Spring 基于注解的 AOP

什么是 AOP

AOP（Aspect-OrientedProgramming，面向切面编程），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。

一 AOP 的基本概念

- (1)Aspect(切面):通常是一个类，里面可以定义切入点和通知
- (2)JointPoint(连接点):程序执行过程中明确的点，一般是方法的调用
- (3)Advice(通知):AOP 在特定的切入点上执行的增强处理，有 before,after,afterReturning,afterThrowing,around
- (4)Pointcut(切入点):就是带有通知的连接点，在程序中主要体现为书写切入点表达式
- (5)AOP 代理：AOP 框架创建的对象，代理就是目标对象的加强。Spring 中的 AOP 代理可以使 JDK 动态代理，也可以是 CGLIB 代理，前者基于接口，后者基于子类

通知方法:

1. 前置通知:在我们执行目标方法之前运行**(@Before)**
2. 后置通知:在我们目标方法运行结束之后 ,不管有没有异常**(@After)**
3. 返回通知:在我们的目标方法正常返回值后运行**(@AfterReturning)**
4. 异常通知:在我们的目标方法出现异常后运行**(@AfterThrowing)**
5. 环绕通知:动态代理，需要手动执行 joinPoint.procced()(其实就是执行我们的目标方法执行之前相当于前置通知，执行之后就相当于我们后置通知**(@Around)**

二 Spring AOP

Spring 中的 AOP 代理还是离不开 Spring 的 IOC 容器，代理的生成，管理及其依赖关系都是由 IOC 容器负责，Spring 默认使用 JDK 动态代理，在需要代

理类而不是代理接口的时候，Spring 会自动切换为使用 CGLIB 代理，不过现在的项目都是面向接口编程，所以 JDK 动态代理相对来说用的还是多一些。

三 基于注解的 AOP 配置方式

切面类：

```
1.  package com.enjoy.cap10.aop;
2.
3.  import org.aspectj.lang.ProceedingJoinPoint;
4.  import org.aspectj.lang.annotation.After;
5.  import org.aspectj.lang.annotation.AfterReturning;
6.  import org.aspectj.lang.annotation.AfterThrowing;
7.  import org.aspectj.lang.annotation.Before;
8.  import org.aspectj.lang.annotation.Pointcut;
9.  import org.aspectj.lang.annotation.Around;
10. import org.aspectj.lang.annotation.Aspect;
11.
12. //日志切面类
13. @Aspect
14. public class LogAspects {
15.         @Pointcut("execution(public int com.enjoy.cap10.aop.Calculator.*(..))")
16.         public void pointCut(){};
17.
18.         //@before 代表在目标方法执行前切入，并指定在哪个方法前切入
19.         @Before("pointCut()")
20.         public void logStart(){
21.                 System.out.println("除法运行....参数列表是:{}");
22.         }
23.         @After("pointCut()")
24.         public void logEnd(){
25.                 System.out.println("除法结束.....");
26.         }
27.         @AfterReturning("pointCut()")
28.         public void logReturn(){
29.                 System.out.println("除法正常返回.....运行结果是:{}");
30.         }
```

```
31.         @AfterThrowing("pointCut()")
32.         public void logException(){
33.                 System.out.println("运行异常.....异常信息是:{}");
34.         }
35.         @Around("pointCut()")
36.         public Object Around(ProceedingJoinPoint proceedingJoinPoint) throws Throwable{
37.                 System.out.println("@Arount:执行目标方法之前...");
38.                 Object obj = proceedingJoinPoint.proceed();//相当于开始调 div 地
39.                 System.out.println("@Arount:执行目标方法之后...");
40.                 return obj;
41.         }
42. }
```

目标方法：

```
1.  package com.enjoy.cap10.aop;
2.
3.  public class Calculator {
4.          //业务逻辑方法
5.          public int div(int i, int j){
6.                  System.out.println("--------");
7.                  return i/j;
8.          }
9.  }
```

配置类：

```
1.  package com.enjoy.cap10.config;
2.
3.  import org.springframework.context.annotation.Bean;
4.  import org.springframework.context.annotation.Configuration;
5.  import org.springframework.context.annotation.EnableAspectJAutoProxy;
6.
7.  import com.enjoy.cap10.aop.Calculator;
8.  import com.enjoy.cap10.aop.LogAspects;
9.
10. @Configuration
11. @EnableAspectJAutoProxy
12. public class Cap10MainConfig {
13.         @Bean
```

```
14.         public Calculator calculator(){
15.                 return new Calculator();
16.         }
17.
18.         @Bean
19.         public LogAspects logAspects(){
20.                 return new LogAspects();
21.         }
22. }
```

测试类：

```
1.  public class Cap10Test {
2.          @Test
3.          public void test01(){
4.                  AnnotationConfigApplicationContext app = new
    AnnotationConfigApplicationContext(Cap10MainConfig.class);
5.                  Calculator c = app.getBean(Calculator.class);
6.                  int result = c.div(4, 3);
7.                  System.out.println(result);
8.                  app.close();
9.
10.         }
11. }
```

结果：

```
1.  @Arount:执行目标方法之前...
2.  除法运行....参数列表是:{}
3.  --------
4.  @Arount:执行目标方法之后...
5.  除法结束.....
6.  除法正常返回.....运行结果是:{}
7.  1
```

AOP 源码赏析

在这个注解比较流行的年代里,当我们想要使用 spring 的某些功能时只需要加上一行代码就可以了,比如:

- @EnableAspectJAutoProxy 开启 AOP
- @EnableTransactionManagement 开启 spring 事务管理,
- @EnableCaching 开启 spring 缓存
- @EnableWebMvc 开启 webMvc

对于我们使用者而言十分简单便利,然而,其背后所做的事,却远远比一个注解复杂的多了,本篇只是简略的介绍一下@EnableAspectJAutoProxy 背后所发生的那些事,了解其工作原理,才能更好的运用,并从中领略大师的智慧.

废话不多说,先来看一下源码:

```
1.  @Target(ElementType.TYPE)
2.  @Retention(RetentionPolicy.RUNTIME)
3.  @Documented
4.  @Import(AspectJAutoProxyRegistrar.class)
5.  public @interface EnableAspectJAutoProxy {
6.
7.      /**
8.       * Indicate whether subclass-based (CGLIB) proxies are to be created as opposed
9.       * to standard Java interface-based proxies. The default is {@code false}.
10.      */
11.     boolean proxyTargetClass() default false;
12.
13.     /**
14.      * Indicate that the proxy should be exposed by the AOP framework as a {@code ThreadLocal}
15.      * for retrieval via the {@link org.springframework.aop.framework.AopContext} class.
16.      * Off by default, i.e. no guarantees that {@code AopContext} access will work.
17.      * @since 4.3.1
18.      */
19.     boolean exposeProxy() default false;
20.
21. }
```

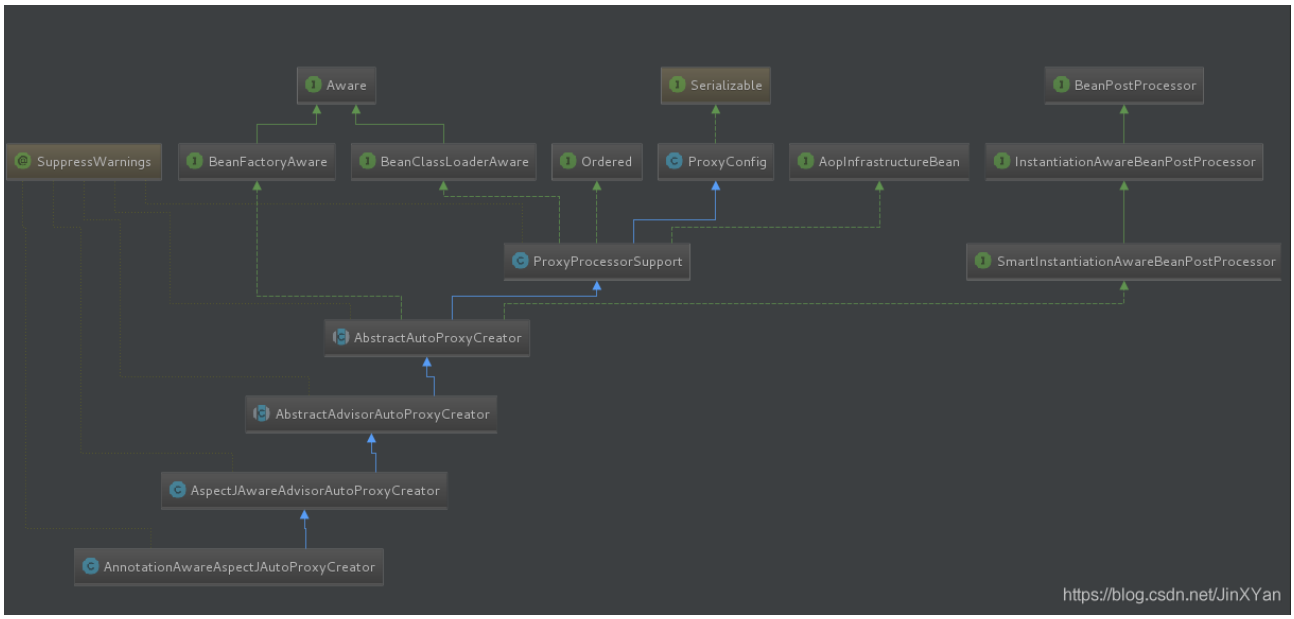
英文注解已经很详细了,这里简单介绍一下两个参数,一个是控制 aop 的具体实现方式,为 true 的话使用 cglib,为 false 的话使用 java 的 Proxy,默认为 false,第二个参数控制代理的暴露方式,解决内部调用不能使用代理的场景，默认为 false.

这里核心是@Import(AspectJAutoProxyRegistrar.class);在 AspectJAutoProxyRegistrar 里,核心的地方是

```
AopConfigUtils.registerAspectJAnnotationAutoProxyCreatorIfNecessary(registry);
```

一个 AOP 的工具类,这个工具类的主要作用是把 AnnotationAwareAspectJAutoProxyCreator 这个类定义为 BeanDefinition 放到 spring 容器中,这是通过实现 ImportBeanDefinitionRegistrar 接口来装载的,具体装载过程不是本篇的重点,这里就不赘述,我们重点看 AnnotationAwareAspectJAutoProxyCreator 这个类.

首先看看这个类图:



从类图是可以大致了解 AnnotationAwareAspectJAutoProxyCreator 这个类的功能.它实现了一系列 Aware 的接口,在 Bean 装载的时候获取

BeanFactory(Bean 容器),Bean 的 ClassLoader,还实现了 order 接口,继承了 PorxyConfig,ProxyConfig 中主要封装了代理的通用处理逻辑,比如设置目标类,设置使用 cglib 还是 java proxy 等一些基础配置.

而能够让这个类参与到 bean 初始化功能,并为 bean 添加代理功能的还是因为它实现了 BeanPostProcessor 这个接口.这个接口的 postProcessAfterInitialization 方法会在 bean 初始化结束后(赋值完成)被调用。

这里先看一下最顶部的抽象类:AbstractAutoProxyCreator,这个抽象类主要抽象了实现代理的逻辑:

```
1.  @Override
2.          public Object postProcessBeforeInitialization(Object bean, String beanName) {
3.                  return bean;
4.          }
5.
6.      // 主要看这个方法，在 bean 初始化之后对生产出的 bean 进行包装
7.          @Override
8.          public Object postProcessAfterInitialization(Object bean, String beanName) throws
    BeansException {
9.                  if (bean != null) {
10.                         Object cacheKey = getCacheKey(bean.getClass(), beanName);
11.                         if (!this.earlyProxyReferences.contains(cacheKey)) {
12.                                 return wrapIfNecessary(bean, beanName, cacheKey);
13.                         }
14.                 }
15.                 return bean;
16.         }
17.
18.     // wrapIfNecessary
19.         protected Object wrapIfNecessary(Object bean, String beanName, Object cacheKey) {
20.                 if (beanName != null && this.targetSourcedBeans.contains(beanName)) {
21.                         return bean;
22.                 }
23.                 if (Boolean.FALSE.equals(this.advisedBeans.get(cacheKey))) {
```

```
24.                         return bean;
25.                 }
26.                 if (isInfrastructureClass(bean.getClass()) || shouldSkip(bean.getClass(), beanName)) {
27.                         this.advisedBeans.put(cacheKey, Boolean.FALSE);
28.                         return bean;
29.                 }
30.
31.                 // Create proxy if we have advice.
32.         // 意思就是如果该类有 advice 则创建 proxy，
33.                 Object[] specificInterceptors = getAdvicesAndAdvisorsForBean(bean.getClass(),
beanName, null);
34.                 if (specificInterceptors != DO_NOT_PROXY) {
35.                         this.advisedBeans.put(cacheKey, Boolean.TRUE);
36.                 // 1.通过方法名也能简单猜测到，这个方法就是把 bean 包装为 proxy 的主要方法，
37.                         Object proxy = createProxy(
38.                                         bean.getClass(), beanName, specificInterceptors, new
SingletonTargetSource(bean));
39.                         this.proxyTypes.put(cacheKey, proxy.getClass());
40.
41.                 // 2.返回该 proxy 代替原来的 bean
42.                         return proxy;
43.                 }
44.
45.                 this.advisedBeans.put(cacheKey, Boolean.FALSE);
46.                 return bean;
47.         }
```

总结：

- **1）将 AnnotationAwareAspectJAutoProxyCreator 注册到 Spring 容器中**
- **2）AnnotationAwareAspectJAutoProxyCreator 类的 postProcessAfterInitialization()方法将所有有 advice 的 bean 重新包装成 proxy**

创建 proxy 过程分析

通过之前的代码结构分析，我们知道，所有的 bean 在返回给用户使用之前都需要经过 AnnotationAwareAspectJAutoProxyCreator 类的 postProcessAfterInitialization()方法，而该方法的主要作用也就是将所有拥有 advice 的 bean 重新包装为 proxy，那么我们接下来直接分析这个包装为 proxy 的方法即可，看一下 bean 如何被包装为 proxy，proxy 在被调用方法时，是具体如何执行的

以下是 AbstractAutoProxyCreator.wrapIfNecessary(Object bean, String beanName, Object cacheKey)中的 createProxy()代码片段分析

```
1.  protected Object createProxy(
2.                  Class<?> beanClass, String beanName, Object[] specificInterceptors,
    TargetSource targetSource) {
3.
4.          if (this.beanFactory instanceof ConfigurableListableBeanFactory) {
5.                  AutoProxyUtils.exposeTargetClass((ConfigurableListableBeanFactory)
    this.beanFactory, beanName, beanClass);
6.          }
7.
8.      // 1.创建 proxyFactory，proxy 的生产主要就是在 proxyFactory 做的
9.          ProxyFactory proxyFactory = new ProxyFactory();
10.         proxyFactory.copyFrom(this);
11.
12.         if (!proxyFactory.isProxyTargetClass()) {
13.                 if (shouldProxyTargetClass(beanClass, beanName)) {
14.                         proxyFactory.setProxyTargetClass(true);
15.                 }
16.                 else {
17.                         evaluateProxyInterfaces(beanClass, proxyFactory);
18.                 }
19.         }
20.
21.     // 2.将当前 bean 适合的 advice，重新封装下，封装为 Advisor 类，然后添加到 ProxyFactory 中
```

```
22.                     Advisor[] advisors = buildAdvisors(beanName, specificInterceptors);
23.                     for (Advisor advisor : advisors) {
24.                             proxyFactory.addAdvisor(advisor);
25.                     }
26.
27.                     proxyFactory.setTargetSource(targetSource);
28.                     customizeProxyFactory(proxyFactory);
29.
30.                     proxyFactory.setFrozen(this.freezeProxy);
31.                     if (advisorsPreFiltered()) {
32.                             proxyFactory.setPreFiltered(true);
33.                     }
34.
35.             // 3.调用getProxy 获取bean 对应的proxy
36.                     return proxyFactory.getProxy(getProxyClassLoader());
37.             }
```

TargetSource 中存放被代理的对象,这段代码主要是为了构建 ProxyFactory,将配置信息(是否使用 java proxy,是否 threadlocal 等),目标类,切面,传入 ProxyFactory 中

1）创建何种类型的 Proxy？JDKProxy 还是 CGLIBProxy？

```
1.  // getProxy()方法
2.          public Object getProxy(ClassLoader classLoader) {
3.                  return createAopProxy().getProxy(classLoader);
4.          }
5.
6.
7.      // createAopProxy()方法就是决定究竟创建何种类型的proxy
8.          protected final synchronized AopProxy createAopProxy() {
9.                  if (!this.active) {
10.                         activate();
11.                 }
12.             // 关键方法createAopProxy()
13.                 return getAopProxyFactory().createAopProxy(this);
14.         }
```

```
15.
16.     // createAopProxy()
17.         public AopProxy createAopProxy(AdvisedSupport config) throws AopConfigException {
18.             // 1.config.isOptimize()是否使用优化的代理策略，目前使用与CGLIB
19.             // config.isProxyTargetClass()  是否目标类本身被代理而不是目标类的接口
20.             // hasNoUserSuppliedProxyInterfaces()是否存在代理接口
21.                         if (config.isOptimize() || config.isProxyTargetClass() ||
    hasNoUserSuppliedProxyInterfaces(config)) {
22.                             Class<?> targetClass = config.getTargetClass();
23.                             if (targetClass == null) {
24.                                 throw new AopConfigException("TargetSource cannot
    determine target class: " +
25.                                                         "Either an interface or a target is required
    for proxy creation.");
26.                             }
27.
28.                 // 2.如果目标类是接口或者是代理类，则直接使用JDKproxy
29.                             if (targetClass.isInterface() || Proxy.isProxyClass(targetClass)) {
30.                                 return new JdkDynamicAopProxy(config);
31.                             }
32.
33.                 // 3.其他情况则使用CGLIBproxy
34.                             return new ObjenesisCglibAopProxy(config);
35.                     }
36.                     else {
37.                             return new JdkDynamicAopProxy(config);
38.                     }
39.         }
```

2）getProxy()方法

由 1）可知，通过 createAopProxy()方法来确定具体使用何种类型的 Proxy，针对于该示例，我们具体使用的为 JdkDynamicAopProxy，下面来看下 JdkDynamicAopProxy.getProxy()方法

```
1.  final class JdkDynamicAopProxy implements AopProxy, InvocationHandler, Serializable//
    JdkDynamicAopProxy 类结构，由此可知，其实现了 InvocationHandler，则必定有 invoke 方法，来被
    调用，也就是用户调用 bean 相关方法时，此 invoke()被真正调用
2.      // getProxy()
3.      public Object getProxy(ClassLoader classLoader) {
4.                      if (logger.isDebugEnabled()) {
5.                              logger.debug("Creating JDK dynamic proxy: target source is " +
    this.advised.getTargetSource());
6.                      }
7.                      Class<?>[] proxiedInterfaces =
    AopProxyUtils.completeProxiedInterfaces(this.advised, true);
8.                      findDefinedEqualsAndHashCodeMethods(proxiedInterfaces);
9.
10.             // JDK proxy  动态代理的标准用法
11.                     return Proxy.newProxyInstance(classLoader, proxiedInterfaces, this);
12.         }
```

3）invoke()方法

以上的代码模式可以很明确的看出来，使用了 JDK 动态代理模式，真正的方法执行在 invoke()方法里，下面我们来看下该方法，来看下 bean 方法如何被代理执行的

```
1.          public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
2.                      MethodInvocation invocation;
3.                      Object oldProxy = null;
4.                      boolean setProxyContext = false;
5.
6.                      TargetSource targetSource = this.advised.targetSource;
7.                      Class<?> targetClass = null;
8.                      Object target = null;
9.
10.                     try {
11.                 // 1.以下的几个判断，主要是为了判断 method 是否为 equals、hashCode 等 Object 的方
    法
12.                             if (!this.equalsDefined && AopUtils.isEqualsMethod(method)) {
```

```
13.                                     // The target does not implement the equals(Object) method
itself.
14.                                     return equals(args[0]);
15.                             }
16.                             else if (!this.hashCodeDefined && AopUtils.isHashCodeMethod(method)) {
17.                                     // The target does not implement the hashCode() method itself.
18.                                     return hashCode();
19.                             }
20.                             else if (method.getDeclaringClass() == DecoratingProxy.class) {
21.                                     // There is only getDecoratedClass() declared -> dispatch to
proxy config.
22.                                     return AopProxyUtils.ultimateTargetClass(this.advised);
23.                             }
24.                             else if (!this.advised.opaque && method.getDeclaringClass().isInterface()
&&
25.                                             method.getDeclaringClass().isAssignableFrom(Advised.class)) {
26.                                     // Service invocations on ProxyConfig with the proxy config...
27.                                     return AopUtils.invokeJoinpointUsingReflection(this.advised,
method, args);
28.                             }
29.
30.                             Object retVal;
31.
32.                             if (this.advised.exposeProxy) {
33.                                     // Make invocation available if necessary.
34.                                     oldProxy = AopContext.setCurrentProxy(proxy);
35.                                     setProxyContext = true;
36.                             }
37.
38.                             // May be null. Get as late as possible to minimize the time we "own" the
target,
39.                             // in case it comes from a pool.
40.                             target = targetSource.getTarget();
41.                             if (target != null) {
42.                                     targetClass = target.getClass();
43.                             }
44.
45.                             // 2.获取当前 bean 被拦截方法链表
46.                             List<Object> chain =
this.advised.getInterceptorsAndDynamicInterceptionAdvice(method, targetClass);
```

```
47.
48.                         // 3.如果为空，则直接调用 target 的 method
49.                         if (chain.isEmpty()) {
50.                                 Object[] argsToUse =
AopProxyUtils.adaptArgumentsIfNecessary(method, args);
51.                                 retVal = AopUtils.invokeJoinpointUsingReflection(target,
method, argsToUse);
52.                         }
53.                 // 4.不为空，则逐一调用 chain 中的每一个拦截方法的 proceed
54.                         else {
55.                                 // We need to create a method invocation...
56.                                 invocation = new ReflectiveMethodInvocation(proxy, target,
method, args, targetClass, chain);
57.                                 // Proceed to the joinpoint through the interceptor chain.
58.                                 retVal = invocation.proceed();
59.                         }
60.
61.                         ...
62.                         return retVal;
63.                 }
64.                 ...
65.         }
```

4）拦截方法真正被执行调用 invocation.proceed()

```
1.          public Object proceed() throws Throwable {
2.                  //          We start with an index of -1 and increment early.
3.                  if (this.currentInterceptorIndex ==
this.interceptorsAndDynamicMethodMatchers.size() - 1) {
4.                          return invokeJoinpoint();
5.                  }
6.
7.                  Object interceptorOrInterceptionAdvice =
8.
          this.interceptorsAndDynamicMethodMatchers.get(++this.currentInterceptorIndex);
9.                  if (interceptorOrInterceptionAdvice instanceof
InterceptorAndDynamicMethodMatcher) {
10.                         // Evaluate dynamic method matcher here: static part will already have
11.                         // been evaluated and found to match.
12.                         InterceptorAndDynamicMethodMatcher dm =
13.                                         (InterceptorAndDynamicMethodMatcher)
interceptorOrInterceptionAdvice;
```

```
14.                         if (dm.methodMatcher.matches(this.method, this.targetClass,
    this.arguments)) {
15.                             return dm.interceptor.invoke(this);
16.                         }
17.                         else {
18.                             // Dynamic matching failed.
19.                             // Skip this interceptor and invoke the next in the chain.
20.                             return proceed();
21.                         }
22.                     }
23.                     else {
24.                         // It's an interceptor, so we just invoke it: The pointcut will have
25.                         // been evaluated statically before this object was constructed.
26.                         return ((MethodInterceptor) interceptorOrInterceptionAdvice).invoke(this);
27.                     }
28.         }
```

总结 4：依次遍历拦截器链的每个元素，然后调用其实现，将真正调用工作委托给各个增强器

总结：

纵观以上过程可知：实际就是为 bean 创建一个 proxy，JDKproxy 或者 CGLIBproxy，然后在调用 bean 的方法时，会通过 proxy 来调用 bean 方法

重点过程可分为：

1）将 AnnotationAwareAspectJAutoProxyCreator 注册到 Spring 容器中

2）AnnotationAwareAspectJAutoProxyCreator 类的 postProcessAfterInitialization()方法将所有有 advice 的 bean 重新包装成 proxy

3）调用 bean 方法时通过 proxy 来调用，proxy 依次调用增强器的相关方法，来实现方法切

总结：

1）、 @EnableAspectJAutoProxy 开启 AOP 功能

2）、 @EnableAspectJAutoProxy 会给容器中注册一个组件 AnnotationAwareAspectJAutoProxyCreator

3）、AnnotationAwareAspectJAutoProxyCreator 是一个后置处理器；

4）、容器的创建流程：

1）、registerBeanPostProcessors（）注册后置处理器；创建 AnnotationAwareAspectJAutoProxyCreator 对象

2）、finishBeanFactoryInitialization（）初始化剩下的单实例 bean

1）、创建业务逻辑组件和切面组件

2）、AnnotationAwareAspectJAutoProxyCreator 拦截组件的创建过程

3）、组件创建完之后，判断组件是否需要增强

是：切面的通知方法，包装成增强器（Advisor）;给业务逻辑组件创建一个代理对象（cglib）;

5）、执行目标方法：

1）、代理对象执行目标方法

2）、CglibAopProxy.intercept();

1）、得到目标方法的拦截器链（增强器包装成拦截器 MethodInterceptor）

2）、利用拦截器的链式机制，依次进入每一个拦截器进行执行；

3）、效果：

正常执行：前置通知-》目标方法-》后置通知-》返回通知

出现异常：前置通知-》目标方法-》后置通知-》异常通知