**3.2.4  CGLIB生成AopProxy代理对象**

在AopProxy接口实现中，可以看到使用CGLIB来生成Proxy代理对象，这个Proxy代理对象的生成，可以在Cglib2AopProxy的代码实现中看到，同样是在AopProxy的接口方法getProxy的实现完成的，如代码清单3-17所示。可以看到具体对CGLIB的使用，比如对Enhancer对象的配置，以及通过Enhancer对象生成代理对象的过程。在这个生成代理对象的过程中，需要注意的是对Enhancer对象callback回调的设置，正是通过这些callback回调，封装了Spring AOP的实现，就像我们在前面看到的JDK的Proxy对象的invoke回调方法一样；在Enhancer的callback回调设置中，实际上是通过设置DynamicAdvisedInterceptor拦截器来完成AOP功能的，如果读者感兴趣的话，可以在getCallbacks方法实现中看到回调DynamicAdvisedInterceptor的设置：

1. Callback aopInterceptor =
2. **new** DynamicAdvisedInterceptor(**this**.advised);

对于这个DynamicAdvisedInterceptor中的回调实现，我们将在后面详细地进行分析。

代码清单3-17  Cglib2AopProxy生成AopProxy代理对象

1. **public** Object getProxy(ClassLoader classLoader) {
2. **if** (logger.isDebugEnabled()) {
3. logger.debug("Creating CGLIB2 proxy: target
4. source is " + **this**.advised.
5. getTargetSource());
6. }
7. //从advised中取得在IoC容器中配置的target对象。
8. **try** {
9. Class rootClass =
10. **this**.advised.getTargetClass();
11. Assert.state(rootClass != **null**, "Target
12. **class** must be available **for**
13. creating a CGLIB proxy");
14. Class proxySuperClass = rootClass;
15. **if** (AopUtils.isCglibProxyClass(rootClass)) {
16. proxySuperClass =
17. rootClass.getSuperclass();
18. Class[] additionalInterfaces =
19. rootClass.getInterfaces();
20. **for** (Class additionalInterface :
21. additionalInterfaces) {
23. **this**.advised.addInterface(additionalInterface);
24. }
25. }
26. // Validate the class, writing log messages
27. as necessary.
28. validateClassIfNecessary(proxySuperClass);
29. // Configure CGLIB Enhancer...
30. //
31. 创建并配置CGLIB的Enhancer，这个Enhancer对象是CGLIB的主要操作
32. 类。
33. Enhancer enhancer = createEnhancer();
34. **if** (classLoader != **null**) {
36. enhancer.setClassLoader(classLoader);
37. **if** (classLoader **instanceof**
38. SmartClassLoader &&
39. ((SmartClassLoader)
40. classLoader).isClassReloadable
41. (proxySuperClass)) {
42. enhancer.setUseCache(**false**);
43. }
44. }
45. /\*\*
47. \*设置Enhancer对象，包括设置代理接口，回调方法，回调方法来自
49. \*advised的IoC配置，比如使用AOP的DynamicAdvisedInterceptor拦
50. 截器。
51. \*/
52. enhancer.setSuperclass(proxySuperClass);
53. enhancer.setStrategy(**new**
54. UndeclaredThrowableStrategy(UndeclaredThrowable
55. Exception.**class**));
57. enhancer.setInterfaces(AopProxyUtils.completeProxiedInterfac
58. es(**this**.advised));
60. enhancer.setInterceptDuringConstruction(**false**);
61. Callback[] callbacks =
62. getCallbacks(rootClass);
63. enhancer.setCallbacks(callbacks);
64. enhancer.setCallbackFilter(**new**
65. ProxyCallbackFilter(
67. **this**.advised.getConfigurationOnlyCopy(),
68. **this**.fixedInterceptorMap,
69. **this**.fixedInterceptorOffset));
70. Class[] types = **new** Class[callbacks.length];
71. **for** (**int** x = 0; x < types.length; x++) {
72. types[x] = callbacks[x].getClass();
73. }
74. enhancer.setCallbackTypes(types);
75. // 通过Enhancer生成代理对象。
76. Object proxy;
77. **if** (**this**.constructorArgs != **null**) {
78. Proxy =
79. enhancer.create(**this**.constructorArgTypes,
80. **this**.constructorArgs);
81. }
82. **else** {
83. proxy = enhancer.create();
84. }
85. **return** proxy;
86. }
87. **catch** (CodeGenerationException ex) {
88. **throw** **new** AopConfigException("Could not
89. generate CGLIB subclass of **class** [" +
91. **this**.advised.getTargetClass() + "]: " +
92. "Common causes of **this**
93. problem include using a **final** **class** or a
94. non-visible **class**",ex);
95. }
96. **catch** (IllegalArgumentException ex) {
97. **throw** **new** AopConfigException("Could not
98. generate CGLIB subclass of **class** [" +
100. **this**.advised.getTargetClass() + "]: " +
101. "Common causes of **this**
102. problem include using a **final** **class** or a
103. non-visible **class**",ex);
104. }
105. **catch** (Exception ex) {
106. // TargetSource.getTarget() failed。
107. **throw** **new** AopConfigException("Unexpected AOP
108. exception", ex);
109. }
110. }

这样，通过使用AopProxy对象封装target目标对象之后，通过ProxyFactoryBean的getObject方法得到的对象就不是一个普通的Java对象了，而是一个AopProxy代理对象。在ProxyFactoryBean中配置的target目标对象，这个时候已经不会直接暴露给应用，而是作为AOP实现的一部分。对target目标对象的方法调用会首先被AopProxy代理对象拦截，对于不同的AopProxy代理对象生成方式，会使用不同的拦截回调入口。例如，对于JDK的AopProxy代理对象，使用的是InvocationHandler的invoke回调入口；而对于CGLIB的AopProxy代理对象，使用的是设置好的callback回调，这是由对CGLIB的使用来决定的；在这些callback回调中，对于AOP实现，是通过DynamicAdvisedInterceptor来完成的，而DynamicAdvisedInterceptor的回调入口是intercept方法。通过这一系列的准备，已经为实现AOP的横切机制奠定了基础，在这个基础上，AOP的Advisor已经可以通过AopProxy代理对象的拦截机制，对需要它进行增强的target目标对象发挥切面的强大威力了。

可以把AOP的实现部分看成由基础设施准备和AOP运行辅助这两个部分组成。这里的AopProxy代理对象的生成，可以看成是一个静态的AOP基础设施的建立过程。通过这个准备过程把代理对象、拦截器这些待调用的部分都准备好，等待着AOP运行过程中对这些基础设施的使用。对于应用触发的AOP应用，会涉及AOP框架的运行和对AOP基础设施的使用。这些动态的运行部分，是从我们前面提到的拦截器回调入口开始的，这些拦截器调用的实现原理和AopProxy代理对象生成一样，也是AOP实现的重要组成部分，同时也是我们下面要重点分析的内容，让我们继续深入到AopProxy代理对象的回调实现中去看一看，慢慢地揭开Spring AOP实现的另一层神秘的面纱。