**3.3.4  AOP拦截器链的调用**

在了解了对目标对象的直接调用以后，我们开始进入AOP实现的核心部分了，对于AOP是怎样完成对目标对象的增强的，这些实现是封装在AOP拦截器链中，由一个个具体的拦截器来完成的。

尽管我们在上面看到，使用JDK和CGLIB会生成不同的AopProxy代理对象，从而构造了不同的回调方法来启动对拦截器链的调用，比如在JdkDynamicAopProxy中的invoke方法，以及Cglib2AopProxy中使用DynamicAdvisedInterceptor的intercept方法。它们都使用了不同的AopProxy代理对象，但最终对AOP拦截的处理可谓殊途同归：它们对拦截器链的调用都是在ReflectiveMethodInvocation中通过proceed方法实现的。在这个proceed方法里，会逐个运行拦截器的拦截方法。在运行拦截器的拦截方法之前，需要对代理方法完成一个匹配判断，通过这个匹配判断来决定拦截器是否满足切面增强的要求。大家一定还记得，我们前面提到的，在Pointcut切点中需要进行matches的匹配过程，就是这个matches调用对方法进行匹配判断，来决定是否需要实行通知增强；以下看到的调用就是进行matches的地方，具体的处理过程在ReflectiveMethodInvocation的proceed方法中，如代码清单3-21所示。在proceed方法中先进行判断，如果现在已经运行到拦截器链的末尾，那么就会直接调用目标对象的实现方法；否则，沿着拦截器链继续进行，得到下一个拦截器，通过这个拦截器进行matches判断，判断是否适用于横切增强的场合，如果是，从拦截器中得到通知器，并启动通知器的invoke方法进行切面增强。在这个过程结束以后，会迭代调用proceed方法，直到拦截器链中的拦截器都完成以上的拦截过程为止。

代码清单3-21  拦截器的运行

1. **public** Object proceed() **throws** Throwable {
2. //  We start with an index of -1 and increment
3. early.
4. /\*\*
5. \*如果拦截器链中的拦截器迭代调用完毕，这里开始调用tar
6. get的函数，
7. \*这个函数是通过反射机制完成的，具体实现在：AopUtils.
8. invokeJoinpointUsingReflection方法里面。
9. \*/
10. **if** (**this**.currentInterceptorIndex ==
11. **this**.interceptorsAndDynamicMethod
12. Matchers.size() - 1) {
13. **return** invokeJoinpoint();
14. }
15. //这里沿着定义好的
16. interceptorOrInterceptionAdvice链进行处理。
17. Object interceptorOrInterceptionAdvice =
18. **this**.interceptorsAndDynamicMethodMatchers.get(++**this**
19. .currentInterceptor
20. Index);
21. **if** (interceptorOrInterceptionAdvice **instanceof**
22. InterceptorAndDynamic
23. MethodMatcher) {
24. /\*\*
25. \* Evaluate dynamic method matcher here:
26. static part will already have
27. \* been evaluated and found to match.
28. \*/
29. /\*\*
31. \*这里对拦截器进行动态匹配的判断，还记得我们前面分析的pointcu
32. t吗？
34. \*这里是触发进行匹配的地方，如果和定义的pointcut匹配，那么这
35. 个advice将会得到执行。
36. \*/
37. InterceptorAndDynamicMethodMatcher dm =
38. (InterceptorAndDynamicMethodMatcher)
39. interceptorOrInterceptionAdvice;
40. **if** (dm.methodMatcher.matches(**this**.method,
41. **this**.targetClass, **this**.arguments)) {
42. **return** dm.interceptor.invoke(**this**);
43. }
44. **else** {
45. // Dynamic matching failed.
46. // Skip this interceptor and invoke
47. the next in the chain.
48. //
49. 如果不匹配，那么proceed会被递归调用，直到所有的拦截器都被运
50. 行过为止。
51. **return** proceed();
52. }
53. }
54. **else** {
55. /\*\*
56. \* It's an interceptor, so we just invoke it:
57. The pointcut will have
58. \* been evaluated statically before this
59. object was constructed.
60. \*/
62. //如果是interceptor，直接调用这个interceptor对应的方法。
63. **return**((MethodInterceptor)
64. interceptorOrInterceptionAdvice).invoke(**this**);
65. }
66. }

以上就是整个拦截器及target目标对象方法被调用的过程。小荷才露尖尖角，我们已经在这里看到对advice通知的调用入口了，虽然这个大名鼎鼎的advice到现在还没有完全现身，但我们已经看到了它的运行轨迹；我们先提出一个疑问来提提大家的兴趣：这些advisor是怎样从配置文件中获得并配置到proxy的拦截器链中去的？我们平常使用的advice通知是怎样起作用的？这些都是了解AOP实现原理的重要问题，下面我们就这些问题已经展示的线索继续展开分析，去寻求这些问题的答案吧。