**3.3  Spring AOP拦截器调用的实现**

**3.3.1  JdkDynamicAopProxy的invoke拦截**

上面我们看到了在Spring中通过ProxyFactoryBean实现AOP功能的第一步，得到AopProxy代理对象的基本过程，以及通过使用JDK和CGLIB最终产生AopProxy代理对象的实现原理。下面我们看看AopProxy代理对象的拦截机制是怎样发挥作用和实现AOP功能的。在JdkDynamicAopProxy中生成Proxy对象时，我们回顾一下它的AopProxy代理对象的生成调用，如下所示。

1. Proxy.newProxyInstance(classLoader, proxiedInterfaces, **this**);

这里的this参数对应的是InvocationHandler对象，InvocationHandler是JDK定义的反射类的一个接口，这个接口定义了invoke方法，而这个invoke方法是作为JDKProxy代理对象进行拦截的回调入口出现的。我们看到，在JdkDynamicAopProxy实现了InvocationHandler接口，也就是说当Proxy对象的代理方法被调用时，JdkDynamicAopProxy的invoke方法作为Proxy对象的回调函数而被触发，从而通过invoke的具体实现，来完成对目标对象方法调用的拦截或者说功能增强的工作。下面我们看看JdkDynamicAopProxy的invoke方法实现，如代码清单3-18所示。可以看到，对Proxy对象完成的代理设置是在invoke方法中完成的，这些设置包括获取目标对象、拦截器链，同时把这些对象作为输入，创建了ReflectiveMethodInvocation对象，通过这个ReflectiveMethodInvocation对象来完成对AOP功能实现的封装。在这个invoke方法中，包含了一个完整的拦截器链对目标对象的拦截过程，比如获得拦截器链并对其中的拦截器进行配置，逐个运行拦截器链里的拦截增强，直到最后对目标对象方法的运行，等等。

代码清单3-18  AopProxy代理对象的回调

1. **public** Object invoke(Object proxy, Method method, Object[]
2. args) **throws** Throwable {
3. MethodInvocation invocation = **null**;
4. Object oldProxy = **null**;
5. **boolean** setProxyContext = **false**;
6. TargetSource targetSource =
7. **this**.advised.targetSource;
8. Class targetClass = **null**;
9. Object target = **null**;
10. **try** {
11. **if** (!**this**.equalsDefined &&
12. AopUtils.isEqualsMethod(method)) {
13. // The target does not implement the
14. equals(Object) method itself.
15. **return** equals(args[0]);
16. }
17. **if** (!**this**.hashCodeDefined &&
18. AopUtils.isHashCodeMethod(method)) {
19. // The target does not implement the
20. hashCode() method itself.
21. **return** hashCode();
22. }
23. **if** (!**this**.advised.opaque &&
24. method.getDeclaringClass().isInterface() &&
26. method.getDeclaringClass().isAssignableFrom(Advised.**class**))
27. {
28. // Service invocations on
29. ProxyConfig with the proxy config...
30. **return**
31. AopUtils.invokeJoinpointUsingReflection(**this**.advised,
32. method, args);
33. }
34. Object retVal = **null**;
35. **if** (**this**.advised.exposeProxy) {
36. // Make invocation available if
37. necessary.
38. oldProxy =
39. AopContext.setCurrentProxy(proxy);
40. setProxyContext = **true**;
41. }
42. /\*\*
43. \* May be null. Get as late as possible to
44. minimize the time we "own" the
45. \* target, in case it comes from a pool.
46. \*/
47. //得到目标对象的地方。
48. target = targetSource.getTarget();
49. **if** (target != **null**) {
50. targetClass = target.getClass();
51. }
52. // Get the interception chain for this
53. method.
54. // 这里获得定义好的拦截器链。
55. List<Object> chain =
56. **this**.advised.getInterceptorsAndDynamicInterception
57. Advice(method, targetClass);
58. /\*\*
59. \* Check whether we have any advice. If we
60. don't, we can fallback on
61. \* direct reflective invocation of the
62. target, and avoid creating a MethodInvocation.
63. \*/
65. //
66. 如果没有设定拦截器，那么我们就直接调用target的对应方法。
67. **if** (chain.isEmpty()) {
68. /\*\*
69. \* We can skip creating a
70. MethodInvocation: just invoke the target directly
71. \* Note that the final invoker must
72. be an InvokerInterceptor so we
73. \* know it does nothing but a
74. reflective operation on the target, and no hot
75. \* swapping or fancy proxying.
76. \*/
77. retVal =
78. AopUtils.invokeJoinpointUsingReflection(target, method,
79. args);
80. }
81. **else** {
82. // We need to create a method
83. invocation...
84. /\*\*
85. \*
86. 如果有拦截器的设定，那么需要调用拦截器之后才调用目标对象的相
87. 应方法，
88. \*
89. 通过构造一个ReflectiveMethodInvocation来实现，下面我们会看
90. \*
91. 这个ReflectiveMethodInvocation类的具体实现。
92. \*/
93. invocation = **new**
94. ReflectiveMethodInvocation(proxy, target, method,
95. args, targetClass, chain);
96. // Proceed to the joinpoint through
97. the interceptor chain.
98. //沿着拦截器链继续前进。
99. retVal = invocation.proceed();
100. }
102. // Massage return value if necessary.
103. **if** (retVal != **null** && retVal == target &&
104. method.getReturnType().
105. isInstance(Proxy) &&
106. !RawTargetAccess.**class**.isAssignableFrom
107. (method.getDeclaringClass())) {
108. /\*\*
109. \* Special case: it returned "this"
110. and the return type of the method
111. \* is type-compatible. Note that we
112. can't help if the target sets
113. \* a reference to itself in another
114. returned object.
115. \*/
116. retVal = Proxy;
117. }
118. **return** retVal;
119. }
120. **finally** {
121. **if** (target != **null** &&
122. !targetSource.isStatic()) {
123. // Must have come from TargetSource.
124. targetSource.releaseTarget(target);
125. }
126. **if** (setProxyContext) {
127. // Restore old proxy.
129. AopContext.setCurrentProxy(oldProxy);
130. }
131. }
132. }