**3.3.3  目标对象方法的调用**

如果没有设置拦截器，那么会对目标对象的方法直接进行调用，对于JdkDynamicAopProxy代理对象，这个对目标对象的方法调用是通过AopUtils使用反射机制完成的，在AopUtils.

invokeJoinpointUsingReflection的方法中实现，如代码清单3-20所示。在这个调用中，首先得到调用方法的反射对象，然后使用invoke启动对方法反射对象的调用。

代码清单3-20  使用反射完成目标对象的方法调用

1. **public** **static** Object invokeJoinpointUsingReflection(Object
2. target, Method method,
3. Object[] args)
4. **throws** Throwable {
5. // Use reflection to invoke the method.
6. // 这里是使用反射调用target对象方法的地方。
7. **try** {
8. ReflectionUtils.makeAccessible(method);
9. **return** method.invoke(target, args);
10. }
11. **catch** (InvocationTargetException ex) {
12. // Invoked method threw a checked exception.
13. // We must rethrow it. The client won't see
14. the interceptor.
15. **throw** ex.getTargetException();
16. }
17. **catch** (IllegalArgumentException ex) {
18. **throw** **new** AopInvocationException("AOP
19. configuration seems to be invalid:
20. tried calling method [" + method + "] on target [" +
21. target + "]", ex);
22. }
23. **catch** (IllegalAccessException ex) {
24. **throw** **new** AopInvocationException("Could not
25. access method [" + method +
26. "]", ex);
27. }
28. }

对于使用Cglib2AopProxy的代理对象，它对目标对象的调用是通过CGLIB的MethodProxy对象来直接完成的，这个对象的使用是由CGLIB的设计来决定的；对于具体的调用在DynamicAdvisedInterceptor的intercept方法中可以看到，使用的是CGLIB封装好的功能，相对JdkDynamicAopProxy的实现来说，形式上看起来较为简单，但它们的功能却都是一样的，都是完成对目标对象方法的调用，具体的代码实现如下。

1. retVal = methodProxy.invoke(target, args);