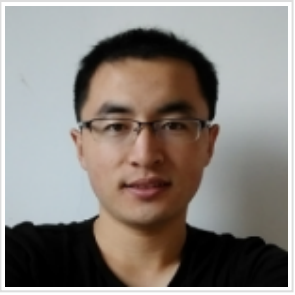
个人资料



终点

访问： 1253121次

积分： 9957

Java之美[从菜鸟到高手演练]之JDK动态代理的实现及原理

标签： JDK动态代理

2015-01-07 22:23 10455人阅读 评论(4) 收藏 举报

分类： J2SE（38）

JDK动态代理的实现及原理

作者：二青

邮箱：xtfggef@gmail.com 微博：http://weibo.com/xtfggef

动态代理，听上去很高大上的技术，在**Java**里应用广泛，尤其是在**hibernate**和**spring**这两种框架里，在AOP，权限控制，事务管理等方面都有动态代理的实现。JDK本身有实现动态代理技术，但是略有限制，即被代理的类必须实现某个接口，否则无法使用JDK自带的动态代理，因此，如果不满足条件，就只能使用另一种更加灵活，功能更加强大的动态代理

等级： BLOG 6
排名： 第1375名
原创： 74篇 转载： 1篇
译文： 0篇 评论： 1110条

技术—— CGLIB。Spring里会自动在JDK的代理和CGLIB之间切换，同时我们也可以强制Spring使用CGLIB。下面我们就动态代理方面的知识点从头至尾依次介绍一下。

我们先来看一个例子：

新建一个接口，UserService.java, 只有一个方法add()。

```java
[java]
01. package com.adam.java.basic;
02.
03. public interface UserService {
04.     public abstract void add();
05. }
```

建一个该接口的实现类UserServiceImpl.java

```java
[java]
01. package com.adam.java.basic;
02. public class UserServiceImpl implements UserService {
03.
04.     @Override
05.     public void add() {
06.         System.out.println("----- add -----");
07.     }
08. }
```

建一个代理处理类MyInvocationHandler.java

```java
[java]
01. package com.adam.java.basic;
02. import java.lang.reflect.InvocationHandler;
03. import java.lang.reflect.Method;
04. import java.lang.reflect.Proxy;
```

```
05.
06. public class MyInvocationHandler implements InvocationHandler {
07.
08.     private Object target;
09.
10.     public MyInvocationHandler(Object target) {
11.         super();
12.         this.target = target;
13.     }
14.
15.     public Object getProxy() {
16.         return Proxy.newProxyInstance(Thread.currentThread()
17.                 .getContextClassLoader(), target.getClass().getInterfaces(),
18.                 this);
19.     }
20.
21.     @Override
22.     public Object invoke(Object proxy, Method method, Object[] args)
23.             throws Throwable {
24.         System.out.println("----- before -----");
25.         Object result = method.invoke(target, args);
26.         System.out.println("----- after -----");
27.         return result;
28.     }
29. }
```

测试类

```
[java]
01. package com.adam.java.basic;
02. public class DynamicProxyTest {
03.
04.     public static void main(String[] args) {
05.         UserService userService = new UserServiceImpl();
06.         MyInvocationHandler invocationHandler = new MyInvocationHandler(
07.                 userService);
08.
```

Java之美[从菜鸟到高手进 (38479)
Java之美[从菜鸟到高手进 (34422)
Java之美[从菜鸟到高手进 (33145)
Java之美[从菜鸟到高手进 (30700)

评论排行

Java之美[从菜鸟到高手进 (138)
Java之美[从菜鸟到高手进 (108)
Java之美[从菜鸟到高手进 (85)
技术之美[程序人生]我在II (80)
Java之美[从菜鸟到高手进 (59)
Java之美[从菜鸟到高手进 (42)
Java之美[从菜鸟到高手进 (37)
Java之美[从菜鸟到高手进 (36)
Java之美[从菜鸟到高手进 (36)
Java之美[从菜鸟到高手进 (35)

推荐文章

* 你的薪水增速跑赢GDP了没
* 为什么Go语言在中国格外的"火"
* 技术宅找女朋友的技术分析
* 【物联网云端对接】通过HTTP协议与微软Azure IoT hub进行云端通信
* iOS狂暴之路---视图控制器(UIViewController)使用详解

最新评论

```
09.         UserService proxy = (UserService) invocationHandler.getProxy();
10.         proxy.add();
11.     }
12. }
```

执行测试类，得到如下输出：

----- before -----

----- add -----

----- after -----

到这里，我们应该会想到点问题：

1. 这个代理对象是由谁且怎么生成的？
2. invoke方法是怎么调用的？
3. invoke和add方法有什么对应关系？
4. 生成的代理对象是什么样子的？

带着这些问题，我们看一下源码。首先，我们的入口便是上面测试类里的getProxy()方法，我们跟进去，看看这个方法：

```
[java]
01. public Object getProxy() {
02.     return Proxy.newProxyInstance(Thread.currentThread()
03.             .getContextClassLoader(), target.getClass().getInterfaces(),this);
04. }
```

也就是说，JDK的动态代理，是通过一个叫Proxy的类来实现的，我们继续跟进去，看看Proxy类的newProxyInstance()方法。先来看看JDK的注释：

```
[plain]
01. /**
02.      * Returns an instance of a proxy class for the specified interfaces
03.      * that dispatches method invocations to the specified invocation
04.      * handler.
```

```
05.      *
06.      * <p>{@code Proxy.newProxyInstance} throws
07.      * {@code IllegalArgumentException} for the same reasons that
08.      * {@code Proxy.getProxyClass} does.
09.      *
10.      * @param   loader the class loader to define the proxy class
11.      * @param   interfaces the list of interfaces for the proxy class
12.      *          to implement
13.      * @param   h the invocation handler to dispatch method invocations to
14.      * @return  a proxy instance with the specified invocation handler of a
15.      *          proxy class that is defined by the specified class loader
16.      *          and that implements the specified interfaces
```

根据JDK注释我们得知，newProxyInstance方法最终将返回一个实现了指定接口的类的实例，其三个参数分别是ClassLoader，指定的接口及我们自己定义的InvocationHandler类。我摘几条关键的代码出来，看看这个代理类的实例对象到底是怎么生成的。

```
[java]
01.  Class<?> cl = getProxyClass0(loader, intfs);
02.  ...
03.  final Constructor<?> cons = cl.getConstructor(constructorParams);
04.  ...
05.  return cons.newInstance(new Object[]{h});
```

有兴趣的同学可以自己看看JDK的源码，当前我用的版本是JDK1.8.25，每个版本实现方式可能会不一样，但基本一致，请研究源码的同学注意这一点。上面的代码表明，首先通过getProxyClass获得这个代理类，然后通过c1.getConstructor()拿到构造函数，最后一步，通过cons.newInstance返回这个新的代理类的一个实例，注意：调用newInstance的时候，传入的参数为h，即我们自己定义好的InvocationHandler类，先记着这一步，后面我们就知道这里这样做的原因。

其实这三条代码，核心就是这个getProxyClass方法，另外两行代码是Java反射的应用，和我们当前的兴趣点没什么关系，所以我们继续研究这个getProxyClass方法。这个方法，注释很简单，如下：

```java
/*
         * Look up or generate the designated proxy class.
         */
        Class<?> cl = getProxyClass0(loader, intfs);
```

就是生成这个关键的代理类，我们跟进去看一下。

```java
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);
    }
```

这里用到了缓存，先从缓存里查一下，如果存在，直接返回，不存在就新创建。在这个get方法里，我们看到了如下代码：

Object subKey = Objects.requireNonNull(subKeyFactory.apply(key, parameter));

此处提到了apply()，是Proxy类的内部类ProxyClassFactory实现其接口的一个方法，具体实现如下：

```java
public Class<?> apply(ClassLoader loader, Class<?>[] interfaces) {

            Map<Class<?>, Boolean> interfaceSet = new IdentityHashMap<>(interfaces.length);
            for (Class<?> intf : interfaces) {
                /*
                 * Verify that the class loader resolves the name of this
```

```
07.                  * interface to the same Class object.
08.                  */
09.                 Class<?> interfaceClass = null;
10.                 try {
11.                     interfaceClass = Class.forName(intf.getName(), false, loader);
12.                 } catch (ClassNotFoundException e) {
13.                 }
14.                 if (interfaceClass != intf) {
15.                     throw new IllegalArgumentException(
16.                         intf + " is not visible from class loader");
17.                 }...
```

看到Class.forName()的时候，我想大多数人会笑了，终于看到熟悉的方法了，没错！这个地方就是要加载指定的接口，既然是生成类，那就要有对应的class字节码，我们继续往下看：

```
[java]
01. /*
02.  * Generate the specified proxy class.
03.  */
04.    byte[] proxyClassFile = ProxyGenerator.generateProxyClass(
05.    proxyName, interfaces, accessFlags);
06.     try {
07.           return defineClass0(loader, proxyName,
08.           proxyClassFile, 0, proxyClassFile.length);
```

这段代码就是利用ProxyGenerator为我们生成了最终代理类的字节码文件，即getProxyClass0()方法的最终返回值。所以让我们回顾一下最初的四个问题：

1. 这个代理对象是由谁且怎么生成的？
2. invoke方法是怎么调用的？
3. invoke和add方法有什么对应关系？
4. 生成的代理对象是什么样子的？

对于第一个问题，我想答案已经清楚了，我再屡一下思路：由Proxy类的getProxyClass0()方法生成目标代理类，然后拿到该类的构造方法，最后通过反射的newInstance方法，产生代理类的实例对象。

接下来，我们看看其他的三个方法，我想先从第四个入手，因为有了上面的生成字节码的代码，那我们可以模仿这一步，自己生成字节码文件看看，所以，我用如下代码，生成了这个最终的代理类。

```java
[java]
01. package com.adam.java.basic;
02.
03. import java.io.FileOutputStream;
04. import java.io.IOException;
05. import sun.misc.ProxyGenerator;
06.
07. public class DynamicProxyTest {
08.
09.     public static void main(String[] args) {
10.         UserService userService = new UserServiceImpl();
11.         MyInvocationHandler invocationHandler = new MyInvocationHandler(
12.                 userService);
13.
14.         UserService proxy = (UserService) invocationHandler.getProxy();
15.         proxy.add();
16.
17.         String path = "C:/$Proxy0.class";
18.         byte[] classFile = ProxyGenerator.generateProxyClass("$Proxy0",
19.                 UserServiceImpl.class.getInterfaces());
20.         FileOutputStream out = null;
21.
22.         try {
23.             out = new FileOutputStream(path);
24.             out.write(classFile);
25.             out.flush();
26.         } catch (Exception e) {
27.             e.printStackTrace();
28.         } finally {
29.             try {
30.                 out.close();
31.             } catch (IOException e) {
```

```
                e.printStackTrace();
            }
        }
    }
}
```

上面测试方法里的proxy.add()，此处的add()方法，就已经不是原始的UserService里的add()方法了，而是新生成的代理类的add()方法，我们将生成的$Proxy0.class文件用jd-gui打开，我去掉了一些代码，add()方法如下：

```java
public final void add()
    throws
  {
    try
    {
      this.h.invoke(this, m3, null);
      return;
    }
    catch (Error|RuntimeException localError)
    {
      throw localError;
    }
    catch (Throwable localThrowable)
    {
      throw new UndeclaredThrowableException(localThrowable);
    }
  }
```

核心就在于this.h.invoke(this. m3, null);此处的h是啥呢？我们看看这个类的类名：

public final class $Proxy0 extends Proxy implements UserService

不难发现，新生成的这个类，继承了Proxy类实现了UserService这个方法，而这个UserService就是我们指定的接口，所以，这里我们基本可以断定，JDK的动态代理，生成的新代理类就是继承了Proxy基类，实现了传入的接口的类。那这个h

到底是啥呢？我们再看看这个新代理类，看看构造函数：

```
[java]
01. public $Proxy0(InvocationHandler paramInvocationHandler)
02.     throws
03.   {
04.     super(paramInvocationHandler);
05.   }
```

构造函数里传入了一个InvocationHandler类型的参数，看到这里，我们就应该想到之前的一行代码：

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

这是newInstance方法的最后一句，传入的h，就是这里用到的h，也就是我们最初自己定义的MyInvocationH[illegible]例。所以，我们发现，其实最后调用的add()方法，其实调用的是MyInvocationHandler的invoke()方法。我们这个方法，找一下m3的含义，继续看代理类的源码：

```
[java]
01. static
02.   {
03.     try
04.     {
05.       m1 = Class.forName("java.lang.Object").getMethod("equals", new Class[] { Class.forName("java.lang.
06.       m2 = Class.forName("java.lang.Object").getMethod("toString", new Class[0]);
07.       m3 = Class.forName("com.adam.java.basic.UserService").getMethod("add", new Class[0]);
08.       m0 = Class.forName("java.lang.Object").getMethod("hashCode", new Class[0]);
09.       return;
10.     }
```

惊喜的发现，原来这个m3，就是原接口的add()方法，看到这里，还有什么不明白的呢？我想2,3,4问题都应该迎刃而解了吧？我们继续，看看原始MyInvocationHandler里的invoke()方法：


```java
<span style="white-space:pre">  </span>@Override
    public Object invoke(Object proxy, Method method, Object[] args)
            throws Throwable {
        System.out.println("----- before -----");
        Object result = method.invoke(target, args);
        System.out.println("----- after -----");
        return result;
    }
```

m3就是将要传入的method，所以，为什么先输出before，后输出after，到这里是不是全明白了呢？这，就是代理整个过程，不难吧？

最后，我稍微总结一下JDK动态代理的操作过程：

1. 定义一个接口，该接口里有需要实现的方法，并且编写实际的实现类。

2. 定义一个InvocationHandler类，实现InvocationHandler接口，重写invoke()方法，且添加getProxy()方法。

总结一下动态代理实现过程：

1. 通过getProxyClass0()生成代理类。

2. 通过Proxy.newProxyInstance()生成代理类的实例对象，创建对象时传入InvocationHandler类型的实例。

3. 调用新实例的方法，即此例中的add()，即原InvocationHandler类中的invoke()方法。

好了，写了这么多，也该结尾了，感谢博友Rejoy的一篇文章，让我有了参考。同时欢迎大家一起提问讨论，如有问题，请留言，我会抽空回复。相关代码已经上传至百度网盘，下载地址。

联系方式：

邮箱：xtfggef@gmail.com

微博：http://weibo.com/xtfggef

- 参考资料：http://rejoy.iteye.com/blog/1627405

上一篇 Java之美[从菜鸟到高手演变]之Spring中Quartz调度器的使用
下一篇 Java之美[从菜鸟到高手演练]之走进全球互联网中枢，顶级域名服务器的分布

我的同类文章

J2SE（38）

- Java之美[从菜鸟到高手演练]... 2015-03-11 阅读 7928
- Java之美[从菜鸟到高手演练]... 2015-01-24 阅读 56084
- Java之美[从菜鸟到高手演变]... 2014-12-08 阅读 10579
- Java之美[从菜鸟到高手演变]... 2014-12-08 阅读 9817
- Java之美[从菜鸟到高手演变]... 2014-11-08 阅读 3643
- Java之美[从菜鸟到高手演练]... 2015-01-26 阅读 7960
- Java之美[从菜鸟到高手演练]... 2015-01-16 阅读 8428
- Java之美[从菜鸟到高手演变]... 2014-12-08 阅读 5734
- Java之美[从菜鸟到高手演变]... 2014-11-24 阅读 7268
- Java之美[从菜鸟到高手演变]... 2014-11-08 阅读 4453

更多文章

参考知识库

Java SE知识库
22303 关注 | 468 收录

Java EE知识库
14932 关注 | 1233 收录

Java 知识库
22942 关注 | 1441 收录

软件测试知识库
3602 关注 | 310 收录

猜你在找

深入浅出Java的反射
java语言从入门到精通2016+项目实训
Java基础核心技术：IO(day15-day16)
Java基础核心技术：Java常用类(day18)
java核心技术精讲
Java Aop原理--利用JDK动态代理
Java动态代理机制原理详解JDK 和CGLIBJavassistASM
菜鸟学习Spring60s让你学会动态代理原理
Java之美从菜鸟到高手演练之网络体系结构的划分
Java之美从菜鸟到高手演练之Linux下单节点安装Hadoop

查看评论

4楼 qq_32079739 2017-01-18 15:11发表

看不懂，怎么办

3楼 ly315131 2016-02-16 16:20发表

好！顶！赞！

2楼 codyCode 2016-01-07 15:24发表

太给力了，受益匪浅。正纠结呢，看了以后，豁然开朗。32个赞

1楼 caishensu 2015-10-15 20:43发表

不错呀，详实生动的一课

您还没有登录,请[登录]或[注册]

* 以上用户言论只代表其个人观点，不代表CSDN网站的观点或立场

核心技术类目

全部主题 Hadoop AWS 移动游戏 Java Android iOS Swift 智能硬件 Docker OpenStack ...
Spark ERP IE10 Eclipse CRM JavaScript 数据库 Ubuntu NFC WAP jQuery BI HTML5
Spring Apache .NET API HTML SDK IIS Fedora XML LBS Unity Splashtop UML
components Windows Mobile Rails QEMU KDE Cassandra CloudStack FTC coremail OPhone
CouchBase 云计算 iOS6 Rackspace Web App SpringSide Maemo Compuware 大数据 aptech
Perl Tornado Ruby Hibernate ThinkPHP HBase Pure Solr Angular Cloud Foundry Redis
Scala Django Bootstrap