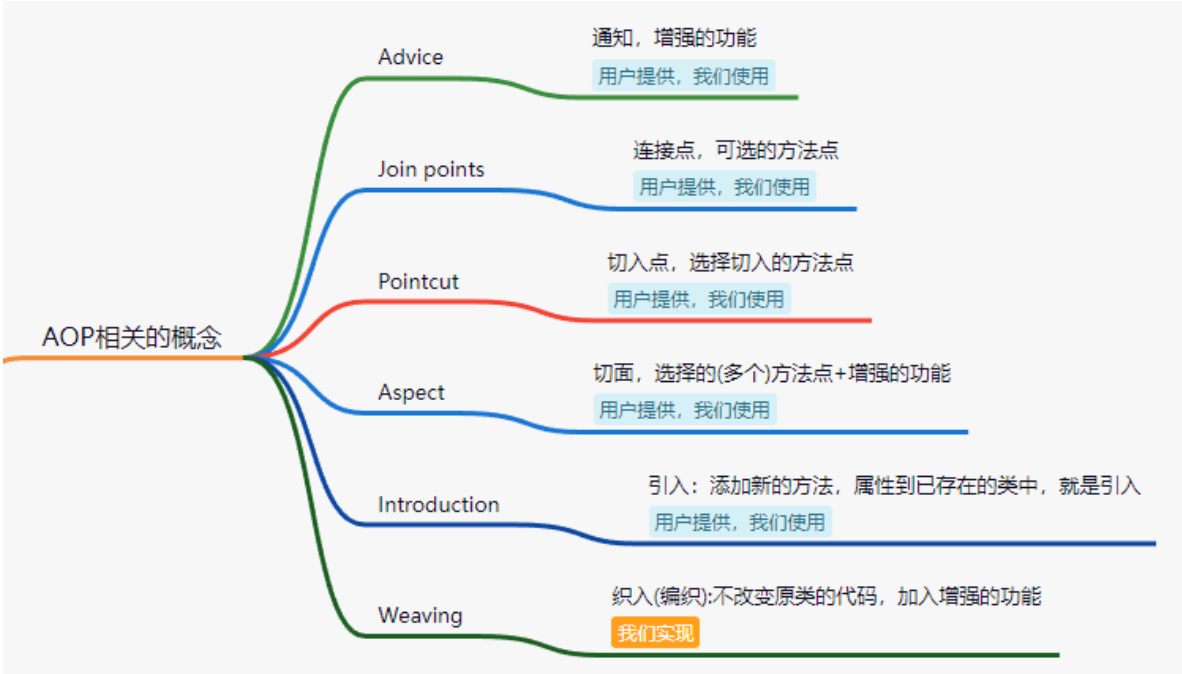
Spring源码-AOP分析

一、手写AOP回顾

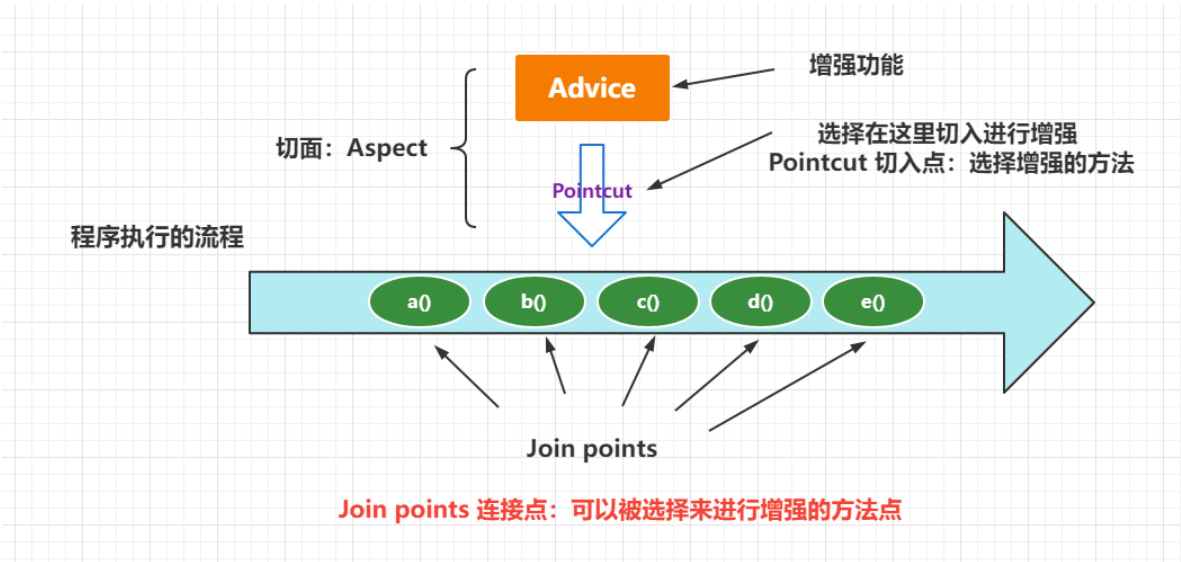
本文我们开始讲解Spring中的AOP原理和源码，我们前面手写了AOP的实现，了解和自己实现AOP应该要具备的内容，我们先回顾下，这对我们理解Spring的AOP是非常有帮助的。

1. 涉及的相关概念

先回顾下核心的概念，比如：Advice，Pointcut，Aspect等

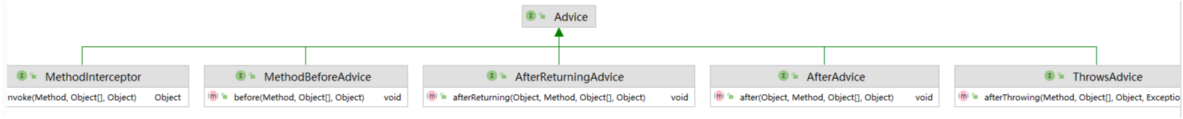


更加形象的描述：

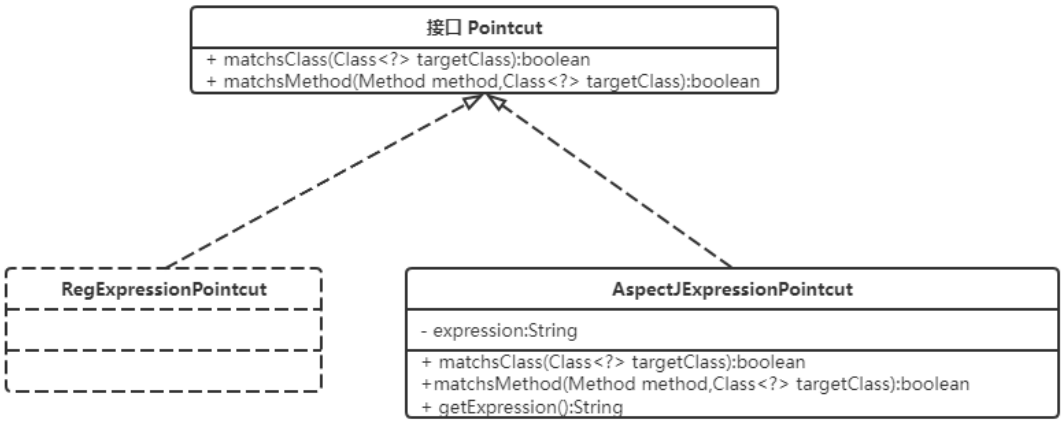


2. 相关核心的设计

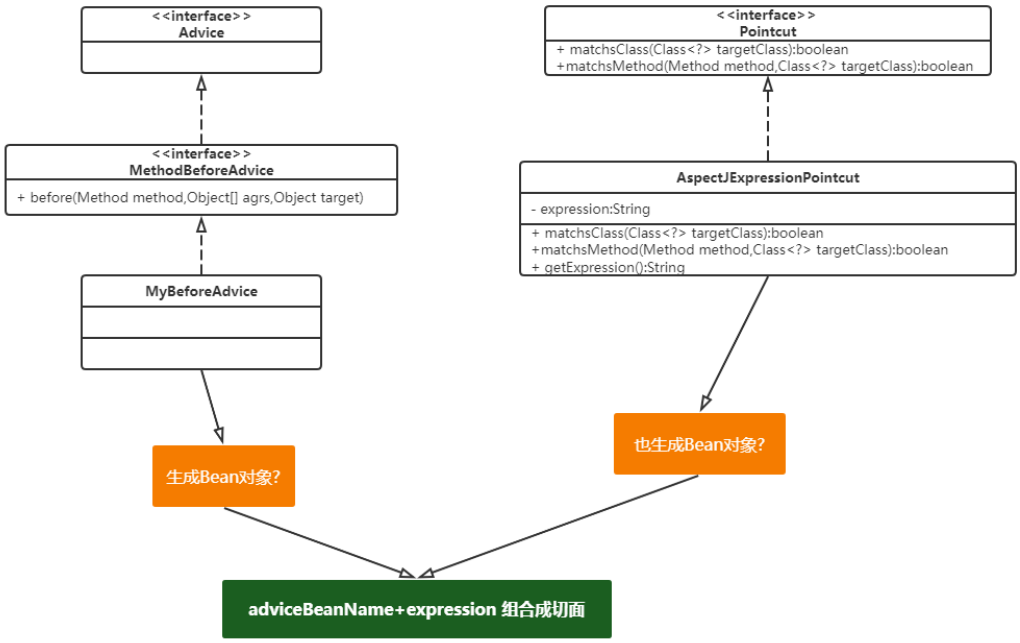
Advice:



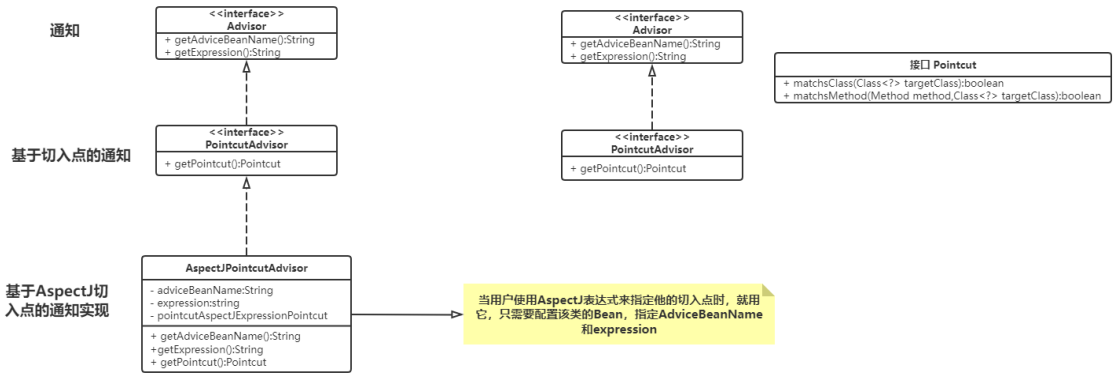
Pointcut:



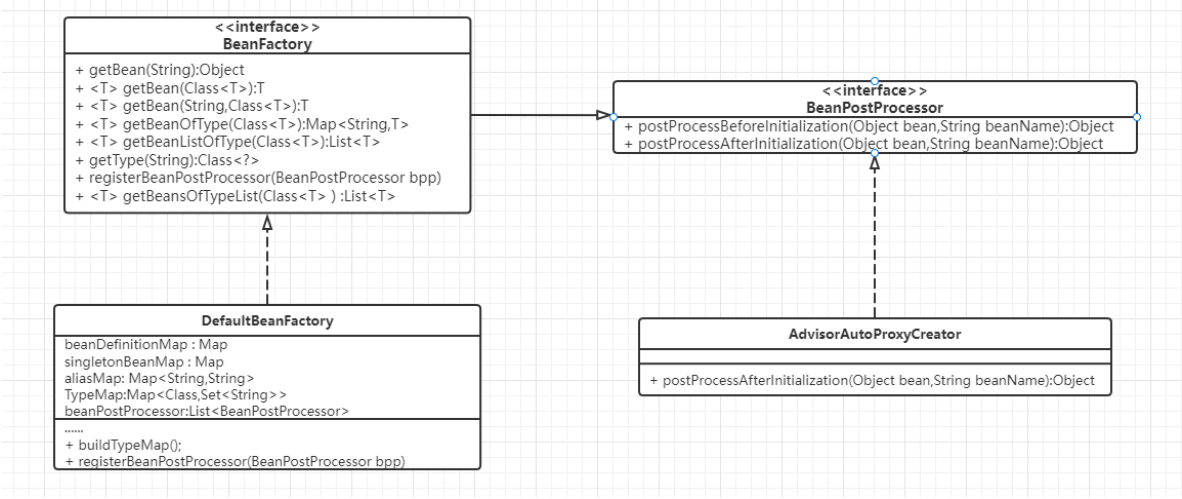
Aspect：

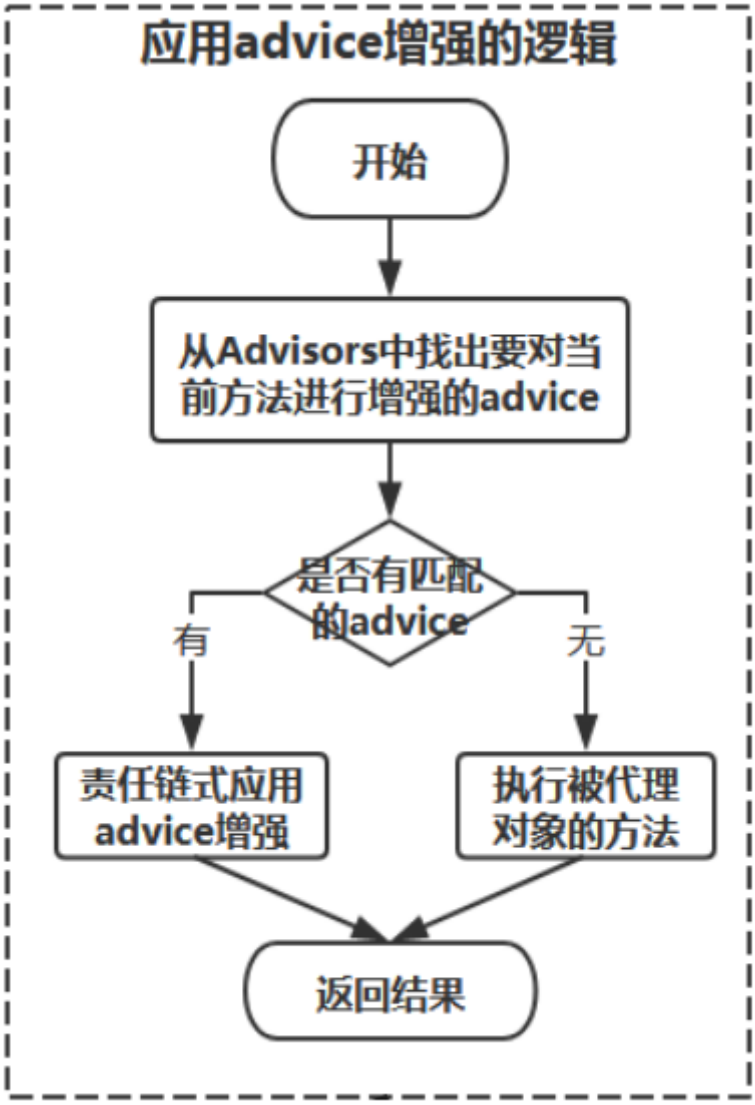


Advisor:



织入：



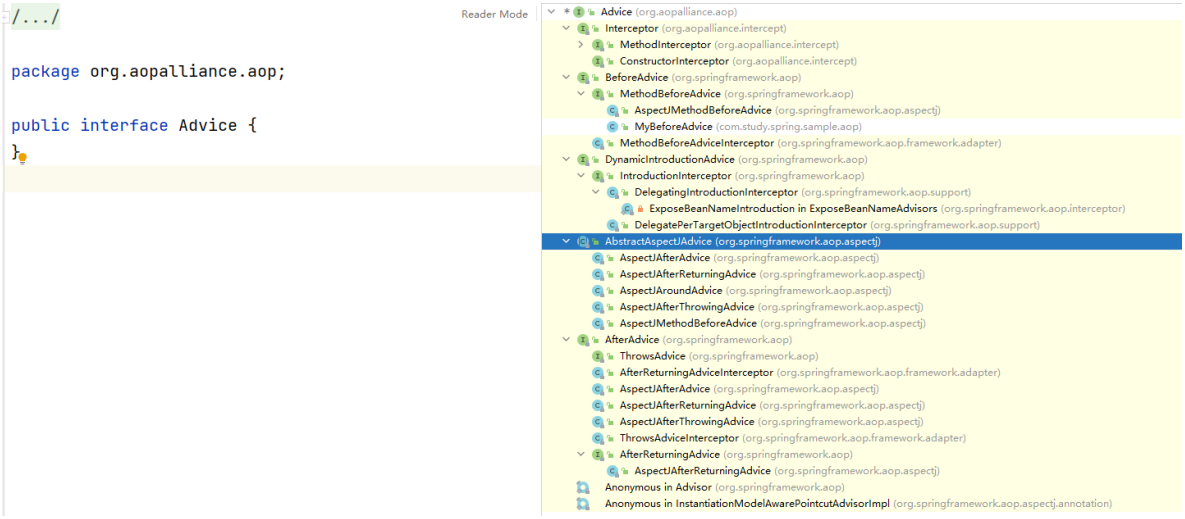


二、AOP相关概念的类结构

回顾了前面的内容，然后我们来看看Spring中AOP是如何来实现的了。

1. Advice类结构

我们先来看看Advice的类结构，advice--》通知，需要增强的功能



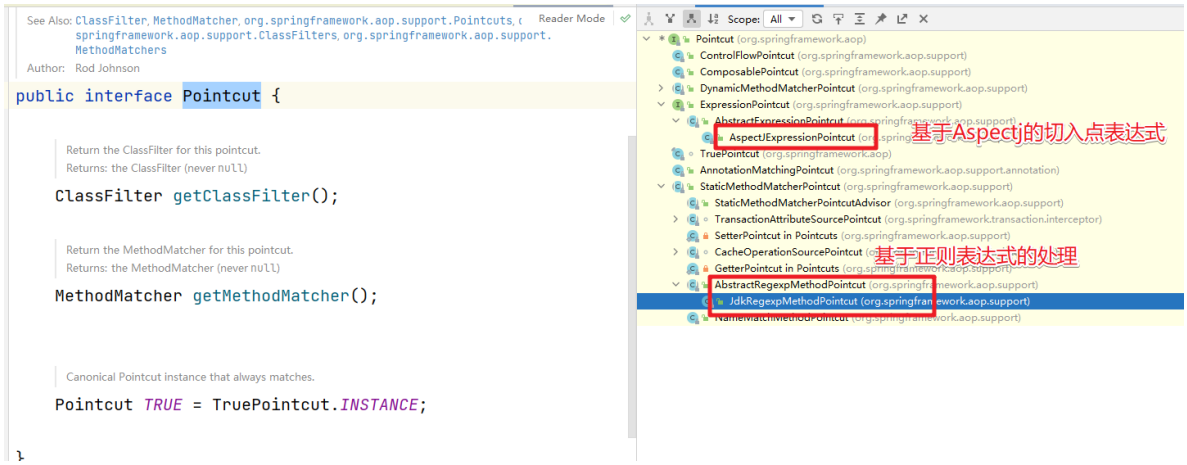
相关的说明

2. Pointcut类结构

然后来看看Pointcut的设计，也就是切入点的处理。

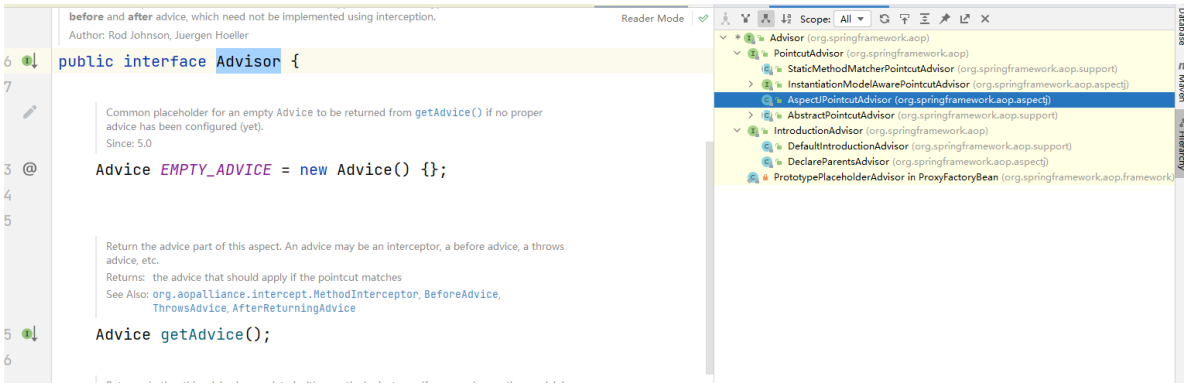
```
Author: Rod Johnson
public interface Pointcut {

    Return the ClassFilter for this pointcut.
    Returns: the ClassFilter (never null)
    ClassFilter getClassFilter();  匹配类型

    Return the MethodMatcher for this pointcut.
    Returns: the MethodMatcher (never null)
    MethodMatcher getMethodMatcher();  匹配方法

    Canonical Pointcut instance that always matches.
    Pointcut TRUE = TruePointcut.INSTANCE;

}
```

Pointcut的两种实现方式



3. Advisor类结构

Advisor的类结构比较简单。一个是PointcutAdvisor,一个是IntroductionAdvisor



我们要看的重点是 PointcutAdvisor 及实现 AspectJPointcutAdvisor。

三、织入的实现

1. BeanPostProcessor

1.1 案例演示

我们通过案例来看，首先使用AOP来增强。

定义切面类

```
/**
 * 切面类
 */
@Component
@EnableAspectJAutoProxy
@Aspect
public class AspectAdviceBeanUseAnnotation {

    // 定义一个全局的Pointcut
    @Pointcut("execution(* com.study.spring.sample.aop.*.do*(..))")
    public void doMethods() {
    }

    @Pointcut("execution(* com.study.spring.sample.aop.*.service*(..))")
    public void services() {
    }

    // 定义一个Before Advice
    @Before("doMethods() and args(tk,..)")
    public void before3(String tk) {
        System.out.println("----------- AspectAdviceBeanUseAnnotation before3
增强  参数tk= " + tk);
    }

    @Around("services() and args(name,..)")
    public Object around2(ProceedingJoinPoint pjp, String name) throws Throwable
{
        System.out.println("--------- AspectAdviceBeanUseAnnotation arround2 参数
name=" + name);
        System.out.println("----------- AspectAdviceBeanUseAnnotation arround2
环绕-前增强 for " + pjp);
        Object ret = pjp.proceed();
        System.out.println("----------- AspectAdviceBeanUseAnnotation arround2
环绕-后增强 for " + pjp);
        return ret;
    }

    @AfterReturning(pointcut = "services()", returning = "retValue")
    public void afterReturning(Object retValue) {
        System.out.println("----------- AspectAdviceBeanUseAnnotation
afterReturning 增强 , 返回值为:  " + retValue);
    }

    @AfterThrowing(pointcut = "services()", throwing = "e")
    public void afterThrowing(JoinPoint jp, Exception e) {
        System.out.println("----------- AspectAdviceBeanUseAnnotation
afterThrowing 增强  for " + jp);
        System.out.println("----------- AspectAdviceBeanUseAnnotation
afterThrowing 增强  异常 : " + e);
```

```
    }

    @After("doMethods()")
    public void after(JoinPoint jp) {
        System.out.println("----------- AspectAdviceBeanUseAnnotation after 增强
for " + jp);
    }

    /*
     * BeanDefinitionRegistryPostProcessor BeanFactoryPostProcessor
     * InstantiationAwareBeanPostProcessor Bean实例创建前后 BeanPostProcessor
     */
}
```

需要增强的目标类

```
@Component
public class BeanQ {

    public void do1(String task, int time) {
        System.out.println("-------------do1 do " + task + " time:" + time);
    }

    public String service1(String name) {
        System.out.println("-------------servce1 do " + name);
        return name;
    }

    public String service2(String name) {
        System.out.println("-------------servce2 do " + name);
        if (!"s1".equals(name)) {
            throw new IllegalArgumentException("参数 name ! = s1, name=" + name);
        }

        return name + " hello!";
    }
}
```

测试代码

```
@Configuration
@ComponentScan
public class AopMainAnno {
    public static void main(String[] args) {
        ApplicationContext context = new
AnnotationConfigApplicationContext(AopMainAnno.class);
        BeanQ bq = context.getBean(BeanQ.class);
        bq.do1("task1", 20);
        System.out.println();

        bq.service1("service1");

        System.out.println();
        bq.service2("s1");
    }
}
```

执行即可看到增强的效果

1.2 @EnableAspectJAutoProxy

我们需要使用代理增强处理，必须添加@EnableAspectJAutoProxy才生效。我们来看看他做了什么事情

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Import(AspectJAutoProxyRegistrar.class)
public @interface EnableAspectJAutoProxy {

    Indicate whether subclass-based (CGLIB) proxies are to be created as opposed to standard Java
    interface-based proxies. The default is false.
    boolean proxyTargetClass() default false;

    Indicate that the proxy should be exposed by the AOP framework as a ThreadLocal for retrieval
    via the org.springframework.aop.framework.AopContext class. Off by default, i.e. no
    guarantees that AopContext access will work.
    Since: 4.3.1
    boolean exposeProxy() default false;

}
```

```
Since:  3.1
See Also: EnableAspectJAutoProxy
Author:  Chris Beams, Juergen Hoeller
class AspectJAutoProxyRegistrar implements ImportBeanDefinitionRegistrar {

    Register, escalate, and configure the AspectJ auto proxy creator based on the value of the
    @EnableAspectJAutoProxy.proxyTargetClass() attribute on the importing
    @Configuration class.
    @Override
    public void registerBeanDefinitions(
            AnnotationMetadata importingClassMetadata, BeanDefinitionRegistry registry) {

        AopConfigUtils.registerAspectJAnnotationAutoProxyCreatorIfNecessary(registry);

        AnnotationAttributes enableAspectJAutoProxy =
                AnnotationConfigUtils.attributesFor(importingClassMetadata, EnableAspectJAutoProxy.class);
        if (enableAspectJAutoProxy != null) {
            if (enableAspectJAutoProxy.getBoolean( attributeName: "proxyTargetClass")) {
                AopConfigUtils.forceAutoProxyCreatorToUseClassProxying(registry);
            }
            if (enableAspectJAutoProxy.getBoolean( attributeName: "exposeProxy")) {
                AopConfigUtils.forceAutoProxyCreatorToExposeProxy(registry);
            }
        }
    }
```

关键进入

```
@Nullable
public static BeanDefinition registerAspectJAnnotationAutoProxyCreatorIfNecessary(
        BeanDefinitionRegistry registry, @Nullable Object source) {

    return registerOrEscalateApcAsRequired(AnnotationAwareAspectJAutoProxyCreator.class, registry, source);
}
```

会把该对象注入到容器中

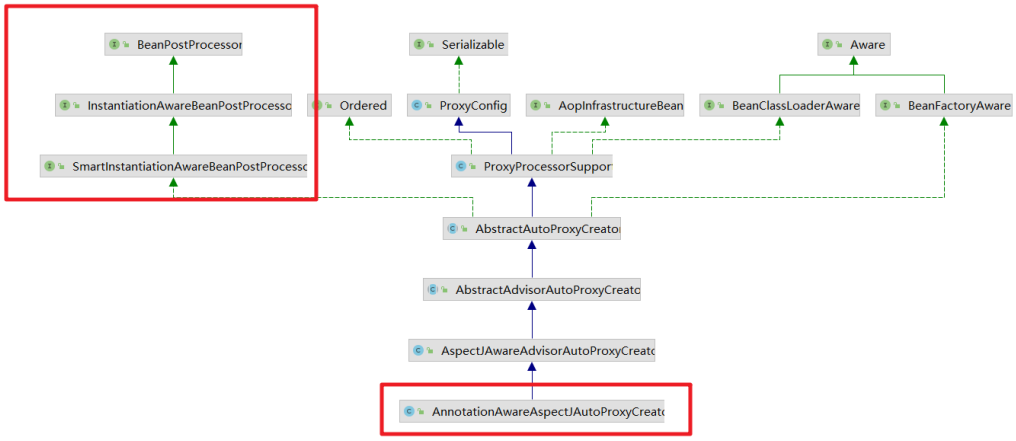
在registerOrEscalateApcAsRequired方法中会把上面的Java类注入到容器中。

```java
@Nullable
private static BeanDefinition registerOrEscalateApcAsRequired(
        Class<?> cls, BeanDefinitionRegistry registry, @Nullable Object source) {

    Assert.notNull(registry, message: "BeanDefinitionRegistry must not be null");

    if (registry.containsBeanDefinition(AUTO_PROXY_CREATOR_BEAN_NAME)) {
        BeanDefinition apcDefinition = registry.getBeanDefinition(AUTO_PROXY_CREATOR_BEAN_NAME);
        if (!cls.getName().equals(apcDefinition.getBeanClassName())) {
            int currentPriority = findPriorityForClass(apcDefinition.getBeanClassName());
            int requiredPriority = findPriorityForClass(cls);
            if (currentPriority < requiredPriority) {
                apcDefinition.setBeanClassName(cls.getName());
            }
        }
        return null;
    }

    RootBeanDefinition beanDefinition = new RootBeanDefinition(cls);
    beanDefinition.setSource(source);
    beanDefinition.getPropertyValues().add( propertyName: "order", Ordered.HIGHEST_PRECEDENCE);
    beanDefinition.setRole(BeanDefinition.ROLE_INFRASTRUCTURE);
    registry.registerBeanDefinition(AUTO_PROXY_CREATOR_BEAN_NAME, beanDefinition);
    return beanDefinition;
```

所以我们需要看看 AnnotationAwareAspectJAutoProxyCreator 的结构

1.3 AnnotationAwareAspectJAutoProxyCreator

我们直接来看类图结构，可以发现其本质就是一个 BeanPostProcessor ,只是扩展了更多的功能。

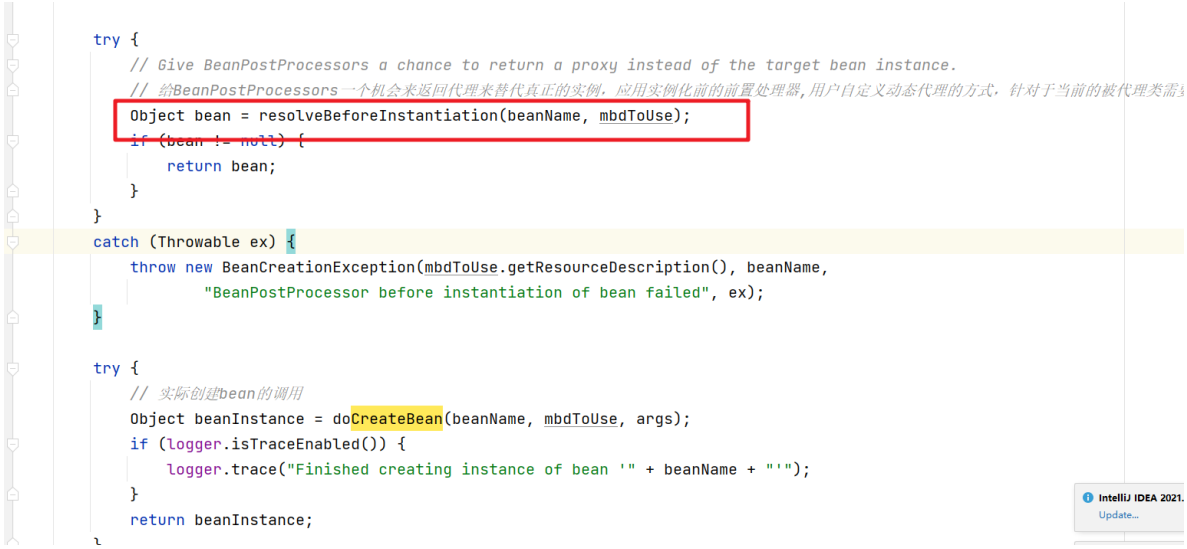


那么具体处理的逻辑

```
        if (!StringUtils.hasLength(beanName) || !this.targetSourcedBeans.contains(beanName)) {
            if (this.advisedBeans.containsKey(cacheKey)) {
                return null;
            }
            if (isInfrastructureClass(beanClass) || shouldSkip(beanClass, beanName)) {
                this.advisedBeans.put(cacheKey, Boolean.FALSE);
                return null;
            }
        }

        // Create proxy here if we have a custom TargetSource.
        // Suppresses unnecessary default instantiation of the target bean:
        // The TargetSource will handle target instances in a custom fashion.
        TargetSource targetSource = getCustomTargetSource(beanClass, beanName);
        if (targetSource != null) {
            if (StringUtils.hasLength(beanName)) {
                this.targetSourcedBeans.add(beanName);
            }
            Object[] specificInterceptors = getAdvicesAndAdvisorsForBean(beanClass, beanName, targetSource);
            Object proxy = createProxy(beanClass, beanName, specificInterceptors, targetSource);
            this.proxyTypes.put(cacheKey, proxy.getClass());
            return proxy;
        }

        return null;
    }
```

1.4 如何串联

Bean的IoC是如何和对应的BeanPostProcessor串联的呢？我们来看看。



```
    try {
        // Give BeanPostProcessors a chance to return a proxy instead of the target bean instance.
        // 给BeanPostProcessors一个机会来返回代理来替代真正的实例，应用实例化前的前置处理器,用户自定义动态代理的方式，针对于当前的被代理类需
        Object bean = resolveBeforeInstantiation(beanName, mbdToUse);
        if (bean != null) {
            return bean;
        }
    }
    catch (Throwable ex) {
        throw new BeanCreationException(mbdToUse.getResourceDescription(), beanName,
                "BeanPostProcessor before instantiation of bean failed", ex);
    }

    try {
        // 实际创建bean的调用
        Object beanInstance = doCreateBean(beanName, mbdToUse, args);
        if (logger.isTraceEnabled()) {
            logger.trace("Finished creating instance of bean '" + beanName + "'");
        }
        return beanInstance;
    }
```

```
@Nullable
protected Object resolveBeforeInstantiation(String beanName, RootBeanDefinition mbd) {
    Object bean = null;
    // 如果beforeInstantiationResolved值为null或者true，那么表示尚未被处理，进行后续的处理
    if (!Boolean.FALSE.equals(mbd.beforeInstantiationResolved)) {
        // Make sure bean class is actually resolved at this point.
        // 确认beanclass确实在此处进行处理
        // 判断当前mbd是否是合成的，只有在实现aop的时候synthetic的值才为true，并且是否实现了InstantiationAwareBeanPostProcessor
        if (!mbd.isSynthetic() && hasInstantiationAwareBeanPostProcessors()) {
            // 获取类型
            Class<?> targetType = determineTargetType(beanName, mbd);
            if (targetType != null) {
                bean = applyBeanPostProcessorsBeforeInstantiation(targetType, beanName);
                if (bean != null) {
                    bean = applyBeanPostProcessorsAfterInitialization(bean, beanName);
                }
            }
        }
        // 是否解析了
        mbd.beforeInstantiationResolved = (bean != null);
    }
    return bean;
}
```

```
    */
    @Nullable
    protected Object applyBeanPostProcessorsBeforeInstantiation(Class<?> beanClass, String beanName) {
        for (BeanPostProcessor bp : getBeanPostProcessors()) {
            if (bp instanceof InstantiationAwareBeanPostProcessor) {
                InstantiationAwareBeanPostProcessor ibp = (InstantiationAwareBeanPostProcessor) bp;
                Object result = ibp.postProcessBeforeInstantiation(beanClass, beanName);
                if (result != null) {
                    return result;
                }
            }
        }
        return null;
    }
```

```
     * @see org.springframework.beans.factory.support.AbstractBeanDefinition#getFactoryMethodName()
     */
    @Nullable
```

关联到了前面介绍的了

```
Choose Overriding Method of postProcessBeforeInstantiation (6 methods found)
AbstractAutoProxyCreator (org.springframework.aop.framework.autoproxy)                    spring.spring-aop.main
CommonAnnotationBeanPostProcessor (org.springframework.context.annotation)                spring.spring-context.main
InstantiationAwareBeanPostProcessorAdapter (org.springframework.beans.factory.config)     spring.spring-beans.main
MyInstantiationAwareBeanPostProcessor (com.mashibing.resolveBeforeInstantiation)          spring.spring-debug.main
PersistenceAnnotationBeanPostProcessor (org.springframework.orm.jpa.support)              spring.spring-orm.main
ScriptFactoryPostProcessor (org.springframework.scripting.support)                        spring.spring-context.main
```

```
     * Perform operations after the bean has been instantiated, via a constructor or factory method.
```

```
    @Override
    public Object postProcessBeforeInstantiation(Class<?> beanClass, String beanName) {
        Object cacheKey = getCacheKey(beanClass, beanName);

        if (!StringUtils.hasLength(beanName) || !this.targetSourcedBeans.contains(beanName)) {
            //查缓存，是否有处理过了，不管是不是需要通知增强的，只要处理过了就会放里面
            if (this.advisedBeans.containsKey(cacheKey)) {
                return null;
            }
            if (isInfrastructureClass(beanClass) || shouldSkip(beanClass, beanName)) {
                // 要跳过的直接设置FALSE
                this.advisedBeans.put(cacheKey, Boolean.FALSE);
                return null;
            }
        }

        // Create proxy here if we have a custom TargetSource.
        // Suppresses unnecessary default instantiation of the target bean:
        // The TargetSource will handle target instances in a custom fashion.
        TargetSource targetSource = getCustomTargetSource(beanClass, beanName);
        if (targetSource != null) {
            if (StringUtils.hasLength(beanName)) {
                this.targetSourcedBeans.add(beanName);
            }
            Object[] specificInterceptors = getAdvicesAndAdvisorsForBean(beanClass, beanName, targetSource);
```

isInfrastructureClass方法判断是否是基础设施

```
    */
    protected boolean isInfrastructureClass(Class<?> beanClass) {
        boolean retVal = Advice.class.isAssignableFrom(beanClass) ||
                Pointcut.class.isAssignableFrom(beanClass) ||
                Advisor.class.isAssignableFrom(beanClass) ||
                AopInfrastructureBean.class.isAssignableFrom(beanClass);
        if (retVal && logger.isTraceEnabled()) {
            logger.trace("Did not attempt to auto-proxy infrastructure class [" + beanClass.getName() + "]");
        }
        return retVal;
    }
```

shouldSkip：是否应该跳过，会完成相关的advisor的收集

```
    @Override
    protected boolean shouldSkip(Class<?> beanClass, String beanName) {
        // TODO: Consider optimization by caching the list of the aspect names
        List<Advisor> candidateAdvisors = findCandidateAdvisors();
        for (Advisor advisor : candidateAdvisors) {
            if (advisor instanceof AspectJPointcutAdvisor &&
                    ((AspectJPointcutAdvisor) advisor).getAspectName().equals(beanName)) {
                return true;
            }
        }
        return super.shouldSkip(beanClass, beanName);
    }
```

具体的处理流程

```
    public List<Advisor> findAdvisorBeans() {
        // Determine list of advisor bean names, if not cached already.
        String[] advisorNames = this.cachedAdvisorBeanNames;
        if (advisorNames == null) {
            // Do not initialize FactoryBeans here: We need to leave all regular
beans
            // uninitialized to let the auto-proxy creator apply to them!
            // 获取当前BeanFactory中所有实现了Advisor接口的bean的名称
            advisorNames = BeanFactoryUtils.beanNamesForTypeIncludingAncestors(
                    this.beanFactory, Advisor.class, true, false);
            this.cachedAdvisorBeanNames = advisorNames;
        }
        if (advisorNames.length == 0) {
            return new ArrayList<>();
        }

        // 对获取到的实现Advisor接口的bean的名称进行遍历
        List<Advisor> advisors = new ArrayList<>();
        // 循环所有的beanName，找出对应的增强方法
        for (String name : advisorNames) {
            // isEligibleBean()是提供的一个hook方法，用于子类对Advisor进行过滤，这里默认
返回值都是true
            if (isEligibleBean(name)) {
                // 如果当前bean还在创建过程中，则略过，其创建完成之后会为其判断是否需要织入
切面逻辑
                if (this.beanFactory.isCurrentlyInCreation(name)) {
                    if (logger.isTraceEnabled()) {
                        logger.trace("Skipping currently created advisor '" +
name + "'");
                    }
                }
                else {
                    try {
                        // 将当前bean添加到结果中
                        advisors.add(this.beanFactory.getBean(name,
Advisor.class));
                    }
                    catch (BeanCreationException ex) {
                        // 对获取过程中产生的异常进行封装
                        Throwable rootCause = ex.getMostSpecificCause();
                        if (rootCause instanceof
BeanCurrentlyInCreationException) {
```

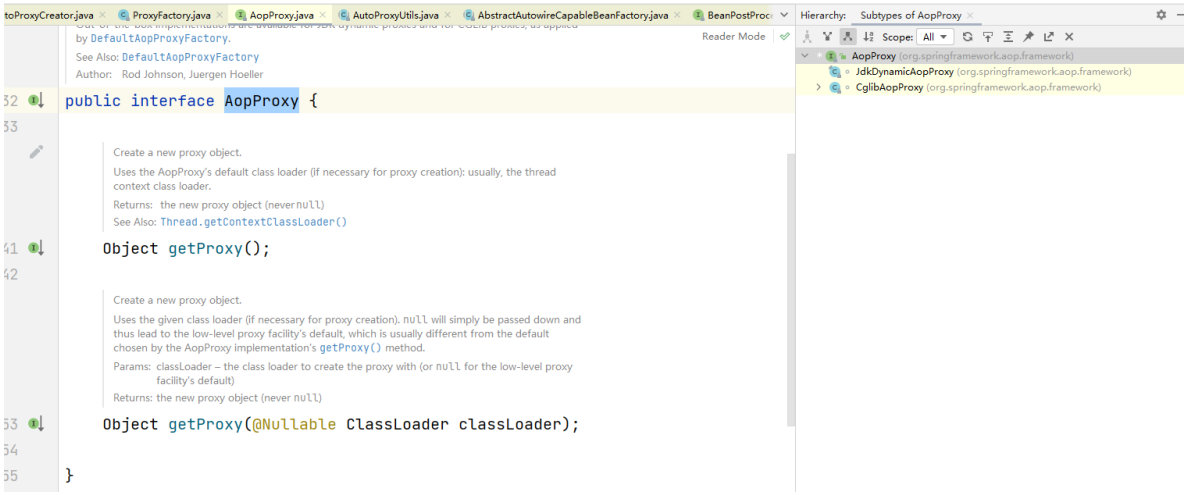
```
                            BeanCreationException bce = (BeanCreationException)
rootCause;
                            String bceBeanName = bce.getBeanName();
                            if (bceBeanName != null &&
this.beanFactory.isCurrentlyInCreation(bceBeanName)) {
                                if (logger.isTraceEnabled()) {
                                    logger.trace("Skipping advisor '" + name +
                                            "' with dependency on currently
created bean: " + ex.getMessage());
                                }
                                // Ignore: indicates a reference back to the
bean we're trying to advise.
                                // We want to find advisors other than the
currently created bean itself.
                                continue;
                            }
                        }
                        throw ex;
                    }
                }
            }
        }
        return advisors;
    }
```

2. 代理类的结构

在上面的分析中出现了很多代理相关的代码，为了更好的理解，我们来梳理下Spring中的代理相关的结构

2.1 AopProxy

在Spring中创建代理对象都是通过AopProxy这个接口的两个具体实现类来实现的，也就是jdk和cglib两种方式。



2.2 AopProxyFactory

在Spring中通过AopProxyFactory这个工厂类来提供AopProxy。

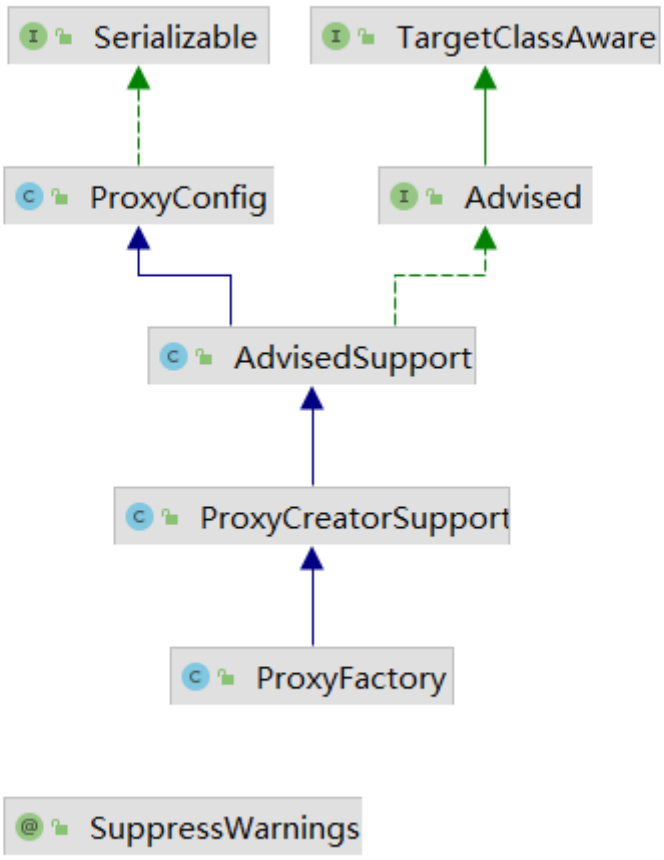
默认的实现类是DefaultAopProxyFactory

```
    /**
     * 真正的创建代理，判断一些列条件，有自定义的接口的就会创建jdk代理，否则就是cglib
     * @param config the AOP configuration in the form of an
     * AdvisedSupport object
     * @return
     * @throws AopConfigException
     */
    @Override
    public AopProxy createAopProxy(AdvisedSupport config) throws
AopConfigException {
        // 这段代码用来判断选择哪种创建代理对象的方式
        // config.isOptimize()   是否对代理类的生成使用策略优化 其作用是和
isProxyTargetClass是一样的 默认为false
        // config.isProxyTargetClass() 是否使用Cglib的方式创建代理对象 默认为false
        // hasNoUserSuppliedProxyInterfaces目标类是否有接口存在 且只有一个接口的时候接
口类型不是SpringProxy类型
        if (config.isOptimize() || config.isProxyTargetClass() ||
hasNoUserSuppliedProxyInterfaces(config)) {
            // 上面的三个方法有一个为true的话，则进入到这里
            // 从AdvisedSupport中获取目标类 类对象
            Class<?> targetClass = config.getTargetClass();
            if (targetClass == null) {
                throw new AopConfigException("TargetSource cannot determine
target class: " +
                        "Either an interface or a target is required for proxy
creation.");
            }
            // 判断目标类是否是接口 如果目标类是接口的话，则还是使用JDK的方式生成代理对象
            // 如果目标类是Proxy类型 则还是使用JDK的方式生成代理对象
            if (targetClass.isInterface() || Proxy.isProxyClass(targetClass)) {
                return new JdkDynamicAopProxy(config);
            }
            // 配置了使用Cglib进行动态代理或者目标类没有接口,那么使用Cglib的方式创建代理对
象
            return new ObjenesisCglibAopProxy(config);
        }
        else {
            // 使用JDK的提供的代理方式生成代理对象
            return new JdkDynamicAopProxy(config);
        }
    }
```

2.3 ProxyFactory

ProxyFactory代理对象的工厂类，用来创建代理对象的工厂。

然后我们来看看 ProxyFactory的体系结构



ProxyConfig

这个类主要保存代理的信息，如果是否使用类代理，是否要暴露代理等。

```
public class ProxyConfig implements Serializable {
```

```
    /** use serialVersionUID from Spring 1.2 for interoperability. */
    private static final long serialVersionUID = -8409359707199703185L;

    // 是否代理的对象是类，动态代理分为代理接口和类，这里的属性默认是代理的接口
    private boolean proxyTargetClass = false;
    // 是否进行主动优化，默认是不会主动优化
    private boolean optimize = false;
    // 是否由此配置创建的代理不能被转成Advised类型，默认时候可转
    boolean opaque = false;
    // 是否会暴露代理在调用的时候，默认是不会暴露
    boolean exposeProxy = false;
    // 是否冻结此配置，不能被修改
    private boolean frozen = false;

}
```

Advised

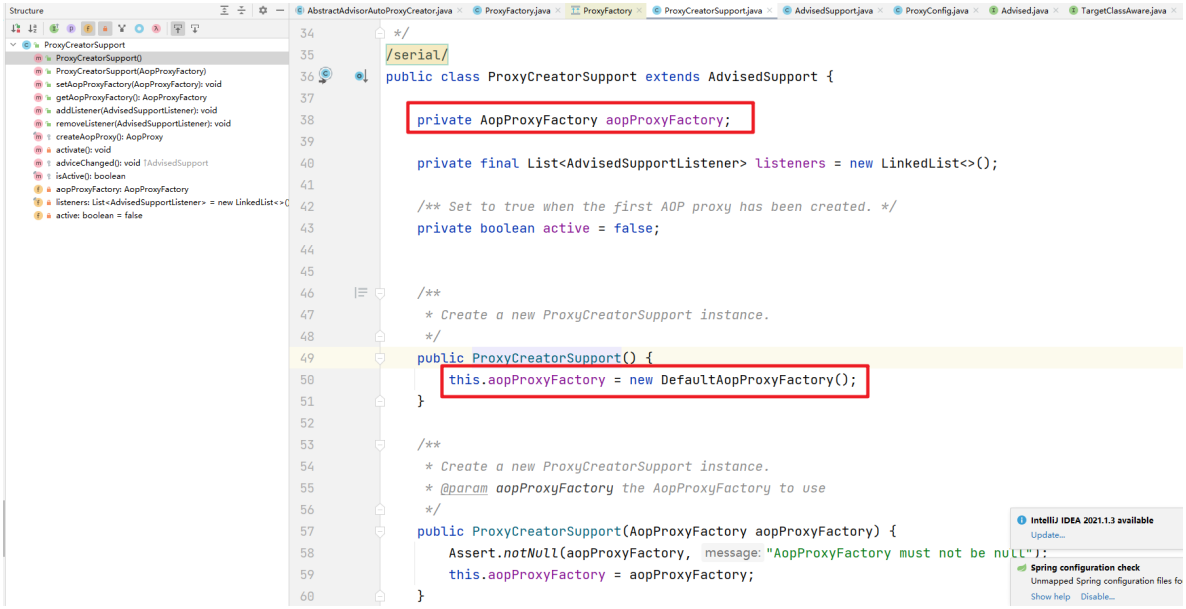
由持有 AOP 代理工厂配置的类实现的接口。此配置包括拦截器和其他advice、advisor和代理接口。从 Spring 获得的任何 AOP 代理都可以转换为该接口，以允许操作其 AOP 通知。

AdvisedSupport

- AOP代理配置管理器的基类。 此类的子类通常是工厂，从中可以直接获取 AOP 代理实例。此类可释放Advices和Advisor的内部管理子类，但实际上并没有实现代理创建方法，实现由子类提供
- AdvisedSupport实现了Advised中处理Advisor和Advice的方法，添加Advice时会被包装成一个Advisor，默认使用的Advisor是DefaultPointcutAdvisor，DefaultPointcutAdvisor默认的Pointcut是TruePointcut（转换为一个匹配所有方法调用的Advisor与代理对象绑定）。
- AdvisedSupport同时会缓存对于某一个方法对应的所有Advisor（Map<MethodCacheKey, List<Object>> methodCache），当Advice或Advisor发生变化时,会清空该缓存。getInterceptorsAndDynamicInterceptionAdvice用来获取对应代理方法对应有效的拦截器链 。

ProxyCreatorSupport

继承了AdvisedSupport,ProxyCreatorSupport正是实现代理的创建方法，ProxyCreatorSupport有一个成员变量AopProxyFactory，而该变量的值默认是DefaultAopProxyFactory



这个也就和前面的AopProxyFactory串联起来了。

3. @Aspect解析

然后我们分析下@Aspect注解的解析过程

```
    @Override
    protected boolean shouldSkip(Class<?> beanClass, String beanName) {
        // TODO: Consider optimization by caching the list of the aspect names
        List<Advisor> candidateAdvisors = findCandidateAdvisors();
        for (Advisor advisor : candidateAdvisors) {
            if (advisor instanceof AspectJPointcutAdvisor &&
                    ((AspectJPointcutAdvisor)
advisor).getAspectName().equals(beanName)) {
                return true;
            }
        }
        return super.shouldSkip(beanClass, beanName);
    }
```

先进入到shouldSkip方法。然后进入到 findCandidateAdvisors方法。

```
    /**
     * 查找通知器
     * @return
     */
    @Override
    protected List<Advisor> findCandidateAdvisors() {
        // Add all the Spring advisors found according to superclass rules.
        // 找到系统中实现了Advisor接口的bean
        List<Advisor> advisors = super.findCandidateAdvisors();
        // Build Advisors for all AspectJ aspects in the bean factory.
        if (this.aspectJAdvisorsBuilder != null) {
            // 找到系统中使用@Aspect标注的bean，并且找到该bean中使用@Before，@After等标
注的方法，
            // 将这些方法封装为一个个Advisor
            advisors.addAll(this.aspectJAdvisorsBuilder.buildAspectJAdvisors());
        }
        return advisors;
```

```
    }
```

在这个方法中就可以看到@Aspect 注解的处理了,进入到buildAspectJAdvisors方法

```
    public List<Advisor> buildAspectJAdvisors() {
        // 获取切面名字列表
        List<String> aspectNames = this.aspectBeanNames;

        // 缓存字段aspectNames没有值,注意实例化第一个单实例bean的时候就会触发解析切面
        if (aspectNames == null) {
            // 双重检查
            synchronized (this) {
                aspectNames = this.aspectBeanNames;
                if (aspectNames == null) {
                    // 用于保存所有解析出来的Advisors集合对象
                    List<Advisor> advisors = new ArrayList<>();
                    // 用于保存切面的名称的集合
                    aspectNames = new ArrayList<>();
                    /**
                     * AOP功能中在这里传入的是Object对象，代表去容器中获取到所有的组件的名
称，然后再
                     * 进行遍历，这个过程是十分的消耗性能的，所以说Spring会再这里加入了保
存切面信息的缓存。
                     * 但是事务功能不一样，事务模块的功能是直接去容器中获取Advisor类型的，
选择范围小，且不消耗性能。
                     * 所以Spring在事务模块中没有加入缓存来保存我们的事务相关的advisor
                     */
                    String[] beanNames =
BeanFactoryUtils.beanNamesForTypeIncludingAncestors(
                            this.beanFactory, Object.class, true, false);
                    // 遍历我们从IOC容器中获取处的所有Bean的名称
                    for (String beanName : beanNames) {
                        // 判断当前bean是否为子类定制的需要过滤的bean
                        if (!isEligibleBean(beanName)) {
                            continue;
                        }
                        // We must be careful not to instantiate beans eagerly
as in this case they
                        // would be cached by the Spring container but would not
have been weaved.
                        // 通过beanName去容器中获取到对应class对象
                        Class<?> beanType = this.beanFactory.getType(beanName,
false);
                        if (beanType == null) {
                            continue;
                        }
                        // 判断当前bean是否使用了@Aspect注解进行标注
                        if (this.advisorFactory.isAspect(beanType)) {
                            aspectNames.add(beanName);
                            // 对于使用了@Aspect注解标注的bean，将其封装为一个
AspectMetadata类型。
                            // 这里在封装的过程中会解析@Aspect注解上的参数指定的切面类
型，如perthis
                            // 和pertarget等。这些被解析的注解都会被封装到其
perClausePointcut属性中
                            AspectMetadata amd = new AspectMetadata(beanType,
beanName);
```

```
                            // 判断@Aspect注解中标注的是否为singleton类型，默认的切面类
都是singleton类型
                            if (amd.getAjType().getPerClause().getKind() ==
PerClauseKind.SINGLETON) {
                                // 将BeanFactory和当前bean封装为
MetadataAwareAspect-
                                // InstanceFactory对象，这里会再次将@Aspect注解中的参
数都封装
                                // 为一个AspectMetadata，并且保存在该factory中
                                MetadataAwareAspectInstanceFactory factory =
                                        new
BeanFactoryAspectInstanceFactory(this.beanFactory, beanName);
                                // 通过封装的bean获取其Advice，如@Before，@After等
等，并且将这些
                                // Advice都解析并且封装为一个个的Advisor
                                List<Advisor> classAdvisors =
this.advisorFactory.getAdvisors(factory);
                                // 如果切面类是singleton类型，则将解析得到的Advisor进
行缓存，
                                // 否则将当前的factory进行缓存，以便再次获取时可以通过
factory直接获取
                                if (this.beanFactory.isSingleton(beanName)) {
                                    this.advisorsCache.put(beanName,
classAdvisors);
                                }
                                else {
                                    this.aspectFactoryCache.put(beanName,
factory);
                                }
                                advisors.addAll(classAdvisors);
                            }
                            else {
                                // Per target or per this.
                                // 如果@Aspect注解标注的是perthis和pertarget类型，说
明当前切面
                                // 不可能是单例的，因而这里判断其如果是单例的则抛出异常
                                if (this.beanFactory.isSingleton(beanName)) {
                                    throw new IllegalArgumentException("Bean
with name '" + beanName +
                                            "' is a singleton, but aspect
instantiation model is not singleton");
                                }
                                // 将当前BeanFactory和切面bean封装为一个多例类型的
Factory
                                MetadataAwareAspectInstanceFactory factory =
                                        new
PrototypeAspectInstanceFactory(this.beanFactory, beanName);
                                // 对当前bean和factory进行缓存
                                this.aspectFactoryCache.put(beanName, factory);

advisors.addAll(this.advisorFactory.getAdvisors(factory));
                            }
                        }
                    }
                    this.aspectBeanNames = aspectNames;
                    return advisors;
                }
            }
```

```
        }

        if (aspectNames.isEmpty()) {
            return Collections.emptyList();
        }
        // 通过所有的aspectNames在缓存中获取切面对应的Advisor，这里如果是单例的，则直接从
advisorsCache
        // 获取，如果是多例类型的，则通过MetadataAwareAspectInstanceFactory立即生成一个
        List<Advisor> advisors = new ArrayList<>();
        for (String aspectName : aspectNames) {
            List<Advisor> cachedAdvisors = this.advisorsCache.get(aspectName);
            // 如果是单例的Advisor bean，则直接添加到返回值列表中
            if (cachedAdvisors != null) {
                advisors.addAll(cachedAdvisors);
            }
            else {
                // 如果是多例的Advisor bean，则通过
MetadataAwareAspectInstanceFactory生成
                MetadataAwareAspectInstanceFactory factory =
this.aspectFactoryCache.get(aspectName);
                advisors.addAll(this.advisorFactory.getAdvisors(factory));
            }
        }
        return advisors;
    }
```

然后我们需要看看 this.advisorFactory.getAdvisors(factory) 方法：完成 切入点表达式和对应Advice增强的方法绑定为Advisor。

```
    @Override
    public List<Advisor> getAdvisors(MetadataAwareAspectInstanceFactory
aspectInstanceFactory) {
        // 获取标记为AspectJ的类
        Class<?> aspectClass =
aspectInstanceFactory.getAspectMetadata().getAspectClass();
        // 获取标记为AspectJ的name
        String aspectName =
aspectInstanceFactory.getAspectMetadata().getAspectName();
        // 对当前切面bean进行校验，主要是判断其切点是否为perflow或者是percflowbelow，
Spring暂时不支持
        // 这两种类型的切点
        validate(aspectClass);

        // We need to wrap the MetadataAwareAspectInstanceFactory with a
decorator
        // so that it will only instantiate once.
        // 将当前aspectInstanceFactory进行封装，这里
LazySingletonAspectInstanceFactoryDecorator
        // 使用装饰器模式，主要是对获取到的切面实例进行了缓存，保证每次获取到的都是同一个切面
实例
        MetadataAwareAspectInstanceFactory lazySingletonAspectInstanceFactory =
                new
LazySingletonAspectInstanceFactoryDecorator(aspectInstanceFactory);

        List<Advisor> advisors = new ArrayList<>();
        // 这里getAdvisorMethods()会获取所有的没有使用@Pointcut注解标注的方法，然后对其
进行遍历
```

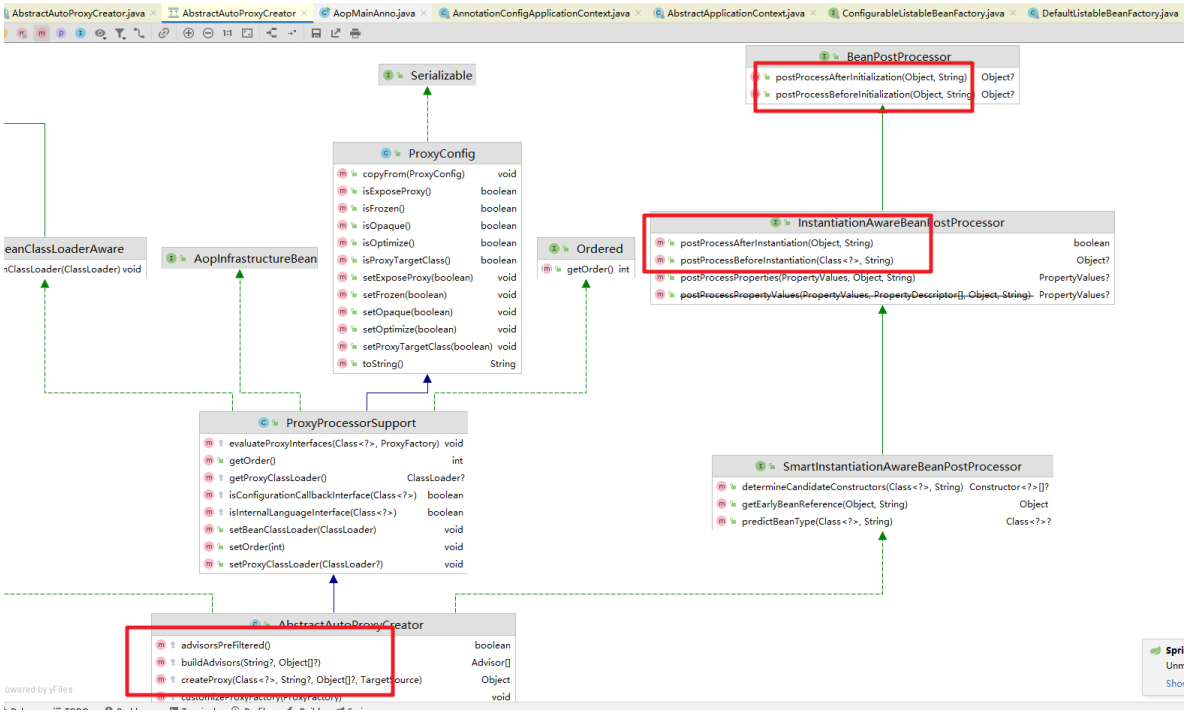
```
        for (Method method : getAdvisorMethods(aspectClass)) {
            // Prior to Spring Framework 5.2.7, advisors.size() was supplied as
the declarationOrderInAspect
            // to getAdvisor(...) to represent the "current position" in the
declared methods list.
            // However, since Java 7 the "current position" is not valid since
the JDK no longer
            // returns declared methods in the order in which they are declared
in the source code.
            // Thus, we now hard code the declarationOrderInAspect to 0 for all
advice methods
            // discovered via reflection in order to support reliable advice
ordering across JVM launches.
            // Specifically, a value of 0 aligns with the default value used in
            // AspectJPrecedenceComparator.getAspectDeclarationOrder(Advisor).
            // 判断当前方法是否标注有@Before，@After或@Around等注解，如果标注了，则将其封
装为一个Advisor
            Advisor advisor = getAdvisor(method,
lazySingletonAspectInstanceFactory, 0, aspectName);
            if (advisor != null) {
                advisors.add(advisor);
            }
        }

        // If it's a per target aspect, emit the dummy instantiating aspect.
        // 这里的isLazilyInstantiated()方法判断的是当前bean是否应该被延迟初始化，其主要是
判断当前
        // 切面类是否为perthis，pertarget或pertypewithiin等声明的切面。因为这些类型所环
绕的目标bean
        // 都是多例的，因而需要在运行时动态判断目标bean是否需要环绕当前的切面逻辑
        if (!advisors.isEmpty() &&
lazySingletonAspectInstanceFactory.getAspectMetadata().isLazilyInstantiated()) {
            // 如果Advisor不为空，并且是需要延迟初始化的bean，则在第0位位置添加一个同步增强
器，
            // 该同步增强器实际上就是一个BeforeAspect的Advisor
            Advisor instantiationAdvisor = new
SyntheticInstantiationAdvisor(lazySingletonAspectInstanceFactory);
            advisors.add(0, instantiationAdvisor);
        }

        // Find introduction fields.
        // 判断属性上是否包含有@DeclareParents注解标注的需要新添加的属性，如果有，则将其封
装为一个Advisor
        for (Field field : aspectClass.getDeclaredFields()) {
            Advisor advisor = getDeclareParentsAdvisor(field);
            if (advisor != null) {
                advisors.add(advisor);
            }
        }

        return advisors;
    }
```

然后我们需要关注的方法 根据对应的方法获取对应的Advisor 通知。

```
        MetadataAwareAspectInstanceFactory lazySingletonAspectInstanceFactory =
                new LazySingletonAspectInstanceFactoryDecorator(aspectInstanceFactory);

        List<Advisor> advisors = new ArrayList<>();
        // 这里getAdvisorMethods()会获取所有的没有使用@Pointcut注解标注的方法，然后对其进行遍历
        for (Method method : getAdvisorMethods(aspectClass)) {
            // Prior to Spring Framework 5.2.7, advisors.size() was supplied as the declarat
            // to getAdvisor(...) to represent the "current position" in the declared metho
            // However, since Java 7 the "current position" is not valid since the JDK no lo
            // returns declared methods in the order in which they are declared in the sourc
            // Thus, we now hard code the declarationOrderInAspect to 0 for all advice metho
            // discovered via reflection in order to support reliable advice ordering across
            // Specifically, a value of 0 aligns with the default value used in
            // AspectJPrecedenceComparator.getAspectDeclarationOrder(Advisor).
            // 判断当前方法是否标注有@Before，@After或@Around等注解，如果标注了，则将其封装为一个Advis
            Advisor advisor = getAdvisor(method, lazySingletonAspectInstanceFactory, declarati
            if (advisor != null) {
                advisors.add(advisor);
            }
        }
```

```
    @Nullable
    public Advisor getAdvisor(Method candidateAdviceMethod,
MetadataAwareAspectInstanceFactory aspectInstanceFactory,
            int declarationOrderInAspect, String aspectName) {
        // 校验当前切面类是否使用了perflow或者percflowbelow标识的切点，Spring暂不支持这两
种切点
        validate(aspectInstanceFactory.getAspectMetadata().getAspectClass());

        // 获取当前方法中@Before，@After或者@Around等标注的注解，并且获取该注解的值，将其
        // 封装为一个AspectJExpressionPointcut对象
        AspectJExpressionPointcut expressionPointcut = getPointcut(
                candidateAdviceMethod,
aspectInstanceFactory.getAspectMetadata().getAspectClass());
        if (expressionPointcut == null) {
            return null;
        }

        // 将获取到的切点，切点方法等信息封装为一个Advisor对象，也就是说当前Advisor包含有所
有
        // 当前切面进行环绕所需要的信息
        return new
InstantiationModelAwarePointcutAdvisorImpl(expressionPointcut,
candidateAdviceMethod,
                this, aspectInstanceFactory, declarationOrderInAspect,
aspectName);
    }
```

3. 多个切面的责任链实现



代理方法

```
    public Object getProxy() { return getProxy(ClassUtils.getDefaultClassLoader()); }

    @Override
    public Object getProxy(@Nullable ClassLoader classLoader) {
        if (logger.isTraceEnabled()) {
            logger.trace( o: "Creating JDK dynamic proxy: " + this.advised.getTargetSource());
        }
        Class<?>[] proxiedInterfaces = AopProxyUtils.completeProxiedInterfaces(this.advised,  decoratingProxy
        findDefinedEqualsAndHashCodeMethods(proxiedInterfaces);
        return Proxy.newProxyInstance(classLoader, proxiedInterfaces,  h: this);
    }
```

返回代理对象，执行的时候调用 invoke 方法

```
    Finds any equals or hashCode method that may be defined on the supplied set of interfaces.
    Params: proxiedInterfaces – the interfaces to introspect
    private void findDefinedEqualsAndHashCodeMethods(Class<?>[] proxiedInterfaces) {
        for (Class<?> proxiedInterface : proxiedInterfaces) {
            Method[] methods = proxiedInterface.getDeclaredMethods();
            for (Method method : methods) {
                if (AopUtils.isEqualsMethod(method)) {
                    this.equalsDefined = true;
                }
                if (AopUtils.isHashCodeMethod(method)) {
                    this.hashCodeDefined = true;
                }
                if (this.equalsDefined && this.hashCodeDefined) {
                    return;
                }
```

invoke方法的处理

```
    public Object invoke(Object proxy, Method method, Object[] args) throws
Throwable {
        Object oldProxy = null;
        boolean setProxyContext = false;

        // 获取到我们的目标对象
        TargetSource targetSource = this.advised.targetSource;
        Object target = null;

        try {
            // 若是equals方法不需要代理
```

```
            if (!this.equalsDefined && AopUtils.isEqualsMethod(method)) {
                // The target does not implement the equals(Object) method
itself.
                return equals(args[0]);
            }
            // 若是hashCode方法不需要代理
            else if (!this.hashCodeDefined && AopUtils.isHashCodeMethod(method))
{
                // The target does not implement the hashCode() method itself.
                return hashCode();
            }
            // 若是DecoratingProxy也不要拦截器执行
            else if (method.getDeclaringClass() == DecoratingProxy.class) {
                // There is only getDecoratedClass() declared -> dispatch to
proxy config.
                return AopProxyUtils.ultimateTargetClass(this.advised);
            }
            // isAssignableFrom方法：如果调用这个方法的class或接口与参数cls表示的类或接
口相同，或者是参数cls表示的类或接口的父类，则返回true
            else if (!this.advised.opaque &&
method.getDeclaringClass().isInterface() &&
                    method.getDeclaringClass().isAssignableFrom(Advised.class))
{
                // Service invocations on ProxyConfig with the proxy config...
                return AopUtils.invokeJoinpointUsingReflection(this.advised,
method, args);
            }

            Object retVal;

            /**
             * 这个配置是暴露我们的代理对象到线程变量中，需要搭配
@EnableAspectJAutoProxy(exposeProxy = true)一起使用
             * 比如在目标对象方法中再次获取代理对象可以使用这个AopContext.currentProxy()
             * 还有的就是事务方法调用事务方法的时候也是用到这个
             */
            if (this.advised.exposeProxy) {
                // Make invocation available if necessary.
                // 把我们的代理对象暴露到线程变量中
                oldProxy = AopContext.setCurrentProxy(proxy);
                setProxyContext = true;
            }

            // Get as late as possible to minimize the time we "own" the target,
            // in case it comes from a pool.
            // 获取我们的目标对象
            target = targetSource.getTarget();
            // 获取我们目标对象的class
            Class<?> targetClass = (target != null ? target.getClass() : null);

            // Get the interception chain for this method.
            // 从Advised中根据方法名和目标类获取AOP拦截器执行链
            List<Object> chain =
this.advised.getInterceptorsAndDynamicInterceptionAdvice(method, targetClass);

            // Check whether we have any advice. If we don't, we can fallback on
direct
```

```
            // reflective invocation of the target, and avoid creating a
MethodInvocation.
            // 如果拦截器链为空
            if (chain.isEmpty()) {
                // We can skip creating a MethodInvocation: just invoke the
target directly
                // Note that the final invoker must be an InvokerInterceptor so
we know it does
                // nothing but a reflective operation on the target, and no hot
swapping or fancy proxying.
                // 通过反射直接调用执行
                Object[] argsToUse =
AopProxyUtils.adaptArgumentsIfNecessary(method, args);
                // 如果没有发现任何拦截器那么直接调用切点方法
                retVal = AopUtils.invokeJoinpointUsingReflection(target, method,
argsToUse);
            }
            else {
                // We need to create a method invocation...
                // 将拦截器封装在ReflectiveMethodInvocation，以便于使用其proceed进行处
理
                MethodInvocation invocation =
                        new ReflectiveMethodInvocation(proxy, target, method,
args, targetClass, chain);
                // Proceed to the joinpoint through the interceptor chain.
                // 执行拦截器链
                retVal = invocation.proceed();
            }

            // Massage return value if necessary.
            // 获取返回类型
            Class<?> returnType = method.getReturnType();
            if (retVal != null && retVal == target &&
                    returnType != Object.class && returnType.isInstance(proxy)
&&

!RawTargetAccess.class.isAssignableFrom(method.getDeclaringClass())) {
                // Special case: it returned "this" and the return type of the
method
                // is type-compatible. Note that we can't help if the target
sets
                // a reference to itself in another returned object.
                retVal = proxy;
            }
            // 返回值类型错误
            else if (retVal == null && returnType != Void.TYPE &&
returnType.isPrimitive()) {
                throw new AopInvocationException(
                        "Null return value from advice does not match primitive
return type for: " + method);
            }
            return retVal;
        }
        finally {
            // 如果目标对象不为空且目标对象是可变的,如prototype类型
            // 通常我们的目标对象都是单例的,即targetSource.isStatic为true
            if (target != null && !targetSource.isStatic()) {
                // Must have come from TargetSource.
```

```
                // 释放目标对象
                targetSource.releaseTarget(target);
            }
            if (setProxyContext) {
                // Restore old proxy.
                // 线程上下文复位
                AopContext.setCurrentProxy(oldProxy);
            }
        }
    }
```

proceed方法

```
    /**
     * 递归获取通知，然后执行
     * @return
     * @throws Throwable
     */
    @Override
    @Nullable
    public Object proceed() throws Throwable {
        // We start with an index of -1 and increment early.
        // 从索引为-1的拦截器开始调用，并按序递增，如果拦截器链中的拦截器迭代调用完毕，开始调
用target的函数，这个函数是通过反射机制完成的
        // 具体实现在AopUtils.invokeJoinpointUsingReflection方法中
        if (this.currentInterceptorIndex ==
this.interceptorsAndDynamicMethodMatchers.size() - 1) {
            return invokeJoinpoint();
        }

        // 获取下一个要执行的拦截器，沿着定义好的interceptorOrInterceptionAdvice链进行处
理
        Object interceptorOrInterceptionAdvice =

this.interceptorsAndDynamicMethodMatchers.get(++this.currentInterceptorIndex);
        if (interceptorOrInterceptionAdvice instanceof
InterceptorAndDynamicMethodMatcher) {
            // Evaluate dynamic method matcher here: static part will already
have
            // been evaluated and found to match.
            // 这里对拦截器进行动态匹配的判断，这里是对pointcut触发进行匹配的地方，如果和定
义的pointcut匹配，那么这个advice将会得到执行
            InterceptorAndDynamicMethodMatcher dm =
                    (InterceptorAndDynamicMethodMatcher)
interceptorOrInterceptionAdvice;
            Class<?> targetClass = (this.targetClass != null ? this.targetClass
: this.method.getDeclaringClass());
            if (dm.methodMatcher.matches(this.method, targetClass,
this.arguments)) {
                return dm.interceptor.invoke(this);
            }
            else {
                // Dynamic matching failed.
                // Skip this interceptor and invoke the next in the chain.
                // 如果不匹配，那么proceed会被递归调用，知道所有的拦截器都被运行过位置
                return proceed();
            }
```

```
        }
        else {
            // It's an interceptor, so we just invoke it: The pointcut will have
            // been evaluated statically before this object was constructed.
            // 普通拦截器，直接调用拦截器，将this作为参数传递以保证当前实例中调用链的执行
            return ((MethodInterceptor)
interceptorOrInterceptionAdvice).invoke(this);
        }
    }
```