在DefaultAdvisorAdapterRegistry的getInterceptors调用中，我们可以看到MethodBeforeAdviceAdapter、AfterReturningAdviceAdapter以及ThrowsAdvice

Adapter这几个通知适配器，从名字上可以看到，它们完全是和advice一一对应的，在这里，它们作为适配器被加入到adaper的List中来了。换一个角度，从这几个类的设计层次和关系上看，它们都是实现AdvisorAdapter接口的同一层次的类，只是各自承担着不同的适配任务，一对一地服务于不同的advice实现。它们的类层次关系如图3-11所示。

|  |
| --- |
| <http://images.51cto.com/files/uploadimg/20091222/152943163.jpg> |
|  |

我们举MethodBeforeAdviceAdapter为例，看看它的具体实现，如代码清单3-28所示。这个MethodBeforeAdviceAdapter的实现并不复杂，它实现了AdvisorAdapter的两个接口方法，一个是supportsAdvice，它对advice的类型进行判断，如果advice是MethodBeforeAdvice的实例，那么返回值为true；另一个是对getInterceptor接口方法的实现，它把advice通知从通知器中取出，然后创建一个MethodBeforeAdviceInterceptor对象，通过这个MethodBeforeAdviceInterceptor对象把取得的advice通知包装起来，然后返回。

代码清单3-28  MethodBeforeAdviceAdapter的实现

1. **class** MethodBeforeAdviceAdapter **implements** AdvisorAdapter,
2. Serializable {
3. **public** **boolean** supportsAdvice(Advice advice) {
4. **return** (advice **instanceof** MethodBeforeAdvice);
5. }
6. **public** MethodInterceptor getInterceptor(Advisor advisor) {
7. MethodBeforeAdvice advice =
8. (MethodBeforeAdvice) advisor.getAdvice();
9. **return** **new** MethodBeforeAdviceInterceptor(advice);
10. }
11. }

到这里就非常清楚了，Spring AOP为了实现advice的织入，设计了特定的拦截器对这些功能进行了封装，虽然应用不会直接用到这些拦截器，但却是advice发挥作用的必不可少的基础设施。接着这条线索，我们还是使用MethodBeforeAdviceInterceptor作为例子，看看它是怎样完成对advice的封装的，如代码清单3-29所示。这个MethodBeforeAdviceInterceptor完成的是对MethodBeforeAdvice通知的封装，可以在MethodBeforeAdviceInterceptor设计的invoke回调方法中看到首先触发了advice的before回调，然后才是MethodInvocation的proceed方法调用。看到这里，就已经和前面我们在ReflectiveMethodInvocation中的proceed分析中联系起来了。在AopProxy代理对象触发的ReflectiveMethodInvocation的proceed方法中，在取得拦截器以后，启动了对拦截器invoke方法的调用。按照AOP的配置规则，ReflectiveMethodInvocation触发的拦截器invoke方法，最终会根据不同的advice类型，触发Spring对不同的advice的拦截器封装，比如对MethodBeforeAdvice，就会最终触发MethodBeforeAdviceInterceptor的invoke方法。在这个MethodBefore-

AdviceInterceptor方法中，先调用advice的before方法，即MethodBeforeAdvice所需要的对目标对象的增强效果：在方法调用之前完成通知增强。

代码清单3-29  MethodBeforeAdviceInterceptor的实现

1. **public** **class** MethodBeforeAdviceInterceptor **implements**
2. MethodInterceptor, Serializable {
3. **private** MethodBeforeAdvice advice;
4. /\*\*
5. \* Create a new MethodBeforeAdviceInterceptor for
6. the given advice.
7. \* @param advice the MethodBeforeAdvice to wrap
8. \*/
9. **public**
10. MethodBeforeAdviceInterceptor(MethodBeforeAdvice advice) {
11. Assert.notNull(advice, "Advice must not be
12. **null**");
13. **this**.advice = advice;
14. }
15. //这个invoke方法是拦截器的回调方法，会在代理对象的方
16. 法被调用的时候触发回调。
17. **public** Object invoke(MethodInvocation mi) **throws**
18. Throwable {
19. **this**.advice.before(mi.getMethod(),
20. mi.getArguments(), mi.getThis() );
21. **return** mi.proceed();
22. }
23. }

了解了MethodBeforeAdviceInterceptor的实现原理，对于其他的advice的实现也是可以举一反三的，比如对于AfterReturningAdviceInterceptor的实现，它和MethodBefore-

AdviceInterceptor实现不同的地方，就是在AfterReturningAdviceInterceptor的invoke方法中，先完成了MethodInvocation的proceed调用，也就是目标对象的方法调用，然后再启动advice的afterReturning回调，这些实现原理在代码中可以很清楚地看到，如代码清单3-30所示。

代码清单3-30  AfterReturningAdviceInterceptor的实现

1. **public** **class** AfterReturningAdviceInterceptor **implements**
2. MethodInterceptor,
3. AfterAdvice,Serializable {
4. **private** **final** AfterReturningAdvice advice;
5. /\*\*
6. \* Create a new AfterReturningAdviceInterceptor for
7. the given advice.
8. \* @param advice the AfterReturningAdvice to wrap
9. \*/
10. **public**
11. AfterReturningAdviceInterceptor(AfterReturningAdvice advice)
12. {
13. Assert.notNull(advice, "Advice must not be
14. **null**");
15. **this**.advice = advice;
16. }
17. **public** Object invoke(MethodInvocation mi) **throws**
18. Throwable {
19. Object retVal = mi.proceed();
20. **this**.advice.afterReturning(retVal,
21. mi.getMethod(), mi.getArguments(),
22. mi.getThis());
23. **return** retVal;
24. }
25. }

对于ThrowAdvice的实现原理，和上面两种情况是非常类似的，也是封装在对应的AdviceInterceptor中实现的，如代码清单3-31所示。只是相对于MethodBeforeAdvice和AfterReturningAdvice的回调方法调用，ThrowAdvice的回调方法调用比前两者的实现要复杂一些；它维护了exceptionHandlerMap来对应不同的方法调用场景，这个exceptionHandlerMap中handler的取得，是与触发ThrowAdvise增强的异常相关的。

代码清单3-31  ThrowsAdviceInterceptor的实现

1. **public** **class** ThrowsAdviceInterceptor **implements**
2. MethodInterceptor, AfterAdvice {
3. **private** **static** **final** String AFTER\_THROWING =
4. "afterThrowing";
5. **private** **static** **final** Log logger =
6. LogFactory.getLog(ThrowsAdviceInterceptor.
7. **class**);
8. **private** **final** Object throwsAdvice;
9. // Methods on throws advice, keyed by exception
10. **class**.
11. **private** **final** Map<Class, Method> exceptionHandlerMap
12. = **new** HashMap<Class,
13. Method>();
14. **public** ThrowsAdviceInterceptor(Object throwsAdvice)
15. {
16. Assert.notNull(throwsAdvice, "Advice must
17. not be **null**");
18. **this**.throwsAdvice = throwsAdvice;
19. //配置ThrowsAdvice的回调方法。
20. Method[] methods =
21. throwsAdvice.getClass().getMethods();
22. **for** (Method method : methods) {
23. **if**
24. (method.getName().equals(AFTER\_THROWING) &&
26. (method.getParameterTypes().length == 1 ||
27. method.getParameter
28. Types().length == 4) &&
29. Throwable.**class**.isAssignableFrom(method.getParameter
30. Types()[method.getParame
31. terTypes().length - 1])
32. ) {
33. // Have an exception
34. handler.
35. **this**.exceptionHandlerMap.put(method.getParameterType
36. s()
37. [method.getParameterTypes().length - 1], method);
38. **if** (logger.isDebugEnabled())
39. {
40. logger.debug("Found
41. exception handler method: " + method);
42. }
43. }
44. }
45. **if** (**this**.exceptionHandlerMap.isEmpty()) {
46. **throw** **new** IllegalArgumentException(
47. "At least one
48. handler method must be found in **class** [" +
49. throwsAdvice.getClass() +
50. "]");
51. }
52. }
53. **public** **int** getHandlerMethodCount() {
54. **return** **this**.exceptionHandlerMap.size();
55. }
56. **private** Method getExceptionHandler(Throwable
57. exception) {
58. Class exceptionClass = exception.getClass();
59. **if** (logger.isTraceEnabled()) {
60. logger.trace("Trying to find handler
61. **for** exception of type [" +
62. exceptionClass.getName() + "]");
63. }
64. Method handler =
65. **this**.exceptionHandlerMap.get(exceptionClass);
66. **while** (handler == **null** &&
67. !exceptionClass.equals(Throwable.**class**)) {
68. exceptionClass =
69. exceptionClass.getSuperclass();
70. handler =
71. **this**.exceptionHandlerMap.get(exceptionClass);
72. }
73. **if** (handler != **null** &&
74. logger.isDebugEnabled()) {
75. logger.debug("Found handler **for**
76. exception of type [" +
77. exceptionClass.getName() + "]: " + handler);
78. }
79. **return** handler;
80. }
81. **public** Object invoke(MethodInvocation mi) **throws**
82. Throwable {
83. /\*\*
85. \*把对目标对象的方法调用放入try/catch中，并在catch中触发Throw
86. Advice的回调，
87. \*然后把异常接着向外抛出，不做过多的处理。
88. \*/
89. **try** {
90. **return** mi.proceed();
91. }
92. **catch** (Throwable ex) {
93. Method handlerMethod =
94. getExceptionHandler(ex);
95. **if** (handlerMethod != **null**) {
96. invokeHandlerMethod(mi, ex,
97. handlerMethod);
98. }
99. **throw** ex;
100. }
101. }
102. //通过反射启动对ThrowAdvice回调方法的调用。
103. **private** **void** invokeHandlerMethod(MethodInvocation
104. mi, Throwable ex, Method
105. method) **throws** Throwable {
106. Object[] handlerArgs;
107. **if** (method.getParameterTypes().length == 1)
108. {
109. handlerArgs = **new** Object[] { ex };
110. }
111. **else** {
112. handlerArgs = **new** Object[]
113. {mi.getMethod(), mi.getArguments(), mi.
114. getThis(), ex};
115. }
116. **try** {
117. method.invoke(**this**.throwsAdvice,
118. handlerArgs);
119. }
120. **catch** (InvocationTargetException targetEx) {
121. **throw** targetEx.getTargetException();
122. }
123. }
124. }