**Spring AOP的底层实现技术**

**三、Spring AOP的实现**

**我们来试着看看能不能理解SpringAOP底层的实现原理。**

**3.1 Spring AOP的几个概念**

Spring AOP中的几个基本概念，每次学习AOP都被这几个概念折腾的很不爽，我们在这里再把这几个概念描述一遍，力争把这几个概念搞清，在每次review这块内容的时候可以很快上手。

**1.切面(Aspect)：切面就是一个关注点的模块化，如事务管理、日志管理、权限管理等；**

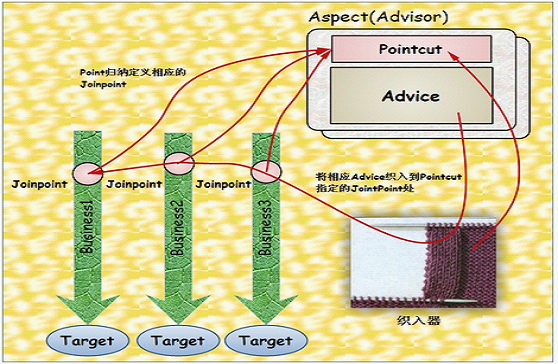
**2.连接点(Joinpoint)：程序执行时的某个特定的点，在Spring中就是一个方法的执行；**

**3.通知(Advice)：通知就是在切面的某个连接点上执行的操作，也就是事务管理、日志管理等；**

**4.切入点(Pointcut)：切入点就是描述某一类选定的连接点，也就是指定某一类要织入通知的方法；**

**5.目标对象(Target)：就是被AOP动态代理的目标对象；**

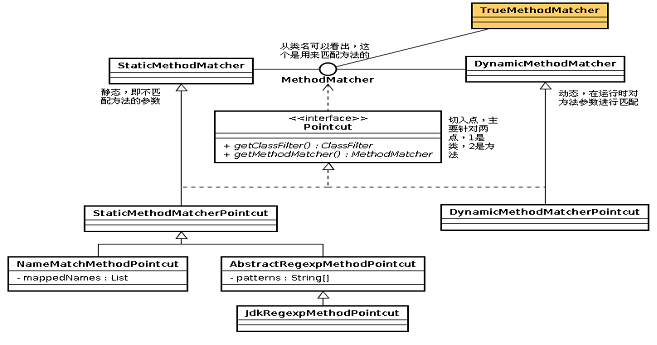
用一张图来形象地表达AOP的概念及其关系如下：



**3.2 Spring AOP中切入点、通知、切面的实现**

理解了上面的几个概念后，我们分别来看看Spring AOP是如何实现这些概念的；

1.切入点(Pointcut)：它定义了**哪些连接点需要被织入横切逻辑**；**在Java中，连接点对应哪些类(接口)的方法**。因此，我们都能猜到，所谓的切入点，就是定义了**匹配哪些娄的哪些方法的一些规则，**可以是静态的基于类(方法)名的值匹配，也可以是基于正则表达式的模式匹配。来看看Spring AOP Pointcut相关的类图：



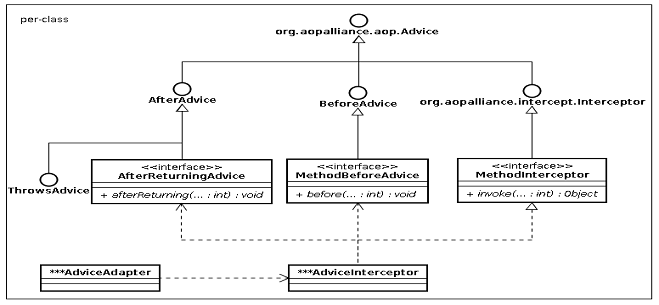
在Pointcut接口的定义中，也许你已经想到了，**ClassFilter是类过滤器**，它定义了哪些**类名需要拦截**；典型的两个实现类为TypePatternClassFilter和TrueClassFilter(所有类均匹配)；而**MethodMatcher**为方法匹配器，定义哪些方法需要拦截。

在上面的类图中：

* StaticMethodMatch与DynamicMethodMatch的区别是后者在运行时会依据方法的参数值进行匹配。
* NameMatchMethodPointCut根据指定的mappedNames来匹配方法。
* AbstractRegexpMethodPointCut根据正则表达式来匹配方法。

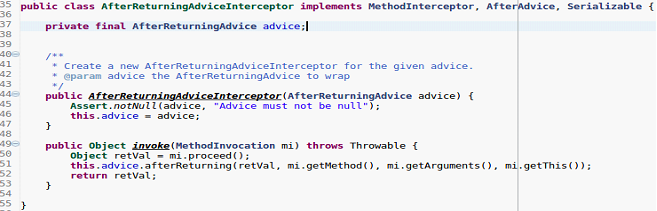
**2.通知(Advice)：通知定义了具体的横切逻辑。**在Spring中，存在两种类型的Advice，即per-class和per-instance的Advice。

所谓per-class，即该类型的Advice只提供方法拦截，不会为目标对象保存任何状态或者添加新的特性，它也是我们最常见的Advice。下面是per-class的类图：



* **BeforeAdvice：在连接点前执行的横切逻辑。**
* **AfterReturningAdvice：在连接点执行后，再执行横切逻辑。**
* **AfterAdvice：一般由程序自己实现，当抛出异常后，执行横切逻辑。**
* AroundAdvice：Spring AOP中并没有提供这个接口，而是采用了AOP Alliance的MethodInteceptor接口；通过看AfterReturningAdvice的源码我们知道，它是不能更改连接点所在方法的返回值的(更改引用)；但使用的MethodInteceptor，所有的事情，都不在话下。

在上面的类图中，还有两种类没有介绍，那就是\*\*\*AdviceAdapter和\*\*\*AdviceInteceptor，我们以**AfterReturningAdviceInterceptor**为例来说明：



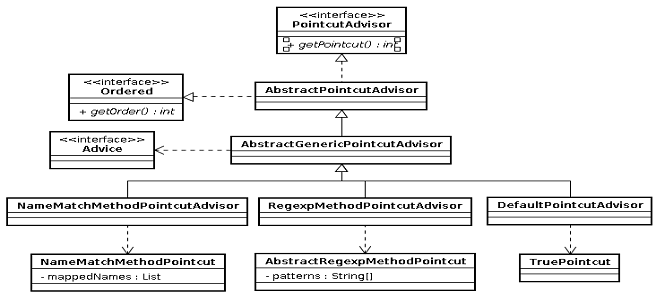
该类实现了MethodInterceptor和AfterAdvice接口，同时构造函数中还有一个AfterReturningAdvice实例的参数；这个类存在的作用是为了什么呢？对，没错，Spring AOP把所有的Advice都适配成了MethodInterceptor，统一的好处是方便后面横切逻辑的执行(参看下一节)，适配的工作即由\*\*\*AdviceAdapter完成；

哈哈，Spring AOP的代码也不过如此嘛：所谓的AfterReturningAdvice，通过适配成MethodInterceptor后，其实就是**在invoke方法中，先执行目标对象的方法，再执行的AfterReturningAdvice所定义的横切逻辑**。你现在明白它为什么不能修改返回值的引用了吧？

对于per-instance的Advice，目前只有一种实现，就是Introduction，使用的场景比较少，有兴趣的同学可以自己研究一下，呵呵！

**3.切面(Aspect)：**在Spring中，Advisor就是切面；但与通常的Aspect不同的是，Advisor通常**只有一个Pointcut和一个Advice，**而Aspect则可以包含多个Pointcut和多个Advice，因此**Advisor是一种特殊的Aspect**。但，这已经够用了！

接下来看下per-class Advisor的类图：



其实没有什么好看的，前面已经说过，Advisor包含一个Pointcut和一个Advisor；在AbstractGenericPointcutAdvisor中，持有一个Advice的引用；下面的几个实现，均是**针对前面提到的几种不同的Pointcut的实现。**

**3.3 Spring AOP实现的基本线索**

我们选择ProxyFactoryBean作为入口点和分析的开始。ProxyFactoryBean是在Spring IoC环境中，创建AOP应用的最底层方法，从中，可以看到一条实现AOP的基本线索。

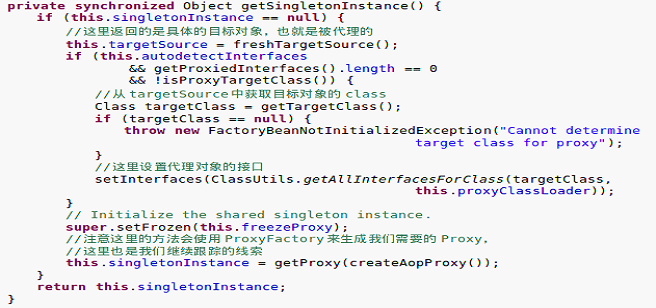
所有的逻辑从以下的方法开始,我们主要针对**单例的代理对象**的生成：



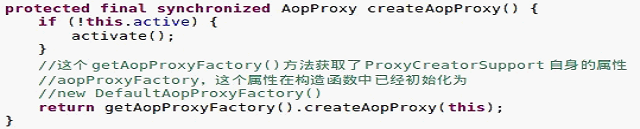
下面我们深入到**SpringAOP核心代码的内部，看看代理对象的生成机制，拦截器横切逻辑以及织入的实现。**

**3.4 代理对象的生成**

对于()方法返回了什么，这就是代理对象如何产生的逻辑了，然后一步一步，看看传说中的proxy到底是如何一步一步的产生的。



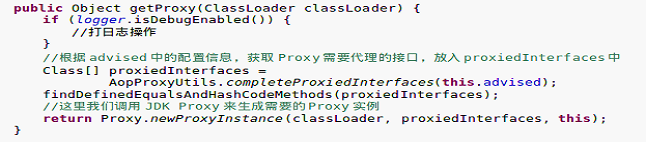
ProxyFactoryBean是AdvisedSupport的子类，Spring使用**AopProxy接口把AOP代理的实现与框架的其他部分分离开来**。在AdvisedSupport中通过这样的方式来得到AopProxy,当然这里需要得到**AopProxyFactory的帮助 ，从JDK或者cglib中得到想要的代理对象：**



这个**DefaultAopProxyFactory**是Spring用来生成AopProxy(生成代理)的地方，它包含JDK和Cglib两种实现方式。让我接着往里面看：



可以看到其中的代理对象可以由JDK或者Cglib来生成，**JdkDynamicAopProxy类和Cglib2AopProxy都实现的是AopProxy的接口**，我们进入JdkDynamicAopProxy实现中看看Proxy是怎样生成的：



用Proxy包装target之后，通过ProxyFactoryBean得到对其方法的调用就被Proxy拦截了， ProxyFactoryBean的**getObject()方法得到的实际上是一个Proxy了**，target对象已经被封装了。对 ProxyFactoryBean这个工厂bean而言，其生产出来的对象是封装了目标对象的代理对象。

**3.5 拦截器的作用**

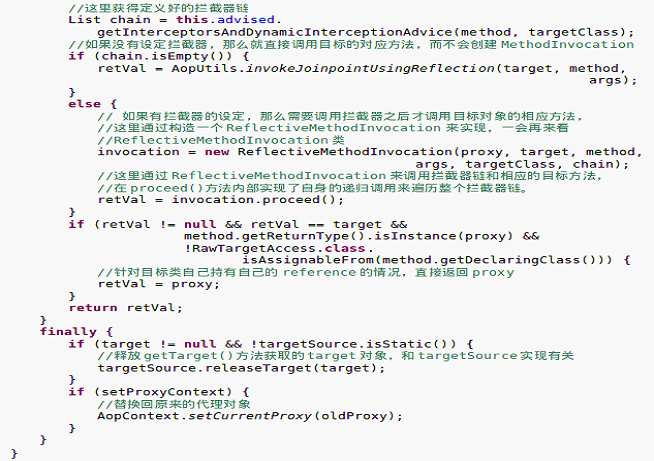
前面分析了SpringAOP实现中得到Proxy对象的过程，接下来我们去探寻Spring AOP中**拦截器链是怎样被调用的，也就是Proxy模式是怎样起作用的。**

还记得在JdkDynamicAopProxy中生成Proxy对象的时候，有一句这样的代码吗？

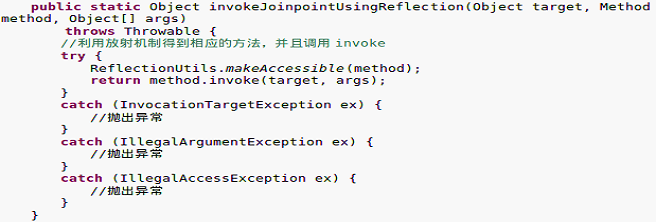
**return Proxy.newProxyInstance(classLoader, proxiedInterfaces, this);**

这里我们的**JdkDynamicAopProxy**实现了InvocationHandler这个接口，**this参数对应的是符合接口的InvocationHandler对象**,也就是说当 Proxy对象的函数被调用的时候，**InvocationHandler的invoke方法会被作为回调函数调用：**





上面所说的目标对象方法的调用，是通过AopUtils的方法调用，使用反射机制来对目标对象的方法进行的：



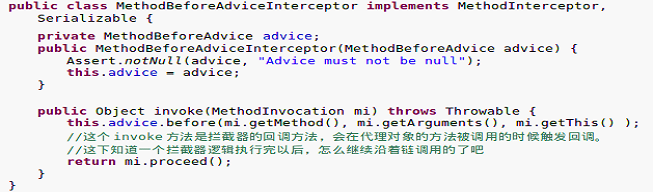
接下来，我们来看具体的ReflectiveMethodInvocation中proceed()方法的实现，也就是拦截器链的实现机制：



从上面的分析我们看到了Spring AOP拦截机制的基本实现，比如Spring怎样得到Proxy，怎样利用JAVA Proxy以及反射机制**对用户定义的拦截器链进行处理。**

**3.6 织入的实现**

在上面调用拦截器的时候，经过一系列的注册，适配的过程以后，拦截器在拦截的时候，会**调用到预置好的一个通知适配器**，**设置通知拦截器**，这是一系列Spring**设计好为通知服务的类的一个，是最终完成通知拦截和实现的地方**，例如对 **MethodBeforeAdviceInterceptor的实现是这样的：**



**可以看到通知适配器将advice适配成Interceptor以后，会调用advice的before方法去执行横切逻辑。这样就成功的完成了before通知的织入。**

再来一次，完整源码（看懂或者看不懂，都坚持一下，加深印象）。

四、Spring AOP代理源码解析

1、声明式Spring AOP代理工厂对象ProxyFactoryBean：

我们以ProxyFactoryBean为例，分析Spring AOP的实现原理，ProxyFactoryBean是Spring中一个非常灵活的创建AOP应用的底层方法，封装了AOP的主要功能。

一个简单的AOP代理工厂对象的配置如下：

**[html]** [view plaincopy](http://blog.csdn.net/leicool_518/article/details/43672313)

1. <!--配置通知器，通知器的实现定义了需要对目标对象进行的增强行为-->
2. <bean id=”**testAdvisor**” class=”com.test.TestAdvisor”/>
3. <!--配置AOP代理，封装AOP功能的主要类-->
4. <bean id=”testAOP” class=”org.springframework.aop.ProxyFactoryBean”>
5. <!--AOP代理接口-->
6. <property name=”**proxyInterfaces**”>
7. <value>com.test.TestProxyInterface</value>
8. </property>
9. <!--需要使用AOP切面增强的对象-->
10. <property name=”**target**”>
11. <bean class=”com.test.TestTarget”/>
12. </property>
13. <!--代理拦截器，配置通知器的名称，即通知器在AOP代理的配置下通过使用代理对象的拦截机制发挥作用-->
14. <property name=”interceptorNames”>
15. <list>
16. <value>testAdvisor</value>
17. </list>
18. </property>
19. </bean>

2、ProxyFactoryBean生成AOP Proxy代理对象：

从上面的ProxyFactoryBean的简单配置例子我们可以看出，**ProxyFactoryBean是用来配置目标对象和切面行为Advice的，**ProxyFactoryBean通过其配置的拦截器名称interceptorNames即通知器Advisor将切面行为Advice应用到目标对象中。

在ProxyFactoryBean中，需要为待增强目标对象目标对象生成Proxy代理对象，从而为AOP切面的编织提供基础，下面通过源码分析ProxyFactoryBean的生成AOPProxy代理对象的实现过程：

(1)ProxyFactoryBean产生代理对象的主要源码：

1. public class ProxyFactoryBean extends ProxyCreatorSupport
2. implements FactoryBean<Object>, BeanClassLoaderAware, BeanFactoryAware {  //标注通知器为全局通用通知器
3. public static final String GLOBAL\_SUFFIX = "\*";
4. //标志通知器链是否已经完成初始化
5. private boolean advisorChainInitialized = false;
6. //单态模式对象
7. private Object singletonInstance;
8. ……
9. //ProxyFactoryBean创建AOPProxy代理的入口方法
10. public Object getObject() throws BeansException {
11. //初始化通知器链
12. initializeAdvisorChain();
13. //如果目标对象是单态模式
14. if (isSingleton()) {
15. //调用获取单态模式对象的方法产生AOPProxy代理
16. return getSingletonInstance();
17. }
18. //如果目标对象是原型模式
19. else {
20. if (this.targetName == null) {
21. logger.warn("Using non-singleton proxies with singleton targets is often undesirable. " +
22. "Enable prototype proxies by setting the 'targetName' property.");
23. }
24. //调用原型模式对象方法每次创建一个新的AOPProxy代理对象
25. return newPrototypeInstance();
26. }
27. }
28. //初始化通知器链
29. private synchronized void initializeAdvisorChain() throws AopConfigException, BeansException {
30. //如果通知器链已经被初始化，则直接返回，即通知器链只在第一次获取代理对象时产生
31. if (this.advisorChainInitialized) {
32. return;
33. }
34. //如果ProxyFactoryBean中配置的连接器列名名称不为空
35. if (!ObjectUtils.isEmpty(this.interceptorNames)) {
36. //如果没有Bean工厂(容器)
37. if (this.beanFactory == null) {
38. throw new IllegalStateException("No BeanFactory available anymore (probably due to serialization) " +
39. "- cannot resolve interceptor names " + Arrays.asList(this.interceptorNames));
40. }
41. //全局通知器不能是通知器链中最后一个，除非显式使用属性指定了目标
42. if (this.interceptorNames[this.interceptorNames.length - 1].endsWith(GLOBAL\_SUFFIX) &&
43. this.targetName == null && this.targetSource == EMPTY\_TARGET\_SOURCE) {
44. throw new AopConfigException("Target required after globals");
45. }
46. //遍历通知器链，向容器添加通知器
47. for (String name : this.interceptorNames) {
48. if (logger.isTraceEnabled()) {
49. logger.trace("Configuring advisor or advice '" + name + "'");
50. }
51. //如果通知器是全局的
52. if (name.endsWith(GLOBAL\_SUFFIX)) {
53. if (!(this.beanFactory instanceof ListableBeanFactory)) {
54. throw new AopConfigException(
55. "Can only use global advisors or interceptors with a ListableBeanFactory");
56. }
57. //向容器中添加全局通知器
58. addGlobalAdvisor((ListableBeanFactory) this.beanFactory,
59. name.substring(0, name.length() - GLOBAL\_SUFFIX.length()));
60. }
61. //如果通知器不是全局的
62. else {
63. Object advice;
64. //如果通知器是单态模式
65. if (this.singleton || this.beanFactory.isSingleton(name)) {
66. //从容器获取单态模式的通知或者通知器
67. advice = this.beanFactory.getBean(name);
68. }
69. //如果通知器是原型模式
70. else {
71. //创建一个新的通知或者通知器对象
72. advice = new PrototypePlaceholderAdvisor(name);
73. }
74. //添加通知器
75. addAdvisorOnChainCreation(advice, name);
76. }
77. }
78. }
79. //设置通知器链已初始化标识
80. this.advisorChainInitialized = true;
81. }
82. //**获取**一个单态模式的AOPProxy代理对象
83. private synchronized Object getSingletonInstance() {
84. //如果单态模式的代理对象还未被创建
85. if (this.singletonInstance == null) {
86. //获取代理的目标源
87. this.targetSource = freshTargetSource();
88. //如果ProxyFactoryBean设置了自动探测接口属性，并且没有配置代理接
89. //且不是目标对象的直接代理类
90. if (this.autodetectInterfaces && getProxiedInterfaces().length == 0 && !isProxyTargetClass()) {
91. //获取代理对象的目标类
92. Class targetClass = getTargetClass();
93. if (targetClass == null) {
94. throw new FactoryBeanNotInitializedException("Cannot determine target class for proxy");
95. }
96. //设置代理对象的接口
97. setInterfaces(ClassUtils.getAllInterfacesForClass(targetClass, this.proxyClassLoader));
98. }
99. //初始化共享的单态模式对象                   super.setFrozen(this.freezeProxy);
100. //调用ProxyFactory生成代理AOPProxy对象
101. this.singletonInstance = getProxy(createAopProxy());
102. }
103. return this.singletonInstance;
104. }
105. //获取一个原型模式的代理对象
106. private synchronized Object **newPrototypeInstance**() {
107. if (logger.isTraceEnabled()) {
108. logger.trace("Creating copy of prototype ProxyFactoryBean config: " + this);
109. }
110. //根据当前的AOPProxyFactory获取一个创建代理的辅助类
111. ProxyCreatorSupport copy = new ProxyCreatorSupport(getAopProxyFactory());
112. //获取一个刷新的目标源
113. TargetSource targetSource = freshTargetSource();
114. //从当前对象中拷贝AOP的配置，为了保持原型模式对象的独立性，每次创建代理
115. //对象时都需要拷贝AOP的配置，以保证原型模式AOPProxy代理对象的独立性
116. copy.copyConfigurationFrom(this, targetSource, freshAdvisorChain());
117. if (this.autodetectInterfaces && getProxiedInterfaces().length == 0 && !isProxyTargetClass()) {
118. //设置代理接口
119. copy.setInterfaces(
120. ClassUtils.getAllInterfacesForClass(targetSource.getTargetClass(), this.proxyClassLoader));
121. }
122. copy.setFrozen(this.freezeProxy);
123. if (logger.isTraceEnabled()) {
124. logger.trace("Using ProxyCreatorSupport copy: " + copy);
125. }
126. //调用ProxyFactory生成AOPProxy代理
127. return getProxy(copy.createAopProxy());
128. }
129. //使用createAopProxy方法返回的AOPProxy对象产生AOPProxy代理对象
130. protected Object getProxy(AopProxy aopProxy) {
131. return aopProxy.getProxy(this.proxyClassLoader);
132. }
133. ……
134. }

通过源码分析，**我们了解到AOPProxyFactory实现了FactoryBean接口，所以本身也是一个Spring的工厂Bean**，AOP代理工厂的主要功能概况为：

**a.初始化通知器链，将配置的通知器链添加到容器存放通知/通知器的集合中。**

**b.根据单态模式/原型模式，获取AOPProxy产生AOPProxy代理对象。**

(2)AOP创建辅助器(AOPCreatorSupport)获取AOP Proxy代理对象：

AOP ProxyFactory的getSingletonInstance和newPrototypeInstance方法均通过调用**AOPCreatorSupport**的createAopProxy()方法获取AOP Proxy，主要源码如下：

1. public class ProxyCreatorSupport extends AdvisedSupport {
2. //AOPProxy工厂
3. private AopProxyFactory aopProxyFactory;
4. //当第一个AOPProxy代理对象被创建时，设置为true
5. private boolean active = false;
6. public AopProxyFactory getAopProxyFactory() {
7. return this.aopProxyFactory;
8. }
9. //默认使用DefaultAopProxyFactory作用AOP代理工厂
10. public ProxyCreatorSupport() {
11. this.aopProxyFactory = new **DefaultAopProxyFactory**();
12. }
13. //创建AOPProxy代理的入口方法
14. protected final synchronized AopProxy createAopProxy() {
15. if (!this.active) {
16. activate();
17. }
18. //调用DefaultAopProxyFactory的创建AOPProxy代理的方法
19. return getAopProxyFactory().createAopProxy(this);
20. }
21. //激活AOP代理配置，向容器注册代理回调监听器，第一次创建AOP代理时调用
22. private void activate() {
23. this.active = true;
24. for (AdvisedSupportListener listener : this.listeners) {
25. listener.activated(this);
26. }
27. }
28. ……
29. }

通过对ProxyCreatorSupport的源码分析，我们知道**真正创建AOPProxy代理对象的是DefaultAopProxyFactory类**。

(3)DefaultAopProxyFactory创建AOP Proxy代理对象：

DefaultAopProxyFactory是AOP创建辅助器(AOPCreatorSupport)默认的AOP代理工厂，DefaultAopProxyFactory的createAopProxy方法实现了创建AOP代理的功能，源码如下：

**[html]** [view plaincopy](http://blog.csdn.net/leicool_518/article/details/43672313)

1. public class DefaultAopProxyFactory implements AopProxyFactory, Serializable {
2. //判断CGLIB类库是否在classpath中
3. private static final boolean cglibAvailable =
4. ClassUtils.isPresent("net.sf.cglib.proxy.Enhancer", DefaultAopProxyFactory.class.getClassLoader());
5. //创建AOP代理对象
6. public AopProxy createAopProxy(AdvisedSupport config) throws AopConfigException {
7. //如果AOP使用显式优化，或者配置了目标类，或者只使用Spring支持的代理接口
8. if (config.isOptimize() || config.isProxyTargetClass() || hasNoUserSuppliedProxyInterfaces(config)) {
9. //获取AOP配置的目标类
10. Class targetClass = config.getTargetClass();
11. if (targetClass == null) {
12. throw new AopConfigException("TargetSource cannot determine target class: " +
13. "Either an interface or a target is required for proxy creation.");
14. }
15. //如果配置的AOP目标类是接口，则使用JDK动态代理机制来生成AOP代理
16. if (targetClass.isInterface()) {
17. return new JdkDynamicAopProxy(config);
18. }
19. //如果AOP配置的目标类不是接口，则使用CGLIB的方式来生成AOP代理
20. if (!cglibAvailable) {
21. throw new AopConfigException(
22. "Cannot proxy target class because CGLIB2 is not available. " +
23. "Add CGLIB to the class path or specify proxy interfaces.");
24. }
25. return CglibProxyFactory.createCglibProxy(config);
26. }
27. else {
28. return new JdkDynamicAopProxy(config);
29. }
30. }
31. //判断AOP是否只配置了SpringProxy代理接口或者没有配置任何代理接口
32. private boolean hasNoUserSuppliedProxyInterfaces(AdvisedSupport config) {
33. //获取AOP配置的所有AOP代理接口
34. Class[] interfaces = config.getProxiedInterfaces();
35. return (interfaces.length == 0 || (interfaces.length == 1 && SpringProxy.class.equals(interfaces[0])));
36. }
37. }

通过对DefaultAopProxyFactory的源码分析，我们了解了Spring在创建AOP代理对象时，如果配置的目标类是接口，则使用JDK的动态代理机制来生成AOP代理，如果使用的不是接口，则使用CGLIB方式来生成AOP的动态代理。

3、JDK动态代理机制创建AOPProxy代理对象：

JDK的动态代理机制只能对接口起作用，即如果要对一个对象使用JDK动态代理方式生成代理对象时，该对象必须实现接口，Spring中通过JdkDynamicAopProxy类使用JDK动态代理机制生成AOPProxy代理对象，JdkDynamicAopProxy的主要源码如下：

**[html]** [view plaincopy](http://blog.csdn.net/leicool_518/article/details/43672313)

1. final class JdkDynamicAopProxy implements AopProxy, InvocationHandler, Serializable {
2. ……
3. //JdkDynamicAopProxy的构造方法
4. public JdkDynamicAopProxy(AdvisedSupport config) throws AopConfigException {
5. Assert.notNull(config, "AdvisedSupport must not be null");
6. //获取AOPBeanFactory中配置的通知器链和目标源
7. if (config.getAdvisors().length == 0 && config.getTargetSource() == AdvisedSupport.EMPTY\_TARGET\_SOURCE) {
8. throw new AopConfigException("No advisors and no TargetSource specified");
9. }
10. //为当前对象设置AOP配置
11. this.advised = config;
12. }
13. //获取AOP代理对象的入口方法
14. public Object getProxy() {
15. return getProxy(ClassUtils.getDefaultClassLoader());
16. }
17. //创建AOP代理对象
18. public Object getProxy(ClassLoader classLoader) {
19. if (logger.isDebugEnabled()) {
20. logger.debug("Creating JDK dynamic proxy: target source is " + this.advised.getTargetSource());
21. }
22. //获取AOPBeanFactory中配置的代理接口
23. Class[] proxiedInterfaces = AopProxyUtils.completeProxiedInterfaces(this.advised);
24. //查找代理目标的接口中是否定义equals()和hashCode()方法
25. findDefinedEqualsAndHashCodeMethods(proxiedInterfaces);
26. //使用JDK的动态代理机制创建AOP代理对象
27. return **Proxy.newProxyInstance**(classLoader, proxiedInterfaces, this);
28. }
29. //查找给定类或接口中是否定义了equals()和hashCode()方法
30. private void findDefinedEqualsAndHashCodeMethods(Class[] proxiedInterfaces) {
31. //遍历给定的类/接口数组
32. for (Class proxiedInterface : proxiedInterfaces) {
33. //或者给定类/接口中所有声明的方法
34. Method[] methods = proxiedInterface.getDeclaredMethods();
35. //遍历类/接口中的声明的方法
36. for (Method method : methods) {
37. //如果方法是equals()方法，则设置当前对象equalsDefined属性
38. if (AopUtils.isEqualsMethod(method)) {
39. this.equalsDefined = true;
40. }
41. //如果方法是hashCode()方法，则设置当前对象hashCodeDefined属性
42. if (AopUtils.isHashCodeMethod(method)) {
43. this.hashCodeDefined = true;
44. }
45. if (this.equalsDefined && this.hashCodeDefined) {
46. return;
47. }
48. }
49. }
50. }
51. //AOP代理对象的回调方法
52. public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
53. MethodInvocation invocation;
54. Object oldProxy = null;
55. boolean setProxyContext = false;
56. //获取通知的目标源
57. TargetSource targetSource = this.advised.targetSource;
58. Class targetClass = null;
59. Object target = null;
60. try {
61. //如果代理目标对象的接口中没有定义equals()方法，且当前调用的方法
62. //是equals()方法，即目标对象没有自己实现equals()方法
63. if (!this.equalsDefined && AopUtils.isEqualsMethod(method)) {
64. return equals(args[0]);
65. }
66. //如果代理目标对象的接口中没有定义hashCode()方法，且当前调用的方法
67. //是hashCode()方法，即目标对象没有自己实现hashCode()方法
68. if (!this.hashCodeDefined && AopUtils.isHashCodeMethod(method)) {
69. return hashCode();
70. }
71. //如果AOP配置了通知，使用反射机制调用通知的同名方法
72. if (!this.advised.opaque && method.getDeclaringClass().isInterface() &&             method.getDeclaringClass().isAssignableFrom(Advised.class)) {
73. return AopUtils.invokeJoinpointUsingReflection(this.advised, method, args);
74. }
75. Object retVal;
76. //如果当前通知暴露了代理，则将当前代理使用currentProxy()方法变为可用代理
77. if (this.advised.exposeProxy) {
78. oldProxy = AopContext.setCurrentProxy(proxy);
79. setProxyContext = true;
80. }
81. //获取目标对象
82. **target = targetSource.getTarget();**
83. if (target != null) {
84. targetClass = target.getClass();
85. }
86. //获取目标对象方法配置的拦截器(通知器)链
87. List<Object> chain = this.advised.getInterceptorsAndDynamicInterceptionAdvice(method, targetClass);
88. //如果没有配置任何通知
89. if (chain.isEmpty()) {
90. //没有配置通知，使用反射直接调用目标对象的方法，并获取方法返回值
91. retVal = AopUtils.invokeJoinpointUsingReflection(target, method, args);
92. }
93. //如果配置了通知
94. else {
95. //为目标对象创建方法回调对象，需要在调用通知之后才调用目标对象的方法
96. invocation = new ReflectiveMethodInvocation(proxy, target, method, args, targetClass, chain);
97. **//调用通知链，沿着通知器链调用所有配置的通知**
98. retVal = invocation.proceed();
99. }
100. //如果方法有返回值，则将代理对象最为方法返回
101. if (retVal != null && retVal == target && method.getReturnType().isInstance(proxy) &&
102. !RawTargetAccess.class.isAssignableFrom(method.getDeclaringClass())) {
103. retVal = proxy;
104. }
105. return retVal;
106. }
107. finally {
108. if (target != null && !targetSource.isStatic()) {
109. //释放目标对象
110. targetSource.releaseTarget(target);
111. }
112. if (setProxyContext) {
113. //存储代理对象
114. AopContext.setCurrentProxy(oldProxy);
115. }
116. }
117. }
118. ……
119. }

通过上述源码分析，我们看到JdkDynamicAopProxy本身实现了InvocationHandler接口和invoke()方法，

JDK的动态代理机制的工作原理是：

**当调用目标对象的方法时，不是直接调用目标对象，而是首先生成一个目标对象的动态代理对象，触发代理对象的invoke()方法，代理的invoke()方法才会真正调用目标对象的方法。Spring AOP的实现原理是在代理对象invoke()方法调用目标对象的方法时，调用配置的通知。**

4、CglibProxyFactory创建AOP Proxy代理：

JDK的动态代理只能针对接口生成代理对象，对于没有实现接口的目标对象，必须通过第3方的CGLIB来生成代理对象，CglibProxyFactory创建AOPProxy代理的主要源码如下：

**[html]** [view plaincopy](http://blog.csdn.net/leicool_518/article/details/43672313)

1. //通过CGLIB方式创建AOP代理对象
2. public Object getProxy(ClassLoader classLoader) {
3. if (logger.isDebugEnabled()) {
4. logger.debug("Creating CGLIB2 proxy: target source is " + this.advised.getTargetSource());
5. }
6. try {
7. //从代理创建辅助类中获取在IoC容器中配置的目标对象
8. Class rootClass = this.advised.getTargetClass();
9. Assert.state(rootClass != null, "Target class must be available for creating a CGLIB proxy");
10. //将目标对象本身做为自己的基类
11. Class proxySuperClass = rootClass;
12. //检查获取到的目标类是否是CGLIB产生的
13. if (AopUtils.isCglibProxyClass(rootClass)) {
14. //如果目标类是有CGLIB产生的，获取目标类的基类
15. proxySuperClass = rootClass.getSuperclass();
16. //获取目标类的接口
17. Class[] additionalInterfaces = rootClass.getInterfaces();
18. //将目标类的接口添加到容器AOP代理创建辅助类的配置中
19. for (Class additionalInterface : additionalInterfaces) {
20. this.advised.addInterface(additionalInterface);
21. }
22. }
23. //校验代理基类
24. validateClassIfNecessary(proxySuperClass);
25. //配置CGLIB的Enhancer类，Enhancer是CGLIB中的主要操作类
26. Enhancer enhancer = createEnhancer();
27. if (classLoader != null) {
28. enhancer.setClassLoader(classLoader);
29. if (classLoader instanceof SmartClassLoader &&
30. ((SmartClassLoader) classLoader).isClassReloadable(proxySuperClass)) {
31. enhancer.setUseCache(false);
32. }
33. }
34. //设置enhancer的基类,继承了父类
35. **enhancer.setSuperclass(proxySuperClass);**
36. enhancer.setStrategy(new UndeclaredThrowableStrategy(UndeclaredThrowableException.class));
37. //设置enhancer的接口
38. enhancer.setInterfaces(AopProxyUtils.completeProxiedInterfaces(this.advised));
39. enhancer.setInterceptDuringConstruction(false);
40. //设置enhancer的回调方法
41. Callback[] callbacks = getCallbacks(rootClass);
42. enhancer.setCallbacks(callbacks);
43. //将通知器中配置作为enhancer的方法过滤
44. enhancer.setCallbackFilter(new ProxyCallbackFilter(
45. this.advised.getConfigurationOnlyCopy(), this.fixedInterceptorMap, this.fixedInterceptorOffset));
46. Class[] types = new Class[callbacks.length];
47. for (int x = 0; x < types.length; x++) {
48. types[x] = callbacks[x].getClass();
49. }
50. //设置enhancer的回调类型
51. enhancer.setCallbackTypes(types);
52. //创建代理对象
53. Object proxy;
54. if (this.constructorArgs != null) {
55. proxy = enhancer.create(this.constructorArgTypes, this.constructorArgs);
56. }
57. else {
58. proxy = enhancer.create();
59. }
60. return proxy;
61. }
62. catch (CodeGenerationException ex) {
63. throw new AopConfigException("Could not generate CGLIB subclass of class [" +
64. this.advised.getTargetClass() + "]: " +
65. "Common causes of this problem include using a final class or a non-visible class",
66. ex);
67. }
68. catch (IllegalArgumentException ex) {
69. throw new AopConfigException("Could not generate CGLIB subclass of class [" +
70. this.advised.getTargetClass() + "]: " +
71. "Common causes of this problem include using a final class or a non-visible class",
72. ex);
73. }
74. catch (Exception ex) {
75. // TargetSource.getTarget() failed
76. throw new AopConfigException("Unexpected AOP exception", ex);
77. }
78. }
79. //获取给定类的回调通知
80. private Callback[] getCallbacks(Class rootClass) throws Exception {
81. //优化参数
82. boolean exposeProxy = this.advised.isExposeProxy();
83. boolean isFrozen = this.advised.isFrozen();
84. boolean isStatic = this.advised.getTargetSource().isStatic();
85. //根据AOP配置创建一个动态通知拦截器，CGLIB创建的动态代理会自动调用
86. //DynamicAdvisedInterceptor类的intercept方法对目标对象进行拦截处理
87. Callback aopInterceptor = new DynamicAdvisedInterceptor(this.advised);
88. Callback targetInterceptor;
89. //根据是否暴露代理，创建直接应用目标的通知
90. if (exposeProxy) {
91. targetInterceptor = isStatic ?
92. new StaticUnadvisedExposedInterceptor(this.advised.getTargetSource().getTarget()) :
93. new DynamicUnadvisedExposedInterceptor(this.advised.getTargetSource());
94. }
95. else {
96. targetInterceptor = isStatic ?
97. new StaticUnadvisedInterceptor(this.advised.getTargetSource().getTarget()) :
98. new DynamicUnadvisedInterceptor(this.advised.getTargetSource());
99. }
100. // 创建目标分发器
101. Callback targetDispatcher = isStatic ?
102. new StaticDispatcher(this.advised.getTargetSource().getTarget()) : new SerializableNoOp();
103. Callback[] mainCallbacks = new Callback[]{
104. aopInterceptor, //普通通知
105. targetInterceptor, // 如果优化则不考虑配置的通知
106. new SerializableNoOp(), //没有被覆盖的方法
107. targetDispatcher, this.advisedDispatcher,
108. new EqualsInterceptor(this.advised),
109. new HashCodeInterceptor(this.advised)
110. };
111. Callback[] callbacks;
112. //如果目标是静态的，并且通知链被冻结，则使用优化AOP调用，直接对方法使用
113. //固定的通知链
114. if (isStatic && isFrozen) {
115. Method[] methods = rootClass.getMethods();
116. Callback[] fixedCallbacks = new Callback[methods.length];
117. this.fixedInterceptorMap = new HashMap<String, Integer>(methods.length);
118. for (int x = 0; x < methods.length; x++) {
119. List<Object> chain = this.advised.getInterceptorsAndDynamicInterceptionAdvice(methods[x], rootClass);
120. fixedCallbacks[x] = new FixedChainStaticTargetInterceptor(
121. chain, this.advised.getTargetSource().getTarget(), this.advised.getTargetClass());
122. this.fixedInterceptorMap.put(methods[x].toString(), x);
123. }
124. //将固定回调和主要回调拷贝到回调数组中
125. callbacks = new Callback[mainCallbacks.length + fixedCallbacks.length];
126. System.arraycopy(mainCallbacks, 0, callbacks, 0, mainCallbacks.length);
127. System.arraycopy(fixedCallbacks, 0, callbacks, mainCallbacks.length, fixedCallbacks.length);
128. this.fixedInterceptorOffset = mainCallbacks.length;
129. }
130. //如果目标不是静态的，或者通知链不被冻结，则使用AOP主要的通知
131. else {
132. callbacks = mainCallbacks;
133. }
134. return callbacks;
135. }

通过上面对CGLIB创建代理和获取回答通知的源码分析，我们了解到CGLIB在获取代理的通知时，会创建DynamicAdvisedInterceptor类，当应用调用目标对象的方法时，**不是直接调用目标对象，**而是通过CGLIB创建的代理对象来调用目标对象，在调用目标对象的方法时，触发DynamicAdvisedInterceptor的intercept回调方法对目标对象进行处理，CGLIB回调拦截器链的源码如下：

1. //CGLIB回调AOP拦截器链
2. public Object intercept(Object proxy, Method method, Object[] args, MethodProxy methodProxy) throws Throwable {
3. Object oldProxy = null;
4. boolean setProxyContext = false;
5. Class targetClass = null;
6. Object target = null;
7. try {
8. //如果通知器暴露了代理
9. if (this.advised.exposeProxy) {
10. //设置给定的代理对象为要被拦截的代理
11. oldProxy = AopContext.setCurrentProxy(proxy);
12. setProxyContext = true;
13. }
14. //获取目标对象
15. target = getTarget();
16. if (target != null) {
17. targetClass = target.getClass();
18. }
19. //获取AOP配置的通知
20. List<Object> chain = this.advised.getInterceptorsAndDynamicInterceptionAdvice(method, targetClass);
21. Object retVal;
22. //如果没有配置通知
23. if (chain.isEmpty() && Modifier.isPublic(method.getModifiers())) {
24. //直接调用目标对象的方法
25. retVal = methodProxy.invoke(target, args);
26. }
27. //如果配置了通知
28. else {
29. //通过CglibMethodInvocation来启动配置的通知
30. retVal = new CglibMethodInvocation(proxy, target, method, args, targetClass, chain, methodProxy).proceed();
31. }
32. //获取目标对象对象方法的回调结果，如果有必要则封装为代理
33. retVal = massageReturnTypeIfNecessary(proxy, target, method, retVal);
34. return retVal;
35. }
36. finally {
37. if (target != null) {
38. releaseTarget(target);
39. }
40. if (setProxyContext) {
41. //存储被回调的代理
42. AopContext.setCurrentProxy(oldProxy);
43. }
44. }
45. }

5、目标对象方法的调用：

(1)JdkDynamicAopProxy直接调用目标对象方法：

JdkDynamicAopProxy中是通过AopUtils.invokeJoinpointUsingReflection方法来直接调用目标对象的方法，源码如下：

**[html]** [view plaincopy](http://blog.csdn.net/leicool_518/article/details/43672313)

1. //通过反射机制直接调用目标对象方法
2. public static Object invokeJoinpointUsingReflection(Object target, Method method, Object[] args)
3. throws Throwable {
4. try {
5. //通过反射使给定的方法可以访问，主要是对protected和private方法使用，//取消严格访问控制权限的限制
6. ReflectionUtils.makeAccessible(method);
7. //使用反射机制调用目标对象的方法
8. return method.invoke(target, args);
9. }
10. catch (InvocationTargetException ex) {
11. throw ex.getTargetException();
12. }
13. catch (IllegalArgumentException ex) {
14. throw new AopInvocationException("AOP configuration seems to be invalid: tried calling method [" +
15. method + "] on target [" + target + "]", ex);
16. }
17. catch (IllegalAccessException ex) {
18. throw new AopInvocationException("Could not access method [" + method + "]", ex);
19. }
20. }

(2)Cglib2AopProxy直接调用目标对象方法：

Cglib2AopProxy是通过methodProxy.invoke来直接调用目标对象的方法，主要源码如下：

retVal = methodProxy.invoke(target, args);

上面讲了怎么多，再简单回顾下代理对象是如何生成的

1、上面讲到了两种生成代理对象的方法，一种是通过ProxyFactory，一种是**通过ProxyFactoryBean**。第一种获取比较简单，但是需要手工的进行写代码，而第二种是通过Spring的**IOC机制来控制Bean的生成**。

2、无论是ProxyFactory或者ProxyFactoryBean都是要通过**createAopProxy().getProxy()**来获取相应的代理对象，而通过Proxyfactory比较直接，上面重点介绍的是通过ProxyFactoryBean获得proxy。

3、首先，找到ProxyFactoryBean的getObject方法，为什么？（主要是跟Bean容器中getObject能返回代理对象）

4、其次调用getSingletonInstance（），在getSingletonInstance方法中引入了super中的方法，super是指ProxyCreatorSupport，这里ProxyCreatorSupport是ProxyFactoryBean和ProxyFactory的父类，已经做了很多工作，只需在ProxyFactoryBean的getObject()方法中通过父类的**createAopProxy()**取得相应的AopProxy。

5、跟踪createAopProxy方法，追踪到了ProxyCreatorSupport中，然后，借助了AopProxyFactory，此时得到的aopProxyFactory，在构造函数中已经定义为new **DefaultAopProxyFactory**()

6、进入DefaultAopProxyFactory中，找到createAopProxy方法，**在这里判断是调用JDK动态或者CGlib动态中的一种。**

7.调用增强方法其实**就是**拦截器在拦截的时候，会**调用到预置好的一个通知适配器**，**设置通知拦截器**，最后把通知织入到切面去。

**最后，总结一下两种方法：**

**JDK动态代理：  ---   InvocationHandler 和 Proxy.newProxyInstance()  -- 动态代理的基本原理为反射 + 多态 + 聚合**       InvocationHandler是一个接口，通过实现该接口定义横切逻辑，并通过反射机制调用目标类的代码，动态将横切逻辑和业务逻辑编织在一起。      JDK动态代理要求**被代理对象（接口实现类）通过反射注入到一个中间对象**（ProxyFactory），而中间对象实现InvocationHandler接口。      然后重写invoke方法，从而实现被代理对象方法被调用时，在调用前后插入一些代码（增强）。       Proxy利用InvocationHandler动态创建一个符合某一接口的实例。        Proxy.newProxyInstance()能够**利用中间对象来生产代理对象**，插入的代码就是切面代码。      通过调用通知链，沿着通知链调用所有的配置通知。                         **局限：**       被代理的对象必须实现接口，而且只有接口中的方法才能被代理。  **cglib动态代理：---   字节码生成技术**   **实现  MethodInterceptor接口，重写其 interceptor()方法**

CGLib采用非常底层的字节码技术，可以为一个类创建子类，并在子类中采用方法拦截的技术拦截所有父类方法的调用，并顺势织入横切逻辑，

**继承被代理对象生成一个子类（enhancer），然后重写被代理的方法**，在覆盖该方法时，插入自己的代码，通过CglibMethodInvocation来启动配置的通知。

**enhancer.create创建代理对象。**

因为需要重写被代理对象的方法，所以被代理的方法不能使final方法，因为final方法不能被覆盖。