[[置顶] Spring AOP 实现原理](http://blog.csdn.net/moreevan/article/details/11977115)

**什么是AOP**

AOP（Aspect-OrientedProgramming，面向方面编程），可以说是OOP（Object-Oriented Programing，面向对象编程）的补充和完善。OOP引入封装、继承和多态性等概念来建立一种对象层次结构，用以模拟公共行为的一个集合。当我们需要为分散的对象引入公共行为的时候，OOP则显得无能为力。也就是说，OOP允许你定义从上到下的关系，但并不适合定义从左到右的关系。例如日志功能。日志代码往往水平地散布在所有对象层次中，而与它所散布到的对象的核心功能毫无关系。对于其他类型的代码，如安全性、异常处理和透明的持续性也是如此。这种散布在各处的无关的代码被称为横切（cross-cutting）代码，在OOP设计中，它导致了大量代码的重复，而不利于各个模块的重用。

而AOP技术则恰恰相反，它利用一种称为“横切”的技术，剖解开封装的对象内部，并将那些影响了多个类的公共行为封装到一个可重用模块，并将其名为“Aspect”，即方面。所谓“方面”，简单地说，就是将那些与业务无关，却为业务模块所共同调用的逻辑或责任封装起来，便于减少系统的重复代码，降低模块间的耦合度，并有利于未来的可操作性和可维护性。AOP代表的是一个横向的关系，如果说“对象”是一个空心的圆柱体，其中封装的是对象的属性和行为；那么面向方面编程的方法，就仿佛一把利刃，将这些空心圆柱体剖开，以获得其内部的消息。而剖开的切面，也就是所谓的“方面”了。然后它又以巧夺天功的妙手将这些剖开的切面复原，不留痕迹。

使用“横切”技术，AOP把软件系统分为两个部分：核心关注点和横切关注点。业务处理的主要流程是核心关注点，与之关系不大的部分是横切关注点。横切关注点的一个特点是，他们经常发生在核心关注点的多处，而各处都基本相似。比如权限认证、日志、事务处理。Aop 的作用在于分离系统中的各种关注点，将核心关注点和横切关注点分离开来。正如Avanade公司的高级方案构架师Adam Magee所说，AOP的核心思想就是“将应用程序中的商业逻辑同对其提供支持的通用服务进行分离。”

实现AOP的技术，主要分为两大类：一是采用动态代理技术，利用截取消息的方式，对该消息进行装饰，以取代原有对象行为的执行；二是采用静态织入的方式，引入特定的语法创建“方面”，从而使得编译器可以在编译期间织入有关“方面”的代码。

**SpringAOP的实现机制：**

Spring AOP采用动态代理机制和字节码生成技术实现。与第一代AspectJ采用编译器将横切逻辑织入目标对象不同，动态代理机制和字节码生成都是在运行期间为目标对象生成一个代理对象，而将横切逻辑织入到这个代理对象中，系统最终使用的是织入了横切逻辑的代理对象，而不是真正的目标对象。

默认情况下，如果Spring AOP发现目标对象实现了相应Interface，则采用动态代理机制为其生成代理对象实例。而如果目标对象没有实现任何Interface，Spring AOP会尝试使用CGLIB(Code Generation Library)这种动态字节码生成类库，为目标对象生成动态的代理对象实例。

通过MethodInterceptor的invoke方法的MethodInvocation参数，我们可以控制对相应的Joinpoint的拦截行为。通过调用MethodInvocation的proceed（）方法，可以让程序执行继续沿着调用链传播。如果在哪一个MethodInterceptor中没有调用proceed（），那么程序的执行将会在当前MethodInterceptor处“短路”，Joinpoint上的调用链将被中断，同一Joinpoint上的其他MethodInterceptor的逻辑以及Joinpoint出的方法逻辑将不会被执行。

Spring默认采取的动态代理机制实现AOP，当动态代理不可用时（代理类无接口）会使用CGlib机制。但Spring的AOP有一定的缺点，第一个只能对方法进行切入，不能对接口，字段，静态代码块进行切入（切入接口的某个方法，则该接口下所有实现类的该方法将被切入）。第二个同类中的互相调用方法将不会使用代理类。因为要使用代理类必须从Spring容器中获取Bean。第三个性能不是最好的，从3.3章节我们得知使用自定义类加载器，性能要优于动态代理和CGlib。

CGlib是一个强大的,高性能,高质量的Code生成类库。它可以在运行期扩展Java类与实现Java接口。 然这些实际的功能是asm所提供的，asm又是什么？Java字节码操控框架，具体是什么大家可以上网查一查，毕竟我们这里所要讨论的是cglib， cglib就是封装了asm，简化了asm的操作，实现了在运行期动态生成新的class。 可能大家还感觉不到它的强大，现在就告诉你。 实际上CGlib为spring aop提供了底层的一种实现;为hibernate使用cglib动态生成VO/PO (接口层对象)。

　　它的原理就是用Enhancer生成一个原有类的子类，并且设置好callback ， 则原有类的每个方法调用都会转成调用实现了MethodInterceptor接口的proxy的intercept() 函数：   
public Object intercept(Object o,Method method,Object[] args,MethodProxy proxy)   
　　在intercept()函数里，你可以在执行Object result=proxy.invokeSuper(o,args);来执行原有函数，在执行前后加入自己的东西，改变它的参数，也可以瞒天过海，完全干别的。说白了，就是AOP中的around advice。

**AOP使用场景**

AOP用来封装横切关注点，具体可以在下面的场景中使用:

Authentication 权限

Caching 缓存

Context passing 内容传递

Error handling 错误处理

Lazy loading　懒加载

Debugging　　调试

logging, tracing, profiling and monitoring　记录跟踪　优化　校准

Performance optimization　性能优化

Persistence　　持久化

Resource pooling　资源池

Synchronization　同步

Transactions 事务

**AOP相关概念**

方面（Aspect）：一个关注点的模块化，这个关注点实现可能另外横切多个对象。事务管理是J2EE应用中一个很好的横切关注点例子。方面用Spring的 Advisor或拦截器实现。

连接点（Joinpoint）: 程序执行过程中明确的点，如方法的调用或特定的异常被抛出。

通知（Advice）: 在特定的连接点，AOP框架执行的动作。各种类型的通知包括“around”、“before”和“throws”通知。通知类型将在下面讨论。许多AOP框架包括Spring都是以拦截器做通知模型，维护一个“围绕”连接点的拦截器链。Spring中定义了四个advice: BeforeAdvice, AfterAdvice, ThrowAdvice和DynamicIntroductionAdvice

切入点（Pointcut）: 指定一个通知将被引发的一系列连接点的集合。AOP框架必须允许开发者指定切入点：例如，使用正则表达式。 Spring定义了Pointcut接口，用来组合MethodMatcher和ClassFilter，可以通过名字很清楚的理解， MethodMatcher是用来检查目标类的方法是否可以被应用此通知，而ClassFilter是用来检查Pointcut是否应该应用到目标类上

引入（Introduction）: 添加方法或字段到被通知的类。 Spring允许引入新的接口到任何被通知的对象。例如，你可以使用一个引入使任何对象实现 IsModified接口，来简化缓存。Spring中要使用Introduction, 可有通过DelegatingIntroductionInterceptor来实现通知，通过DefaultIntroductionAdvisor来配置Advice和代理类要实现的接口

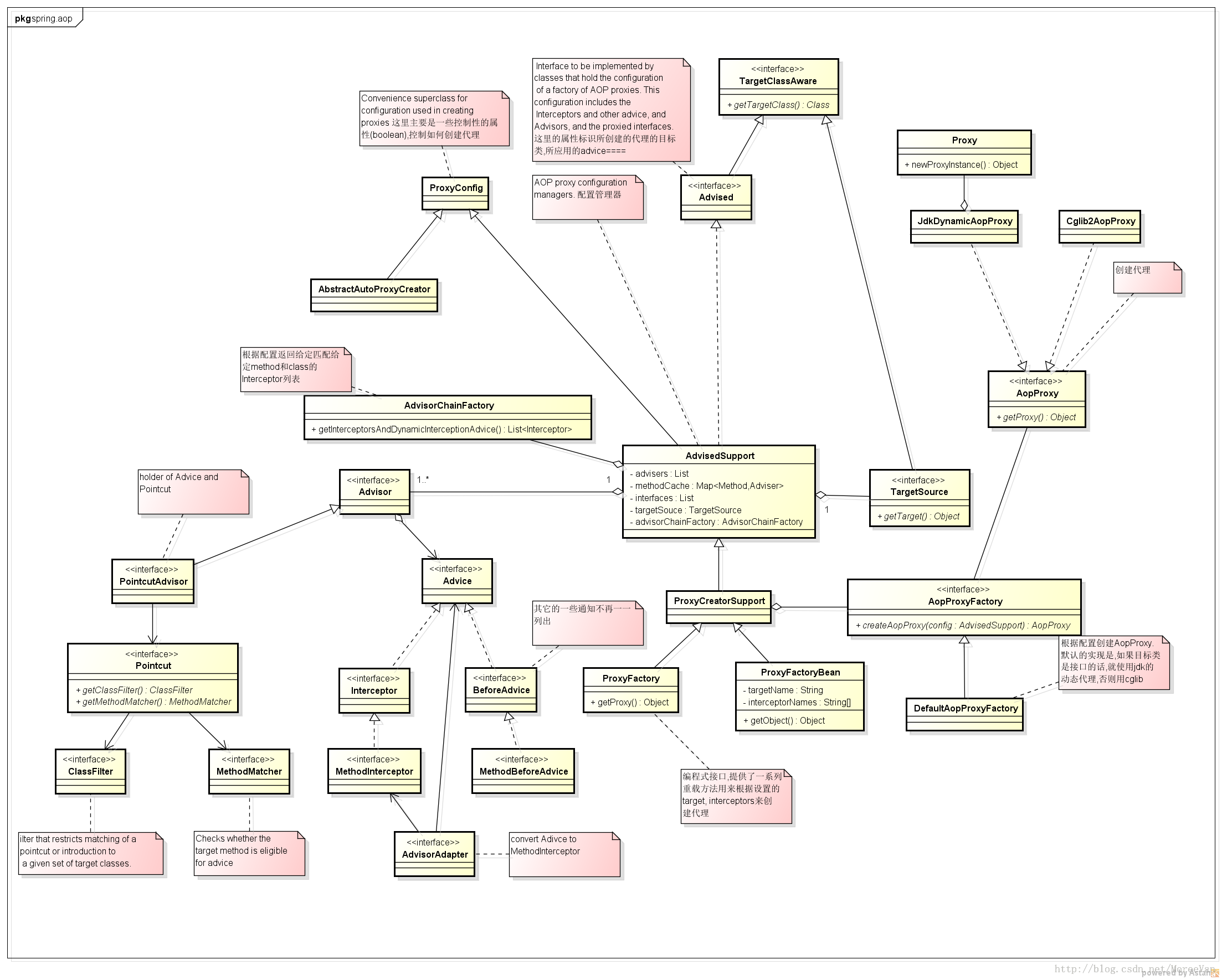
目标对象（Target Object）: 包含连接点的对象。也被称作被通知或被代理对象。POJO

AOP代理（AOP Proxy）: AOP框架创建的对象，包含通知。 在Spring中，AOP代理可以是JDK动态代理或者CGLIB代理。

织入（Weaving）: 组装方面来创建一个被通知对象。这可以在编译时完成（例如使用AspectJ编译器），也可以在运行时完成。Spring和其他纯Java AOP框架一样，在运行时完成织入。

**Spring AOP组件**

下面这种类图列出了Spring中主要的AOP组件



**如何使用Spring AOP**

可以通过配置文件或者编程的方式来使用Spring AOP。

配置可以通过xml文件来进行，大概有四种方式：

1.        配置ProxyFactoryBean，显式地设置advisors, advice, target等

2.        配置AutoProxyCreator，这种方式下，还是如以前一样使用定义的bean，但是从容器中获得的其实已经是代理对象

3.        通过<aop:config>来配置

4.        通过<aop: aspectj-autoproxy>来配置，使用AspectJ的注解来标识通知及切入点

也可以直接使用ProxyFactory来以编程的方式使用Spring AOP，通过ProxyFactory提供的方法可以设置target对象, advisor等相关配置，最终通过 getProxy()方法来获取代理对象

具体使用的示例可以google. 这里略去

**Spring AOP代理对象的生成**

Spring提供了两种方式来生成代理对象: JDKProxy和Cglib，具体使用哪种方式生成由AopProxyFactory根据AdvisedSupport对象的配置来决定。默认的策略是如果目标类是接口，则使用JDK动态代理技术，否则使用Cglib来生成代理。下面我们来研究一下Spring如何使用JDK来生成代理对象，具体的生成代码放在JdkDynamicAopProxy这个类中，直接上相关代码：

**[java]** [view plain](http://blog.csdn.net/moreevan/article/details/11977115) [copy](http://blog.csdn.net/moreevan/article/details/11977115)

1. /\*\*
2. \* <ol>
3. \* <li>获取代理类要实现的接口,除了Advised对象中配置的,还会加上SpringProxy, Advised(opaque=false)
4. \* <li>检查上面得到的接口中有没有定义 equals或者hashcode的接口
5. \* <li>调用Proxy.newProxyInstance创建代理对象
6. \* </ol>
7. \*/
8. **public** Object getProxy(ClassLoader classLoader) {
9. **if** (logger.isDebugEnabled()) {
10. logger.debug("Creating JDK dynamic proxy: target source is " +**this**.advised.getTargetSource());
11. }
12. Class[] proxiedInterfaces =AopProxyUtils.completeProxiedInterfaces(**this**.advised);
13. findDefinedEqualsAndHashCodeMethods(proxiedInterfaces);
14. **return** Proxy.newProxyInstance(classLoader, proxiedInterfaces, **this**);
15. }

那这个其实很明了，注释上我也已经写清楚了，不再赘述。

下面的问题是，代理对象生成了，那切面是如何织入的？

我们知道InvocationHandler是JDK动态代理的核心，生成的代理对象的方法调用都会委托到InvocationHandler.invoke()方法。而通过JdkDynamicAopProxy的签名我们可以看到这个类其实也实现了InvocationHandler，下面我们就通过分析这个类中实现的invoke()方法来具体看下Spring AOP是如何织入切面的。

**[java]** [view plain](http://blog.csdn.net/moreevan/article/details/11977115) [copy](http://blog.csdn.net/moreevan/article/details/11977115)

1. publicObject invoke(Object proxy, Method method, Object[] args) throwsThrowable {
2. MethodInvocation invocation = **null**;
3. Object oldProxy = **null**;
4. **boolean** setProxyContext = **false**;
6. TargetSource targetSource = **this**.advised.targetSource;
7. Class targetClass = **null**;
8. Object target = **null**;
10. **try** {
11. //eqauls()方法，具目标对象未实现此方法
12. **if** (!**this**.equalsDefined && AopUtils.isEqualsMethod(method)){
13. **return** (equals(args[0])? Boolean.TRUE : Boolean.FALSE);
14. }
16. //hashCode()方法，具目标对象未实现此方法
17. **if** (!**this**.hashCodeDefined && AopUtils.isHashCodeMethod(method)){
18. **return** newInteger(hashCode());
19. }
21. //Advised接口或者其父接口中定义的方法,直接反射调用,不应用通知
22. **if** (!**this**.advised.opaque &&method.getDeclaringClass().isInterface()
23. &&method.getDeclaringClass().isAssignableFrom(Advised.**class**)) {
24. // Service invocations onProxyConfig with the proxy config...
25. **return** AopUtils.invokeJoinpointUsingReflection(**this**.advised,method, args);
26. }
28. Object retVal = **null**;
30. **if** (**this**.advised.exposeProxy) {
31. // Make invocation available ifnecessary.
32. oldProxy = AopContext.setCurrentProxy(proxy);
33. setProxyContext = **true**;
34. }
36. //获得目标对象的类
37. target = targetSource.getTarget();
38. **if** (target != **null**) {
39. targetClass = target.getClass();
40. }
42. //获取可以应用到此方法上的Interceptor列表
43. List chain = **this**.advised.getInterceptorsAndDynamicInterceptionAdvice(method,targetClass);
45. //如果没有可以应用到此方法的通知(Interceptor)，此直接反射调用 method.invoke(target, args)
46. **if** (chain.isEmpty()) {
47. retVal = AopUtils.invokeJoinpointUsingReflection(target,method, args);
48. } **else** {
49. //创建MethodInvocation
50. invocation = newReflectiveMethodInvocation(proxy, target, method, args, targetClass, chain);
51. retVal = invocation.proceed();
52. }
54. // Massage return value if necessary.
55. **if** (retVal != **null** && retVal == target &&method.getReturnType().isInstance(proxy)
56. &&!RawTargetAccess.**class**.isAssignableFrom(method.getDeclaringClass())) {
57. // Special case: it returned"this" and the return type of the method
58. // is type-compatible. Notethat we can't help if the target sets
59. // a reference to itself inanother returned object.
60. retVal = proxy;
61. }
62. **return** retVal;
63. } **finally** {
64. **if** (target != **null** && !targetSource.isStatic()) {
65. // Must have come fromTargetSource.
66. targetSource.releaseTarget(target);
67. }
68. **if** (setProxyContext) {
69. // Restore old proxy.
70. AopContext.setCurrentProxy(oldProxy);
71. }
72. }
73. }

主流程可以简述为：获取可以应用到此方法上的通知链（Interceptor Chain）,如果有,则应用通知,并执行joinpoint; 如果没有,则直接反射执行joinpoint。而这里的关键是通知链是如何获取的以及它又是如何执行的，下面逐一分析下。

首先，从上面的代码可以看到，通知链是通过Advised.getInterceptorsAndDynamicInterceptionAdvice()这个方法来获取的,我们来看下这个方法的实现:

**[java]** [view plain](http://blog.csdn.net/moreevan/article/details/11977115) [copy](http://blog.csdn.net/moreevan/article/details/11977115)

1. **public** List<Object>getInterceptorsAndDynamicInterceptionAdvice(Method method, Class targetClass) {
2. MethodCacheKeycacheKey = **new** MethodCacheKey(method);
3. List<Object>cached = **this**.methodCache.get(cacheKey);
4. **if**(cached == **null**) {
5. cached= **this**.advisorChainFactory.getInterceptorsAndDynamicInterceptionAdvice(
6. **this**,method, targetClass);
7. **this**.methodCache.put(cacheKey,cached);
8. }
9. returncached;
10. }

可以看到实际的获取工作其实是由AdvisorChainFactory. getInterceptorsAndDynamicInterceptionAdvice()这个方法来完成的，获取到的结果会被缓存。

下面来分析下这个方法的实现：

**[java]** [view plain](http://blog.csdn.net/moreevan/article/details/11977115) [copy](http://blog.csdn.net/moreevan/article/details/11977115)

1. /\*\*
2. \* 从提供的配置实例config中获取advisor列表,遍历处理这些advisor.如果是IntroductionAdvisor,
3. \* 则判断此Advisor能否应用到目标类targetClass上.如果是PointcutAdvisor,则判断
4. \* 此Advisor能否应用到目标方法method上.将满足条件的Advisor通过AdvisorAdaptor转化成Interceptor列表返回.
5. \*/
6. publicList getInterceptorsAndDynamicInterceptionAdvice(Advised config, Methodmethod, Class targetClass) {
7. // This is somewhat tricky... we have to process introductions first,
8. // but we need to preserve order in the ultimate list.
9. List interceptorList = **new** ArrayList(config.getAdvisors().length);
11. //查看是否包含IntroductionAdvisor
12. **boolean** hasIntroductions = hasMatchingIntroductions(config,targetClass);
14. //这里实际上注册一系列AdvisorAdapter,用于将Advisor转化成MethodInterceptor
15. AdvisorAdapterRegistry registry = GlobalAdvisorAdapterRegistry.getInstance();
17. Advisor[] advisors = config.getAdvisors();
18. **for** (**int** i = 0; i <advisors.length; i++) {
19. Advisor advisor = advisors[i];
20. **if** (advisor **instanceof** PointcutAdvisor) {
21. // Add it conditionally.
22. PointcutAdvisor pointcutAdvisor= (PointcutAdvisor) advisor;
23. **if**(config.isPreFiltered() ||pointcutAdvisor.getPointcut().getClassFilter().matches(targetClass)) {
24. //TODO: 这个地方这两个方法的位置可以互换下
25. //将Advisor转化成Interceptor
26. MethodInterceptor[]interceptors = registry.getInterceptors(advisor);
28. //检查当前advisor的pointcut是否可以匹配当前方法
29. MethodMatcher mm =pointcutAdvisor.getPointcut().getMethodMatcher();
31. **if** (MethodMatchers.matches(mm,method, targetClass, hasIntroductions)) {
32. **if**(mm.isRuntime()) {
33. // Creating a newobject instance in the getInterceptors() method
34. // isn't a problemas we normally cache created chains.
35. **for** (intj = 0; j < interceptors.length; j++) {
36. interceptorList.add(**new** InterceptorAndDynamicMethodMatcher(interceptors[j],mm));
37. }
38. } **else** {
39. interceptorList.addAll(Arrays.asList(interceptors));
40. }
41. }
42. }
43. } **else** **if** (advisor **instanceof** IntroductionAdvisor){
44. IntroductionAdvisor ia =(IntroductionAdvisor) advisor;
45. **if**(config.isPreFiltered() || ia.getClassFilter().matches(targetClass)) {
46. Interceptor[] interceptors= registry.getInterceptors(advisor);
47. interceptorList.addAll(Arrays.asList(interceptors));
48. }
49. } **else** {
50. Interceptor[] interceptors =registry.getInterceptors(advisor);
51. interceptorList.addAll(Arrays.asList(interceptors));
52. }
53. }
54. **return** interceptorList;
55. }

这个方法执行完成后，Advised中配置能够应用到连接点或者目标类的Advisor全部被转化成了MethodInterceptor.

接下来我们再看下得到的拦截器链是怎么起作用的。

**[java]** [view plain](http://blog.csdn.net/moreevan/article/details/11977115) [copy](http://blog.csdn.net/moreevan/article/details/11977115)

1. **if** (chain.isEmpty()) {
2. retVal = AopUtils.invokeJoinpointUsingReflection(target,method, args);
3. } **else** {
4. //创建MethodInvocation
5. invocation = newReflectiveMethodInvocation(proxy, target, method, args, targetClass, chain);
6. retVal = invocation.proceed();
7. }

         从这段代码可以看出，如果得到的拦截器链为空，则直接反射调用目标方法，否则创建MethodInvocation，调用其proceed方法，触发拦截器链的执行，来看下具体代码

**[java]** [view plain](http://blog.csdn.net/moreevan/article/details/11977115) [copy](http://blog.csdn.net/moreevan/article/details/11977115)

1. **public** Object proceed() **throws** Throwable {
2. //  We start with an index of -1and increment early.
3. **if** (**this**.currentInterceptorIndex == **this**.interceptorsAndDynamicMethodMatchers.size()- 1) {
4. //如果Interceptor执行完了，则执行joinPoint
5. **return** invokeJoinpoint();
6. }
8. Object interceptorOrInterceptionAdvice =
9. **this**.interceptorsAndDynamicMethodMatchers.get(++**this**.currentInterceptorIndex);
11. //如果要动态匹配joinPoint
12. **if** (interceptorOrInterceptionAdvice **instanceof** InterceptorAndDynamicMethodMatcher){
13. // Evaluate dynamic method matcher here: static part will already have
14. // been evaluated and found to match.
15. InterceptorAndDynamicMethodMatcher dm =
16. (InterceptorAndDynamicMethodMatcher)interceptorOrInterceptionAdvice;
17. //动态匹配：运行时参数是否满足匹配条件
18. **if** (dm.methodMatcher.matches(**this**.method, **this**.targetClass,**this**.arguments)) {
19. //执行当前Intercetpor
20. returndm.interceptor.invoke(**this**);
21. }
22. **else** {
23. //动态匹配失败时,略过当前Intercetpor,调用下一个Interceptor
24. **return** proceed();
25. }
26. }
27. **else** {
28. // It's an interceptor, so we just invoke it: The pointcutwill have
29. // been evaluated statically before this object was constructed.
30. //执行当前Intercetpor
31. **return** ((MethodInterceptor) interceptorOrInterceptionAdvice).invoke(**this**);
32. }
33. }

代码也比较简单，这里不再赘述。

# Spring实现AOP的4种方式

  先了解AOP的相关术语:  
1.通知(Advice):  
通知定义了切面是什么以及何时使用。描述了切面要完成的工作和何时需要执行这个工作。Sleepable类是advice，Beforesleep，aftersleep函数是要完成的工作，before，after是要完成的时间  
2.连接点(Joinpoint):  
程序能够应用通知的一个“时机”，这些“时机”就是连接点，例如方法被调用时、异常被抛出时等等，方法被调用时就是before，after函数触发，异常被抛出时就是around触发，这个触发advice里函数的点就叫连接点Joinpoint，函数需要join进来的point。  
3.切入点(Pointcut)  
通知定义了切面要发生的“故事”和时间，那么切入点就定义了“故事”发生的地点，例如某个类或方法的名称，Spring中允许我们方便的用正则表达式来指定，比如\*sleep函数，在sleep函数调用时发生advice的事情。函数join进来后需要知道在哪cut，具体触发函数为pointcut，而时机比如方法调用，异常抛出就是joinpoint。  
4.切面(Aspect)  
通知和切入点共同组成了切面：时间、地点和要发生的“故事”，在哪切入，做什么，组成了一个aspect  
5.引入(Introduction)  
引入允许我们向现有的类添加新的方法和属性(Spring提供了一个方法注入的功能）  
6.目标(Target)  
即被通知的对象，如果没有AOP,那么它的逻辑将要交叉别的事务逻辑，有了AOP之后它可以只关注自己要做的事（AOP让他做爱做的事）  
7.代理(proxy)  
应用通知的对象，详细内容参见设计模式里面的代理模式，实际执行的类，通过weaving把目标对象切面都用来创建代理对象，真正执行的是代理对象，里面有before，after，sleep，还有保证调用序列的方法（比如，before-sleep-after），还有proceed函数来保证这个序列一个一个传下去，before后proceed到sleep，sleep后proceed到after  
8.织入(Weaving)  
把切面应用到目标对象来创建新的代理对象的过程，织入一般发生在如下几个时机:  
(1)编译时：当一个类文件被编译时进行织入，这需要特殊的编译器才可以做的到，例如AspectJ的织入编译器  
(2)类加载时：使用特殊的ClassLoader在目标类被加载到程序之前增强类的字节代码  
(3)运行时：切面在运行的某个时刻被织入,SpringAOP就是以这种方式织入切面的，原理应该是使用了JDK的动态代理技术  
  
Spring提供了4种实现AOP的方式：  
1.经典的基于代理的AOP  
2.@AspectJ注解驱动的切面  
3.纯POJO切面  
4.注入式AspectJ切面  
  
首先看经典的基于代理的AOP:  
Spring支持五种类型的通知：  
Before(前)  org.apringframework.aop.MethodBeforeAdvice  
After-returning(返回后) org.springframework.aop.AfterReturningAdvice  
After-throwing(抛出后) org.springframework.aop.ThrowsAdvice  
Arround(周围) org.aopaliance.intercept.MethodInterceptor  
Introduction(引入) org.springframework.aop.IntroductionInterceptor  
  
值的说明的是周围通知，他是由AOP Alliance中的接口定义的而非Spring,周围通知相当于前通知、返回后通知、抛出后通知的结合（传说中的完全体？好吧，我看日和看多  
  
了）还有引入通知怎么玩我还没搞清楚，等心无杂念的时候玩玩  
  
这东西怎么玩？这么几个步骤：  
1.创建通知：实现这几个接口，把其中的方法实现了  
2.定义切点和通知者：在Spring配制文件中配置这些信息  
3.使用ProxyFactoryBean来生成代理  
  
具体做法。。。大晚上的就举个睡觉的例子吧:  
  
首先写一个接口叫Sleepable,这是一个牛X的接口,所有具有睡觉能力的东西都可以实现该接口（不光生物，包括关机选项里面的休眠）  
  
package test.spring.aop.bean  
  
public interface Sleepable{  
   
    void sleep();   
}  
  
然后写一个Human类，他实现了这个接口  
  
package test.spring.aop.bean  
  
public Human implements Sleepable{  
     
   /\*这人莫非跟寡人差不多？  
    \*除了睡觉睡的比较好之外其余的什么也不会做？\*/  
   public void sleep(){  
      System.out.println("睡觉了！梦中自有颜如玉！");  
   }  
  
}  
  
  
好了，这是主角，不过睡觉前后要做些辅助工作的，最基本的是脱穿衣服，失眠的人还要吃安眠药什么的,但是这些动作与纯粹的睡觉这一“业务逻辑”是不相干的，如果把  
  
这些代码全部加入到sleep方法中，是不是有违单一职责呢？，这时候我们就需要AOP了。  
  
编写一个SleepHelper类，它里面包含了睡觉的辅助工作,用AOP术语来说它就应该是通知了，我们需要实现上面的接口  
  
package test.spring.aop.bean;  
  
import java.lang.reflect.Method;  
  
import org.springframework.aop.AfterReturningAdvice;  
import org.springframework.aop.MethodBeforeAdvice;  
  
public class SleepHelper implements MethodBeforeAdvice,AfterReturningAdvice{  
  
    public void before(Method mtd, Object[] arg1, Object arg2)  
            throws Throwable {  
        System.out.println("通常情况下睡觉之前要脱衣服！");  
    }  
  
    public void afterReturning(Object arg0, Method arg1, Object[] arg2,  
            Object arg3) throws Throwable {  
        System.out.println("起床后要先穿衣服！");  
    }  
      
}  
  
然后在spring配置文件中进行配置:  
<bean id="sleepHelper" class="test.spring.aop.bean.SleepHelper">  
</bean>  
  
OK!现在创建通知的工作就完成了.  
  
第二步是进行配置，这是很令人蛋疼的操作，尤其是这么热的天，Spring又把东西的名字起的见鬼的长！它为啥不能像usr这种风格呢？  
  
首先要做的是配置一个切点,据说切点的表示方式在Spring中有好几种，但是常用的只有两种：1.使用正则表达式 2.使用AspectJ表达式 AspectJ我不是很熟悉(我也是熟悉  
  
党 or 精通党？),我还是习惯用正则表达式  
  
Spring使用org.springframework.aop.support.JdkRegexpMethodPointcut来定义正则表达式切点  
<bean id="spleepPointcut" class="org.springframework.aop.support.JdkRegexpMethodPointcut">  
  <property name="pattern" value=".\*sleep"/>  
</bean>  
  
pattern属性指定了正则表达式，它匹配所有的sleep方法  
  
切点仅仅是定义了故事发生的地点，还有故事发生的时间以及最重要的故事的内容,就是通知了，我们需要把通知跟切点结合起来，我们要使用的通知者是:  
org.springframework.aop.support.DefaultPointcutAdvisor  
  
<bean id="sleepHelperAdvisor" class="org.springframework.aop.support.DefaultPointcutAdvisor">  
     <property name="advice" ref="sleepHelper"/>  
     <property name="pointcut" ref="sleepPointcut"/>  
</bean>  
  
切入点和通知都配置完成，接下来该调用ProxyFactoryBean产生代理对象了  
  
<bean id="humanProxy" class="org.springframework.aop.framework.ProxyFactoryBean">  
     <property name="target" ref="human"/>  
     <property name="interceptorNames" value="sleepHelperAdvisor" />  
     <property name="proxyInterfaces" value="test.spring.aop.bean.Sleepable" />  
</bean>  
  
ProxyFactoryBean是一个代理，我们可以把它转换为proxyInterfaces中指定的实现该interface的代理对象:  
  
import org.springframework.aop.framework.ProxyFactoryBean;  
import org.springframework.context.ApplicationContext;  
import org.springframework.context.support.ClassPathXmlApplicationContext;  
  
import test.spring.aop.bean.Sleepable;  
  
  
public class Test {  
  
    public static void main(String[] args){  
        ApplicationContext appCtx = new ClassPathXmlApplicationContext("applicationContext.xml");  
        Sleepable sleeper = (Sleepable)appCtx.getBean("humanProxy");  
        sleeper.sleep();  
    }  
}  
  
程序运行产生结果：  
通常情况下睡觉之前要脱衣服！  
睡觉啦~梦中自有颜如玉!  
起床后要先穿衣服！  
  
OK!这是我们想要的结果，但是上面这个过程貌似有点复杂，尤其是配置切点跟通知,Spring提供了一种自动代理的功能，能让切点跟通知自动进行匹配，修改配置文件如下:  
 <bean id="sleepHelper" class="test.spring.aop.bean.SleepHelper">  
  </bean>  
  <bean id="sleepAdvisor" class="org.springframework.aop.support.RegexpMethodPointcutAdvisor">  
    <property name="advice" ref="sleepHelper"/>  
    <property name="pattern" value=".\*sleep"/>  
  </bean>  
  <bean id="human" class="test.spring.aop.bean.Human">  
  </bean>  
  <bean class="org.springframework.aop.framework.autoproxy.DefaultAdvisorAutoProxyCreator"/>  
  
执行程序：  
public class Test {  
  
    public static void main(String[] args){  
        ApplicationContext appCtx = new ClassPathXmlApplicationContext("applicationContext.xml");  
        Sleepable sleeper = (Sleepable)appCtx.getBean("human");  
        sleeper.sleep();  
    }  
}  
成功输出结果跟前面一样!  
只要我们声明了org.springframework.aop.framework.autoproxy.DefaultAdvisorAutoProxyCreator(我勒个去的，名太长了)就能为方法匹配的bean自动创建代理！  
  
但是这样还是要有很多工作要做,有更简单的方式吗?有！  
  
一种方式是使用AspectJ提供的注解:  
  
package test.mine.spring.bean;  
  
import org.aspectj.lang.annotation.AfterReturning;  
import org.aspectj.lang.annotation.Aspect;  
import org.aspectj.lang.annotation.Before;  
import org.aspectj.lang.annotation.Pointcut;  
@Aspect  
public class SleepHelper {  
  
    public SleepHelper(){  
          
    }  
      
    @Pointcut("execution(\* \*.sleep())")  
    public void sleeppoint(){}  
      
    @Before("sleeppoint()")  
    public void beforeSleep(){  
        System.out.println("睡觉前要脱衣服!");  
    }  
      
    @AfterReturning("sleeppoint()")  
    public void afterSleep(){  
        System.out.println("睡醒了要穿衣服！");  
    }  
      
}  
  
用@Aspect的注解来标识切面,注意不要把它漏了，否则Spring创建代理的时候会找不到它,@Pointcut注解指定了切点，@Before和@AfterReturning指定了运行时的通知，注  
  
意的是要在注解中传入切点的名称  
  
然后我们在Spring配置文件上下点功夫,首先是增加AOP的XML命名空间和声明相关schema  
命名空间:  
xmlns:aop="http://www.springframework.org/schema/aop"  
schema声明:  
http://www.springframework.org/schema/aop  
http://www.springframework.org/schema/aop/spring-aop-2.0.xsd  
  
然后加上这个标签:  
<aop:aspectj-autoproxy/> 有了这个Spring就能够自动扫描被@Aspect标注的切面了  
  
最后是运行，很简单方便了：  
public class Test {  
  
    public static void main(String[] args){  
        ApplicationContext appCtx = new ClassPathXmlApplicationContext("applicationContext.xml");  
        Sleepable human = (Sleepable)appCtx.getBean("human");  
        human.sleep();  
    }  
}  
  
下面我们来看最后一种常用的实现AOP的方式:使用Spring来定义纯粹的POJO切面  
  
前面我们用到了<aop:aspectj-autoproxy/>标签,Spring在aop的命名空间里面还提供了其他的配置元素:  
<aop:advisor> 定义一个AOP通知者  
<aop:after> 后通知  
<aop:after-returning> 返回后通知  
<aop:after-throwing> 抛出后通知  
<aop:around> 周围通知  
<aop:aspect>定义一个切面  
<aop:before>前通知  
<aop:config>顶级配置元素，类似于<beans>这种东西  
<aop:pointcut>定义一个切点  
  
我们用AOP标签来实现睡觉这个过程:  
代码不变，只是修改配置文件,加入AOP配置即可:  
<aop:config>  
    <aop:aspect ref="sleepHelper">  
    <aop:before method="beforeSleep" pointcut="execution(\* \*.sleep(..))"/>  
    <aop:after method="afterSleep" pointcut="execution(\* \*.sleep(..))"/>  
    </aop:aspect>  
</aop:config>  
  
完！  
  
OK~~基本上就这么多了吧，要想用好还得多折腾折腾，另外玩玩AspectJ~不就是个玩！