spring aop 面向切面编程

时间：2019-4-23

author: 陈秀林，衣玉鑫

1.spring AOP介绍

- 参考链接：https://docs.spring.io/spring/docs/2.5.x/reference/aop.html AOP（面向切面编程）对OOP（面向对象编程）是一种补充，它提供了另一种程序结构的思路。OOP的模块单元是class，而AOP的模块单元是aspect。Spring中一个关键的组件是AOP框架，然而，Spring IoC容器并不依赖于AOP，也就是说如果你不想用AOP的话可以不用。 在Spring框架中AOP主要作用：
- 提供声明式的企业服务，特别是代替EJB的声明式服务，最重要的服务是声明式事物管理；
- 允许用户实现自定义式的aspect

2.AOP基础

2.1AOP概念

- Aspect（切面）：横切多个class的一个关注点的模块化。事务管理是一个很好的例子。在Spring AOP中，aspect可以用普通类或者带有@Aspect注解的普通类来实现。

```
@Aspect
@Component
public class LogProxy {
    ...
}
```

- Join point（连接点）：程序执行期间的一个点，比如方法的执行或者异常的处理。在Spring AOP中，一个连接点总是代表一个方法执行。

```
public interface TestAop {
    int add(int num1, int num2);// 连接点

    int delete(int num1, int num2);//连接点

    int dev(int num1, int num2);//连接点
}
```

- Weaving(织入)：织入是指将切面代码插入到目标对象的过程。代理的invoke方法完成的工作，可以称为织入，Spring和其他纯Java AOP框架一样，在运行时完成织入。

- Introduction（引入）：用来给一个类型声明额外的方法或属性（也被称为连接类型声明（inter-type declaration））。Spring允许引入新的接口（以及一个对应的实现）到任何被代理的对象。例如，你可以使用引入来使一个bean实现IsModified接口，以便简化缓存机制。
- Advice（通知）：一个aspect在一个特定的连接点所采取的动作。它是某个连接点所采用的处理逻辑，也就是向连接点注入的代码，比如输出的日志信息。知的类型包括"around"、"before"、"after"。许多AOP框架，包括Spring也是，它们把一个通知作为一个拦截器，维护一个拦截器链环绕在连接点上。
- Pointcut（切入点）：匹配连接点的一个谓词。Advice关联一个切点表达式。Spring默认用AspectJ切点表达式。
-

  支持的切入点指示符

  Spring AOP支持在切入点表达式中使用如下的AspectJ切入点指示符：

  Spring AOP支持的切入点指示符可能会在将来的版本中得到扩展，从而支持更多的AspectJ切入点指示符。

  - *execution* - 匹配方法执行的连接点，这是Spring的最主要的切入点指示符。

    执行表达式的格式如下：

    ```
    execution (modifiers-pattern? ret-type-pattern declaring-type-pattern? name-pattern (param-pattern)  throws-pattern?)
    ```

    这里问号表示当前项可以有也可以没有，其中各项的语义如下：

    - modifiers-pattern：方法的可见性，如public，protected；
    - ret-type-pattern：方法的返回值类型，如int，void等；
    - declaring-type-pattern：方法所在类的全路径名，如com.spring.Aspect；
    - name-pattern：方法名类型，如buisinessService()；
    - param-pattern：方法的参数类型，如java.lang.String；
    - throws-pattern：方法抛出的异常类型，如java.lang.Exception；

      参数模式稍微有点复杂：`()`匹配了一个不接受任何参数的方法， 而`(..)`匹配了一个接受任意数量参数的方法（零或者更多）。 模式`(*)`匹配了一个接受一个任何类型的参数的方法。

      在类名模式串中，“.”*表示包下的所有类，而*“..”表示包、子孙包下的所有类。

    - 任意公共方法的执行：

      ```
      execution (public * * (..) )
      ```

    - 任何一个名字以“set”开始的方法的执行：

      ```
      execution (* set* (..) )
      ```

    - `AccountService`接口定义的任意方法的执行：

      ```
      execution (* com.xyz.service.AccountService.* (..) )
      ```

    - 在service包中定义的任意方法的执行：

```
execution (* com.xyz.service.*.* (..) )
```

- 在service包或其子包中定义的任意方法的执行：

```
execution (* com.xyz.service..*.* (..) )
```

- 在service包中的任意连接点（在Spring AOP中只是方法执行）：

```
within (com.xyz.service.*)
```

- 在service包或其子包中的任意连接点（在Spring AOP中只是方法执行）：

```
within (com.xyz.service..*)
```

- 实现了`AccountService`接口的代理对象的任意连接点 （在Spring AOP中只是方法执行）：

```
this (com.xyz.service.AccountService)
```

- 实现`AccountService`接口的目标对象的任意连接点 （在Spring AOP中只是方法执行）：

```
target (com.xyz.service.AccountService)
```

- *within* - 限定匹配特定类型的连接点（在使用Spring AOP的时候，在匹配的类型中定义的方法的执行）。

  within表达式的粒度为类，其参数为全路径的类名（可使用通配符），表示匹配当前表达式的所有类都将被当前方法环绕。如下是within表达式的语法：

```
within(declaring-type-pattern)
例如：
within(com.spring.service.*)
```

- *this* - 限定匹配特定的连接点（使用Spring AOP的时候方法的执行），其中bean reference（Spring AOP代理）是指定类型的实例。
- *target* - 限定匹配特定的连接点（使用Spring AOP的时候方法的执行），其中目标对象（被代理的应用对象）是指定类型的实例。
- *args* - 限定匹配特定的连接点（使用Spring AOP的时候方法的执行），其中参数是指定类型的实例。

```
args(param-pattern)
```

- *@target* - 限定匹配特定的连接点（使用Spring AOP的时候方法的执行），其中正执行对象的类持有指定类型的注解。

  目标对象具有`@Transactional`注释的任何连接点（仅在Spring AOP中执行方法）：

```
@target (org.springframework.transaction.annotation.Transactional)
```

- *@args* - 限定匹配特定的连接点（使用Spring AOP的时候方法的执行），其中实际传入参数的运行时类型持有指定类型的注解。
- *@within* - 限定匹配特定的连接点，其中连接点所在类型已指定注解（在使用Spring AOP的时候，所执行的方法所在类型已指定注解）。
- *@annotation* - 限定匹配特定的连接点（使用Spring AOP的时候方法的执行），其中连接点的主题持有指定的注解。

  @within和@annotation分别表示匹配使用指定注解标注的类和标注的方法将会被匹配，@args则表示使用指定注解标注的类作为某个方法的参数时该方法将会被匹配。如下是@args注解的语法：

  ```
  @args(annotation-type)
  ```

  如下示例表示匹配使用了com.spring.annotation.FruitAspect注解标注的类作为参数的方法：

  ```
  @args(com.spring.annotation.FruitAspect)
  ```

```
@Pointcut("@annotation(com.iflytek.stp.speech.common.log.Logging)")
public void loggingPointCut() {
}
```

- Target object（目标对象）：被一个或多个aspect通知的对象。在Spring AOP中是用运行时代理来实现的，因此这个对象总是一个被代理对象。
- AOP proxy（AOP代理）：为了实现aspect而被AOP框架所创建的一个对象。在Spring框架中，一个AOP proxy可能是一个JDK动态代理或者一个CGLIB代理。
- 具体的详见连接http://shouce.jb51.net/spring/aop.html

关于join point 和 point cut 的区别 在 Spring AOP 中, 所有的方法执行都是 join point. 而 point cut 是一个描述信息, 它修饰的是 join point, 通过 point cut, 我们就可以确定哪些 join point 可以被织入 Advice. 因此 join point 和 point cut 本质上就是两个不同纬度上的东西. advice 是在 join point 上执行的, 而 point cut 规定了哪些 join point 可以执行哪些 advice

2.2通知的类型

- Before advice（前置通知）：在一个连接点之前执行的通知，但是没有能力阻止后面的执行（除非它抛异常）
- After returning advice（返回通知）：在一个连接点正常执行完以后执行的通知：例如，在一个不抛异常的方法返回时执行
- After throwing advice（异常通知）：如果方法因为抛出异常而退出了才会执行的通知

  AfterThrowing Advice对应的是切入点方法执行对外抛出异常的拦截。因为当一个切入点方法可以同时被Around Advice和AfterThrowing Advice拦截时，实际上AfterThrowing Advice拦截的是Around Advice处理后的结果，所以这种情况下最终AfterThrowing Advice是否能被触发，还要看Around Advice自身是否对外抛出异常，即算是目标方法对外抛出了异常，但是被Around Advice处理了又没有向外抛出异常的时候AfterThrowing Advice也不会被触发的。如果希望在AfterThrowing Advice处理方法中获取到被抛出的异常，可以给对应的Advice处理方法加一个Exception或其子类型（能确定抛出的异常类型）的方法参数，然后通

  @AfterThrowing的throwing属性指定拦截到的异常对象对应的Advice处理方法的哪个参数。如下就指定了拦截到的异常对象将传递给Advice处理方法的ex参数。

```
@AfterThrowing(value="bean(userService)", throwing="ex")
public void afterThrowing(Exception ex) {
    System.out.println("-----after throwing with pointcut expression:
bean(userService)------" + ex);
}
```

AfterThrowing是用于在切入点方法抛出异常时进行某些特殊的处理，但是它不会阻止方法调用者看到异常结果。

- After (finally) advice（后置通知）：无论连接点正常退出还是异常退出都会执行
- Around advice（环绕通知）：环绕一个连接点比如方法调用的通知。这是最强的一种通知。环绕通知可以在方法调用之前或之后执行自定义的行为。它也负责选择是否处理连接点方法执行，通过返回一个它自己的返回或者抛出异常。环绕通知是用得最普遍的一种通知。

总结：

当方法符合切点规则不符合环绕通知的规则时候，执行的顺序如下

@Before→@After→@AfterRunning(如果有异常→@AfterThrowing)

当方法符合切点规则并且符合环绕通知的规则时候，执行的顺序如下

@Around→@Before→@After→@Around执行 ProceedingJoinPoint.proceed() 之后的操作→@AfterRunning(如果有异常→@AfterThrowing)

2.3通知参数

Spring提供完全类型的通知 - 意味着您在通知签名中声明了所需的参数（正如我们在上面看到的返回和抛出示例所示），而不是一直使用`Object[]`数组。我们将看到如何在一瞬间为通知主体提供参数和其他上下文值。首先让我们来看看如何编写通用通知。

- 访问当前的JoinPoint

  任何通知方法都可以声明为它的第一个参数，类型的参数`org.aspectj.lang.JoinPoint`（请注意，周围的建议*需要*声明类型的第一个参数`ProceedingJoinPoint`，它是一个子类`JoinPoint`。

```
Caused by: java.lang.IllegalArgumentException: ProceedingJoinPoint is only supported for
around advice
```

```
public interface JoinPoint {
    Object getThis();
    Object getTarget();
    Object[] getArgs();
    /**
     * 方法描述
     */
    Signature getSignature();
    ...
}
```

2.4Aop代理

Spring AOP默认使用AOP代理的标准JDK *动态*代理。这使得任何接口（或接口集）都可以被代理。

Spring AOP也可以使用CGLIB代理。这是代理类而不是接口所必需的。如果业务对象未实现接口，则默认使用CGLIB。因为优良的做法是编程接口而不是类; 业务类通常会实现一个或多个业务接口。可以 强制使用CGLIB，在那些需要建议未在接口上声明的方法，或者需要将代理对象作为具体类型传递给方法的情况下。

掌握Spring AOP是*基于代理的*这一事实非常重要。

局限性：

1）切入点只能在方法级别，无法对一些字段进行拦截；

2）springAop只能拦截spring管理的bean对象，不能拦截例如domain等对象；

3）jdk代理和cglib自身的局限性，jdk代理必须实现接口；Cglib原理是针对目标类生成一个子类，覆盖其中的所有方法，所以目标类和方法不能声明为final类型。

2.5spring AOP和AspectJ之间的差别

- AspectJ是一个比较牛逼的AOP框架，他可以对类的成员变量，方法进行拦截。由于 AspectJ 是 Java 语言语法和语义的扩展，所以它提供了自己的一套处理方面的关键字。除了包含字段和方法之外，AspectJ 的方面声明还包含切入点和通知成员。
- Spring AOP依赖的是 Spring 框架方便的、最小化的运行时配置，所以不需要独立的启动器。但是，使用这个技术，只能通知从 Spring 框架检索出的对象。Spring的AOP技术只能是对方法进行拦截。
- 在spring AOP中我们同样也可以使用类似AspectJ的注解来实现AOP功能，但是这里要注意一下，使AspectJ的注解时，AOP的实现方式还是Spring AOP。Spring缺省使用J2SE动态代理来作为AOP的代理，这样任何接口都可以被代理，Spring也可以使用CGLIB代理，对于需要代理类而不是代理接口的时候CGLIB是很有必要的。如果一个业务对象没有实现接口，默认就会使用CGLIB代理。

3.spring AOP的流程

spring AOP的流程如下：

- 1.Spring加载自动代理AnnotationAwareAspectJAutoProxyCreator，当作一个系统组件
- 2.当一个bean加载到Spring中时，会触发自动代理器中的bean后置处理
- 3.bean后置处理，会先扫描bean中所有的Advisor
- 4.然后用这些Adviosr和其他参数构建ProxyFactory
- 5.ProxyFactory会根据配置和目标对象的类型寻找代理的方式（JDK动态代理或CGLIG代理）
- 6.然后代理出来的对象放回context中，完成Spring AOP代理

spring aop实现大致过程，链接：https://www.processon.com/view/5979d599e4b06b35d2f9a65a

4.@AspectJ支持-源码详解

4.1回顾后置处理器的注册过程

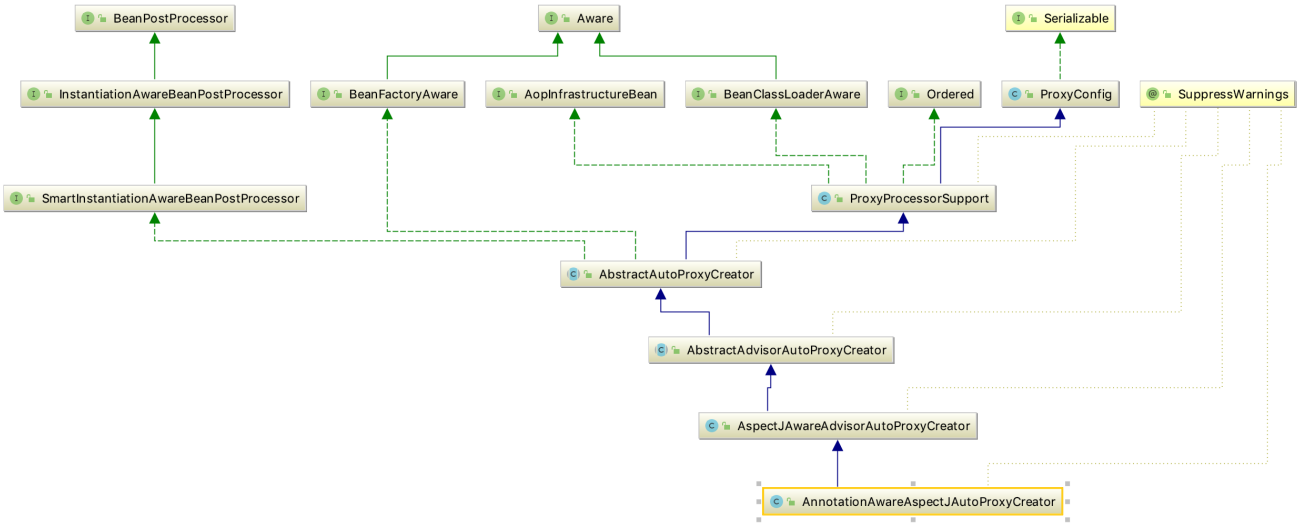
AnnotationAwareAspectJAutoProxyCreator 继承了 AbstractAutoProxyCreator 抽象类，该类实现了后置处理器接口的方法，Spring 容器初始化的方法 refresh，所有的逻辑都是从这里作为入口：

```
@Override
```

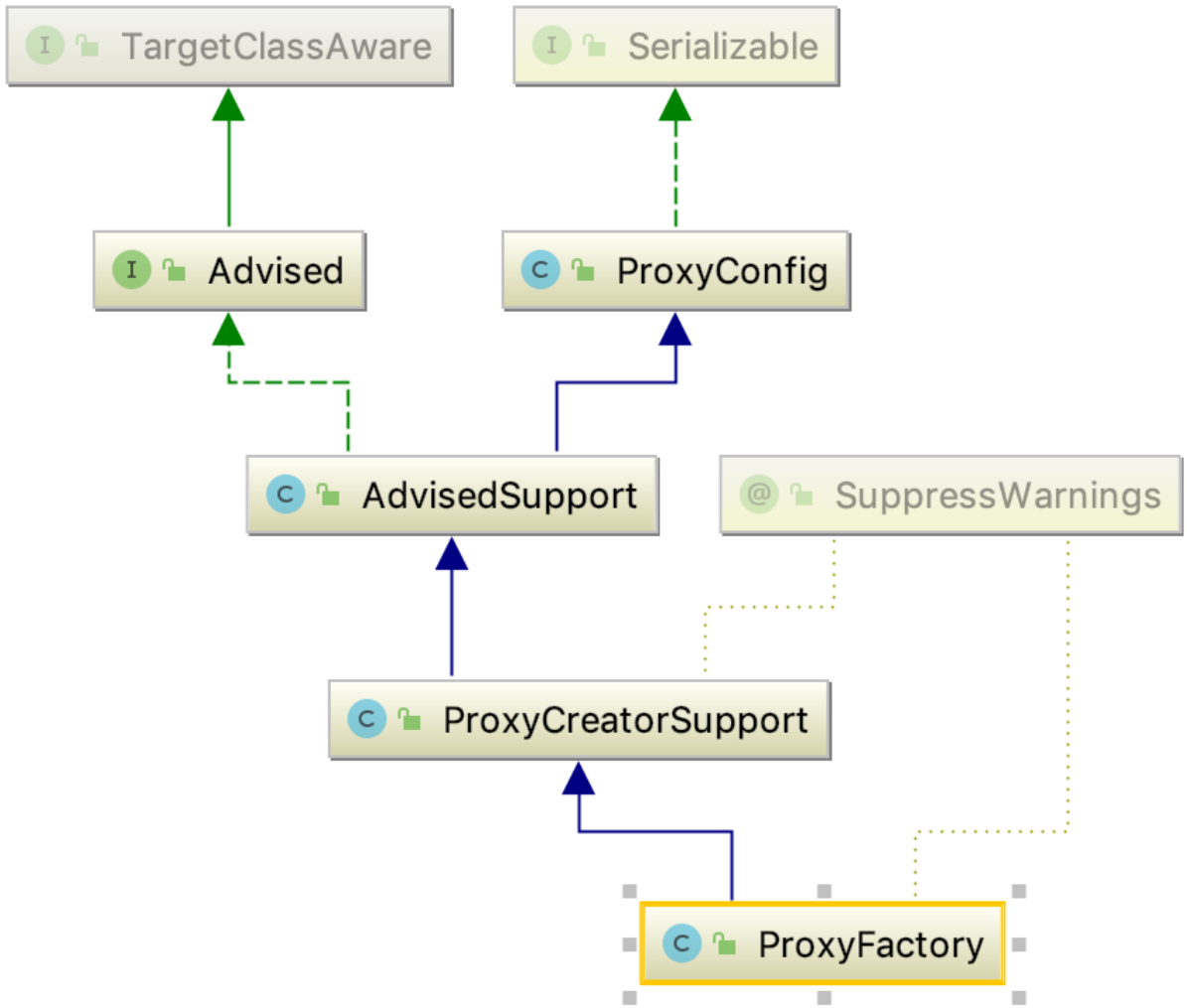
```
    public void refresh() throws BeansException, IllegalStateException {
        synchronized (this.startupShutdownMonitor) {
            // 为刷新准备应用上下文
            prepareRefresh();
            // 告诉子类刷新内部bean工厂，即在子类中启动refreshBeanFactory()的地方----创建bean工厂，
根据配置文件生成bean定义
            ConfigurableListableBeanFactory beanFactory = obtainFreshBeanFactory();
            // 在这个上下文中使用bean工厂
            prepareBeanFactory(beanFactory);
            try {
                // 设置BeanFactory的后置处理器// 默认什么都不做
                postProcessBeanFactory(beanFactory);
                // 调用BeanFactory的后处理器，这些后处理器是在Bean定义中向容器注册的
                invokeBeanFactoryPostProcessors(beanFactory);
                // 注册Bean的后处理器，在Bean创建过程中调用
                registerBeanPostProcessors(beanFactory);
                //对上下文的消息源进行初始化
                initMessageSource();
                // 初始化上下文中的事件机制
                initApplicationEventMulticaster();
                // 初始化其他的特殊Bean
                onRefresh();
                // 检查监听Bean并且将这些Bean向容器注册
                registerListeners();
                // 实例化所有的（non-lazy-init）单件
                finishBeanFactoryInitialization(beanFactory);
                //  发布容器事件，结束refresh过程
                finishRefresh();
            } catch (BeansException ex) {
                // 为防止bean资源占用，在异常处理中，销毁已经在前面过程中生成的单件bean
                destroyBeans();
                // 重置“active”标志
                cancelRefresh(ex);
                // Propagate exception to caller.
                throw ex;
            } finally {
                resetCommonCaches();
            }
        }
    }
```

4.2重要组建类结构

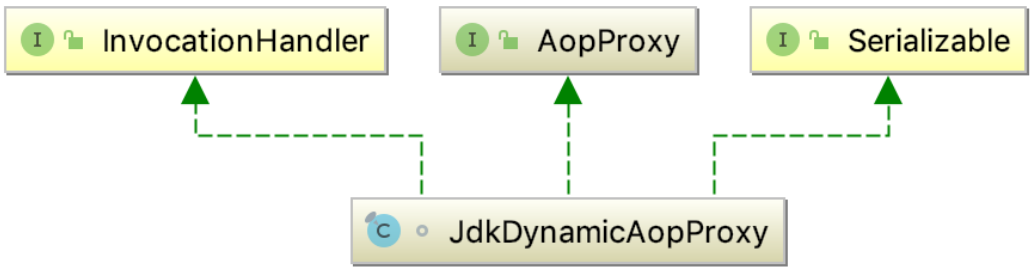
AnnotationAwareAspectJAutoProxyCreator



ProxyFactory



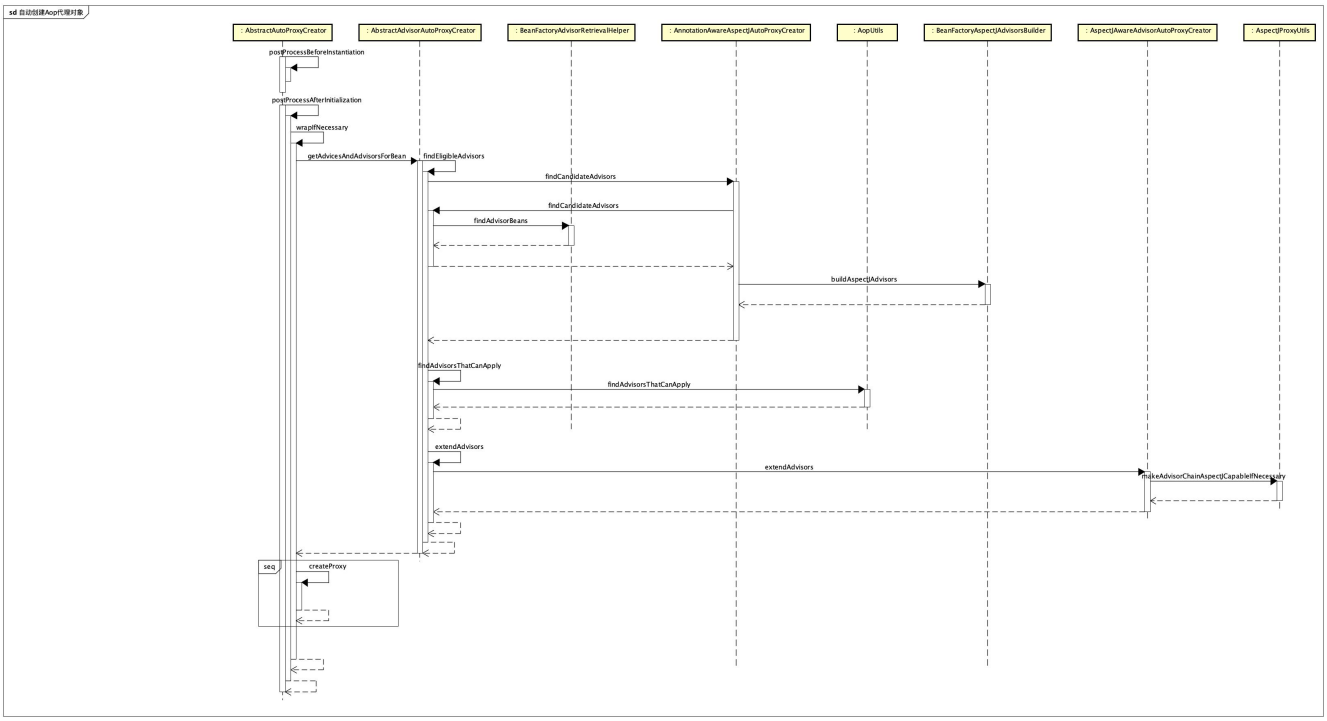
JdkDynamicAopProxy

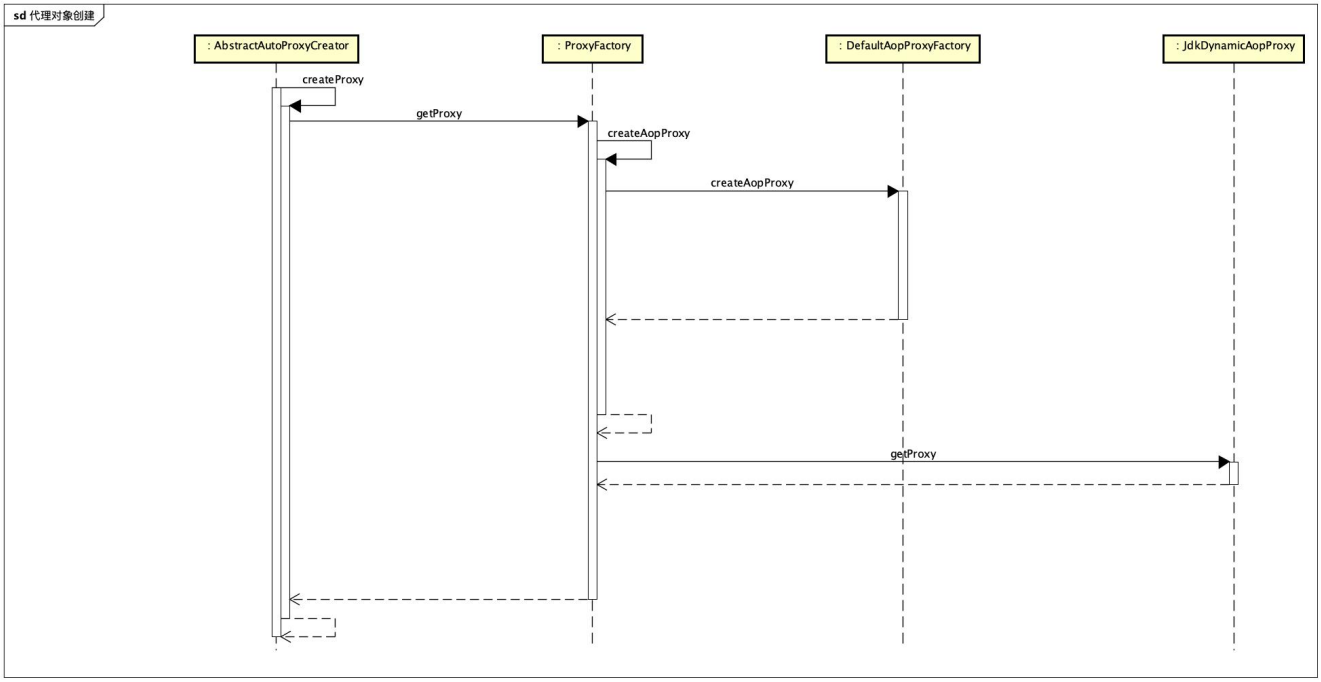


4.3 重要流程讲解

自动代理对象的创建

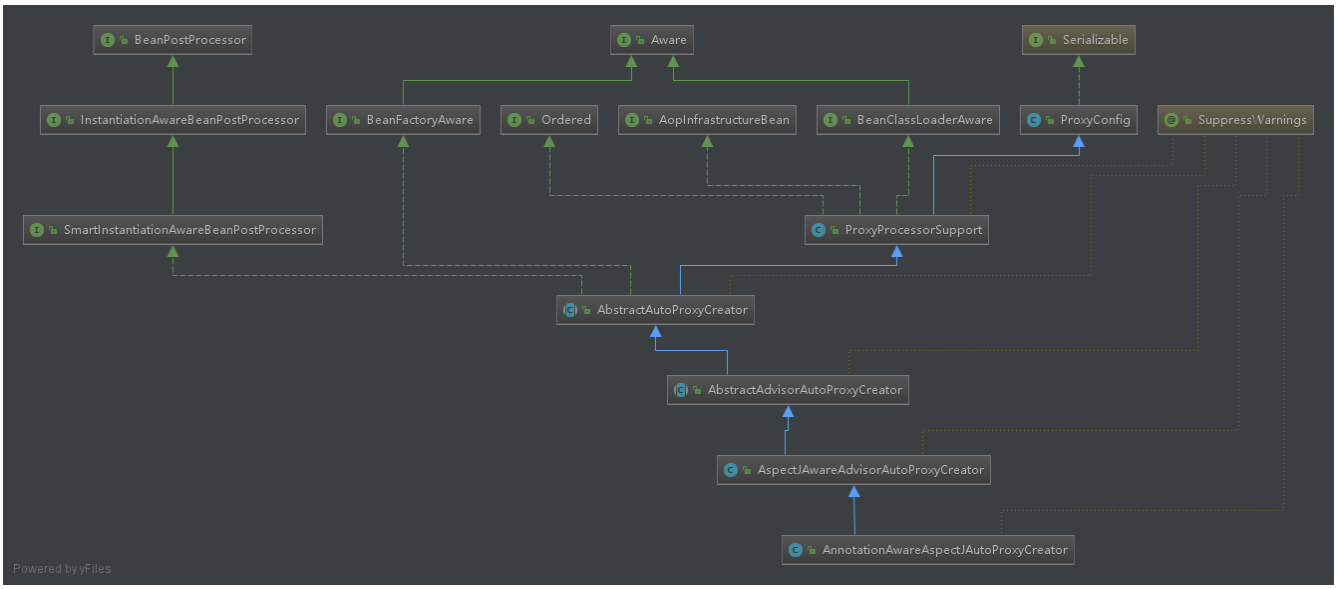
时序图





4.4创建AOP代理具体流程

AnnotationAwareAspectJAutoProxyCreator实现了BeanPostProcessor接口，当spring加载这个bean时会在实例化前调用其AbstractAutoProxyCreator的postProcessAfterInitialization：



代码如下：

```
public Object postProcessAfterInitialization(Object bean, String beanName) throws
BeansException {
        if (bean != null) {
            ////根据给定的class和name构建出key
            // 看缓存中是否以及包含该bean和name
            Object cacheKey = getCacheKey(bean.getClass(), beanName);
            if (!this.earlyProxyReferences.contains(cacheKey)) {
                // 如果需要，为bean生成代理对象
                return wrapIfNecessary(bean, beanName, cacheKey);
            }
        }
        return bean;
    }
```

```
protected Object wrapIfNecessary(Object bean, String beanName, Object cacheKey) {
        // 如果已经处理过则直接返回
        if (beanName != null && this.targetSourcedBeans.contains(beanName)) {
            return bean;
        }
        // 无需增强，直接返回
        //postProcessBeforeInstantiation方法已经将排除的bean放置到advisedBeans
        if (Boolean.FALSE.equals(this.advisedBeans.get(cacheKey))) {
            return bean;
        }
        /**
         * 如果是基础设施类（Pointcut、Advice、Advisor等接口的实现类），或者是应该
         * 跳过的类，则不应该生成代理，此时直接返回bean
         */
        if (isInfrastructureClass(bean.getClass()) || shouldSkip(bean.getClass(),
beanName)) {
            this.advisedBeans.put(cacheKey, Boolean.FALSE);
            return bean;
        }

        //为bean查找合适的通知器，如果存在增强方法则创建代理
        // Create proxy if we have advice.
        Object[] specificInterceptors = getAdvicesAndAdvisorsForBean(bean.getClass(),
beanName, null);
        // 如果获取到了增强则需要针对增强创建代理
        if (specificInterceptors != DO_NOT_PROXY) {
            this.advisedBeans.put(cacheKey, Boolean.TRUE);
            // 创建代理
            Object proxy = createProxy(
                    bean.getClass(), beanName, specificInterceptors, new
SingletonTargetSource(bean));
            this.proxyTypes.put(cacheKey, proxy.getClass());
            /**
             * 返回代理对象，此时IOC容器时输入bean，得到proxy,此时
             * beanName对应的bean是代理对象，而非原始的bean
             */
            return proxy;
        }
```

```
        this.advisedBeans.put(cacheKey, Boolean.FALSE);
        return bean;
    }
```

- 创建代理主要包含了两个步骤：

(1) 获取增强方法或者增强器

(2) 根据获取的增强进行代理

```
protected Object[] getAdvicesAndAdvisorsForBean(Class<?> beanClass, String beanName,
TargetSource targetSource) {
   // 查找合适的通知器
   List<Advisor> advisors = findEligibleAdvisors(beanClass, beanName);
   if (advisors.isEmpty()) {
      return DO_NOT_PROXY;
   }
   return advisors.toArray();
}
```

```
protected List<Advisor> findEligibleAdvisors(Class<?> beanClass, String beanName) {
        //查找所有的通知器
        List<Advisor> candidateAdvisors = findCandidateAdvisors();
        /**
         * 筛选可应用在beanClass上的advisor，通过ClassFilter和MethodMatch
         * 对目标类和方法进行匹配
         */
        List<Advisor> eligibleAdvisors = findAdvisorsThatCanApply(candidateAdvisors,
beanClass, beanName);
        //拓展操作
        extendAdvisors(eligibleAdvisors);
        if (!eligibleAdvisors.isEmpty()) {
            eligibleAdvisors = sortAdvisors(eligibleAdvisors);
        }
        return eligibleAdvisors;
    }
```

4.4.1获取增强器

由于分析的使用注解的方式实现AOP，对于findCandidateAdvisors是在AnnotationAwareAspectJAutoProxyCreator完成的。

```
protected List<Advisor> findCandidateAdvisors() {
        // Add all the Spring advisors found according to superclass rules.
        // 调用父类方法从容器中查找所有通知器
        List<Advisor> advisors = super.findCandidateAdvisors();
        // Build Advisors for all AspectJ aspects in the bean factory.
        // 解析@Aspect注解，并构建通知器
        advisors.addAll(this.aspectJAdvisorsBuilder.buildAspectJAdvisors());
        return advisors;
    }
```

```
public List<Advisor> buildAspectJAdvisors() {
        List<String> aspectNames = this.aspectBeanNames;

        if (aspectNames == null) {
            synchronized (this) {
                aspectNames = this.aspectBeanNames;
                if (aspectNames == null) {
                    List<Advisor> advisors = new LinkedList<Advisor>();
                    aspectNames = new LinkedList<String>();
                    //获取所有的beanName
                    String[] beanNames =
BeanFactoryUtils.beanNamesForTypeIncludingAncestors(
                            this.beanFactory, Object.class, true, false);
                    //循环所有的beanName找出对于的增强方法
                    for (String beanName : beanNames) {
                        //不合法的bean则略过，由子类定义规则，默认返回true
                        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.
                        // 获取对应的bean的类型
                        Class<?> beanType = this.beanFactory.getType(beanName);
                        if (beanType == null) {
                            continue;
                        }
                        // 如果存在Aspect注解
                        if (this.advisorFactory.isAspect(beanType)) {
                            aspectNames.add(beanName);
                            AspectMetadata amd = new AspectMetadata(beanType, beanName);
                            if (amd.getAjType().getPerClause().getKind() ==
PerClauseKind.SINGLETON) {
                                MetadataAwareAspectInstanceFactory factory =
                                        new
BeanFactoryAspectInstanceFactory(this.beanFactory, beanName);
                                //解析标记Aspect注解中的增强方法
                                List<Advisor> classAdvisors =
this.advisorFactory.getAdvisors(factory);
                                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.
                                if (this.beanFactory.isSingleton(beanName)) {
                                    throw new IllegalArgumentException("Bean with name '" +
beanName +
```

```
                                                "' is a singleton, but aspect instantiation
model is not singleton");
                                }
                                MetadataAwareAspectInstanceFactory factory =
                                        new
PrototypeAspectInstanceFactory(this.beanFactory, beanName);
                                this.aspectFactoryCache.put(beanName, factory);
                                advisors.addAll(this.advisorFactory.getAdvisors(factory));
                            }
                        }
                    }
                    this.aspectBeanNames = aspectNames;
                    return advisors;
                }
            }
        }
        ...
        return advisors;
    }
```

提取Advisor的步骤如下：

（1）获取所有的beanName，所有在beanFactory注册的bean都会被提取出来；

（2）遍历所有的beanName,并找出声明Aspectj注解的类，进行进一步的处理；

（3）对标记为Aspectj注解的类进行增强器的提取；

（4）将提取的结果加入缓存。

- 其中最为重要也是最为复杂的就是增强器获取，是由this.advisorFactory.getAdvisors(factory)实现的：

```
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();
        //验证
        validate(aspectClass);

        // We need to wrap the MetadataAwareAspectInstanceFactory with a decorator
        // so that it will only instantiate once.
MetadataAwareAspectInstanceFactory lazySingletonAspectInstanceFactory =
            new LazySingletonAspectInstanceFactoryDecorator(aspectInstanceFactory);

        List<Advisor> advisors = new LinkedList<Advisor>();
        for (Method method : getAdvisorMethods(aspectClass)) {
            // 普通增强器的获取
            Advisor advisor = getAdvisor(method, lazySingletonAspectInstanceFactory,
advisors.size(), aspectName);
            if (advisor != null) {
                advisors.add(advisor);
```

```
            }
        }

        // If it's a per target aspect, emit the dummy instantiating aspect.
        //如果在配置中将增强配置为延迟初始化，需要在首位加入同步实例化增强器以保证增强使用之前的实例化
        if (!advisors.isEmpty() &&
lazySingletonAspectInstanceFactory.getAspectMetadata().isLazilyInstantiated()) {
            Advisor instantiationAdvisor = new
SyntheticInstantiationAdvisor(lazySingletonAspectInstanceFactory);
            advisors.add(0, instantiationAdvisor);
        }

        // Find introduction fields.
        // 获取DeclareParents注解
        for (Field field : aspectClass.getDeclaredFields()) {
            Advisor advisor = getDeclareParentsAdvisor(field);
            if (advisor != null) {
                advisors.add(advisor);
            }
        }

        return advisors;
    }
```

下面将详细介绍每一个步骤：

普通增强器的获取

```
public Advisor getAdvisor(Method candidateAdviceMethod,
MetadataAwareAspectInstanceFactory aspectInstanceFactory,
            int declarationOrderInAspect, String aspectName) {

        validate(aspectInstanceFactory.getAspectMetadata().getAspectClass());

        // 切点信息的获取
        AspectJExpressionPointcut expressionPointcut = getPointcut(
                candidateAdviceMethod,
aspectInstanceFactory.getAspectMetadata().getAspectClass());
        if (expressionPointcut == null) {
            return null;
        }

        //根据切点信息生成增强器
        return new InstantiationModelAwarePointcutAdvisorImpl(expressionPointcut,
candidateAdviceMethod,
                this, aspectInstanceFactory, declarationOrderInAspect, aspectName);
    }
```

- 切点信息的获取，指定注解表达式的获取，如@Before("test())")

```
private AspectJExpressionPointcut getPointcut(Method candidateAdviceMethod, Class<?>
candidateAspectClass) {
        // 获取方法上的注解
        AspectJAnnotation<?> aspectJAnnotation =

AbstractAspectJAdvisorFactory.findAspectJAnnotationOnMethod(candidateAdviceMethod);
        if (aspectJAnnotation == null) {
            return null;
        }

        //使用AspectJExpressionPointcut实例封装获取的信息
        AspectJExpressionPointcut ajexp =
                new AspectJExpressionPointcut(candidateAspectClass, new String[0], new
Class<?>[0]);
        //提取得到的注解表达式，@Pointcut("execution (public * * (..) )")
        ajexp.setExpression(aspectJAnnotation.getPointcutExpression());
        ajexp.setBeanFactory(this.beanFactory);
        return ajexp;
    }
```

- 根据切点信息生成增强器，是由InstantiationModelAwarePointcutAdvisorImpl统一封装完成的，不同的增强需要不同的增强器来完成不同的逻辑：

```
//根据不同的注解类型封装不同的增强器
switch (aspectJAnnotation.getAnnotationType()) {
   case AtBefore:
      springAdvice = new AspectJMethodBeforeAdvice(
            candidateAdviceMethod, expressionPointcut, aspectInstanceFactory);
      break;
   case AtAfter:
      springAdvice = new AspectJAfterAdvice(
            candidateAdviceMethod, expressionPointcut, aspectInstanceFactory);
      break;
   case AtAfterReturning:
      springAdvice = new AspectJAfterReturningAdvice(
            candidateAdviceMethod, expressionPointcut, aspectInstanceFactory);
      AfterReturning afterReturningAnnotation = (AfterReturning)
aspectJAnnotation.getAnnotation();
      if (StringUtils.hasText(afterReturningAnnotation.returning())) {
         springAdvice.setReturningName(afterReturningAnnotation.returning());
      }
      break;
   case AtAfterThrowing:
      springAdvice = new AspectJAfterThrowingAdvice(
            candidateAdviceMethod, expressionPointcut, aspectInstanceFactory);
      AfterThrowing afterThrowingAnnotation = (AfterThrowing)
aspectJAnnotation.getAnnotation();
      if (StringUtils.hasText(afterThrowingAnnotation.throwing())) {
         springAdvice.setThrowingName(afterThrowingAnnotation.throwing());
      }
      break;
   case AtAround:
      springAdvice = new AspectJAroundAdvice(
```

```
                candidateAdviceMethod, expressionPointcut, aspectInstanceFactory);
         break;
      case AtPointcut:
         if (logger.isDebugEnabled()) {
            logger.debug("Processing pointcut '" + candidateAdviceMethod.getName() +
"'");
         }
         return null;
      default:
         throw new UnsupportedOperationException(
               "Unsupported advice type on method: " + candidateAdviceMethod);
   }
```

@Before会对应AspectJMethodBeforeAdvice，实现增强逻辑。以MethodBeforeAdviceInterceptor为例，前置增强在拦截器链中放置MethodBeforeAdviceInterceptor，而在MethodBeforeAdviceInterceptor中又放置了AspectJMethodBeforeAdvice，并在调用invoke时首先串联调用。

```
public class MethodBeforeAdviceInterceptor implements MethodInterceptor, Serializable {

    private MethodBeforeAdvice advice;


    /**
     * Create a new MethodBeforeAdviceInterceptor for the given advice.
     * @param advice the MethodBeforeAdvice to wrap
     */
    public MethodBeforeAdviceInterceptor(MethodBeforeAdvice advice) {
        Assert.notNull(advice, "Advice must not be null");
        this.advice = advice;
    }

    @Override
    public Object invoke(MethodInvocation mi) throws Throwable {
        // 设置前置通知逻辑
        this.advice.before(mi.getMethod(), mi.getArguments(), mi.getThis() );
        // 通过 MethodInvocation 调用下一个拦截器，若所有拦截器均执行完，则调用目标方法
        return mi.proceed();
    }

}
```

```
public void before(Method method, Object[] args, Object target) throws Throwable {
   invokeAdviceMethod(getJoinPointMatch(), null, null);
}
```

```
protected Object invokeAdviceMethod(JoinPointMatch jpMatch, Object returnValue, Throwable
ex) throws Throwable {
   return invokeAdviceMethodWithGivenArgs(argBinding(getJoinPoint(), jpMatch, returnValue,
ex));
}
```

```
protected Object invokeAdviceMethodWithGivenArgs(Object[] args) throws Throwable {
   Object[] actualArgs = args;
   if (this.aspectJAdviceMethod.getParameterTypes().length == 0) {
      actualArgs = null;
   }
   try {
      ReflectionUtils.makeAccessible(this.aspectJAdviceMethod);
      // TODO AopUtils.invokeJoinpointUsingReflection
      //增强激活方法
      return
this.aspectJAdviceMethod.invoke(this.aspectInstanceFactory.getAspectInstance(),
actualArgs);
   }
   catch (IllegalArgumentException ex) {
      throw new AopInvocationException("Mismatch on arguments to advice method [" +
            this.aspectJAdviceMethod + "]; pointcut expression [" +
            this.pointcut.getPointcutExpression() + "]", ex);
   }
   catch (InvocationTargetException ex) {
      throw ex.getTargetException();
   }
}
```

4.4.2寻找匹配的增强器

前面已经完成了所有增强器的解析，但是还需要根据规则挑选出满足我们配置的通配符的增强器，具体实现在findAdvisorsThatCanApply中：

```
protected List<Advisor> findAdvisorsThatCanApply(
      List<Advisor> candidateAdvisors, Class<?> beanClass, String beanName) {

   ProxyCreationContext.setCurrentProxiedBeanName(beanName);
   try {
      // 过滤已经得到的advisors
      return AopUtils.findAdvisorsThatCanApply(candidateAdvisors, beanClass);
   }
   finally {
      ProxyCreationContext.setCurrentProxiedBeanName(null);
   }
}
```

```
public static List<Advisor> findAdvisorsThatCanApply(List<Advisor> candidateAdvisors,
Class<?> clazz) {
   if (candidateAdvisors.isEmpty()) {
      return candidateAdvisors;
   }
   List<Advisor> eligibleAdvisors = new LinkedList<Advisor>();
   // 首先处理引介增强筛选 IntroductionAdvisor 类型的通知器
   for (Advisor candidate : candidateAdvisors) {
      if (candidate instanceof IntroductionAdvisor && canApply(candidate, clazz)) {
         eligibleAdvisors.add(candidate);
      }
```

```
    }
    boolean hasIntroductions = !eligibleAdvisors.isEmpty();
    for (Advisor candidate : candidateAdvisors) {
       //引介增强已经处理
       if (candidate instanceof IntroductionAdvisor) {
          // already processed
          continue;
       }
       //对于普通bean的处理
       if (canApply(candidate, clazz, hasIntroductions)) {
          eligibleAdvisors.add(candidate);
       }
    }
    return eligibleAdvisors;
}
```

4.4.3创建代理

在获取了所有对应的bean的增强器之后，便可以进行代理的创建了

```
protected Object createProxy(
            Class<?> beanClass, String beanName, Object[] specificInterceptors,
TargetSource targetSource) {

        if (this.beanFactory instanceof ConfigurableListableBeanFactory) {
            //指定bean的targetClass
            AutoProxyUtils.exposeTargetClass((ConfigurableListableBeanFactory)
this.beanFactory, beanName, beanClass);
        }

        ProxyFactory proxyFactory = new ProxyFactory();
        //获取当前类中相关属性
        proxyFactory.copyFrom(this);

        //决定对于给定的bean是否应该使用targetClass而不是其他的代理接口
        if (!proxyFactory.isProxyTargetClass()) {
            if (shouldProxyTargetClass(beanClass, beanName)) {
                proxyFactory.setProxyTargetClass(true);
            }
            else {
                //添加代理接口
                evaluateProxyInterfaces(beanClass, proxyFactory);
            }
        }

        Advisor[] advisors = buildAdvisors(beanName, specificInterceptors);
        for (Advisor advisor : advisors) {
            //加入增强器
            proxyFactory.addAdvisor(advisor);
        }

        //设置要代理的类
        proxyFactory.setTargetSource(targetSource);
```

```
        //定制代理
        customizeProxyFactory(proxyFactory);

        //用来控制代理工厂被配置之后，是否还允许修改通知
        //默认值为false，即被代理之后，不允许修改代理的配置
        proxyFactory.setFrozen(this.freezeProxy);
        if (advisorsPreFiltered()) {
            proxyFactory.setPreFiltered(true);
        }

        return proxyFactory.getProxy(getProxyClassLoader());
    }
```

对于代理类的创建及处理，spring委托给ProxyFactory去处理，该函数仅仅是初始化操作，为真正的创建代理做准备：

（1）获取当前类中的属性；

（2）添加代理接口；

（3）封装Advisor并加入到ProxyFactory中；

（4）设置要代理的类；

（5）为子类提供了定制的函数cunstomProxyFactory，子类可以在此函数中进行对ProxyFactory的进一步封装；

（6）进行获取代理操作。

- 创建代理

```
protected final synchronized AopProxy createAopProxy() {
        if (!this.active) {
            //激活代理的配置
            activate();
        }
        //创建代理
        return getAopProxyFactory().createAopProxy(this);
    }
```

```
 public AopProxy createAopProxy(AdvisedSupport config) throws AopConfigException {
        if (config.isOptimize() || config.isProxyTargetClass() ||
hasNoUserSuppliedProxyInterfaces(config)) {
            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.");
            }
            if (targetClass.isInterface() || Proxy.isProxyClass(targetClass)) {
                return new JdkDynamicAopProxy(config);
            }
            return new ObjenesisCglibAopProxy(config);
```

```
        }
        else {
            return new JdkDynamicAopProxy(config);
        }
    }
```

至此已经完成了代理的创建。在JDK代理中，在自定义的InvocationHandler中需要重写3个函数：

- 构造函数：将代理的对象传入；
- invoke方法，此方法中实现了AOP增强的所有逻辑；
- getProxy方法，创造一个新的代理对象，如下所示：

```
public Object getProxy(ClassLoader classLoader) {
   if (logger.isDebugEnabled()) {
      logger.debug("Creating JDK dynamic proxy: target source is " +
this.advised.getTargetSource());
   }
   Class<?>[] proxiedInterfaces =
AopProxyUtils.completeProxiedInterfaces(this.advised, true);
   findDefinedEqualsAndHashCodeMethods(proxiedInterfaces);
   return Proxy.newProxyInstance(classLoader, proxiedInterfaces, this);
}
```

下面重点来看一下invoke方法：

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

   TargetSource targetSource = this.advised.targetSource;
   Class<?> targetClass = null;
   Object target = null;

   try {
      if (!this.equalsDefined && AopUtils.isEqualsMethod(method)) {
         // The target does not implement the equals(Object) method itself.
         //equals方法的处理
         return equals(args[0]);
      }
      else if (!this.hashCodeDefined && AopUtils.isHashCodeMethod(method)) {
         // The target does not implement the hashCode() method itself.
         //hash方法的处理
         return hashCode();
      }
      else if (method.getDeclaringClass() == DecoratingProxy.class) {
         // There is only getDecoratedClass() declared -> dispatch to proxy config.
         return AopProxyUtils.ultimateTargetClass(this.advised);
      }
      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;

        // 如果exposeProxy属性为true,则暴露代理对象
        if (this.advised.exposeProxy) {
            // Make invocation available if necessary.
                  // 向AopContext中设置代理对象
            oldProxy = AopContext.setCurrentProxy(proxy);
            setProxyContext = true;
        }

        // May be null. Get as late as possible to minimize the time we "own" the target,
        // in case it comes from a pool.
        target = targetSource.getTarget();
        if (target != null) {
            targetClass = target.getClass();
        }

        // Get the interception chain for this method.
             // 获取适合当前方法的拦截器
        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，则直接执行切点方法
             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.
            Object[] argsToUse = AopProxyUtils.adaptArgumentsIfNecessary(method, args);
            // 通过反射执行目标方法
            retVal = AopUtils.invokeJoinpointUsingReflection(target, method, argsToUse);
        }
        else {
            // We need to create a method invocation...
                  // 创建一个方法调用器，并将拦截器链传入其中
            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.
             // 获取方法返回值类型
        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.
         //如果方法返回this，即return this,将代理对象赋值给retVal
               retVal = proxy;
            }
            // 如果返回值类型为基础类型，比如 int，long 等，当返回值为 null，抛出异常
      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 {
      if (target != null && !targetSource.isStatic()) {
         // Must have come from TargetSource.
         targetSource.releaseTarget(target);
      }
      if (setProxyContext) {
         // Restore old proxy.
         AopContext.setCurrentProxy(oldProxy);
      }
   }
}
```

invoke方法的执行流程如下所示：

1. 检测 expose-proxy 是否为 true，若为 true，则暴露代理对象
2. 获取适合当前方法的拦截器
3. 如果拦截器链为空，则直接通过反射执行目标方法
4. 若拦截器链不为空，则创建方法调用 ReflectiveMethodInvocation 对象
5. 调用 ReflectiveMethodInvocation 对象的 proceed() 方法启动拦截器链
6. 处理返回值，并返回该值

   ReflectiveMethodInvocation 贯穿于拦截器链执行的始终，该类的proceed方法用于启动拦截器链

```
public Object proceed() throws Throwable {
   // We start with an index of -1 and increment early.
   //拦截器链中的最后一个拦截器执行完后，即可执行目标方法
   if (this.currentInterceptorIndex == this.interceptorsAndDynamicMethodMatchers.size() -
1) {
      //执行目标方法
      return invokeJoinpoint();
   }

   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.
             /*
              * 调用具有三个参数（3-args）的 matches 方法动态匹配目标方法，
              * 两个参数（2-args）的 matches 方法用于静态匹配
              */
        InterceptorAndDynamicMethodMatcher dm =
              (InterceptorAndDynamicMethodMatcher) interceptorOrInterceptionAdvice;
        if (dm.methodMatcher.matches(this.method, this.targetClass, this.arguments)) {
           //调用拦截器逻辑
            return dm.interceptor.invoke(this);
        }
        else {
           // Dynamic matching failed.
           // Skip this interceptor and invoke the next in the chain.
                 //如果匹配失败，则忽略当前的拦截器
           return proceed();
        }
     }
     else {
        // It's an interceptor, so we just invoke it: The pointcut will have
        // been evaluated statically before this object was constructed.
        //调用拦截器逻辑，并传递ReflectiveMethodInvocation 对象
             return ((MethodInterceptor) interceptorOrInterceptionAdvice).invoke(this);
     }
  }
```

如上，proceed 根据 currentInterceptorIndex 来确定当前应执行哪个拦截器，并在调用拦截器的 invoke 方法时，将自己作为参数传给该方法。

5.问题回顾

- 1.spring aop的局限性?
- 2.五种通知的执行顺序?
- 3.代理对象可以在运行过程中动态添加拦截器吗?
- 4.代理对象内部方法调用怎么通过代理?