博客来源地址: http://rejoy.iteye.com/blog/1627405

[**JDK动态代理实现原理**](http://rejoy.iteye.com/blog/1627405)

**博客分类：**

* [java](http://rejoy.iteye.com/category/237889)

之前虽然会用JDK的动态代理，但是有些问题却一直没有搞明白。比如说：InvocationHandler的invoke方法是由谁来调用的，代理对象是怎么生成的，直到前几个星期才把这些问题全部搞明白了。   
    废话不多说了，先来看一下JDK的动态是怎么用的。

**Java代码  [收藏代码](javascript:void())**

1. **package** dynamic.proxy;
3. **import** java.lang.reflect.InvocationHandler;
4. **import** java.lang.reflect.Method;
5. **import** java.lang.reflect.Proxy;
7. /\*\*
8. \* 实现自己的InvocationHandler
9. \* @author zyb
10. \* @since 2012-8-9
11. \*
12. \*/
13. **public** **class** MyInvocationHandler **implements** InvocationHandler {
15. // 目标对象
16. **private** Object target;
18. /\*\*
19. \* 构造方法
20. \* @param target 目标对象
21. \*/
22. **public** MyInvocationHandler(Object target) {
23. **super**();
24. **this**.target = target;
25. }

28. /\*\*
29. \* 执行目标对象的方法
30. \*/
31. **public** Object invoke(Object proxy, Method method, Object[] args) **throws** Throwable {
33. // 在目标对象的方法执行之前简单的打印一下
34. System.out.println("------------------before------------------");
36. // 执行目标对象的方法
37. Object result = method.invoke(target, args);
39. // 在目标对象的方法执行之后简单的打印一下
40. System.out.println("-------------------after------------------");
42. **return** result;
43. }
45. /\*\*
46. \* 获取目标对象的代理对象
47. \* @return 代理对象
48. \*/
49. **public** Object getProxy() {
50. **return** Proxy.newProxyInstance(Thread.currentThread().getContextClassLoader(),
51. target.getClass().getInterfaces(), **this**);
52. }
53. }
55. **package** dynamic.proxy;
57. /\*\*
58. \* 目标对象实现的接口，用JDK来生成代理对象一定要实现一个接口
59. \* @author zyb
60. \* @since 2012-8-9
61. \*
62. \*/
63. **public** **interface** UserService {
65. /\*\*
66. \* 目标方法
67. \*/
68. **public** **abstract** **void** add();
70. }
72. **package** dynamic.proxy;
74. /\*\*
75. \* 目标对象
76. \* @author zyb
77. \* @since 2012-8-9
78. \*
79. \*/
80. **public** **class** UserServiceImpl **implements** UserService {
82. /\* (non-Javadoc)
83. \* @see dynamic.proxy.UserService#add()
84. \*/
85. **public** **void** add() {
86. System.out.println("--------------------add---------------");
87. }
88. }
90. **package** dynamic.proxy;
92. **import** org.junit.Test;
94. /\*\*
95. \* 动态代理测试类
96. \* @author zyb
97. \* @since 2012-8-9
98. \*
99. \*/
100. **public** **class** ProxyTest {
102. @Test
103. **public** **void** testProxy() **throws** Throwable {
104. // 实例化目标对象
105. UserService userService = **new** UserServiceImpl();
107. // 实例化InvocationHandler
108. MyInvocationHandler invocationHandler = **new** MyInvocationHandler(userService);
110. // 根据目标对象生成代理对象
111. UserService proxy = (UserService) invocationHandler.getProxy();
113. // 调用代理对象的方法
114. proxy.add();
116. }
117. }

执行结果如下：   
**------------------before------------------   
--------------------add---------------   
-------------------after------------------**   
  
   用起来是很简单吧，其实这里基本上就是AOP的一个简单实现了，在目标对象的方法执行之前和执行之后进行了增强。Spring的AOP实现其实也是用了Proxy和InvocationHandler这两个东西的。   
  
    用起来是比较简单，但是如果能知道它背后做了些什么手脚，那就更好不过了。首先来看一下JDK是怎样生成代理对象的。既然生成代理对象是用的Proxy类的静态方newProxyInstance，那么我们就去它的源码里看一下它到底都做了些什么？

**Java代码  [收藏代码](javascript:void())**

1. /\*\*
2. \* loader:类加载器
3. \* interfaces:目标对象实现的接口
4. \* h:InvocationHandler的实现类
5. \*/
6. **public** **static** Object newProxyInstance(ClassLoader loader,
7. Class<?>[] interfaces,
8. InvocationHandler h)
9. **throws** IllegalArgumentException
10. {
11. **if** (h == **null**) {
12. **throw** **new** NullPointerException();
13. }
15. /\*
16. \* Look up or generate the designated proxy class.
17. \*/
18. Class cl = getProxyClass(loader, interfaces);
20. /\*
21. \* Invoke its constructor with the designated invocation handler.
22. \*/
23. **try** {
24. // 调用代理对象的构造方法（也就是$Proxy0(InvocationHandler h)）
25. Constructor cons = cl.getConstructor(constructorParams);
26. // 生成代理类的实例并把MyInvocationHandler的实例传给它的构造方法
27. **return** (Object) cons.newInstance(**new** Object[] { h });
28. } **catch** (NoSuchMethodException e) {
29. **throw** **new** InternalError(e.toString());
30. } **catch** (IllegalAccessException e) {
31. **throw** **new** InternalError(e.toString());
32. } **catch** (InstantiationException e) {
33. **throw** **new** InternalError(e.toString());
34. } **catch** (InvocationTargetException e) {
35. **throw** **new** InternalError(e.toString());
36. }
37. }

   我们再进去getProxyClass方法看一下

**Java代码  [收藏代码](javascript:void())**

1. **public** **static** Class<?> getProxyClass(ClassLoader loader,
2. Class<?>... interfaces)
3. **throws** IllegalArgumentException
4. {
5. // 如果目标类实现的接口数大于65535个则抛出异常（我XX，谁会写这么NB的代码啊？）
6. **if** (interfaces.length > 65535) {
7. **throw** **new** IllegalArgumentException("interface limit exceeded");
8. }
10. // 声明代理对象所代表的Class对象（有点拗口）
11. Class proxyClass = **null**;
13. String[] interfaceNames = **new** String[interfaces.length];
15. Set interfaceSet = **new** HashSet();   // for detecting duplicates
17. // 遍历目标类所实现的接口
18. **for** (**int** i = 0; i < interfaces.length; i++) {
20. // 拿到目标类实现的接口的名称
21. String interfaceName = interfaces[i].getName();
22. Class interfaceClass = **null**;
23. **try** {
24. // 加载目标类实现的接口到内存中
25. interfaceClass = Class.forName(interfaceName, **false**, loader);
26. } **catch** (ClassNotFoundException e) {
27. }
28. **if** (interfaceClass != interfaces[i]) {
29. **throw** **new** IllegalArgumentException(
30. interfaces[i] + " is not visible from class loader");
31. }
33. // 中间省略了一些无关紧要的代码 .......
35. // 把目标类实现的接口代表的Class对象放到Set中
36. interfaceSet.add(interfaceClass);
38. interfaceNames[i] = interfaceName;
39. }
41. // 把目标类实现的接口名称作为缓存（Map）中的key
42. Object key = Arrays.asList(interfaceNames);
44. Map cache;
46. **synchronized** (loaderToCache) {
47. // 从缓存中获取cache
48. cache = (Map) loaderToCache.get(loader);
49. **if** (cache == **null**) {
50. // 如果获取不到，则新建地个HashMap实例
51. cache = **new** HashMap();
52. // 把HashMap实例和当前加载器放到缓存中
53. loaderToCache.put(loader, cache);
54. }
56. }
58. **synchronized** (cache) {
60. **do** {
61. // 根据接口的名称从缓存中获取对象
62. Object value = cache.get(key);
63. **if** (value **instanceof** Reference) {
64. proxyClass = (Class) ((Reference) value).get();
65. }
66. **if** (proxyClass != **null**) {
67. // 如果代理对象的Class实例已经存在，则直接返回
68. **return** proxyClass;
69. } **else** **if** (value == pendingGenerationMarker) {
70. **try** {
71. cache.wait();
72. } **catch** (InterruptedException e) {
73. }
74. **continue**;
75. } **else** {
76. cache.put(key, pendingGenerationMarker);
77. **break**;
78. }
79. } **while** (**true**);
80. }
82. **try** {
83. // 中间省略了一些代码 .......
85. // 这里就是动态生成代理对象的最关键的地方
86. **byte**[] proxyClassFile = ProxyGenerator.generateProxyClass(
87. proxyName, interfaces);
88. **try** {
89. // 根据代理类的字节码生成代理类的实例
90. proxyClass = defineClass0(loader, proxyName,
91. proxyClassFile, 0, proxyClassFile.length);
92. } **catch** (ClassFormatError e) {
93. **throw** **new** IllegalArgumentException(e.toString());
94. }
95. }
96. // add to set of all generated proxy classes, for isProxyClass
97. proxyClasses.put(proxyClass, **null**);
99. }
100. // 中间省略了一些代码 .......
102. **return** proxyClass;
103. }

进去ProxyGenerator类的静态方法generateProxyClass，这里是真正生成代理类class字节码的地方。

**Java代码  [收藏代码](javascript:void())**

1. **public** **static** **byte**[] generateProxyClass(**final** String name,
2. Class[] interfaces)
3. {
4. ProxyGenerator gen = **new** ProxyGenerator(name, interfaces);
5. // 这里动态生成代理类的字节码，由于比较复杂就不进去看了
6. **final** **byte**[] classFile = gen.generateClassFile();
8. // 如果saveGeneratedFiles的值为true，则会把所生成的代理类的字节码保存到硬盘上
9. **if** (saveGeneratedFiles) {
10. java.security.AccessController.doPrivileged(
11. **new** java.security.PrivilegedAction<Void>() {
12. **public** Void run() {
13. **try** {
14. FileOutputStream file =
15. **new** FileOutputStream(dotToSlash(name) + ".class");
16. file.write(classFile);
17. file.close();
18. **return** **null**;
19. } **catch** (IOException e) {
20. **throw** **new** InternalError(
21. "I/O exception saving generated file: " + e);
22. }
23. }
24. });
25. }
27. // 返回代理类的字节码
28. **return** classFile;
29. }

现在，JDK是怎样动态生成代理类的字节的原理已经一目了然了。   
  
好了，再来解决另外一个问题，那就是由谁来调用InvocationHandler的invoke方法的。要解决这个问题就要看一下JDK到底为我们生成了一个什么东西。用以下代码可以获取到JDK为我们生成的字节码并写到硬盘中。

**Java代码  [收藏代码](javascript:void())**

1. **package** dynamic.proxy;
3. **import** java.io.FileOutputStream;
4. **import** java.io.IOException;
6. **import** sun.misc.ProxyGenerator;
8. /\*\*
9. \* 代理类的生成工具
10. \* @author zyb
11. \* @since 2012-8-9
12. \*/
13. **public** **class** ProxyGeneratorUtils {
15. /\*\*
16. \* 把代理类的字节码写到硬盘上
17. \* @param path 保存路径
18. \*/
19. **public** **static** **void** writeProxyClassToHardDisk(String path) {
20. // 第一种方法，这种方式在刚才分析ProxyGenerator时已经知道了
21. // System.getProperties().put("sun.misc.ProxyGenerator.saveGeneratedFiles", true);
23. // 第二种方法
25. // 获取代理类的字节码
26. **byte**[] classFile = ProxyGenerator.generateProxyClass("$Proxy11", UserServiceImpl.**class**.getInterfaces());
28. FileOutputStream out = **null**;
30. **try** {
31. out = **new** FileOutputStream(path);
32. out.write(classFile);
33. out.flush();
34. } **catch** (Exception e) {
35. e.printStackTrace();
36. } **finally** {
37. **try** {
38. out.close();
39. } **catch** (IOException e) {
40. e.printStackTrace();
41. }
42. }
43. }
44. }
46. **package** dynamic.proxy;
48. **import** org.junit.Test;
50. /\*\*
51. \* 动态代理测试类
52. \* @author zyb
53. \* @since 2012-8-9
54. \*
55. \*/
56. **public** **class** ProxyTest {
58. @Test
59. **public** **void** testProxy() **throws** Throwable {
60. // 实例化目标对象
61. UserService userService = **new** UserServiceImpl();
63. // 实例化InvocationHandler
64. MyInvocationHandler invocationHandler = **new** MyInvocationHandler(userService);
66. // 根据目标对象生成代理对象
67. UserService proxy = (UserService) invocationHandler.getProxy();
69. // 调用代理对象的方法
70. proxy.add();
72. }
74. @Test
75. **public** **void** testGenerateProxyClass() {
76. ProxyGeneratorUtils.writeProxyClassToHardDisk("F:/$Proxy11.class");
77. }
78. }

通过以上代码，就可以在F盘上生成一个$Proxy.class文件了，现在用反编译工具来看一下这个class文件里面的内容。

**Java代码  [收藏代码](javascript:void())**

1. // Decompiled by DJ v3.11.11.95 Copyright 2009 Atanas Neshkov  Date: 2012/8/9 20:11:32
2. // Home Page: http://members.fortunecity.com/neshkov/dj.html  http://www.neshkov.com/dj.html - Check often for new version!
3. // Decompiler options: packimports(3)
5. **import** dynamic.proxy.UserService;
6. **import** java.lang.reflect.\*;
8. **public** **final** **class** $Proxy11 **extends** Proxy
9. **implements** UserService
10. {
12. // 构造方法，参数就是刚才传过来的MyInvocationHandler类的实例
13. **public** $Proxy11(InvocationHandler invocationhandler)
14. {
15. **super**(invocationhandler);
16. }
18. **public** **final** **boolean** equals(Object obj)
19. {
20. **try**
21. {
22. **return** ((Boolean)**super**.h.invoke(**this**, m1, **new** Object[] {
23. obj
24. })).booleanValue();
25. }
26. **catch**(Error \_ex) { }
27. **catch**(Throwable throwable)
28. {
29. **throw** **new** UndeclaredThrowableException(throwable);
30. }
31. }
33. /\*\*
34. \* 这个方法是关键部分
35. \*/
36. **public** **final** **void** add()
37. {
38. **try**
39. {
40. // 实际上就是调用MyInvocationHandler的public Object invoke(Object proxy, Method method, Object[] args)方法，第二个问题就解决了
41. **super**.h.invoke(**this**, m3, **null**);
42. **return**;
43. }
44. **catch**(Error \_ex) { }
45. **catch**(Throwable throwable)
46. {
47. **throw** **new** UndeclaredThrowableException(throwable);
48. }
49. }
51. **public** **final** **int** hashCode()
52. {
53. **try**
54. {
55. **return** ((Integer)**super**.h.invoke(**this**, m0, **null**)).intValue();
56. }
57. **catch**(Error \_ex) { }
58. **catch**(Throwable throwable)
59. {
60. **throw** **new** UndeclaredThrowableException(throwable);
61. }
62. }
64. **public** **final** String toString()
65. {
66. **try**
67. {
68. **return** (String)**super**.h.invoke(**this**, m2, **null**);
69. }
70. **catch**(Error \_ex) { }
71. **catch**(Throwable throwable)
72. {
73. **throw** **new** UndeclaredThrowableException(throwable);
74. }
75. }
77. **private** **static** Method m1;
78. **private** **static** Method m3;
79. **private** **static** Method m0;
80. **private** **static** Method m2;
82. // 在静态代码块中获取了4个方法：Object中的equals方法、UserService中的add方法、Object中的hashCode方法、Object中toString方法
83. **static**
84. {
85. **try**
86. {
87. m1 = Class.forName("java.lang.Object").getMethod("equals", **new** Class[] {
88. Class.forName("java.lang.Object")
89. });
90. m3 = Class.forName("dynamic.proxy.UserService").getMethod("add", **new** Class[0]);
91. m0 = Class.forName("java.lang.Object").getMethod("hashCode", **new** Class[0]);
92. m2 = Class.forName("java.lang.Object").getMethod("toString", **new** Class[0]);
93. }
94. **catch**(NoSuchMethodException nosuchmethodexception)
95. {
96. **throw** **new** NoSuchMethodError(nosuchmethodexception.getMessage());
97. }
98. **catch**(ClassNotFoundException classnotfoundexception)
99. {
100. **throw** **new** NoClassDefFoundError(classnotfoundexception.getMessage());
101. }
102. }
103. }

好了，到目前为止，前面 的两个问题都已经知道回事了，现在再用JDK动态代理的时候就不只会用而已了，真正的达到了“知其然，知其所以然”的目的。。。http://rejoy.iteye.com/images/smiles/icon_biggrin.gif http://rejoy.iteye.com/images/smiles/icon_biggrin.gif http://rejoy.iteye.com/images/smiles/icon_biggrin.gif   
  
就写到这了，累死了。。http://rejoy.iteye.com/images/smiles/icon_cry.gif http://rejoy.iteye.com/images/smiles/icon_cry.gif