**3.2.2 ProxyFactoryBean生成AopProxy**

在Spring AOP的使用中，我们已经了解到，可以通过ProxyFactoryBean来配置目标对象和方面行为。这个ProxyFactoryBean是一个FactoryBean，对FactoryBean这种Spring应用中经常出现的Bean的工作形式，读者一定不会感到陌生，对于FactoryBean的工作原理，我们在结合IoC容器的实现原理分析中已经做过阐述。在ProxyFactoryBean中，通过interceptorNames属性来配置已经定义好的通知器Advisor。虽然这里的名字叫interceptNames，但值得注意的是，实际上却是供AOP应用配置通知器的地方。在ProxyFactoryBean中，需要为target目标对象生成Proxy代理对象，从而为AOP横切面的编织做好准备工作。这些具体的代理对象生成工作，在以后的实现原理分析中，我们可以看到是通过JDK的Proxy或CGLIB来完成的。

在ProxyFactoryBean中，它的AOP实现需要依赖JDK或者CGLIB提供的Proxy特性。从FactoryBean中获取对象，是用getObject()方法作为入口完成的；让我们进入ProxyFactoryBean的实现中去，这个我们一点也不陌生的getObject方法，是FactoryBean需要实现的接口。对ProxyFactoryBean来说，把需要对target目标对象增加的增强处理，都通过getObject方法进行封装了，这些增强处理是为AOP功能的实现提供服务的。对于getObject的实现如代码清单3-11所示。getObject方法首先对通知器链进行初始化，通知器链封装了一系列的拦截器，这些拦截器都要从配置中读取，然后为代理对象的生成做好准备。在生成代理对象的时候，因为Spring中有sigleton类型和prototype类型这两种不同的Bean，所以这里对代理对象的生成需要做一个区分。

代码清单3-11 ProxyFactoryBean的getObject

1. **public** Object getObject() **throws** BeansException {
2. //这里初始化通知器链。
3. initializeAdvisorChain();
4. //这里对singleton和protype的类型进行区分,生成对应的proxy。
5. **if** (isSingleton()) {
6. **return** getSingletonInstance();
7. }
8. **else** {
9. **if** (**this**.targetName == **null**) {
10. logger.warn("Using non-singleton
11. proxies with singleton targets is
12. often undesirable. " +
13. "Enable prototype proxies by setting the 'targetName' property.");
14. }
15. **return** newPrototypeInstance();
16. }
17. }

为Proxy代理对象配置Advisor链是在initializeAdvisorChain方法中完成的，如代码清单3-12所示。这个初始化过程有一个标志位advisorChainInitialized，这个标志用来表示通知器链是否已经初始化。如果已经初始化，那么这里就不会再初始化，而是直接返回。也就是说，这个初始化的工作，发生在应用第一次通过ProxyFactoryBean去获取代理对象的时候。在完成这个初始化之后，接着会读取配置中出现的所有通知器，这个取得通知器的过程也比较简单，把通知器的名字交给容器的getBean方法就可以了，这是通过对IoC容器实现的一个回调来完成的。然后把从IoC容器中取得的通知器加入到拦截器链中，这个动作是由addAdvisorOn-ChainCreation方法来实现的。

代码清单3-12 对Advisor配置链的初始化

1. **private** **synchronized** **void** initializeAdvisorChain() **throws**
2. AopConfigException,
3. BeansException {
4. **if** (**this**.advisorChainInitialized) {
5. **return**;
6. }
7. **if** (!ObjectUtils.isEmpty(**this**.interceptorNames)) {
8. **if** (**this**.beanFactory == **null**) {
9. **throw** **new** IllegalStateException("No
10. BeanFactory available anymore
11. (probably due to serialization) " +
12. "- cannot resolve interceptor names
13. " + Arrays.asList(**this**.
14. interceptorNames));
15. }
16. /\*\*
17. \* Globals can't be last unless we specified
18. a targetSource
19. \* using the property...
20. \*/
21. **if**
22. (**this**.interceptorNames[**this**.interceptorNames.length -
23. 1].endsWith
24. (GLOBAL\_SUFFIX) && **this**.targetName == **null** &&
25. **this**.targetSource ==
26. EMPTY\_TARGET\_
27. SOURCE) {
28. **throw** **new** AopConfigException("Target
29. required after globals");
30. }
32. // Materialize interceptor chain from bean
33. names.
34. //
35. 这里是添加advisor链的调用，是通过interceptorNames属性来进行
36. 配置的。
37. **for** (String name : **this**.interceptorNames) {
38. **if** (logger.isTraceEnabled()) {
39. logger.trace("Configuring
40. advisor or advice '" + name + "'");
41. }
43. **if** (name.endsWith(GLOBAL\_SUFFIX)) {
44. **if** (!(**this**.beanFactory
45. **instanceof** ListableBeanFactory)) {
46. **throw** **new**
47. AopConfigException(
48. "Can
49. only use global advisors or interceptors with
50. a
51. ListableBeanFactory");
52. }
54. addGlobalAdvisor((ListableBeanFactory) **this**.beanFactory,
56. name.substring(0, name.length() - GLOBAL\_SUFFIX.length()));
57. }
59. **else** {
60. // If we get here, we need
61. to add a named interceptor.
62. // We must check if it's a
63. singleton or prototype.
64. Object advice;
65. **if** (**this**.singleton ||
66. **this**.beanFactory.isSingleton(name)) {
67. // Add the real
68. Advisor/Advice to the chain.
69. advice =
70. **this**.beanFactory.getBean(name);
71. }
72. **else** {
73. // It's a prototype
74. Advice or Advisor: replace with a prototype.
75. /\*\*
76. \*Avoid unnecessary
77. creation of prototype bean just for
78. \*advisor chain
79. initialization.
80. \*/
81. advice = **new**
82. PrototypePlaceholderAdvisor(name);
83. }
85. addAdvisorOnChainCreation(advice, name);
86. }
87. }
88. }
89. **this**.advisorChainInitialized = **true**;
90. }

生成singleton的代理对象在getSingletonInstance()的代码中完成，这个方法是ProxyFactoryBean生成AopProxy代理对象的调用入口。在代理对象中会封装对target目标对象的调用，也就是说针对target对象的方法调用行为，会被这里生成的代理对象所拦截。具体的生成过程是，首先需要读取ProxyFactoryBean中的配置，为生成代理对象做好必要的准备，比如设置代理的方法调用接口等。对于在getSingletonInstance()方法中代理对象的生成过程，如代码清单3-13所示。

代码清单3-13 生成单件代理对象

1. **private** **synchronized** Object getSingletonInstance() {
2. **if** (**this**.singletonInstance == **null**) {
3. **this**.targetSource = freshTargetSource();
4. **if** (**this**.autodetectInterfaces &&
5. getProxiedInterfaces().length == 0
6. && !isProxyTargetClass()) {
7. // Rely on AOP infrastructure to
8. tell us what interfaces to proxy.
9. Class targetClass =
10. getTargetClass();
11. **if** (targetClass == **null**) {
12. **throw** **new**
13. FactoryBeanNotInitializedException("Cannot determine
14. target **class** **for** proxy");
15. }
16. // 这里设置代理对象的接口。
17. setInterfaces(ClassUtils.getAllInterfacesForClass(targetClas
18. s, **this**.proxyClassLoader));
19. }
20. // Initialize the shared singleton instance.
21. **super**.setFrozen(**this**.freezeProxy);
22. //
23. 注意这里的方法会使用ProxyFactory来生成我们需要的Proxy。
24. **this**.singletonInstance =
25. getProxy(createAopProxy());
26. }
27. **return** **this**.singletonInstance;
28. }
29. //使用createAopProxy返回的AopProxy来得到代理对象。
30. **protected** Object getProxy(AopProxy aopProxy) {
31. **return** aopProxy.getProxy(**this**.proxyClassLoader);
32. }

这里出现了AopProxy类型的对象，Spring使用这个AopProxy接口类把AOP代理对象的实现与框架的其他部分有效地分离开来。AopProxy是一个接口，它有两个子类实现，一个是Cglib2AopProxy，另一个是JdkDynamicProxy。顾名思义，对这两个AopProxy接口的子类实现，Spring分别使用CGLIB和JDK来生成需要的Proxy代理对象。

具体的代理对象的生成，是在ProxyFactoryBean的基类AdvisedSupport的实现中通过AopProxyFactory完成的，这个代理对象要么从JDK中生成，要么借助CGLIB获得。因为ProxyFactoryBean本身就是AdvisedSupport的子类，在ProxyFactoryBean中获得AopProxy是很方便的，具体的AopProxy生成过程，我们可以在ProxyCreatorSupport中看到。至于需要生成什么样的代理对象，信息都封装在AdvisedSupport里，这个对象也是生成AopProxy的方法的输入参数，这里设置为this本身，因为ProxyCreatorSupport本身就是AdvisedSupport的子类。在ProxyCreatorSupport中生成代理对象的入口实现，如代码清单3-14所示。

代码清单3-14 ProxyCreatorSupport生成AopProxy对象

1. **protected** **final** **synchronized** AopProxy createAopProxy() {
2. **if** (!**this**.active) {
3. activate();
4. }
5. /\*\*
6. \*通过AopProxyFactory取得AopProxy，
7. 这个AopProxyFactory是在
8. \*初始化函数中定义的，使用的是DefaultAopProxyFactory。
9. \*/
10. **return** getAopProxyFactory().createAopProxy(**this**);
11. }

这里使用了AopProxyFactory来创建AopProxy，AopProxyFactory使用的是DefaultAopProxyFactory。这个被使用的AopProxyFactory，作为AopProxy的创建工厂对象，是在ProxyFactoryBean的基类ProxyCreatorSupport中被创建的。在创建AopProxyFactory的时候，它被设置为DefaultAopProxyFactory。很显然，Spring给出了这个默认的AopProxyFactory工厂的实现。有了这个AopProxyFactory对象以后，问题就转换为：在DefaultAopProxyFactory中AopProxy是怎样生成的问题了。

关于AopProxy代理对象的生成，需要考虑使用哪种生成方式，如果目标对象是接口类，那么适合使用JDK来生成代理对象，否则Spring会使用CGLIB来生成目标对象的代理对象。为了满足不同的代理对象生成的要求，DefaultAopProxyFactory作为AopProxy对象的生产工厂，可以根据不同的需要生成这两种AopProxy对象。对于AopProxy对象的生产过程，在DefaultAopProxyFactory中创建AopProxy的代码中可以清楚地看到，但这是一个比较高层次的AopProxy代理对象的生成过程，如代码清单3-15所示。所谓高层次，是指在DefaultAopProxyFactory创建AopProxy的过程中，对不同的AopProxy代理对象的生成，所涉及的生成策略和场景做了相应的设计，但是对于具体的AopProxy代理对象的生成，最终并没有由这个DefaultAopProxyFactory来完成，比如对JDK和CGLIB这些具体的技术的使用，对具体的实现层次的代理对象的生成，是由Spring封装的JdkDynamicAopProxy和CglibProxyFactory类来完成的。

在AopProxy代理对象的生成过程中，首先要从AdvisedSupport对象中取得配置的目标对象，这个目标对象是实现AOP功能所必需的，道理很简单，AOP完成的是切面应用对目标对象的增强，皮之不存，毛将焉附，这个目标对象可以看做是"皮"，而AOP切面增强就是依附于这块皮上的"毛发"。如果这里发现没有配置目标对象的话，会直接抛出异常，提醒AOP应用，需要提供正确的目标对象的配置。在对目标对象配置的检查完成以后，需要根据配置的情况来决定使用什么方式来创建AopProxy代理对象。一般而言，默认的方式是使用JDK来产生AopProxy代理对象，但是如果遇到配置的目标对象不是接口类的实现的时候，会使用CGLIB来产生AopProxy代理对象；在使用CGLIB来产生AopProxy代理对象的时候，因为CGLIB是一个第三方的类库，本身不在JDK的基本类库中，所以需要在CLASSPATH路径中正确地配置，以便能够加载和使用。在Spring中，使用JDK和CGLIB来生成AopProxy代理对象的工作，是由JdkDynamicAopProxy和CglibProxyFactory来完成的。对于详细的代理对象的生成过程，在下面的小节中，将逐个进行详细的分析。

代码清单3-15 在DefaultAopProxyFactory中创建AopProxy

1. **public** AopProxy createAopProxy(AdvisedSupport config) **throws**
2. AopConfigException {
3. **if** (config.isOptimize() ||
4. config.isProxyTargetClass() || hasNoUserSupplied
5. ProxyInterfaces(config)) {
6. Class targetClass = config.getTargetClass();
7. **if** (targetClass == **null**) {
8. **throw** **new**
9. AopConfigException("TargetSource cannot determine target
10. **class**: " +
11. "Either an **interface** or a target is
12. required **for** proxy creation.");
14. }//如果targetClass是接口类，使用JDK来生成Proxy。
15. **if** (targetClass.isInterface()) {
16. **return** **new**
17. JdkDynamicAopProxy(config);
18. }
19. **if** (!cglibAvailable) {
20. **throw** **new** AopConfigException(
21. "Cannot proxy target
22. **class** because CGLIB2 is not available. " +
23. "Add CGLIB to the
24. **class** path or specify proxy interfaces.");
25. }
26. //如果不是接口类要生成proxy，那么使用cglib来生成。
27. **return**
28. CglibProxyFactory.createCglibProxy(config);
29. }
30. **else** {
31. **return** **new** JdkDynamicAopProxy(config);
32. }
33. }