本文侧重不在于讲什么是动态代理，而是说明muppet中是如何封装动态代理，以便于解耦与快速的开发

对动态代理比较熟悉的可以直接跳到第三部分：muppet如何封装动态代理的

1.何为动态代理

1.1 代理与静态代理

代理中分被代理和代理，很多情况下，我们想控制和目标对象的交互，例如：古代曹操挟天子以令诸侯，汉献帝是被代理的，曹操是代理，曹操想代理皇帝做某些事情，就必须把皇帝代理起来，于是就把汉献帝迎来放在笼子里（皇宫），然后皇上想发个诏书，必须经过曹丞相，下面人给皇帝发奏文同样也是曹操先拦截下。

可见代理有两个基本要素：1）代理持有被代理对象，被代理类是被代理类控制的，2）外部要想和目标对象有交互，必须先经过代理对象。这就是代理模式要完成的目标。这个目标不会凭空实现，下面说明下如何实现这个目标

用一个日志记录例子说明下：

如果我想记录这个类中方法的调用日志，即何时调用了这个类的哪个方法

//目标接口即被代理的接口

interface TargetInterface{

public void target1();

public void target2();

}

//有一个类实现了这个接口，现在我们要针对这个类的对象进行代理

Class TargetClass implements TargetInterface{

public void target1(){

System.out.println("target1调用");

}

public void target2(){

}

}

如果我想获取一个实现了TargetInterface接口的 对象，怎么办

public TargetInterface getTargetInterface(){

TargetInterface targetClass = new TargetClass();

return targetClass;

}

//于是返回了一个 TargetClass对象，这个对象实现了TargetInterface接口

//为了代理它，我必须也得装成TargetInterface一样。

//先实现这个接口，但是至于具体请求转发给TargetClass类对象实现

Class ProxyClass implements TargetInterface{

private TargetInterface targetClass;

public ProxyClass(TargetInterface targetClass){

this.targetClass = targetClass;

}

public void target1(){

System.out.println("调用了target1方法");

this.targetClass.target1();

}

public void target2(){

this.targetClass.target2();

}

}

现在我要实现代理TargetClass对象的目标

public TargetInterface getTargetInterface(){

TargetInterface targetClass = new TargetClass();//被代理对象已经出现

return proxyClass = new ProxyClass(targetClass);//代理对象也出现，同时代理对象将被代理对象控制起来

}

通过此方法获取到的targetInterface接口实例，已经不是TargetClass类型对象了，而是ProxyInterface类型对象，

但是外部看不到，他们只知道他们获取了一个 TargetInterface类型实例，而不在乎获取的对象到底是TargetClass,还是

TargetProxy类型，但是ProxyClass却是代理了TargetClass.现在TargetClass的方法访问都想经过ProxyClass这一关。

本来是TargetClass声明我实现了TargetInterface，现在ProxyClass也声明自己实现了TargetInterface解耦，但是它是在

获取TargetClass对象后，将其封装起来，实现了对代理对象的控制，但是对于外界这一切都是透明的，因为他们都实现了

TargetInterface接口。

这就是代理：

也是静态代理，静态的原因是任何一个接口我们都需要提供一个代理类来实现，但是我们看到这些代码都是非常简单的。

先不提动态代理，通过这个例子，我们也可以总结下，

代理模式的弱点：

代理对象通过持有被代理的引用，控制记录被代理方法的调用，但是，代理对象并不知道被代理对象内部实现的逻辑，

不清楚被代理对象在方法体中做了什么，所以注定代理对象在其方法体中只能做一些与被代理对象中的逻辑无关的一些操作，

如果逻辑有关，那么代理对象如何知道呢？因为不知道所以只能做一些无关的操作，例如，日志记录，权限拦截，访问控制，

异常拦截。等通用性功能。也就是说代理模式更多实现的功能点是与业务关联不大的操作。也就是与接口关联不大的操作，

那么如何实现对所有接口的代理呢？

这就引出了动态代理：

1.2 动态代理

动态代理可以一次代理许多许多接口，但是前提是这些 接口被代理的原因是相同的，也就是都是为了记录日志，权限控制。

2.如何实现动态代理

2.1 jdk的实现

jdk中提供了动态代理的支持，但是功能有限

示例代码如下；

class DefaultBindInvocationHandler implements InvocationHandler{

public Object bind(Object target)

throws NullPointerException,IllegalArgumentException{

this.target = (target);//持有被代理对象

/\*\*

\*是否实现了接口，如果没有实现接口，jdk是无法代理的，这个方法的第二个参数

\*要求不能为空，所以需要检验下被代理对象是否实现了接口，如果没有实现，那么

\*生成对类的代理。对类的代理将在后面细说。

\*/

if(isHaveInterface(target)){

return Proxy.newProxyInstance(target.getClass().getClassLoader()

, target.getClass().getInterfaces(), this);

}

else{

return createProxy(target.getClass());

}

}

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

throws Throwable {

before(proxy,method,args);

Object result = method.invoke(target, args);

after(proxy,method,args);

return result;

}

}

我们想要获取TargetClass对象的代理怎么办呢？

TargetClass targetClassObject = new TargetClassObject();

DefaultBindInvocationHandler proxyHandler = new DefaultBindInvocationHandler();

proxyHandler.bind(targetClassObject);

其他需要代理的类只要 要被代理的原因相同，反映到代码就是 invoke方法的逻辑满足这个类被代理的需要，那么他就可以使用这个类

进行代理。invoke方法在执行目标类方法前，执行了before,之后执行了after，用于前置后置增强。

如何有需要特殊的增强，额外的增强，可以新增子类重写before，after，实现增强，同时也可以重写invoke方法。

但是jdk代理存在的遗憾的是当类没有实现任何接口时，jdk就不能代理了

而且jdk代理的限制在于，我代理一个对象也许就是拦截对象的所有方法，为什么你要求我一定要实现一个接口呢，如果我要拦截私有方法难道就不行吗？

很明显，jdk代理无法完成这个目标。

于是cglib给我们提供了这种方便。

2.2 cglib代理

cglib是专门针对类进行代理的，它通过一种运行时生成字节吗文件，的技术，动态生成类，生成字节吗，实现继承某个类，实现某个接口，最大限度的

实现了动态生成类技术，Spring基于cglib甚至可以获取到方法体上参数的名字。在大量的框架都使用了cglib动态生成类技术。

cglib生成代理类很容易

Enhancer enhancer = new Enhancer(); //启动容器

enhancer.setSuperclass(targetClass);// 设置代理目标，被代理的类的Class对象

enhancer.setCallback(this);// 设置回调 ，要求参数为实现了MethodInterceptor接口的类

enhancer.setClassLoader(targetClass.getClassLoader());//指定加载器为加载目标类的加载器。（不能指定为加载本类的加载器）

return enhancer.create();///创建一个对象

这样一个继承了targetClass类型的对象就创建成功了。

3.muppet如何封装动态代理的

一切都应准备就绪，但是如何提供一个工具类，以后更加便捷的操作呢？

首先明确一下，我们不能完全抛弃jdk代理，毕竟jdk代理更加快速一些，而且我们希望我们尽量与cglib的 耦合弱一点，最好将用到cglib的地方控制在一个静态的值域中，而不能随业务扩展随着扩展。

也就是我们要封装动态代理。

1.考虑输入。无怪乎，被代理对象，代理增强器，在jdk中是InvocationHandler,在cglib中是MethodInterceptor。两个接口。

2.代理增强器是随着代理逻辑增多，而增多的，如何包装cglib的增强器，使之出现的地方有限而最少呢？

我们是这样设计的。

1）/\*\*

\*首先定义绑定代理增强接口，何为绑定代理增强呢，因为我们一般在获取代理对象时都是通过代理增强器类的绑定操作，

\*绑定要被代理的对象，然后返回代理对象。不如我们就定义一绑定接口。同时让他继承InvocationHandler实现jdk代理

\*这个接口是为jdk动态代理服务的

\*

interface BindInvocationHandler extends InvocationHandler{

/\*\*

\* 绑定被代理对象，返回代理对象

\*/

public Object bind(Object target)throws NullPointerException,IllegalArgumentException;

/\*\*

\*绑定被代理对象，返回代理对象，根据指定的接口返回实现指定接口的代理

\*/

public <T> T bind(Object target,Class<T> targetInterface)throws NullPointerException,IllegalArgumentException;

}

2）定义默认绑定代理增强类，这个类实现了绑定代理增强，和cglib代理增强。注定是最复杂的环节，

class DefaultBindInvocationHandler implements BindInvocationHandler

,MethodInterceptor{

}

先明确一下这个类的功能

2.1）实现InvocationHandler接口

这个接口只需要实现一个方法

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

参数分别为 生成的代理对象，即将执行的方法对象，方法中参数

看看我们实现的方法体

before(proxy,method,args,null);//默认是空方法体

Object result = method.invoke(target, args);//执行被代理对象方法

after(proxy,method,args,null);//默认是空方法体

return result;

很简单对不对，before，和after都是protected类型方法，很明显是用来子类 继承，按照模板方法，将行为转移到子类实现，

2.2）实现MethodIntercept接口

这个接口也只有一个方法与invoke类似

@Override

public Object intercept(Object proxy, Method method, Object[] params,

MethodProxy methodProxy) throws Throwable {

before(proxy,method,params,methodProxy);

Object result = methodProxy.invokeSuper(proxy, params);

after(proxy,method,params,methodProxy);

return result;

}

平白无奇，同样是通过methodProxy方法实现被代理对象方法执行。只不过before方法和after方法第四个参数不为空

2.3）jdk绑定被代理对象同时返回代理对象

public Object bind(Object target)

throws NullPointerException,IllegalArgumentException

{

this.target = (target);

if(isHaveInterface(target)){//判断有没有实现接口，如果实现了，那么按照jdk动态代理，否则按照 cglib代理。

return Proxy.newProxyInstance(target.getClass().getClassLoader()

, target.getClass().getInterfaces(), this);

}

else{

return createProxy(target.getClass());

}

}

先绑定被代理对象，然后，创建代理对象，返回代理对象，同样平淡无奇

2.4）cglib绑定被代理对象，同时返回代理对象。

为了区分和jdk的区别，方法名改为getClassProxy，表示是对类的代理，而不是jdk的对接口的代理

public <T> T getClassProxy(T target){

this.target = target;

return (T) createProxy(target.getClass());

}

private final <T> T createProxy(Class<T> targetClass) {

Enhancer enhancer = new Enhancer(); //启动容器

enhancer.setSuperclass(targetClass);// 设置代理目标，被代理的类的Class对象

enhancer.setCallback(this);// 设置回调 ，要求参数为实现了MethodInterceptor接口的类

enhancer.setClassLoader(targetClass.getClassLoader());//指定加载器为加载目标类的加载器.(不能指定为加载本类的加载器）

return enhancer.create();///创建一个对象

}

这段代码在上面 已经解释过了，即使平淡，也不平淡，不平淡的地方是this，在设置回调类时是设置this,如果是本类对象调用，那么

this，当然是本类，也就是代理对象的方法调用都会引起intercept方法的调用。但是一旦子类重写了intercept方法，那回调类

将转移到子类对象，也就是子类的实现的intercept方法会被调用，而正是我们希望看到的结果。即我们创建的代理对象随着子类重写

intercept方法，代理增强也不同

2.5）到这我们可以新增子类，DefaultBindInvocationHandler做一些前后置增强了，但是我们的需求止于此了吗

首先新增子类，对于子类来说功能实现选择 较多，可以重写before，after,也可以重写invoke，但是每次增加一个代理加强

都需要继承一个父类，是不是太过于多余，限制太多。耦合性太强，我们期待什么呢？

像这样定一个代理增强、

public class ClosedInvocationHandler implements InvocationHandler{

@Override

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

throws Throwable {

if(Closed.class.isInstance(proxy)){

Closed close = (Closed)proxy;

if(close.hasClosed()){

throw new IllealInvokeException("已经关闭，没有权限访问");

}else{

return method.invoke(proxy, args);

}

}

return method.invoke(proxy, args);

}

}

ReflectUtil.getClassProxy(new A(),new ClosedInvocationHandler());

这样我们只需要定义一个实现了jdk接口的实现类，实现了与cglib解耦，同时屏蔽了jdk代理与cglib接口的区别。

同时不需要代理增强不需要额外定义绑定操作。

让客户端不关心代理实现的细节，将关注点放在定义代理增强上。

为此我们定义了一个标准绑定增强类

class StandardBindInvocationHandler extends DefaultBindInvocationHandler{

private InvocationHandler handler;

public StandardBindInvocationHandler(InvocationHandler handler){

this.handler = handler;

}

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

throws Throwable {

before(proxy,method,args,null);

Object result = handler.invoke(target, method, args);

after(proxy,method,args,null);

return result;

}

@Override

public Object intercept(Object proxy, Method method, Object[] params,

MethodProxy methodProxy) throws Throwable {

before(proxy,method,params,methodProxy);

Object object = handler.invoke(target, method, params);

after(proxy,method,params,methodProxy);

return object;

}

}

这个类可以看到是包内可见，并不对包外开放，也就是在其他地方定义代理增强完全不需要知道这么一个类的存在。

这个类其实本身就是代理模式的应用，代理的接口就是InvocationHandler接口，在初始化这个类对象的时候需要指定

一个代理增强对象。然后可以看到不论是jdk代理，还是cglib代理 ，都将请求转发到这个代理对象，但是有一点

需要注意，在传给代理增强类参数中有一丝变化

Object object = handler.invoke(target, method, params);//在 jdk中，第一个参数是生成的代理对象，但是

在我们这里传给他的事被代理对象，也就是用户输入的类。这一点在定义代理增强类时需要注意，我觉得这很好，对于

客户端调用，他不关注生成的代理对象是什么玩意，他只关注我被代理的对象什么样，然后指定特定的逻辑 处理。

所以在我们刚才自定的代理增强类中。

public class ClosedInvocationHandler implements InvocationHandler{

@Override

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

throws Throwable {

if(Closed.class.isInstance(proxy)){

Closed close = (Closed)proxy;

if(close.hasClosed()){

throw new IllealInvokeException("已经关闭，没有权限访问");

}else{

return method.invoke(proxy, args);

}

}

return method.invoke(proxy, args);

}

}

我们可以直接调用method.invoke(proxy,args);在使用原生jdk代理使绝对不可以的，因为这句话的意思 是执行指定对象的这个方法，因为代理

对象执行了一个方法，引起invoke调用,在invoke方法中在对代理对象的方法进行调用，那同样会引起invoke方法的调用，正确是调用

被代理对象的这个方法，而我们给自定义代理增强传入的参数就是被代理对象。所以很方便使用。

那么其他的黑箱是啥呢，当然是ReflectUtil的getClassProxy方法

StandardBindInvocationHandler handler = new StandardBindInvocationHandler(invocationHandler);

return handler.getClassProxy(target);

只有简单的两行

现在梳理下方法调用

1.创建StandardBindInvocationHandler，指定了代理增强，这个代理增强是我们的特定的业务实现，实现接口为InvocationHandler接口

2.调用standardBindInvocatinoHandler的getClassProxy，StandardBindInvocationHandler本身没有重写，所以还是父类的方法

this.target = target;

return (T) createProxy(target.getClass(),constructorArgsClazzs,constructorArgsValues);

3。绑定被代理对象，注意这个target是保护类型，也就是子类可以访问，也就是在StandardBingInvocationHandler中的invoke,intercept方法中的target参数

就是这个target,而它的赋值就是在这里。

4.然后createProxy方法就是我们之前说的

Enhancer enhancer = new Enhancer();

enhancer.setSuperclass(targetClass);// 设置代理目标

enhancer.setCallback(this);// 设置回调

enhancer.setClassLoader(targetClass.getClassLoader());

return (T)enhancer.create();

同样传神的是this，因此代理对象的默认代理增强是子类对象，也就是代理对象上的方法调用请求 将调用StandardBingInvocationHandler类的intercept方法，然后。

5.当调用intercept时，intercept会将请求转发给用户自定义的InvocationHandler实现上，也就是ClosedInvocationHandler类型的代理增强。

至此，整个流程结束

缘起ReflectUtil.创建代理类交给了DefaultBindInvocationHandler类，回调处理交给了子类的实现，子类的实现又把请求转给了用户的自定义请求。

由于采用jdk自定义的InvocationHandler接口，所以实现了和cglib的解耦。

jdk的代理同样一个思路，都是面向对象的简单使用。

总之最后的调用变成这样

public class ClosedInvocationHandler implements InvocationHandler{

@Override

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

throws Throwable {

if(Closed.class.isInstance(proxy)){

Closed close = (Closed)proxy;

if(close.hasClosed()){

throw new IllealInvokeException("已经关闭，没有权限访问");

}else{

return method.invoke(proxy, args);

}

}

return method.invoke(proxy, args);

}

}

对类代理 ReflectUtil.getClassProxy(new A(),new ClosedInvocationHandler());

对接口代理 ReflectUtil.getProxy(new A(),new ClosedInvocationHandler());

解耦，关注应该关注的地方，使用简洁，是我们追求的目标。欢迎关注muppet最新进展，同时欢迎加入我们，做些有意义的开心的事情。

svn地址：https://123.56.225.214/svn/muppet 用户名：muppet 密码:muppet 也可以关注微信公众平台 IT之路 （头像是彩色条纹） 内有优质原创文章