**3.3.5  配置通知器**

在整个AopProxy代理对象的拦截回调过程中，让我们先回到ReflectiveMethod Invocation类的proceed方法，在这个方法里，可以看到得到了配置的interceptor-OrInterceptionAdvice，如下所示。

1. Object interceptorOrInterceptionAdvice =
2. **this**.interceptorsAndDynamicMethodMatchers.
3. get(++**this**.currentInterceptorIndex);

这个interceptorOrInterceptionAdvice是获得的拦截器，它通过拦截器机制对目标对象的行为增强起作用。这个拦截器来自于interceptorsAndDynamicMethodMatchers，具体来说，它是interceptorsAndDynamicMethodMatchers持有的List中的一个元素。关于如何配置拦截器的问题，就被转化为这个List中的拦截器元素是从哪里来、在哪里配置的问题。我们接着对invoke调用进行回放，回到JdkDynamicAopProxy中的invoke方法中，可以看到这个List中的interceptors是在哪个调用中获取的。对于Cglib2AopProxy，也有类似的过程，只不过这个过程是在DynamicAdvisedInterceptor的intercept回调中实现的，如下所示。  
List<Object> chain =  
this.advised.getInterceptorsAndDynamicInterceptionAdvice(method, targetClass);

在上面的代码中可以看到，获取interceptors的操作是由advised对象完成的，这个advised是一个AdvisedSupport对象，从类的继承关系上看，这个AdvisedSupport类同时也是ProxyFactoryBean的基类。从AdvisedSupport的代码里可以看到 getInterceptorsAndDynamic

InterceptionAdvice的实现，如代码清单3-22所示。在这个方法里取得了拦截器链，在取得拦截器链的时候，为提高取得拦截器链的效率，还为这个拦截器链设置了缓存。

代码清单3-22  AdvisedSupport取得拦截器

1. **public** List<Object>
2. getInterceptorsAndDynamicInterceptionAdvice(Method method,
3. Class targetClass) {
4. /\*\*
5. \*这里使用了cache，会从cache去取已有的inteceptor链，
6. 但是第一次
7. \*还是需要自己动手生成的。这个inteceptor链的生成是由
8. advisorChainFactory完成的，
9. \*在这里使用的是DefaultAdvisorChainFactory。
10. \*/
11. MethodCacheKey cacheKey = **new**
12. MethodCacheKey(method);
13. List<Object> cached =
14. **this**.methodCache.get(cacheKey);
15. **if** (cached == **null**) {
16. cached =
17. **this**.advisorChainFactory.getInterceptorsAndDynamicIntercepti
18. onAdvice(
19. **this**, method, targetClass);
20. **this**.methodCache.put(cacheKey, cached);
21. }
22. **return** cached;
23. }

取得拦截器链的工作是由配置好的advisorChainFactory 来完成的，从名字上可以猜到，它是一个生成通知器链的工厂；在这里，advisorChainFactory 被配置成一个DefaultAdvisorChainFactory对象，在DefaultAdvisorChainFactory中实现了interceptor链的获取过程，如代码清单3-23所示。在这个获取过程中，首先设置了一个List，其长度是由配置的通知器的个数来决定的，这个配置就是我们在XML中对ProxyFactoryBean做的interceptNames属性的配置；然后，DefaultAdvisorChain-

Factory会通过一个AdvisorAdapterRegistry来实现拦截器的注册，在后面我们会看到，这个AdvisorAdapterRegistry对advice通知的织入功能起了很大的作用，关于这个AdvisorAdapterRegistry对象的实现原理，我们会在下面分析通知是如何实现增强的部分进行详细阐述。有了这个AdvisorAdapterRegistry注册器，由它来对从ProxyFactoryBean配置中得到的通知进行适配，从而获得相应的拦截器，再把它加入到前面设置好的List中去，完成这个所谓的拦截器注册过程。在这些拦截器适配和注册过程完成以后，这个List中的拦截器会被JDK生成的AopProxy代理对象的invoke方法，或者CGLIB代理对象的intercept拦截方法取得，并启动拦截器的invoke调用，最终触发通知的切面增强。

代码清单3-23  DefaultAdvisorChainFactory生成拦截器链

1. **public** List<Object>
2. getInterceptorsAndDynamicInterceptionAdvice(
3. Advised config, Method method, Class
4. targetClass) {
6. /\*\*
7. \* This is somewhat tricky... we have to process
8. introductions first,
9. \* but we need to preserve order in the ultimate
10. list.
11. \*/
12. //advisor链已经在config中持有了，这里我们可以直接使
13. 用。
14. List<Object> interceptorList = **new** ArrayList<Object>
15. (config.getAdvisors().length);
16. **boolean** hasIntroductions =
17. hasMatchingIntroductions(config, targetClass);
18. AdvisorAdapterRegistry registry =
19. GlobalAdvisorAdapterRegistry.getInstance();
20. **for** (Advisor advisor : config.getAdvisors()) {
21. **if** (advisor **instanceof**
22. PointcutAdvisor) {
23. // Add it conditionally.
24. PointcutAdvisor pointcutAdvisor =
25. (PointcutAdvisor) advisor;
26. **if** (config.isPreFiltered() ||
27. pointcutAdvisor.getPointcut().getClass
28. Filter().matches(targetClass)) {
29. /\*\*
30. \*拦截器链是通过AdvisorAdapterRegistry来加入的，这个AdvisorAd
31. apterRegistry
32. \*对advice织入起了很大的作用，在后面的分析中会看到。
33. \*/
34. MethodInterceptor[] interceptors =
35. registry.getInterceptors(advisor);
36. MethodMatcher mm =
37. pointcutAdvisor.getPointcut().gEtMethodMatcher();
38. //使用MethodMatchers的matches方法进行匹配判断。
39. **if**
40. (MethodMatchers.matches(mm, method, targetClass,
41. hasIntroductions)) {
42. **if** (mm.isRuntime())
43. {
44. /\*\*
45. \* Creating a new
46. object instance in the getInterceptors()method
47. \* isn't a problem as
48. we normally cache created chains.
49. \*/
50. **for**
51. (MethodInterceptor interceptor : interceptors) {
53. interceptorList.add(**new** InterceptorAndDynamicMethod
55. Matcher(interceptor, mm));
56. }
57. }
58. **else** {
60. interceptorList.addAll(Arrays.asList(interceptors));
61. }
62. }
63. }
64. }
65. **else** **if** (advisor **instanceof**
66. IntroductionAdvisor) {
67. IntroductionAdvisor ia =
68. (IntroductionAdvisor) advisor;
69. **if** (config.isPreFiltered() ||
70. ia.getClassFilter().matches(targetClass)) {
71. Interceptor[] interceptors =
72. registry.getInterceptors(advisor);
74. interceptorList.addAll(Arrays.asList(interceptors));
75. }
76. }
77. **else** {
78. Interceptor[] interceptors =
79. registry.getInterceptors(advisor);
81. interceptorList.addAll(Arrays.asList(interceptors));
82. }
83. }
84. **return** interceptorList;
85. }
86. // Determine whether the Advisors contain matching
87. introductions.
88. **private** **static** **boolean** hasMatchingIntroductions(Advised
89. config, ClasstargetClass) {
90. **for** (**int** i = 0; i < config.getAdvisors().length;
91. i++) {
92. Advisor advisor = config.getAdvisors()[i];
93. **if** (advisor **instanceof** IntroductionAdvisor)
94. {
95. IntroductionAdvisor ia =
96. (IntroductionAdvisor) advisor;
97. **if**
98. (ia.getClassFilter().matches(targetClass)) {
99. **return** **true**;
100. }
101. }
102. }
103. **return** **false**;
104. }

事实上，这里的advisor通知器是从AdvisorSupport中取得的，从对它调用过程上看会非常地清楚，如图3-10所示。

在ProxyFactoryBean的getObject方法中对advisor进行初始化的时候，从XML配置中获取了advisor通知器。在ProxyFactoryBean中，我们看看对advisor进行初始化的代码实现，如代码清单3-24所示。在这个初始化的advisor的取得中，可以看到对IoC容器的一个getBean回调，由这个对IoC容器的getBean调用来得到配置好的advisor通知器。

|  |
| --- |
| <http://images.51cto.com/files/uploadimg/20091222/151934984.jpg> |
|  |