Spring AOP 之动态代理剖析

作者：valarchie
原文链接：https://ld246.com/article/1570729100128
来源网站：链滴

代理模式

一说到代理，很多人都会立马想到设计模型中的代理模式，通过持有被代理对象并继承被代理对象的便可以实现代理。假设我们要给ServiceA代理日志功能，就需要声明并实现日志代理类。如果要给SericeA代理事务功能，就又需要声明并实现事务代理类。这时如何整合日志和事务代理功能就是一个问了。其次，假设ServiceA当中有100个方法，都需要手工加上100次日志代码。再其次，假设Service、ServiceB、ServiceC都需要代理日志的话，还得针对这三个Service生成不同的代理类。

我们可以看到静态代理的局限性：

- 难以整合不同的代理类去代理同一个对象。
- 难以在类内部的各个方法复用代理逻辑。
- 难以在不同的类之间复用代理逻辑。
- 需要大量的代理类才能满足我们的庞大的业务需求。

而动态代理是如何灵活的解决这些问题的呢？

JDK动态代理

首先我们先使用动态代理最基础的用法，不涉及Spring框架的最原始方法。

先声明一个HelloService接口：

```
public interface HelloService {
    void hello();
}
```

HelloService的实现类：

```
public class HelloServiceImpl implements HelloService {
    @Override
    public void hello() {
        System.out.println("hello!");
    }
}
```

动态代理通过一个自定义的InvocationHandler来实现代理，简而言之就是InvocationHandler中会入我们想代理的对象，并在invoke方法当中进行方法的增强。这里我们实现一个模拟事务的InvocatioHandler:

```
public class TransactionHandler implements InvocationHandler {
    // 被代理的对象
    private Object target;

    public TransactionHandler(Object target) {
        this.target = target;
    }

    @Override
    public Object invoke(Object proxy, Method method, Object[] args)
            throws IllegalArgumentException, InvocationTargetException, IllegalAccessException {
```

```
        System.out.println("开启事务！");
        Object retVal = method.invoke(target, args);
        System.out.println("结束事务！");

        return retVal;
    }


}
```

测试动态代理代码：

```
public class TestJdkProxy {

    public static void main(String[] args) {
        // 填入以下参数可以将动态生成的类 保存到项目中
        System.getProperties().put("sun.misc.ProxyGenerator.saveGeneratedFiles", "true");

        HelloService helloService = new HelloServiceImpl();
        // 将被代理的对象填入InvocationHandler当中，本次实现的是添加事务功能
        TransactionHandler transactionHandler = new TransactionHandler(helloService);

        // 通过JDK自带的Proxy类，填入当前的类加载器，需要实现的接口，以及有增强功能的Handler
        HelloService proxyHelloService = (HelloService) Proxy.newProxyInstance(helloService.ge
Class().getClassLoader(),
                new Class[]{HelloService.class}, transactionHandler);

        proxyHelloService.hello();

        System.out.println("被代理的proxyHelloService，它的类型是：" + proxyHelloService.getCl
ss().getName());

    }

}
```

运行结果：

```
开启事务!
hello!
结束事务!
被代理的proxyHelloService，它的类型是：com.sun.proxy.$Proxy0
```

我们可以看到代理类proxyHelloService的类型是com.sun.proxy.$Proxy0类。

为什么呢?

关键就在于Proxy.newProxyInstance()方法当中，我们进入到方法中分析。

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

```
        throws IllegalArgumentException
    {
        //...略

        /*
         * 查找或生成指定的代理类
         */
        Class<?> cl = getProxyClass0(loader, intfs);

        /*
         * 使用指定的InvocationHandler作为参数调用上一步声明的代理类的构造器
         * 并生成实例
         */
        try {
            if (sm != null) {
                checkNewProxyPermission(Reflection.getCallerClass(), cl);
            }

            final Constructor<?> cons = cl.getConstructor(constructorParams);
            final InvocationHandler ih = h;
            if (!Modifier.isPublic(cl.getModifiers())) {
                AccessController.doPrivileged(new PrivilegedAction<Void>() {
                    public Void run() {
                        cons.setAccessible(true);
                        return null;
                    }
                });
            }
            return cons.newInstance(new Object[]{h});
        }

        //....略
    }
```

笔者省略了方法内的检验和异常等代码。

方法内主要最主要的是getProxyClass0(loader, intfs)和cons.newInstance(new Object[]{h})这两个法：

1. getProxyClass0(loader, intfs)方法根据填入的类加载器和所需代理的接口动态地生成出一个代理。
2. cons.newInstance(new Object[]{h})方法将所需增强的InvocationHandler填入到上一步生成的理类的构造器并生成实例。

我们进入到getProxyClass0方法中查看

```
private static Class<?> getProxyClass0(ClassLoader loader,
                                       Class<?>... interfaces) {
        if (interfaces.length > 65535) {
            throw new IllegalArgumentException("interface limit exceeded");
        }

        // If the proxy class defined by the given loader implementing
        // the given interfaces exists, this will simply return the cached copy;
```

```
        // otherwise, it will create the proxy class via the ProxyClassFactory
        return proxyClassCache.get(loader, interfaces);
    }
```

在代码中我们可以清晰的看到注释内容为：如果类加载器中存在实现了给定接口的代理类，那么就直返回缓存好的类对象。否则，将通过ProxyClassFactory动态生成我们所需实现给定接口的代理类。

代码中的proxyClassCache.get(loader, interfaces)方法中比较复杂，我们就不进入分析，代码中会用到ProxyFactory的apply方法，而ProxyFactory是Proxy类的内部类。接下来开始分析ProxyFactor类。

```
 private static final class ProxyClassFactory
        implements BiFunction<ClassLoader, Class<?>[], Class<?>>
    {
        // 设定动态代理类的名字前缀为 $Proxy
        // 这就是为什么开头看到动态代理类的名称为com.sun.proxy.$Proxy0
        private static final String proxyClassNamePrefix = "$Proxy";

        // 这是一个递增的数字，所以代理类的名称都是$ProxyN的形式
        private static final AtomicLong nextUniqueNumber = new AtomicLong();

        @Override
        public Class<?> apply(ClassLoader loader, Class<?>[] interfaces) {

            Map<Class<?>, Boolean> interfaceSet = new IdentityHashMap<>(interfaces.length);

            // 循环检查传入的接口类对象数组
            for (Class<?> intf : interfaces) {
                /*
                 * Verify that the class loader resolves the name of this
                 * interface to the same Class object.
                 */
                Class<?> interfaceClass = null;
                try {
                    interfaceClass = Class.forName(intf.getName(), false, loader);
                } catch (ClassNotFoundException e) {
                }
                if (interfaceClass != intf) {
                    throw new IllegalArgumentException(
                        intf + " is not visible from class loader");
                }
                // 确定传入的接口类对象是接口
                if (!interfaceClass.isInterface()) {
                    throw new IllegalArgumentException(
                        interfaceClass.getName() + " is not an interface");
                }
                // 确定传入的接口类对象没有重复
                if (interfaceSet.put(interfaceClass, Boolean.TRUE) != null) {
                    throw new IllegalArgumentException(
                        "repeated interface: " + interfaceClass.getName());
                }
            }

            String proxyPkg = null;     // 声明代理类的包名
```

```
        int accessFlags = Modifier.PUBLIC | Modifier.FINAL;

        // 确认所有非公共的接口都来自同一个包，如果不同包的话，jdk不知道要给这个代理类取什
包名
        // 如果接口是公共的话 直接使用默认的com.sun.proxy包名
        // 所以，上文HelloService接口的修饰符如果是包限定的话，生成的包名就会是
        // 我们自定义的com.valarchie.aop.$Proxy0
        for (Class<?> intf : interfaces) {
            int flags = intf.getModifiers();
            if (!Modifier.isPublic(flags)) {
                accessFlags = Modifier.FINAL;
                String name = intf.getName();
                int n = name.lastIndexOf('.');
                String pkg = ((n == -1) ? "" : name.substring(0, n + 1));
                if (proxyPkg == null) {
                    proxyPkg = pkg;
                } else if (!pkg.equals(proxyPkg)) {
                    throw new IllegalArgumentException(
                        "non-public interfaces from different packages");
                }
            }
        }

        if (proxyPkg == null) {
            // 如果接口都是公共的，那么直接使用 com.sun.proxy 包名
            proxyPkg = ReflectUtil.PROXY_PACKAGE + ".";
        }

        // 生成$ProxyN后缀的这个N数字，它是原子递增的
        long num = nextUniqueNumber.getAndIncrement();
        String proxyName = proxyPkg + proxyClassNamePrefix + num;

        // 动态拼接生成代理类并存入二进制数组
        byte[] proxyClassFile = ProxyGenerator.generateProxyClass(
            proxyName, interfaces, accessFlags);
        try {
            // 将类文件二进制数组转成类对象返回
            return defineClass0(loader, proxyName,
                                proxyClassFile, 0, proxyClassFile.length);
        } catch (ClassFormatError e) {

            throw new IllegalArgumentException(e.toString());
        }
    }
}
```

看到以上源码，大家就明白为什么代理类的名字是com.sun.proxy.$Proxy0这种形式了。

方法内最关键的是ProxyGenerator.generateProxyClass(proxyName, interfaces, accessFlags)方，它将我们所需代理的接口HelloService接口传入然后生成了一个类型为$Proxy0的类并实现了Helloervice接口。

我们先将生成的代理类com.sun.proxy.$Proxy0使用jad或者其他反编译工具反编译出来查看一下：

```
package com.sun.proxy;
```

```
import com.valarchie.aop.HelloService;
import java.lang.reflect.*;

public final class $Proxy0 extends Proxy
    implements HelloService
{
    // 构造函数传入我们之前的InvocationHandler
    public $Proxy0(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 void hello()
    {
        try
        {
            // 使用我们的Handler来代理hello方法，m3其实就是hello方法，
            // 在下面的静态初始化块中生成
            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);
```

```
        }
    }

    public final int hashCode()
    {
        try
        {
            return ((Integer)super.h.invoke(this, m0, null)).intValue();
        }
        catch(Error _ex) { }
        catch(Throwable throwable)
        {
            throw new UndeclaredThrowableException(throwable);
        }
    }

    private static Method m1;
    private static Method m3;
    private static Method m2;
    private static Method m0;

    static
    {
        try
        {
            m1 = Class.forName("java.lang.Object").getMethod("equals", new Class[] {
                Class.forName("java.lang.Object")
            });
            // 找到我们所需代理的hello方法
            m3 = Class.forName("com.valarchie.aop.HelloService").getMethod("hello", new Class[0
);
            m2 = Class.forName("java.lang.Object").getMethod("toString", new Class[0]);
            m0 = Class.forName("java.lang.Object").getMethod("hashCode", new Class[0]);
        }
        catch(NoSuchMethodException nosuchmethodexception)
        {
            throw new NoSuchMethodError(nosuchmethodexception.getMessage());
        }
        catch(ClassNotFoundException classnotfoundexception)
        {
            throw new NoClassDefFoundError(classnotfoundexception.getMessage());
        }
    }
}
```

可以看到代理类实际上是一个继承Proxy类并实现了HelloService接口的动态生成的类（这也是为什当没有定义接口的话，不能使用JDK动态代理的原因）。在每个方法中都去回调了InvocationHandle中的invoke方法。简而言之，这个动态生成的代理类有以下三个重点：

1. 生成出所有被代理接口类中含有的方法。

2. 在构造函数当中传入定义好的InvocationHandler，而InvocationHandler内已经持有真正被代理对象。

3. 在每个方法中都去回调InvocationHandler中的invoke方法。

这与我们设计模式中的代理模式有何不同呢?

动态代理类通过InvocationHandler对象间接的持有被代理对象，所有方法也都是通过回调invoke方来间接调用被代理类的方法，并加上自己的增强实现。看到这里，读者就会发现之所以动态代理更加活的关键原因就是使用了InvocationHandler这个中间类来进行解耦。

看到代理类的内部构造之后，那JDK是如何生成这个类的呢?

ProxyGenerator.generateProxyClass方法的源码可以在官方找到，该方法的核心内容就是去调用generateClassFile()实例方法来生成Class文件，具体源码如下：

```
private byte[] generateClassFile() {
    // 第一步, 将所有的方法组装成ProxyMethod对象
    // 首先为代理类生成通用的toString, hashCode, equals方法
    addProxyMethod(hashCodeMethod, Object.class);
    addProxyMethod(equalsMethod, Object.class);
    addProxyMethod(toStringMethod, Object.class);
    // 遍历每一个接口的所有方法, 并且为其生成ProxyMethod对象
    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]);
        }
    }
    //对于具有相同签名的代理方法, 检验方法的返回值是否兼容
    for (List<ProxyMethod> sigmethods : proxyMethods.values()) {
        checkReturnTypes(sigmethods);
    }

    //第二步, 组装要生成的class文件的所有的字段信息和方法信息
    try {
        //添加构造器方法
        methods.add(generateConstructor());
        //遍历缓存中的代理方法
        for (List<ProxyMethod> sigmethods : proxyMethods.values()) {
            for (ProxyMethod pm : sigmethods) {
                //添加代理类的静态字段, 例如:private static Method m1;
                fields.add(new FieldInfo(pm.methodFieldName,
                        "Ljava/lang/reflect/Method;", ACC_PRIVATE | ACC_STATIC));
                //添加代理类的代理方法
                methods.add(pm.generateMethod());
            }
        }
        //添加代理类的静态字段初始化方法
        methods.add(generateStaticInitializer());
    } catch (IOException e) {
        throw new InternalError("unexpected I/O Exception");
    }

    //验证方法和字段集合不能大于65535
    if (methods.size() > 65535) {
        throw new IllegalArgumentException("method limit exceeded");
    }
```

```
if (fields.size() > 65535) {
    throw new IllegalArgumentException("field limit exceeded");
}

//第三步, 写入最终的class文件
//验证常量池中存在代理类的全限定名
cp.getClass(dotToSlash(className));
//验证常量池中存在代理类父类的全限定名, 父类名为:"java/lang/reflect/Proxy"
cp.getClass(superclassName);
//验证常量池存在代理类接口的全限定名
for (int i = 0; i < interfaces.length; i++) {
    cp.getClass(dotToSlash(interfaces[i].getName()));
}
//接下来要开始写入文件了,设置常量池只读
cp.setReadOnly();

ByteArrayOutputStream bout = new ByteArrayOutputStream();
DataOutputStream dout = new DataOutputStream(bout);
try {
    //1.写入魔数 CAFEBABE 咖啡宝贝~
    dout.writeInt(0xCAFEBABE);
    //2.写入次版本号
    dout.writeShort(CLASSFILE_MINOR_VERSION);
    //3.写入主版本号
    dout.writeShort(CLASSFILE_MAJOR_VERSION);
    //4.写入常量池
    cp.write(dout);
    //5.写入访问修饰符
    dout.writeShort(ACC_PUBLIC | ACC_FINAL | ACC_SUPER);
    //6.写入类索引
    dout.writeShort(cp.getClass(dotToSlash(className)));
    //7.写入父类索引, 生成的代理类都继承自Proxy
    dout.writeShort(cp.getClass(superclassName));
    //8.写入接口计数值
    dout.writeShort(interfaces.length);
    //9.写入接口集合
    for (int i = 0; i < interfaces.length; i++) {
        dout.writeShort(cp.getClass(dotToSlash(interfaces[i].getName())));
    }
    //10.写入字段计数值
    dout.writeShort(fields.size());
    //11.写入字段集合
    for (FieldInfo f : fields) {
        f.write(dout);
    }
    //12.写入方法计数值
    dout.writeShort(methods.size());
    //13.写入方法集合
    for (MethodInfo m : methods) {
        m.write(dout);
    }
    //14.写入属性计数值, 代理类class文件没有属性所以为0
    dout.writeShort(0);
} catch (IOException e) {
```

```
            throw new InternalError("unexpected I/O Exception");
        }
        //转换成二进制数组输出
        return bout.toByteArray();
    }
```

以上就是JDK动态代理的详细过程。

笔者水平有限，如有错误恳请网友评论指正。

转自我的个人博客 vc2x.com