转载自IBM developerworks：

https://www.ibm.com/developerworks/cn/java/j-lo-proxy1/

https://www.ibm.com/developerworks/cn/java/j-lo-proxy2/

# 引言

Java动态代理机制的出现，使得Java开发人员不用手工编写代理类，只要简单地指定一组接口及委托类对象，便能动态地获得代理类。代理类会负责将所有的方法调用分派到委托对象上反射执行，在分派执行的过程中，开发人员还可以按需调整委托类对象及功能，这是一套非常灵活有弹性的代理框架。

# 代理：设计模式

代理是一种常用的设计模式，其目的就是为其他对象提供一个代理以控制对某个对象的访问。代理类负责为委托类预处理消息、过滤消息并转发消息，以及进行消息被委托类执行后的后续处理。

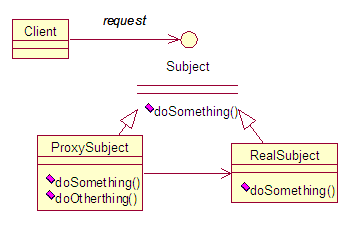


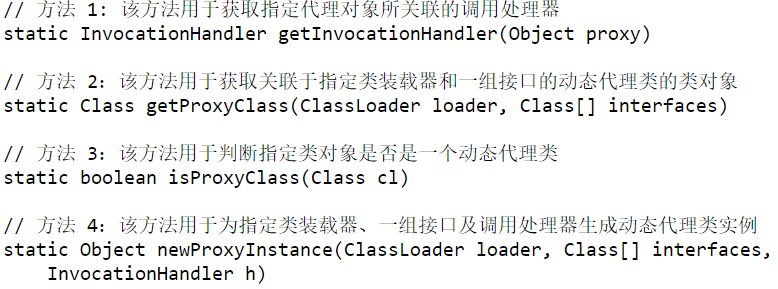
图1 代理模式

为了保持行为的一致性，代理类和委托类通常会实现相同的接口，所以在访问者看来，两者没有丝毫的区别。通过代理类这一中间层，能有效控制对委托类对象的访问，也可以很好地隐藏和保护委托类对象，同时也为实施不同控制策略预留了空间，从而在设计上获得了更大的灵活性。Java动态代理机制以巧妙的方式近乎完美地实践了代理模式的设计理念。

# 相关的类和接口

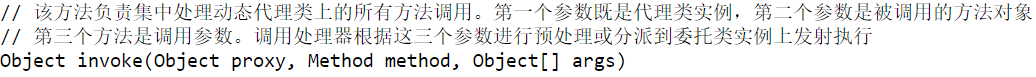
要了解Java动态代理机制，首先需要了解以下相关的类或接口：

* java.lang.reflect.Proxy：这是Java动态代理机制的主类，它提供了一组静态方法来为一组接口动态地生成代理类及其对象。



清单1 Proxy的静态方法

* java.lang.reflect.InvocatoinHandler：这是调用处理器接口，它自定义了一个invoke方法，用于集中处理在动态代理类对象上的方法调用，通常在该方法中实现对委托类的代理访问。



清单2 InvocationHandler的核心方法

* java.lang.ClassLoader：这是类装载器类，负责将类的字节码加载到JVM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装载器来进行装载才能使用，它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，而非预存在任何一个.class文件中。

# 代理机制及其特点

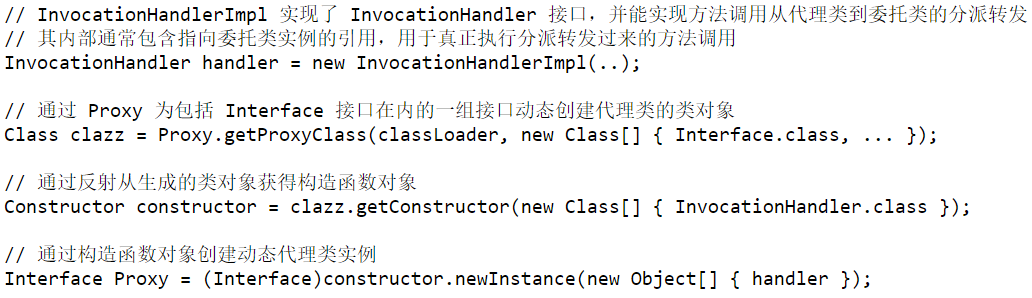
首先让我们来了解一下如何使用Java动态代理，具体有如下四个步骤：

1、通过实现InvocatioinHandler接口创建自己的调用处理器；

2、通过为Proxy类指定ClassLoader对象和一组Interface来创建动态代理类；

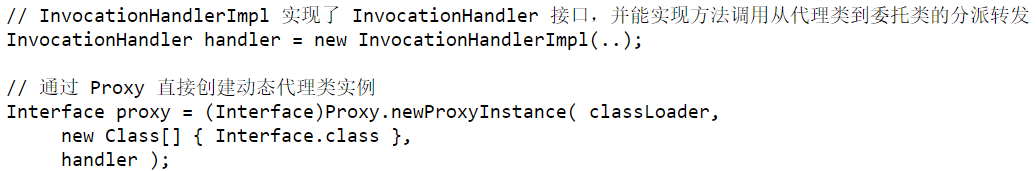
3、通过反射机制获得动态代理类的构造函数，其唯一参数类型为调用处理器接口类型；

4、通过构造函数创建动态代理类实例，构造时调用处理器对象作为参数被传入。



清单3 动态代理对象创建过程

实际使用过程更加简单，因为Proxy的静态方法newProxyInstance已经为我们封装了步骤2到步骤4的过程，所以简化后的过程如下：



清单4 简化的动态代理对象创建过程

接下来让我们了解一下Java动态代理的一些特点。

首先是动态生成代理类本身的一些特点。

（1）包：如果所代理的接口都是public的，那么它被定义在顶层包（即包路径为空）；如果所代理的接口中有非public的（因为接口不能被定义为protect或private，所以除public之外就是默认的package访问级别），那么它将被定义在该接口所在包（假设代理了com.ibm.developerworks包中的某非public接口A，那么新生成的代理类所在的包就是com.ibm.developerworks），这样的设计目的是为了最大程度的保证动态代理类不会因为包管理的问题而无法被成功定义并访问。

（2）修饰符：该代理类具有final和public修饰符，意味着它可以被所有的类访问，但是不能被再度继承。

（3）类名：格式是"$ProxyN"，其中N是一个逐一递增的阿拉伯数字，代表Proxy类第N次生成的动态代理类，值得注意的一点是，并不是每次调用Proxy的静态方法创建动态代理类都会使得N值增加，原因是如果对同一组接口（接口的排列顺序也相同）试图重复创建动态代理类，它会很聪明地返回先前已经创建好的代理类的类对象，而不会再尝试去创建一个全新的代理类，这样可以节省不必要的代码重复生成，提高了代理类的创建效率。

（4）类继承关系：该类的继承关系如图：

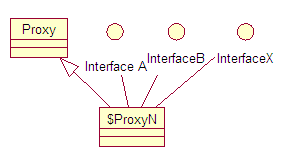


图2 动态代理类的继承图

由图可见，Proxy类是它的父类，这个规则适用于所有由Proxy创建的动态代理类。另外，该类还实现了其所代理的一组接口，这就是为什么它能够被安全地类型转换到其他所代理的接口的根本原因。

然后让我们了解一下代理类实例的一些特点。

（1）每个实例都会关联一个调用处理器对象，可以通过Proxy提供的静态方法getInvacationHandler去获得代理类实例的调用处理器对象。在代理类实例上调用其代理的接口中所声明的方法时，这些方法最终都会由调用处理器的invoke方法执行。

（2）此外，值得注意的是，代理类的根类java.lang.Object中有三个方法也同样会被分派到调用处理器的invoke方法执行，它们是hashCode、equals和toString，可能的原因是：一是因为这些方法为public且非final类型，能够被代理类覆盖；二是因为这些方法往往呈现出一个类的某种特征属性，具有一定的区分度，所以为了保证代理类与委托类对外的一致性，这三个方法也应该被分派到委托类执行。

（3）当代理的一组接口有重复声明的方法，且该方法被调用时，代理类总是从排在最前面的接口中获取方法对象并分派给调用处理器，而无论代理类实例是否正在以该接口（或者是以继承于该接口的某子接口）的形式被外部引用，因为在代理类内部无法区分其当前的被引用类型。

接下来了解一下被代理的一组接口有哪些特点。

（1）不能有重复的接口，以避免动态代理类代码生成时的编译错误。

（2）这些接口对于类装载器必须可见，否则类装载器将无法链接它们，从而导致类定义失败。

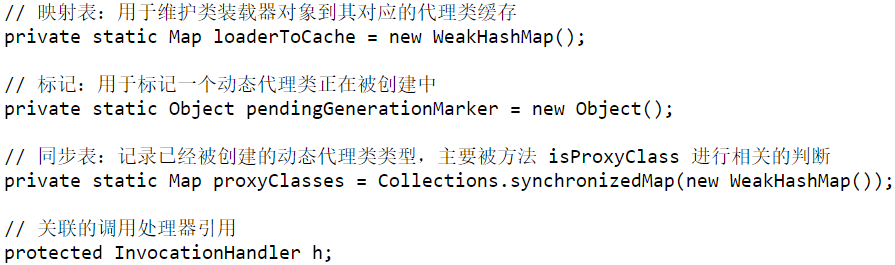
（3）需要被代理的所有非public接口必须在同一个包中，否则代理类也会生成失败。

（4）接口的数目不能超过65535个，这是JVM设定的限制。

最后来了解一下异常处理方面的特点。InvocationHandler的invoke方法理论上可以抛出任意类型的异常，因为所有的异常都继承自Throwable接口，但实际上并非如此，因为我们必须遵守一个继承原则：子类覆盖父类或实现父接口的方法时，抛出的异常必须在原方法支持的异常列表之内。因此虽然调用处理器理论上可以抛出任意类型的异常，但实际上往往受到限制，除非父接口的方法支持抛出Throwable异常。如果在invoke方法产生了接口方法声明中不支持的异常，Java动态代理给出的解决方案是：它将会抛出UndeclaredThrowableException异常，这个异常是一个RuntimeException类型，因此不会引起编译错误，通过该异常的getCause方法还可以获得原来的那个不受支持的异常对象，以便于错误诊断。

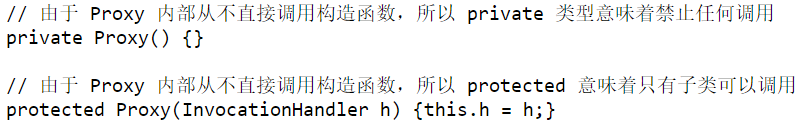
# Proxy源码分析

首先记住Proxy的几个重要的静态变量。



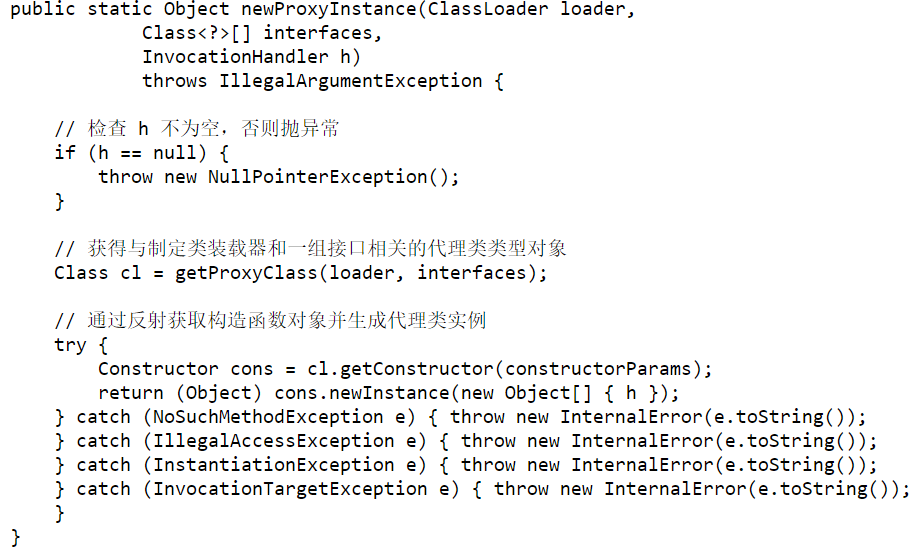
清单5 Proxy的重要静态变量

然后来看一下Proxy的构造方法。



清单6 Proxy构造方法

接着看一下newProxyInstance方法。

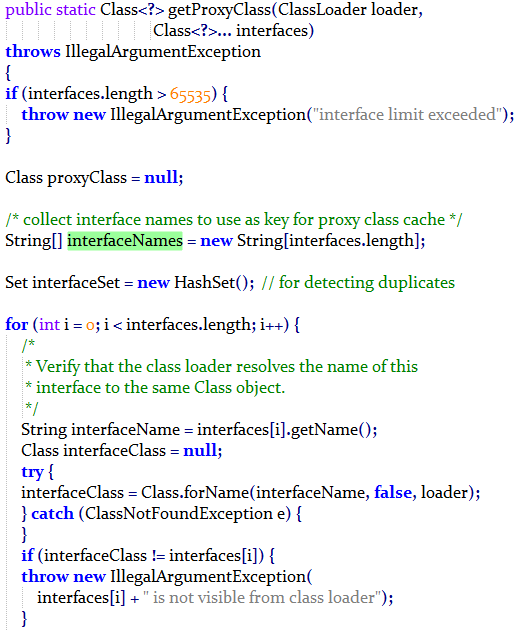


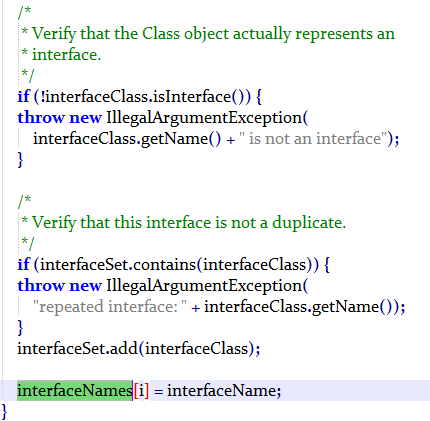
清单7 Proxy静态方法newProxyInstance

由此可见，动态代理真正的关键在于getProxyClass方法，该方法负责为一组接口动态地生成代理类类型对象。

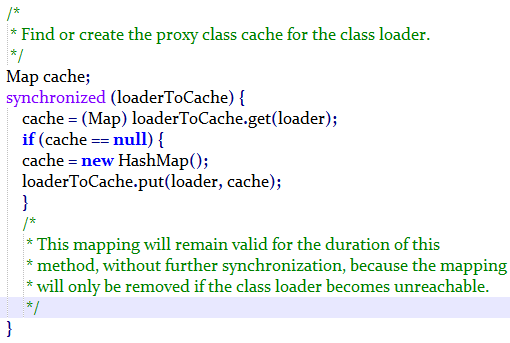
下面我们分析getProxyClass方法，该方法总共可以分为四个步骤：

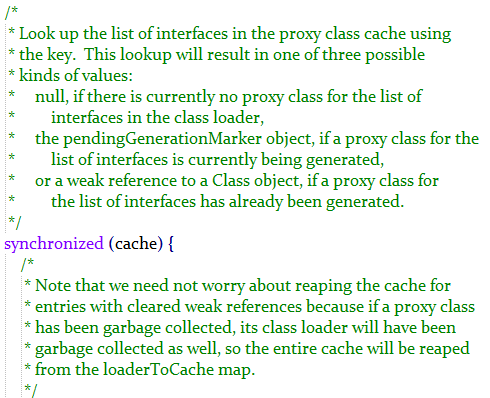
1、对这组接口进行一定程序的安全检查，包括检查接口类对象是否对类装载器可见，并且该接口类对象是否与类装载器所能识别的接口类对象是完全相同的；检查确保是interfaces类型而不是class类型；检查接口名是否重复。如果检查通过，则最后得到一个包含所有接口名称的字符串数组。



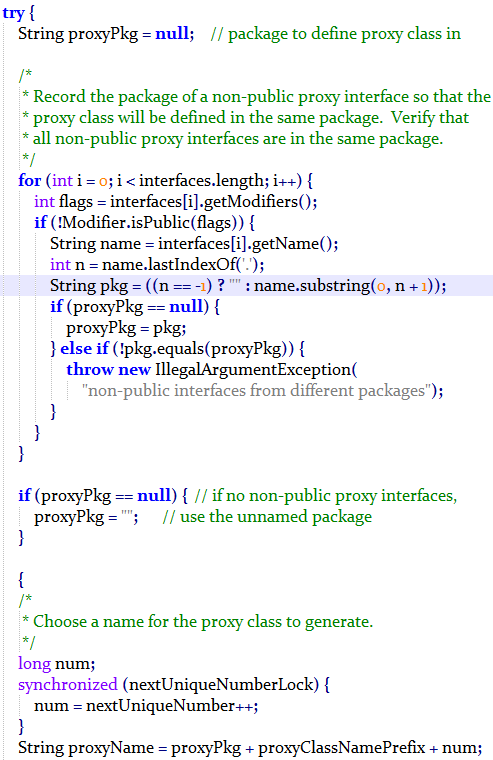


2、从loaderToCache映射表中获取以类装载器对象为关键字所对应的缓存表，如果不存在就创建一个新的缓存表并更新到loaderToCache。缓存表是一个HashMap实例，正常情况下，缓存表中的键是接口名称列表，值是动态生成的代理类的类对象引用；当代理类正在被创建时，缓存表中的键是接口名称列表，值是pendingGenerationMarker，其中pendingGenerationMarker的作用是通知后续的同类请求（即接口数组相同且接口排列顺序也相同）代理类正在被创建，并该请求等待直至创建完成。

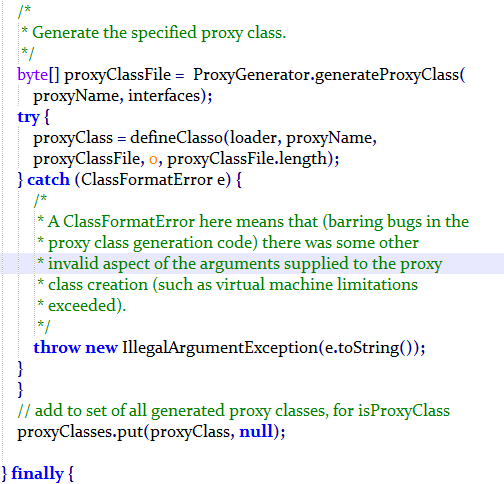




3、动态创建代理类的类对象。首先要确定代理类所在的包，其原则如前所述：如果都为public接口，则包名为空字符串表示顶层包；如果所有非public接口都在同一个包，则包名与这些接口的包名相同；如果有多个非public接口且不同包，则抛异常终止代理类的生成。确定了包后，就开始生成代理类的类名，同样如前所述按格式“$ProxyN”生成。

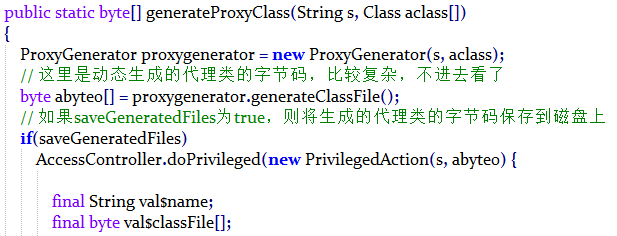


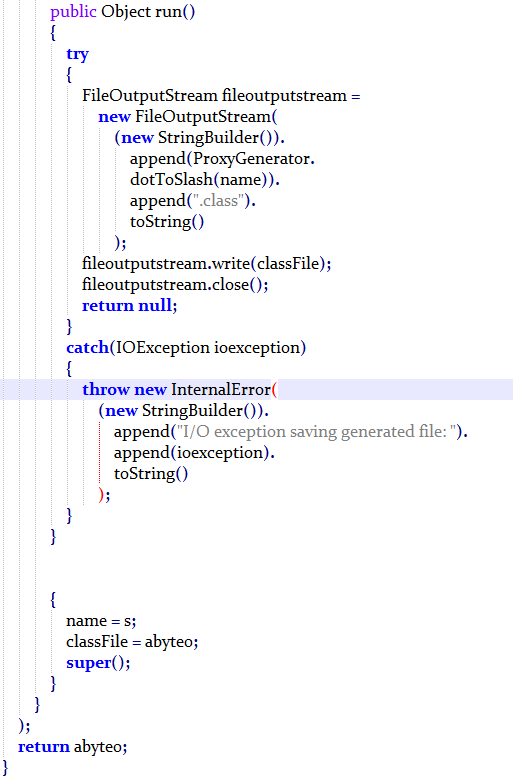
4、包名确定了，类名也确定了，接下来就是动态生成代理类了。



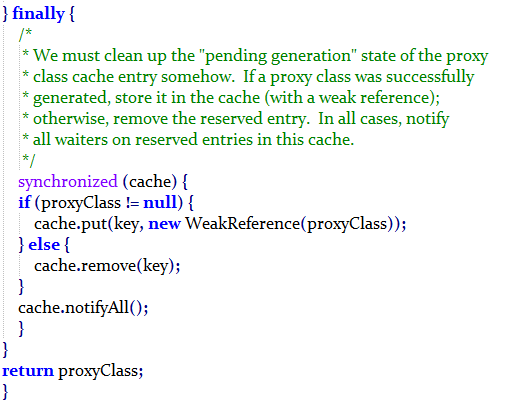
由此可见，所有的代码生成工作均由ProxyGenerator完成，该类在rt.jar中的sum.misc包中，sun没有提供源码以供研读。至于动态类的定义，则由Proxy的native静态方法definneClass0完成。

我们可以反编译一下ProxyGenerator，结果如下：





5、代码生成过程的结尾阶段是根据结果更新缓存表，如果成功则将代理类的类对象引用更新进缓存表，否则清除缓存表中对应的关键值，最后唤醒所有可能的处于等待状态的线程。



# InvocationHandler源码分析

我们来看一下究竟由谁来调用InvocationHandler的invoke方法。要解决这个问题，就需要看一下jdk为我们生成的代理类字节码到底是什么样的。用以下代码可以将jdk生成的代理类字节码写入到磁盘中。

package com.kevin.decompile;  
  
import sun.misc.ProxyGenerator;  
  
import java.io.FileNotFoundException;  
import java.io.FileOutputStream;  
import java.io.IOException;  
  
*/\*\*  
 \** ***@author*** *kevin  
 \** ***@version*** *1.0  
 \** ***@des*** *代理类生成工具  
 \** ***@data*** *2017/5/30 23:46  
 \*/*public class ProxyGeneratorUtils {  
  
 */\*\*  
 \* 将代理类的字节码写到磁盘上  
 \** ***@param*** *path 保存路径  
 \*/* public static void writeProxyClassToDisk(String path) {  
 // 第一种方法，根据反编译ProxyGenerator.class可知，将saveGeneratedFiles设置为true即可  
 System.*getProperties*().put("sun.misc.ProxyGenerator.saveGeneratedFiles", true);  
  
 // 第二种方法  
 byte[] classFile = ProxyGenerator.*generateProxyClass*("$Proxy4", UserServiceImpl.class.getInterfaces());  
 FileOutputStream out = null;  
 try {  
 out = new FileOutputStream(path);  
 out.write(classFile);  
 out.flush();  
 } catch (FileNotFoundException e) {  
 e.printStackTrace();  
 } catch (IOException e) {  
 e.printStackTrace();  
 } finally {  
 try {  
 out.close();  
 } catch (IOException e) {  
 e.printStackTrace();  
 }  
 }  
 }  
}

测试用例如下：

package com.kevin.aop;  
  
import com.kevin.decompile.MyInvocationHandler;  
import com.kevin.decompile.ProxyGeneratorUtils;  
import com.kevin.decompile.UserService;  
import com.kevin.decompile.UserServiceImpl;  
import org.junit.Test;  
  
*/\*\*  
 \** ***@author*** *kevin  
 \** ***@version*** *1.0  
 \** ***@des*** *\** ***@data*** *2017/5/30 23:53  
 \*/*public class TestDecompile {  
  
 @Test  
 public void testProxy() {  
 UserService userService = new UserServiceImpl();  
 MyInvocationHandler invocationHandler = new MyInvocationHandler(userService);  
 UserService proxy = (UserService) invocationHandler.getProxyInstance();  
 proxy.add();  
 }  
  
 @Test  
 public void testCompile() {  
 String path = "D:\\Workspace\\aop\\src\\main\\java\\com\\kevin\\decompile\\$Proxy4.class";  
 ProxyGeneratorUtils.*writeProxyClassToDisk*(path);  
 }  
}

运行以上代码，就可以在相应目录下生成一个$Proxy4.class文件，我们用反编译工具来看一下这个class文件中的内容。

//  
// Source code recreated from a .class file by IntelliJ IDEA  
// (powered by Fernflower decompiler)  
//  
  
import com.kevin.decompile.UserService;  
import java.lang.reflect.InvocationHandler;  
import java.lang.reflect.Method;  
import java.lang.reflect.Proxy;  
import java.lang.reflect.UndeclaredThrowableException;  
  
public final class $Proxy4 extends Proxy implements UserService {  
 private static Method m1;  
 private static Method m2;  
 private static Method m3;  
 private static Method m0;

// 构造方法，参数就是刚才传进来的MyInvocationHandler类实例  
 public $Proxy4(InvocationHandler var1) throws {  
 super(var1);  
 }  
  
 public final boolean equals(Object var1) throws {  
 try {

// 调用invocatoinHandler中的invoke方法  
 return ((Boolean)super.h.invoke(this, m1, new Object[]{var1})).booleanValue();  
 } catch (RuntimeException | Error var3) {  
 throw var3;  
 } catch (Throwable var4) {  
 throw new UndeclaredThrowableException(var4);  
 }  
 }  
  
 public final String toString() throws {  
 try {

// 调用invocatoinHandler中的invoke方法  
 return (String)super.h.invoke(this, m2, (Object[])null);  
 } catch (RuntimeException | Error var2) {  
 throw var2;  
 } catch (Throwable var3) {  
 throw new UndeclaredThrowableException(var3);  
 }  
 }  
  
 public final void add() throws {  
 try {

// 调用invocatoinHandler中的invoke方法  
 super.h.invoke(this, m3, (Object[])null);  
 } catch (RuntimeException | Error var2) {  
 throw var2;  
 } catch (Throwable var3) {  
 throw new UndeclaredThrowableException(var3);  
 }  
 }  
  
 public final int hashCode() throws {  
 try {

// 调用invocatoinHandler中的invoke方法  
 return ((Integer)super.h.invoke(this, m0, (Object[])null)).intValue();  
 } catch (RuntimeException | Error var2) {  
 throw var2;  
 } catch (Throwable var3) {  
 throw new UndeclaredThrowableException(var3);  
 }  
 }

// 在静态代码块中获取了4个方法，Object中的equals、hashCode、

// toString方法和UserService的add方法  
 static {  
 try {  
 m1 = Class.forName("java.lang.Object").getMethod("equals", new Class[]{Class.forName("java.lang.Object")});  
 m2 = Class.forName("java.lang.Object").getMethod("toString", new Class[0]);  
 m3 = Class.forName("com.kevin.decompile.UserService").getMethod("add", new Class[0]);  
 m0 = Class.forName("java.lang.Object").getMethod("hashCode", new Class[0]);  
 } catch (NoSuchMethodException var2) {  
 throw new NoSuchMethodError(var2.getMessage());  
 } catch (ClassNotFoundException var3) {  
 throw new NoClassDefFoundError(var3.getMessage());  
 }  
 }  
}

# Proxy的不足之处

虽然Proxy已经设计得十分优美，但是还是有一点点小遗憾，那就是它始终只支持interface代理。因为通过Proxy动态生成的代理类一定有一个父类叫Proxy，Java的继承机制注定了这些动态代理类无法实现对class的动态代理，原因是Java不支持多继承。

当然，不完美并不代表不伟大，伟大是一种本质，Java动态代理就是如此。