)

IMG_283

public static Object newProxyInstance(ClassLoader loader, Class<?>[]interfaces,InvocationHandler h) throws IllegalArgumentException {

// 检查 h 不为空，否则抛异常

if (h == null) {

throw new NullPointerException();

}

// 获得与指定类装载器和一组接口相关的代理类类型对象

Class cl = getProxyClass(loader, interfaces);

// 通过反射获取构造函数对象并生成代理类实例

try {

Constructor cons = cl.getConstructor(constructorParams);

return (Object) cons.newInstance(new Object[] { h });

} catch (NoSuchMethodException e) { throw new InternalError(e.toString());

} catch (IllegalAccessException e) { throw new InternalError(e.toString());

} catch (InstantiationException e) { throw new InternalError(e.toString());

} catch (InvocationTargetException e) { throw new InternalError(e.toString());

}

}

IMG_284

[IMG_285](http://www.cnblogs.com/duanxz/archive/2012/12/03/javascript:void(0);" \o "复制代码)

类**Proxy**的**getProxyClass**方法调用**ProxyGenerator**的 **generateProxyClass**方法产生ProxySubject.class的二进制数据：

public static byte[] generateProxyClass(final String name, Class[] interfaces)

我们可以**import sun.misc.ProxyGenerator**，调用 **generateProxyClass**方法产生binary data，然后写入文件，最后通过反编译工具来查看内部实现原理。 反编译后的**ProxySubject.java** **Proxy**静态方法**newProxyInstance**

[IMG_286](http://www.cnblogs.com/duanxz/archive/2012/12/03/javascript:void(0);" \o "复制代码)

IMG_287

import java.lang.reflect.\*;

public final class ProxySubject extends Proxy

implements Subject

{

private static Method m1;

private static Method m0;

private static Method m3;

private static Method m2;

public ProxySubject(InvocationHandler invocationhandler)

{

super(invocationhandler);

}

public final boolean equals(Object obj)

{

try

{

return ((Boolean)super.h.invoke(this, m1, new Object[] {

obj

})).booleanValue();

}

catch(Error \_ex) { }

catch(Throwable throwable)

{

throw new UndeclaredThrowableException(throwable);

}

}

public final int hashCode()

{

try

{

return ((Integer)super.h.invoke(this, m0, null)).intValue();

}

catch(Error \_ex) { }

catch(Throwable throwable)

{

throw new UndeclaredThrowableException(throwable);

}

}

public final void doSomething()

{

try

{

super.h.invoke(this, m3, null);

return;

}

catch(Error \_ex) { }

catch(Throwable throwable)

{

throw new UndeclaredThrowableException(throwable);

}

}

public final String toString()

{

try

{

return (String)super.h.invoke(this, m2, null);

}

catch(Error \_ex) { }

catch(Throwable throwable)

{

throw new UndeclaredThrowableException(throwable);

}

}

static

{

try

{

m1 = Class.forName("java.lang.Object").getMethod("equals", new Class[] {

Class.forName("java.lang.Object")

});

m0 = Class.forName("java.lang.Object").getMethod("hashCode", new Class[0]);

m3 = Class.forName("Subject").getMethod("doSomething", new Class[0]);

m2 = Class.forName("java.lang.Object").getMethod("toString", new Class[0]);

}

catch(NoSuchMethodException nosuchmethodexception)

{

throw new NoSuchMethodError(nosuchmethodexception.getMessage());

}

catch(ClassNotFoundException classnotfoundexception)

{

throw new NoClassDefFoundError(classnotfoundexception.getMessage());

}

}

}

IMG_288

[IMG_289](http://www.cnblogs.com/duanxz/archive/2012/12/03/javascript:void(0);" \o "复制代码)

[ProxyGenerator](http://www.docjar.com/html/api/sun/misc/ProxyGenerator.java.html)内部是如何生成class二进制数据，可以参考源代码。

[IMG_290](http://www.cnblogs.com/duanxz/archive/2012/12/03/javascript:void(0);" \o "复制代码)

IMG_291

private byte[] generateClassFile() {

/\*

\* Record that proxy methods are needed for the hashCode, equals,

\* and toString methods of java.lang.Object. This is done before

\* the methods from the proxy interfaces so that the methods from

\* java.lang.Object take precedence over duplicate methods in the

\* proxy interfaces.

\*/

addProxyMethod(hashCodeMethod, Object.class);

addProxyMethod(equalsMethod, Object.class);

addProxyMethod(toStringMethod, Object.class);

/\*

\* Now record all of the methods from the proxy interfaces, giving

\* earlier interfaces precedence over later ones with duplicate

\* methods.

\*/

for (int i = 0; i < interfaces.length; i++) {

Method[] methods = interfaces[i].getMethods();

for (int j = 0; j < methods.length; j++) {

addProxyMethod(methods[j], interfaces[i]);

}

}

/\*

\* For each set of proxy methods with the same signature,

\* verify that the methods' return types are compatible.

\*/

for (List<ProxyMethod> sigmethods : proxyMethods.values()) {

checkReturnTypes(sigmethods);

}

/\* ============================================================

\* Step 2: Assemble FieldInfo and MethodInfo structs for all of

\* fields and methods in the class we are generating.

\*/

try {

methods.add(generateConstructor());

for (List<ProxyMethod> sigmethods : proxyMethods.values()) {

for (ProxyMethod pm : sigmethods) {

// add static field for method's Method object

fields.add(new FieldInfo(pm.methodFieldName,

"Ljava/lang/reflect/Method;",

ACC\_PRIVATE | ACC\_STATIC));

// generate code for proxy method and add it

methods.add(pm.generateMethod());

}

}

methods.add(generateStaticInitializer());

} catch (IOException e) {

throw new InternalError("unexpected I/O Exception");

}

/\* ============================================================

\* Step 3: Write the final class file.

\*/

/\*

\* Make sure that constant pool indexes are reserved for the

\* following items before starting to write the final class file.

\*/

cp.getClass(dotToSlash(className));

cp.getClass(superclassName);

for (int i = 0; i < interfaces.length; i++) {

cp.getClass(dotToSlash(interfaces[i].getName()));

}

/\*

\* Disallow new constant pool additions beyond this point, since

\* we are about to write the final constant pool table.

\*/

cp.setReadOnly();

ByteArrayOutputStream bout = new ByteArrayOutputStream();

DataOutputStream dout = new DataOutputStream(bout);

try {

/\*

\* Write all the items of the "ClassFile" structure.

\* See JVMS section 4.1.

\*/

// u4 magic;

dout.writeInt(0xCAFEBABE);

// u2 minor\_version;

dout.writeShort(CLASSFILE\_MINOR\_VERSION);

// u2 major\_version;

dout.writeShort(CLASSFILE\_MAJOR\_VERSION);

cp.write(dout); // (write constant pool)

// u2 access\_flags;

dout.writeShort(ACC\_PUBLIC | ACC\_FINAL | ACC\_SUPER);

// u2 this\_class;

dout.writeShort(cp.getClass(dotToSlash(className)));

// u2 super\_class;

dout.writeShort(cp.getClass(superclassName));

// u2 interfaces\_count;

dout.writeShort(interfaces.length);

// u2 interfaces[interfaces\_count];

for (int i = 0; i < interfaces.length; i++) {

dout.writeShort(cp.getClass(

dotToSlash(interfaces[i].getName())));

}

// u2 fields\_count;

dout.writeShort(fields.size());

// field\_info fields[fields\_count];

for (FieldInfo f : fields) {

f.write(dout);

}

// u2 methods\_count;

dout.writeShort(methods.size());

// method\_info methods[methods\_count];

for (MethodInfo m : methods) {

m.write(dout);

}

// u2 attributes\_count;

dout.writeShort(0); // (no ClassFile attributes for proxy classes)

} catch (IOException e) {

throw new InternalError("unexpected I/O Exception");

}

return bout.toByteArray();

IMG_292

[IMG_293](http://www.cnblogs.com/duanxz/archive/2012/12/03/javascript:void(0);" \o "复制代码)

## **总结**

一个典型的动态代理创建对象过程可分为以下四个步骤：  
1、通过实现InvocationHandler接口创建自己的调用处理器 IvocationHandler handler = new InvocationHandlerImpl(...);  
2、通过为Proxy类指定ClassLoader对象和一组interface创建动态代理类  
Class clazz = Proxy.getProxyClass(classLoader,new Class[]{...});  
3、通过反射机制获取动态代理类的构造函数，其参数类型是调用处理器接口类型  
Constructor constructor = clazz.getConstructor(new Class[]{InvocationHandler.class});  
4、通过构造函数创建代理类实例，此时需将调用处理器对象作为参数被传入  
Interface Proxy = (Interface)constructor.newInstance(new Object[] (handler));  
为了简化对象创建过程，Proxy类中的newInstance方法封装了2~4，只需两步即可完成代理对象的创建。  
生成的ProxySubject继承Proxy类实现Subject接口，实现的Subject的方法实际调用处理器的invoke方法，而invoke方法利用反射调用的是被代理对象的的方法（Object result=method.invoke(proxied,args)）

## **美中不足**

诚然，**Proxy**已经设计得非常优美，但是还是有一点点小小的遗憾之处，那就是它始终无法摆脱仅支持**interface**代理的桎梏，因为它的设计注定了这个遗憾。回想一下那些动态生成的代理类的继承关系图，它们已经注定有一个共同的父类叫**Proxy**。Java的继承机制注定了这些动态代理类们无法实现对class的动态代理，原因是多继承在Java中本质上就行不通。有很多条理由，人们可以否定对 class代理的必要性，但是同样有一些理由，相信支持class动态代理会更美好。接口和类的划分，本就不是很明显，只是到了Java中才变得如此的细化。如果只从方法的声明及是否被定义来考量，有一种两者的混合体，它的名字叫抽象类。实现对抽象类的动态代理，相信也有其内在的价值。此外，还有一些历史遗留的类，它们将因为没有实现任何接口而从此与动态代理永世无缘。如此种种，不得不说是一个小小的遗憾。但是，不完美并不等于不伟大，伟大是一种本质，Java动态代理就是佐例。

## **cglib**

jdk给目标类提供动态要求目标类必须实现接口，当一个目标类不实现接口时，jdk是无法为其提供动态代理的。cglib 却能给这样的类提供动态代理。

 详细见：[Java之代理（jdk静态代理，jdk动态代理，cglib动态代理，aop，aspectj）](http://www.cnblogs.com/duanxz/p/4410405.html)

## **Spring AOP**

详细见：[Spring AOP 实现原理](http://www.cnblogs.com/duanxz/archive/2012/12/03/2799636.html)

## **拦截器**

**[struts2拦截器的实现原理及源码剖析](http://www.cnblogs.com/duanxz/archive/2012/12/03/2800004.html)**