说起java动态代理，在我刚开始学java时对这项技术也是十分困惑，明明可以直接调通的对象方法为什么还要使用动态代理？随着学习的不断深入和工作经验的积累，慢慢的体会并理解了java动态代理机制。昨天再给公司新同事做技术培训时有同学就对动态代理产生了疑问，我这里梳理一遍一并记录一下，方便大家查看对自己也是加深记忆。

(1)什么是代理？

大道理上讲代理是一种软件设计模式，目的地希望能做到代码重用。具体上讲，代理这种设计模式是通过不直接访问被代理对象的方式，而访问被代理对象的方法。这个就好比 商户---->明星经纪人(代理)---->明星这种模式。我们可以不通过直接与明星对话的情况下，而通过明星经纪人(代理)与其产生间接对话。

(2)什么情况下使用代理？

(1)设计模式中有一个设计原则是开闭原则，是说对修改关闭对扩展开放，我们在工作中有时会接手很多前人的代码，里面代码逻辑让人摸不着头脑(sometimes the code is really like shit)，这时就很难去下手修改代码，那么这时我们就可以通过代理对类进行增强。

(2)我们在使用RPC框架的时候，框架本身并不能提前知道各个业务方要调用哪些接口的哪些方法 。那么这个时候，就可用通过动态代理的方式来建立一个中间人给客户端使用，也方便框架进行搭建逻辑，某种程度上也是客户端代码和框架松耦合的一种表现。

(3)Spring的AOP机制就是采用动态代理的机制来实现切面编程。

(3)静态代理和动态代理

我们根据加载被代理类的时机不同，将代理分为静态代理和动态代理。如果我们在代码编译时就确定了被代理的类是哪一个，那么就可以直接使用静态代理；如果不能确定，那么可以使用类的动态加载机制，在代码运行期间加载被代理的类这就是动态代理，比如RPC框架和Spring AOP机制。

(4)静态代理

我们先创建一个接口，遗憾的是java api代理机制求被代理类必须要实现某个接口，对于静态代理方式代理类也要实现和被代理类相同的接口；对于动态代理代理类则不需要显示的实现被代理类所实现的接口。

/\*\*

\* 顶层接口

\* @author yujie.wang

\*

\*/

public interface Person {

public void sayHello(String content, int age);

public void sayGoodBye(boolean seeAgin, double time);

}

/\*\*

\* 需要被代理的类 实现了一个接口Person

\* @author yujie.wang

\*

\*/

public class Student implements Person{

@Override

public void sayHello(String content, int age) {

// TODO Auto-generated method stub

System.out.println("student say hello" + content + " "+ age);

}

@Override

public void sayGoodBye(boolean seeAgin, double time) {

// TODO Auto-generated method stub

System.out.println("student sayGoodBye " + time + " "+ seeAgin);

}

}

/\*\*

\* 静态代理，这个代理类也必须要实现和被代理类相同的Person接口

\* @author yujie.wang

\*

\*/

public class ProxyTest implements Person{

private Person o;

public ProxyTest(Person o){

this.o = o;

}

public static void main(String[] args) {

// TODO Auto-generated method stub

//s为被代理的对象，某些情况下 我们不希望修改已有的代码，我们采用代理来间接访问

Student s = new Student();

//创建代理类对象

ProxyTest proxy = new ProxyTest(s);

//调用代理类对象的方法

proxy.sayHello("welcome to java", 20);

System.out.println("\*\*\*\*\*\*");

//调用代理类对象的方法

proxy.sayGoodBye(true, 100);

}

@Override

public void sayHello(String content, int age) {

// TODO Auto-generated method stub

System.out.println("ProxyTest sayHello begin");

//在代理类的方法中 间接访问被代理对象的方法

o.sayHello(content, age);

System.out.println("ProxyTest sayHello end");

}

@Override

public void sayGoodBye(boolean seeAgin, double time) {

// TODO Auto-generated method stub

System.out.println("ProxyTest sayHello begin");

//在代理类的方法中 间接访问被代理对象的方法

o.sayGoodBye(seeAgin, time);

System.out.println("ProxyTest sayHello end");

}

}

测试代码输出：

ProxyTest sayHello begin

student say hellowelcome to java 20

ProxyTest sayHello end

\*\*\*\*\*\*

ProxyTest sayHello begin

student sayGoodBye 100.0 true

ProxyTest sayHello end

静态代理看起来是比较简单的，没有什么问题只不过是在代理类中引入了被代理类的对象而已。

那么接下来我们看看动态代理。

(5)动态代理

我们先直接上动态代理的代码，之后再分析代码的行为，上面的Person接口和Student被代理类保持不变。

/\*\*

\* 动态代理,动态代理类不要显示的实现被代理类所实现的接口

\* @author yujie.wang

\*

\*/

public class MyInvocationHandler implements InvocationHandler{

private Object object;

public MyInvocationHandler(Object object){

this.object = object;

}

@Override

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

throws Throwable {

// TODO Auto-generated method stub

System.out.println("MyInvocationHandler invoke begin");

System.out.println("proxy: "+ proxy.getClass().getName());

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

for(Object o : args){

System.out.println("arg: "+ o);

}

//通过反射调用 被代理类的方法

method.invoke(object, args);

System.out.println("MyInvocationHandler invoke end");

return null;

}

public static void main(String [] args){

//创建需要被代理的类

Student s = new Student();

//这一句是生成代理类的class文件，前提是你需要在工程根目录下创建com/sun/proxy目录，不然会报找不到路径的io异常

System.getProperties().put("sun.misc.ProxyGenerator.saveGeneratedFiles","true");

//获得加载被代理类的 类加载器

ClassLoader loader = Thread.currentThread().getContextClassLoader();

//指明被代理类实现的接口

Class<?>[] interfaces = s.getClass().getInterfaces();

// 创建被代理类的委托类,之后想要调用被代理类的方法时，都会委托给这个类的invoke(Object proxy, Method method, Object[] args)方法

MyInvocationHandler h = new MyInvocationHandler(s);

//生成代理类

Person proxy = (Person)Proxy.newProxyInstance(loader, interfaces, h);

//通过代理类调用 被代理类的方法

proxy.sayHello("yujie.wang", 20);

proxy.sayGoodBye(true, 100);

System.out.println("end");

}

}

运行测试代码输出如下结果：

MyInvocationHandler invoke begin

proxy: com.sun.proxy.$Proxy0

method: sayHello

arg: yujie.wang

arg: 20

student say helloyujie.wang 20

MyInvocationHandler invoke end

MyInvocationHandler invoke begin

proxy: com.sun.proxy.$Proxy0

method: sayGoodBye

arg: true

arg: 100.0

student sayGoodBye 100.0 true

MyInvocationHandler invoke end

end

仔细分析上面的动态代理实现代码，我们看到这里涉及到java反射包下的一个接口InvocationHandler和一个类Proxy。

package java.lang.reflect;

public interface InvocationHandler {

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

throws Throwable;

}

这个接口只有一个invoke方法，我们在通过代理类调用被代理类的方法时，最终都会委托给这个invoke方法执行，

//通过代理类调用 被代理类的方法

proxy.sayHello("yujie.wang", 20);

proxy.sayGoodBye(true, 100);

所以我们就可以在这个invoke方法中对被代理类进行增强或做一些其他操作。

Proxy类的public static Object newProxyInstance(ClassLoader loader,Class<?>[] interfaces,InvocationHandler h)方法内部通过拼接字节码的方式来创建代理类，后面我会反编译出它所创建的代理类看看内容。

我们看这个方法的三个参数：

ClassLoader loader：指定一个动态加载代理类的类加载器

Class<?>[] interfaces：指明被代理类实现的接口，之后我们通过拼接字节码生成的类才能知道调用哪些方法。

InvocationHandler h：这是一个方法委托类，我们通过代理调用被代理类的方法时，就可以将方法名和方法参数都委托给这个委托类。

你看我们现在有了类加载器、类实现的接口、要调用方法的方法名和参数，那么我们就可以做很多事情了。

(6)反编译Proxy.newProxyInstance所创建的代理类

//这一句是生成代理类的class文件，前提是你需要在工程根目录下创建com/sun/proxy目录，不然会报找不到路径的io异常

System.getProperties().put("sun.misc.ProxyGenerator.saveGeneratedFiles","true");

我们在代码中加入上述代码，代码就会保存生成的代理类，名称为$Proxy0.class

通过jd-gui反编译代码如下，其中注释是我加上去的：

package com.sun.proxy;

import com.yujie.proxy.dynamic.Person;

import java.lang.reflect.InvocationHandler;

import java.lang.reflect.Method;

import java.lang.reflect.Proxy;

import java.lang.reflect.UndeclaredThrowableException;

/\*\*

\*代理类也实现了Person接口，看起来和静态代理的方式也会一样的

\*同时代理类也继承了Proxy类

\*/

public final class $Proxy0 extends Proxy implements Person{

private static Method m4;

private static Method m1;

private static Method m0;

private static Method m3;

private static Method m2;

public $Proxy0(InvocationHandler paramInvocationHandler)

throws

{

super(paramInvocationHandler);

}

//实现了Person接口的方法，这就是我们调用这个方法Proxy.newProxyInstance必须提供第二个参数的作用

public final void sayGoodBye(boolean paramBoolean, double paramDouble)

throws

{

try

{

// 我们看到通过调用代理类的方法时，最终方法都会委托给InvocationHandler实现类的invoke方法

// m4为代理类通过反射获得的Method

this.h.invoke(this, m4, new Object[] { Boolean.valueOf(paramBoolean), Double.valueOf(paramDouble) });

return;

}

catch (Error|RuntimeException localError)

{

throw localError;

}

catch (Throwable localThrowable)

{

throw new UndeclaredThrowableException(localThrowable);

}

}

public final boolean equals(Object paramObject)

throws

{

try

{

return ((Boolean)this.h.invoke(this, m1, new Object[] { paramObject })).booleanValue();

}

catch (Error|RuntimeException localError)

{

throw localError;

}

catch (Throwable localThrowable)

{

throw new UndeclaredThrowableException(localThrowable);

}

}

public final int hashCode()

throws

{

try

{

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

}

catch (Error|RuntimeException localError)

{

throw localError;

}

catch (Throwable localThrowable)

{

throw new UndeclaredThrowableException(localThrowable);

}

}

//实现了Person接口的方法，这就是我们调用这个方法Proxy.newProxyInstance必须提供第二个参数的作用

public final void sayHello(String paramString, int paramInt)

throws

{

try

{

// 我们看到通过调用代理类的方法时，最终方法都会委托给InvocationHandler实现类的invoke方法

// m4为代理类通过反射获得的Method

this.h.invoke(this, m3, new Object[] { paramString, Integer.valueOf(paramInt) });

return;

}

catch (Error|RuntimeException localError)

{

throw localError;

}

catch (Throwable localThrowable)

{

throw new UndeclaredThrowableException(localThrowable);

}

}

public final String toString()

throws

{

try

{

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

}

catch (Error|RuntimeException localError)

{

throw localError;

}

catch (Throwable localThrowable)

{

throw new UndeclaredThrowableException(localThrowable);

}

}

static

{

try

{//代理类通过反射 获得的接口方法Method

m4 = Class.forName("com.yujie.proxy.dynamic.Person").getMethod("sayGoodBye", new Class[] { Boolean.TYPE, Double.TYPE });

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]);

//代理类通过反射 获得的接口方法Method

m3 = Class.forName("com.yujie.proxy.dynamic.Person").getMethod("sayHello", new Class[] { Class.forName("java.lang.String"), Integer.TYPE });

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

return;

}

catch (NoSuchMethodException localNoSuchMethodException)

{

throw new NoSuchMethodError(localNoSuchMethodException.getMessage());

}

catch (ClassNotFoundException localClassNotFoundException)

{

throw new NoClassDefFoundError(localClassNotFoundException.getMessage());

}

}

}

总结一下：

jdk的代理让我们在不直接访问某些对象的情况下，通过代理机制也可以访问被代理对象的方法，这种技术可以应用在很多地方比如RPC框架，Spring AOP机制，但是我们看到jdk的代理机制必须要求被代理类实现某个方法，这样在生成代理类的时候才能知道重新那些方法。这样一个没有实现任何接口的类就无法通过jdk的代理机制进行代理，当然解决方法是使用cglib的代理机制进行代理。

————————————————

版权声明：本文为CSDN博主「英雄郑旺」的原创文章，遵循 CC 4.0 BY-SA 版权协议，转载请附上原文出处链接及本声明。

原文链接：https://blog.csdn.net/u011784767/article/details/78281384