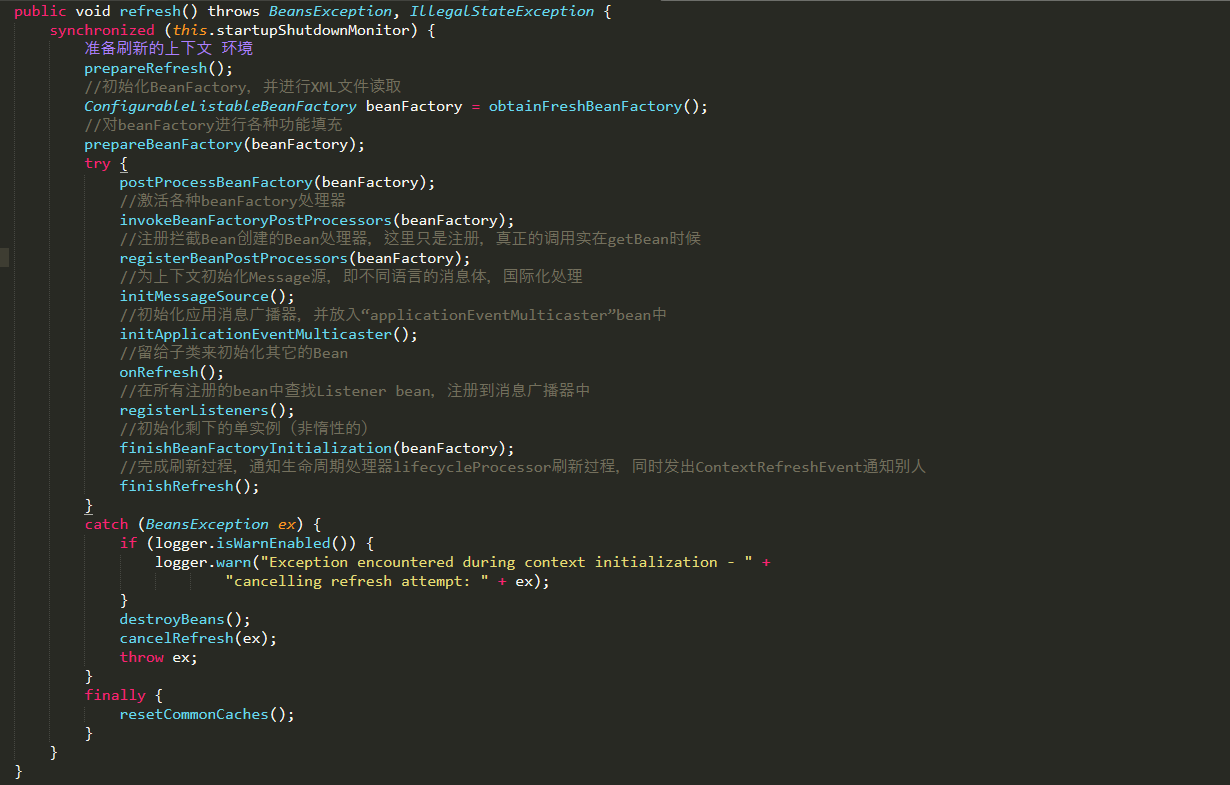
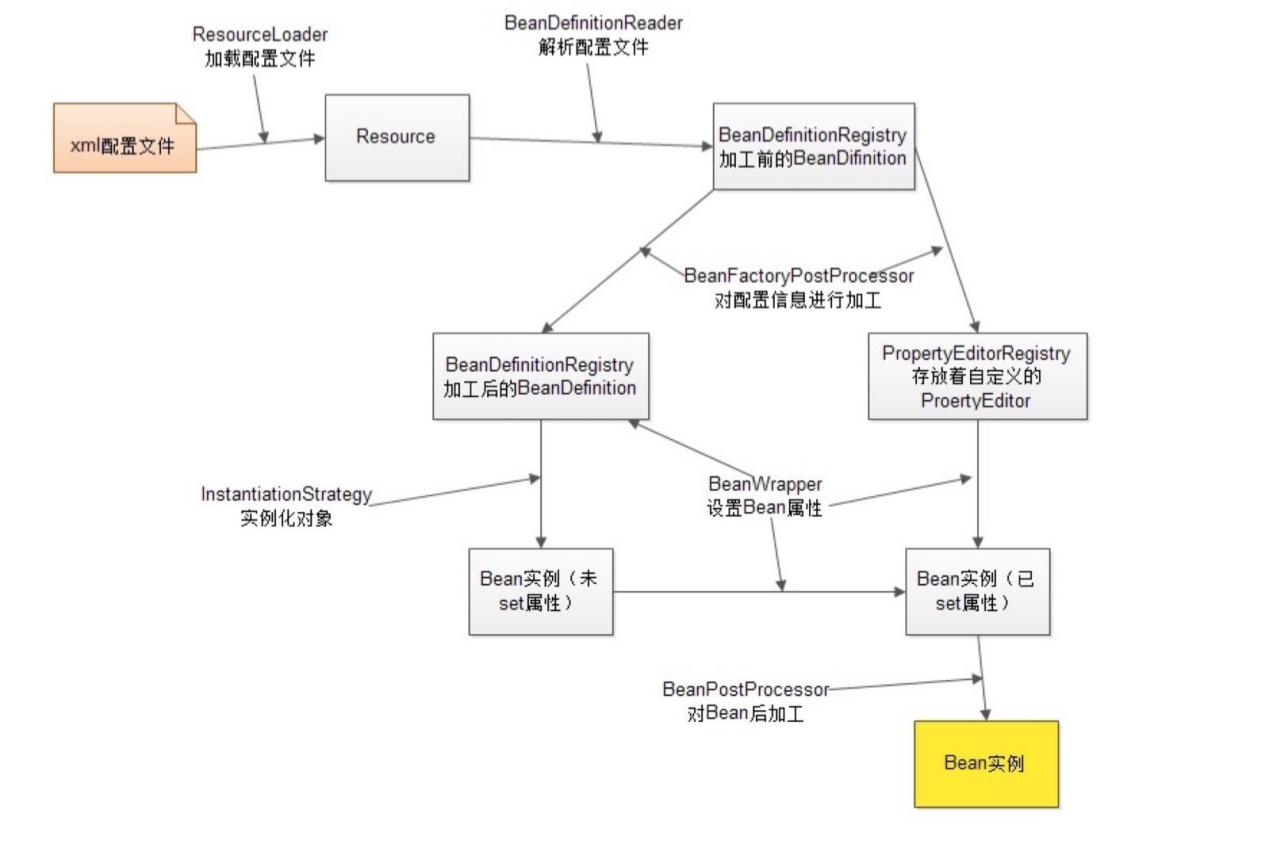
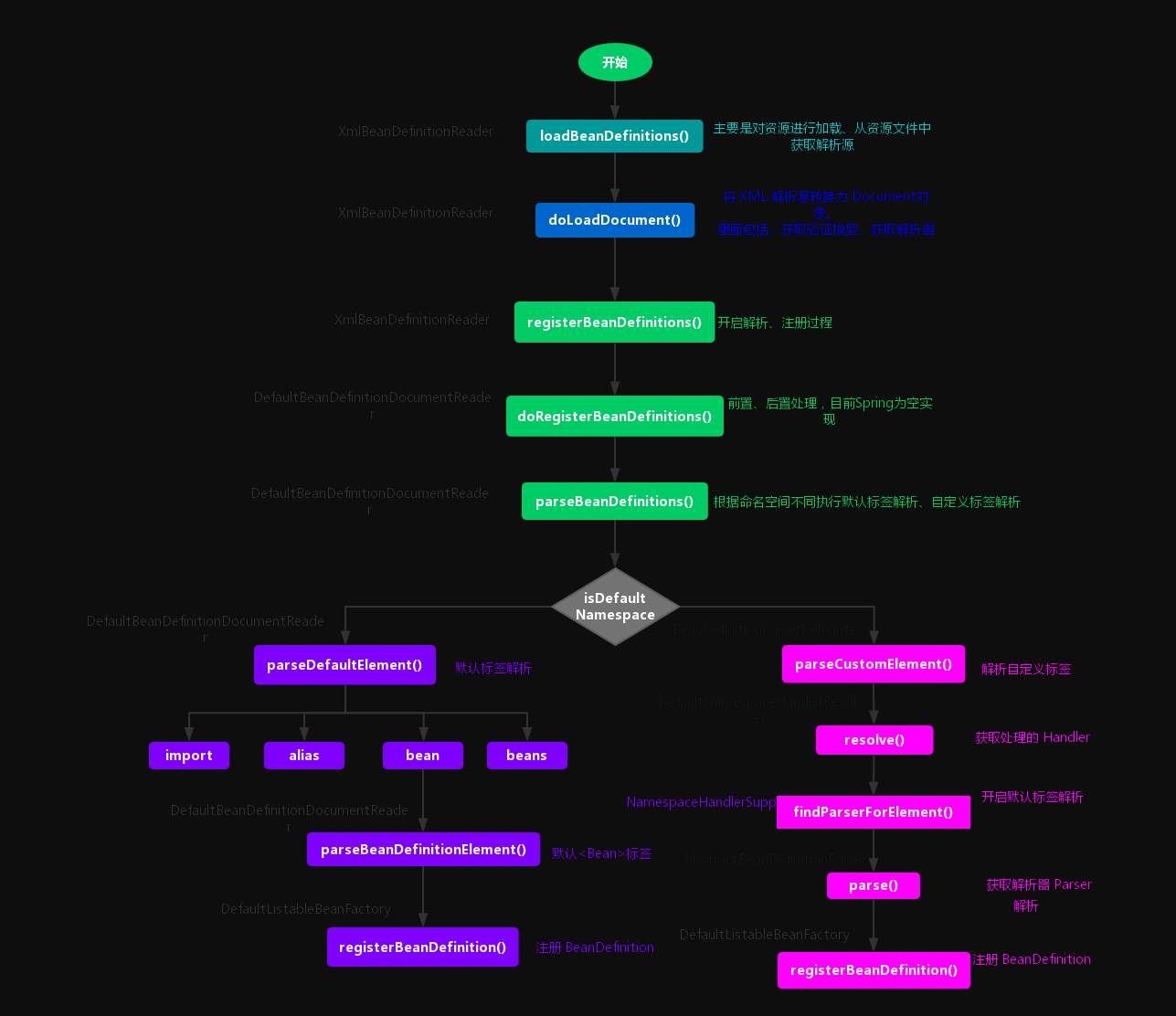
### Spring的refresh



### SpringIOC加载过程图解



### SpringIOC转载Bean到BeanDefinition过程



### SpringIOC实例化bean过程(getBean(XXX.class))

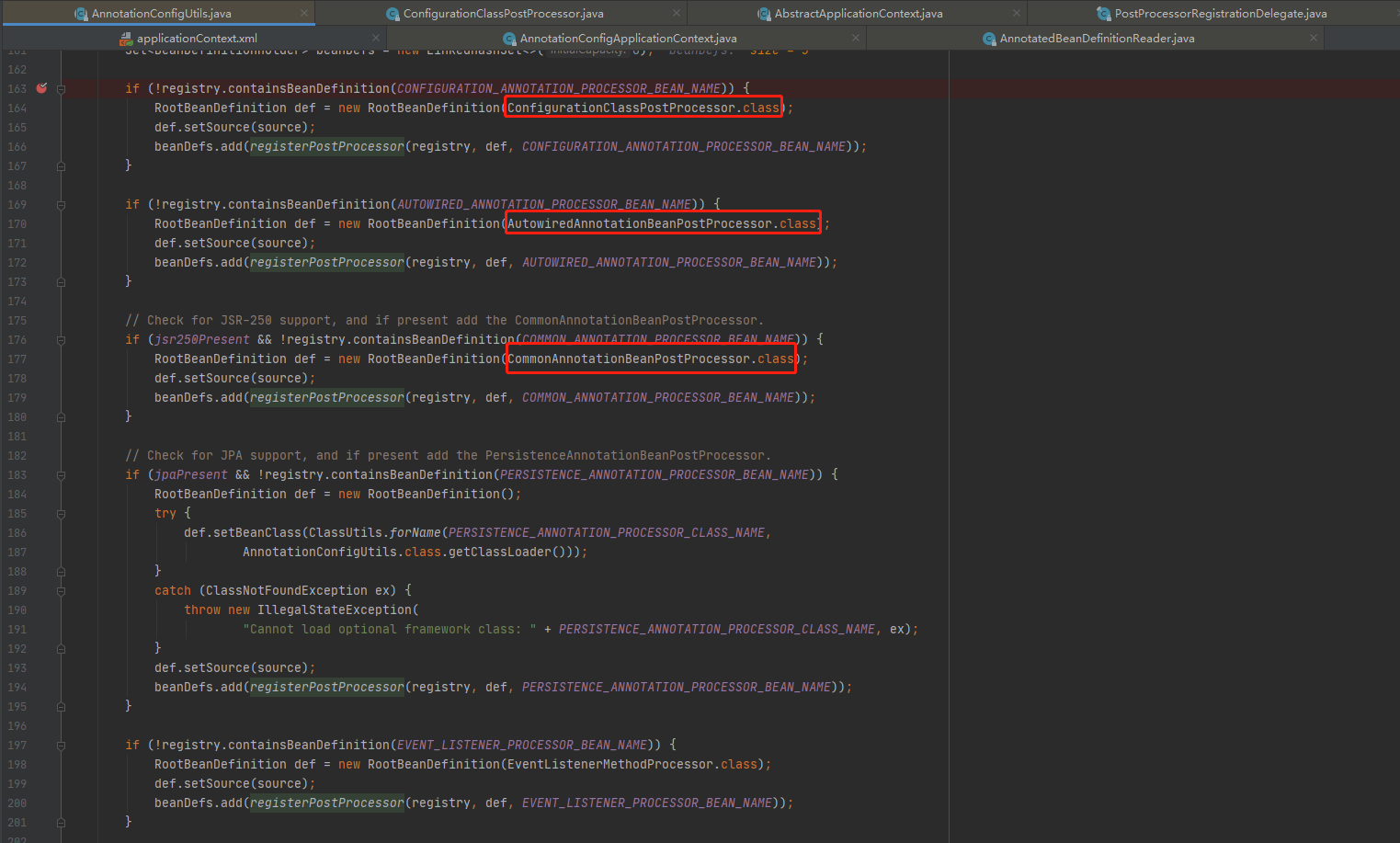


### Spring中<context:component-scan/>原理作用

**注: <context:component-scan/>和<context:annotation-config>的区别是，两者都是自动注入了下面4个后处理器，但是component-scan功能更多，还额外支持扫描注解等功能**

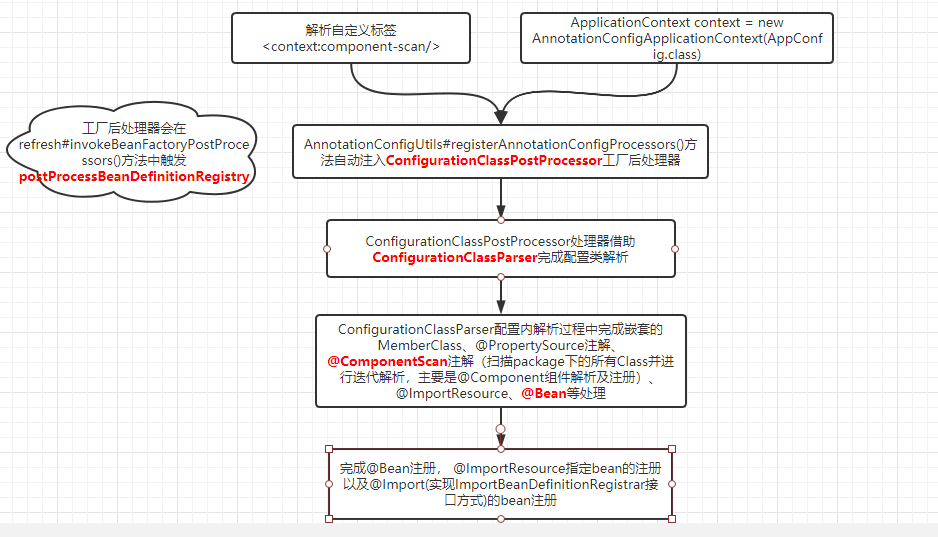
这个标签作用是扫描指定类下的注解例如@Component,@Autowired等将其转成bean。实际上看代码这个标签为我们自动注册了一系列的后处理器和工厂后处理器，以至于我们能实现加载注解的到bean的功能。

org.springframework.context.annotation.AnnotationConfigUtils#registerAnnotationConfigProcessors()



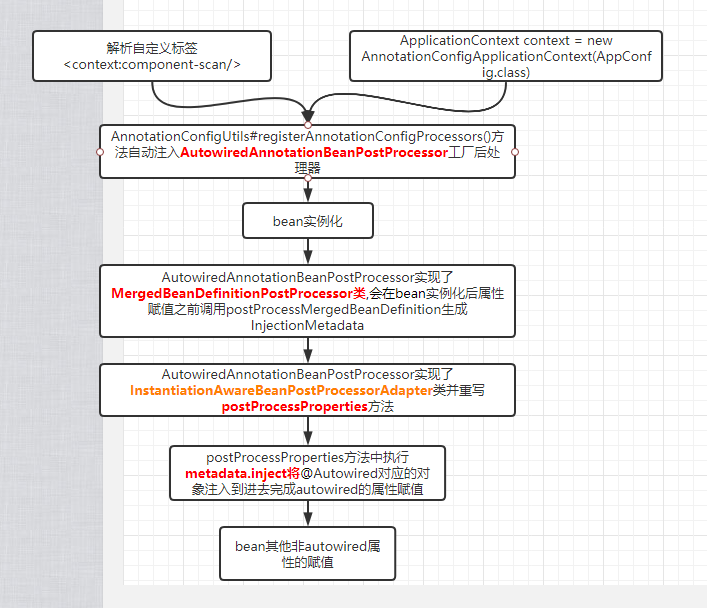
#### ConfigurationClassPostProcessor

**这个后处理器主要帮我们处理了@Configuration，@ComponentScan，@Bean等注解的处理，帮我们将注解类注册到spring容器中。**



#### AutowiredAnnotationBeanPostProcessor

这个下面@Autowried自动装配的有解释



#### CommonAnnotationBeanPostProcessor

它负责解析@Resource、@WebServiceRef、@EJB三个注解

这个类的方式跟AutowiredAnnotationBeanPostProcessor有点像。参考AutowiredAnnotationBeanPostProcessor就能知道他的方式，毕竟@Resource和@AutoWried差不多

#### EventListenerMethodProcessor

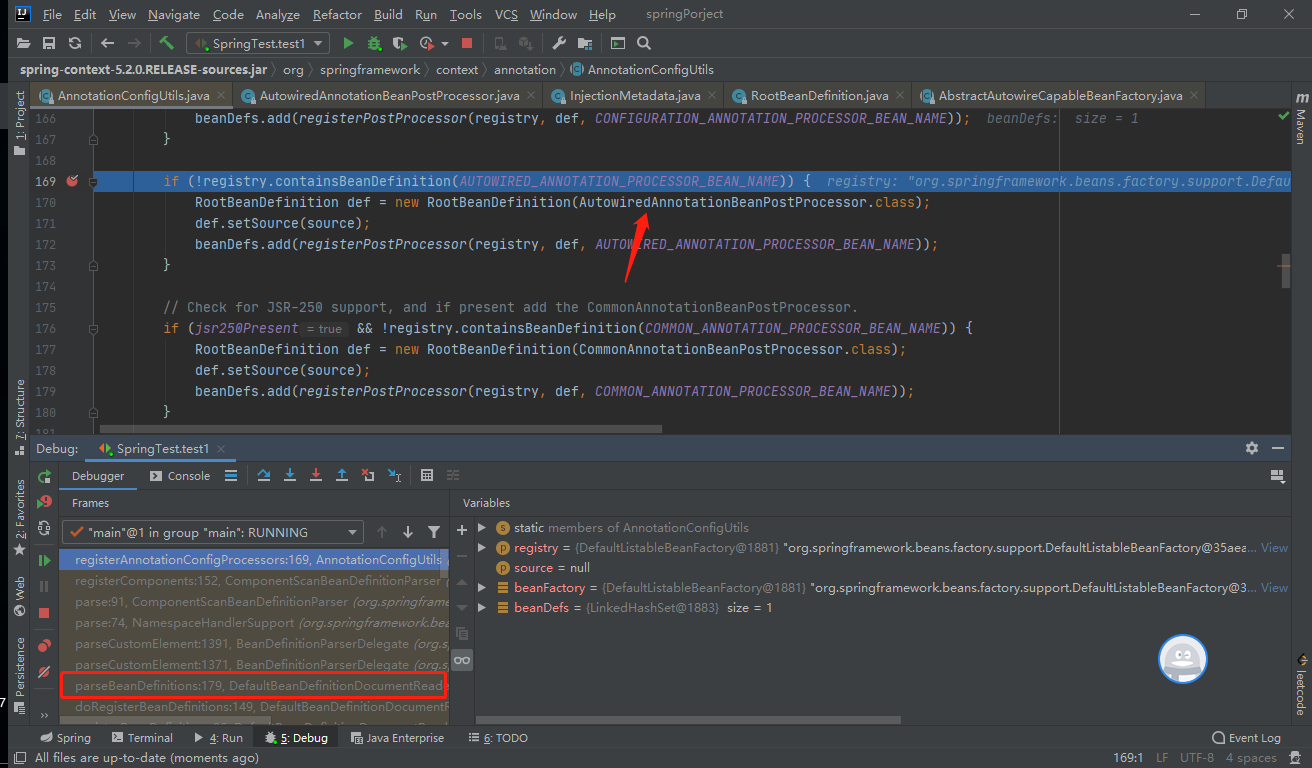
#### DefaultEventListenerFactory

### Spring中@Autowried自动装配是如何实现的

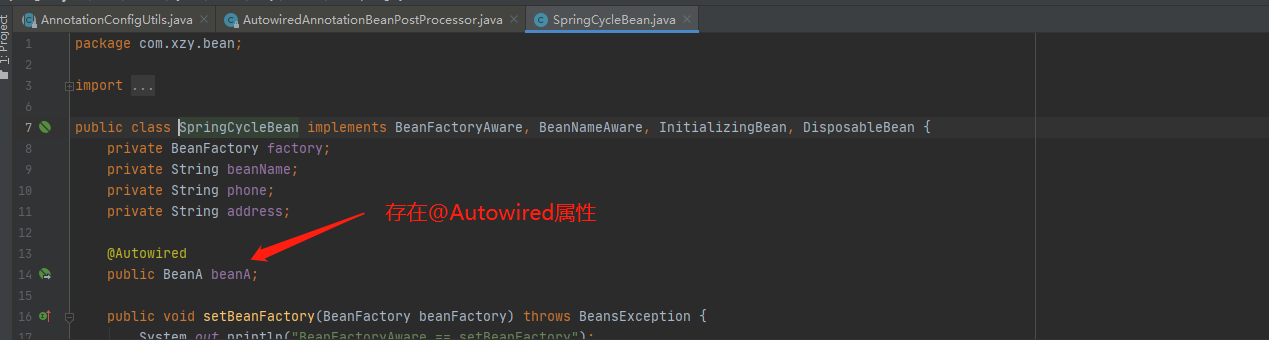
<https://zhuanlan.zhihu.com/p/28775709>

首先，@Autowried是通过后处理器实现的

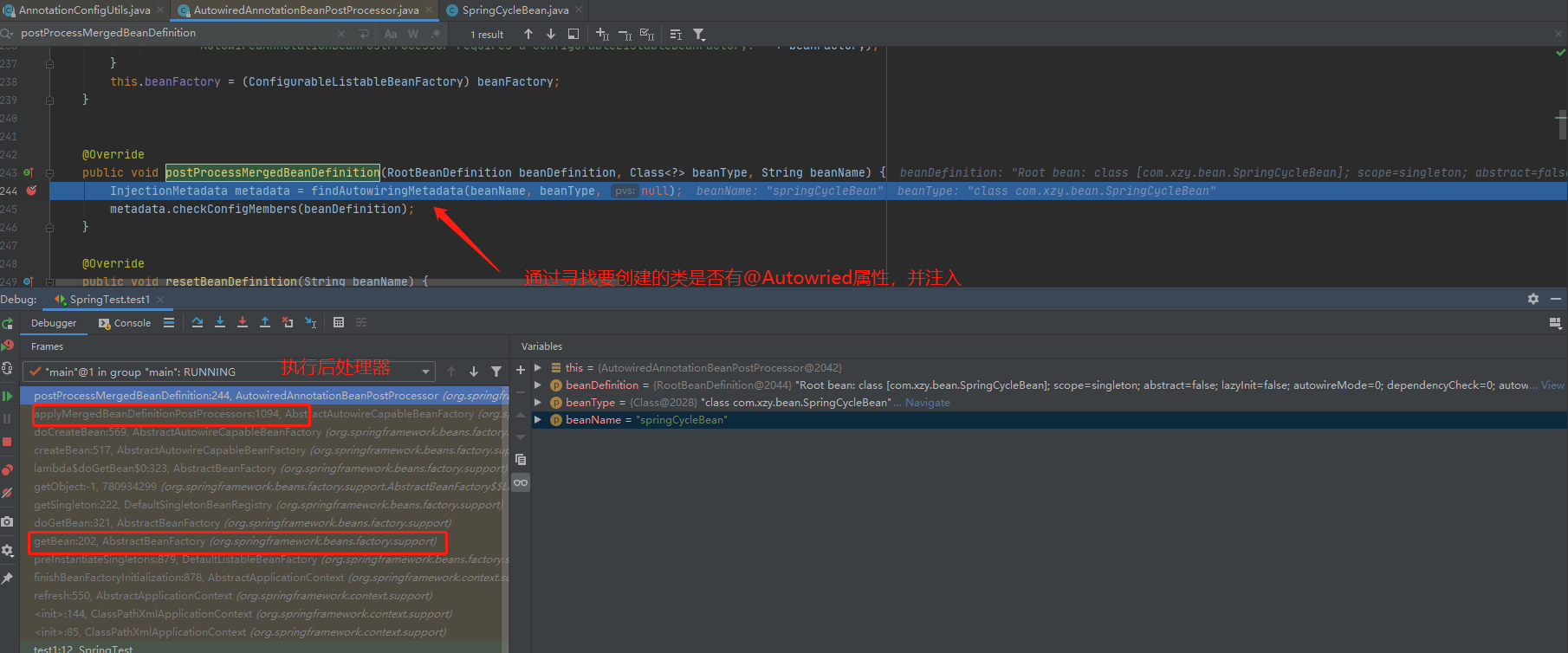
当我们在配置文件里面添加注释<context:component-scan/>[<context:annotation-config/>也有一样的效果]时。Spring通过parseBeanDefinitions解析标签，会自动注入AutowiredAnnotationBeanPostProcessor后处理器。

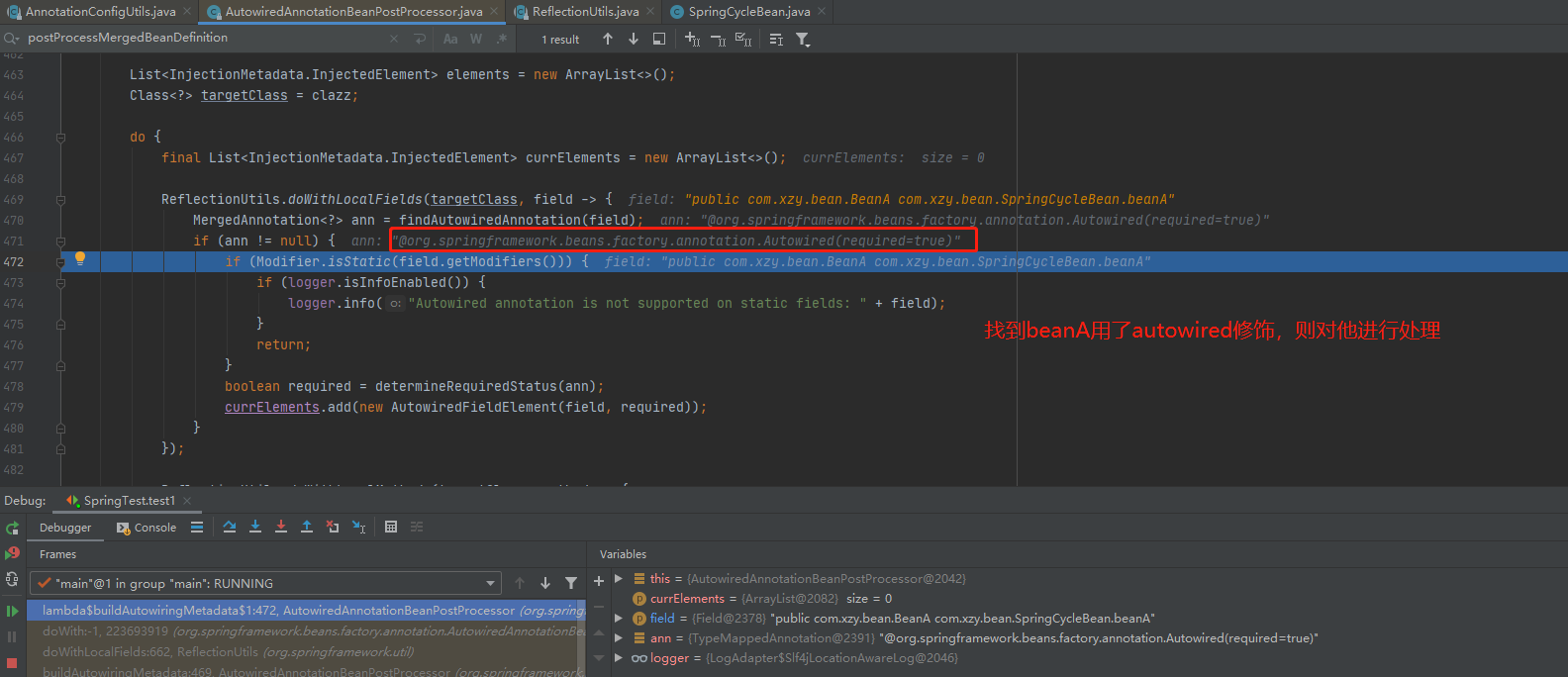


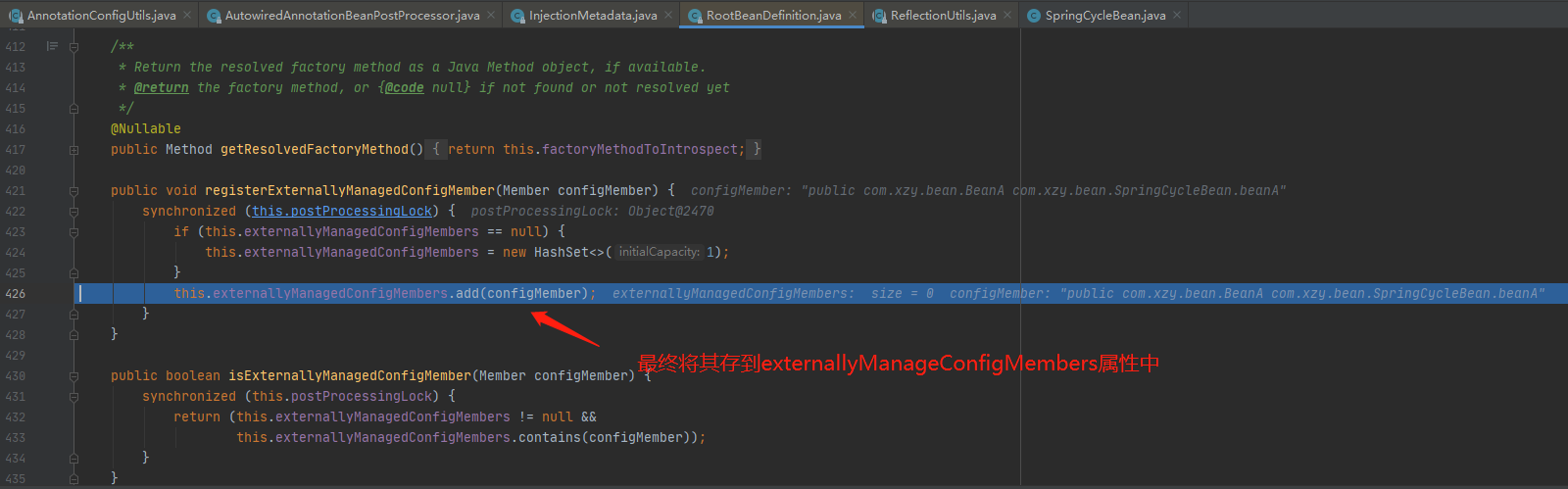
假如一个对象有@Autowried属性，则在getBean时，会通过AutowiredAnnotationBeanPostProcessor来解析@Autowried



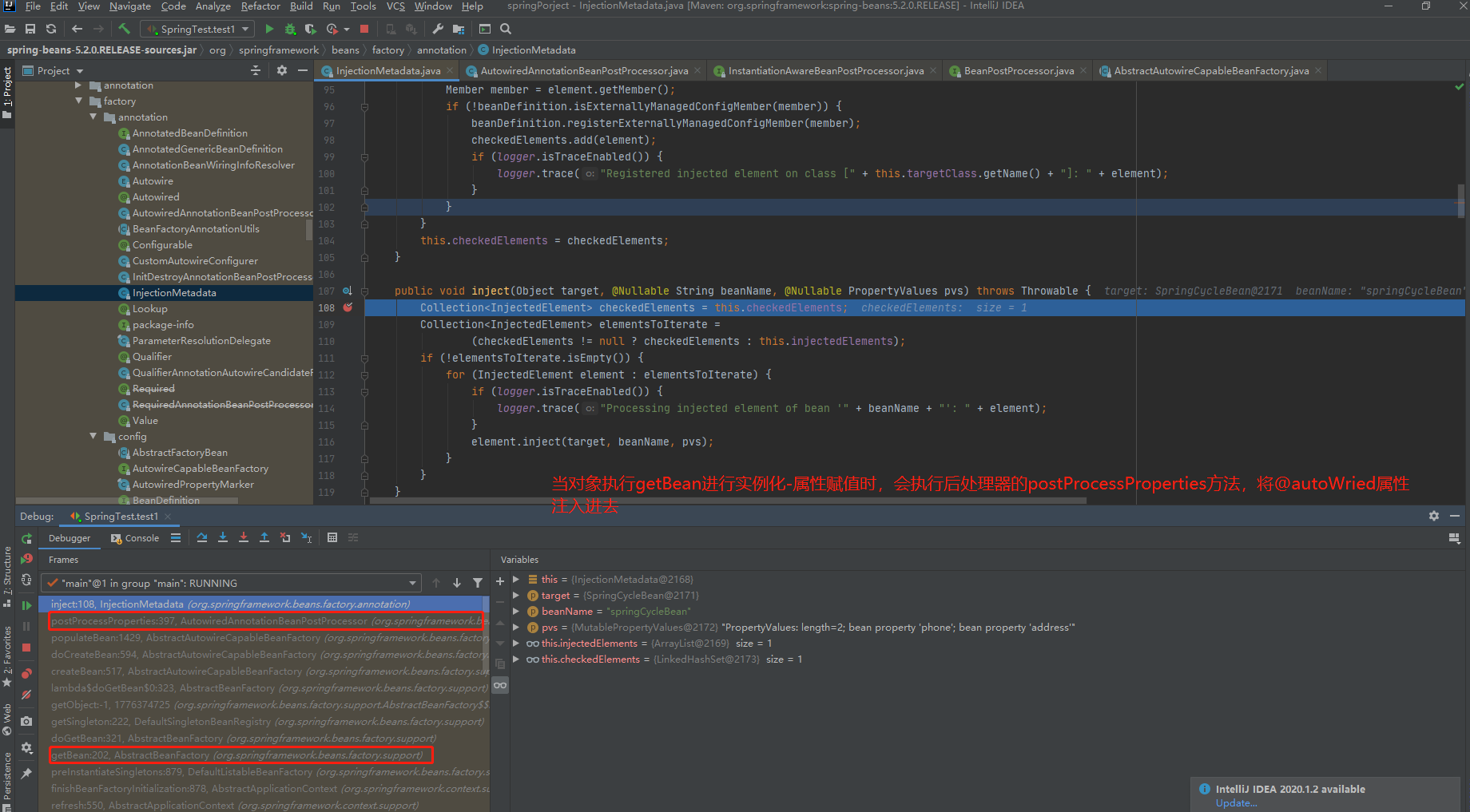
下图，在实例化之后，设置属性之前，由于实现了MergedBeanDefinitionPostProcessor，则会调用postProcessMergedBeanDefinition[这个方法在实例化后属性赋值之前调用]，找到对应的Autowired







当创建的对象要设置属性时，触发了AutowiredAnnotationBeanPostProcessor实现的后处理赋值方法postProcessPropertyValues(此方法是在InstantiationAwareBeanPostProcessorAdapter类中，需要子类重写),此方法中会将@autowired注解的bean先完成初始化后然后注入到对应的属性中,例如A中包含了@autowired B,则这时候A在属性赋值之前调用这个方法然后这时候如果B还没实例化，那会先实例化B然后将B注入到A中对应的属性中



总结:

1.xml配置文件配置上<context:component-scan>后，启动时通过对应的NameSpaceHandler类处理，自动注入了AutowiredAnnotationBeanPostProcessor后处理器。

2.AutowiredAnnotationBeanPostProcessor这个处理器实现了MergedBeanDefinitionPostProcessor和InstantiationAwareBeanPostProcessorAdapter处理器。

3.MergedBeanDefinitionPostProcessor中的postProcessMergedBeanDefinition方法会在实例化后属性赋值前执行，通过实现这个方法，然后利用反射机制将标注了@AutoWrited的字段或方法标注出来为elment。

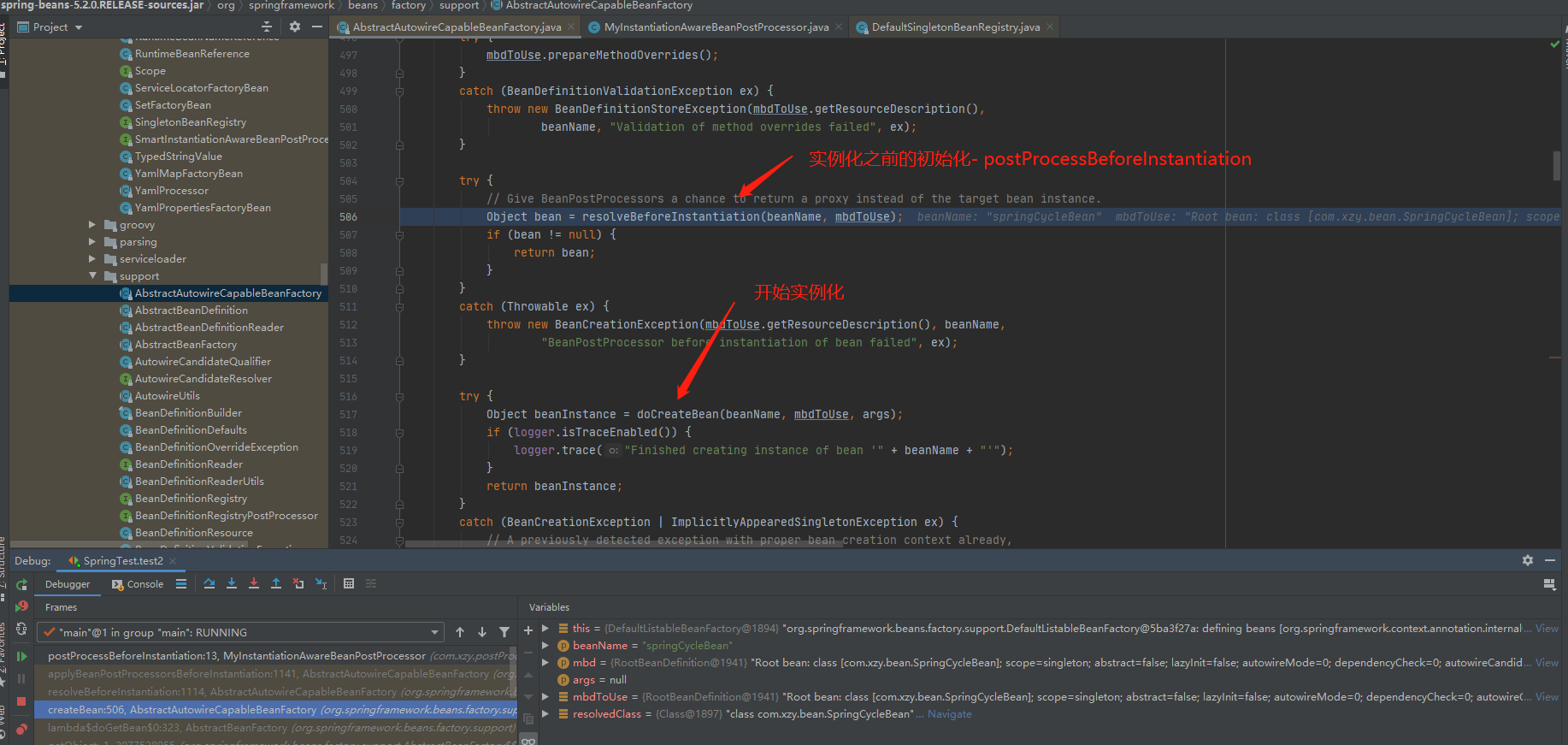
4.通过InstantiationAwareBeanPostProcessorAdapter中的postProcessPropertyValues方法在属性赋值时获取之前的需要注入的elment,然后从BeanFactory中获取并注入。

### Spring各种后处理器执行的代码

#### postProcessBeforeInstantiation实例化之前调用

org.springframework.beans.factory.support.AbstractAutowireCapableBeanFactory

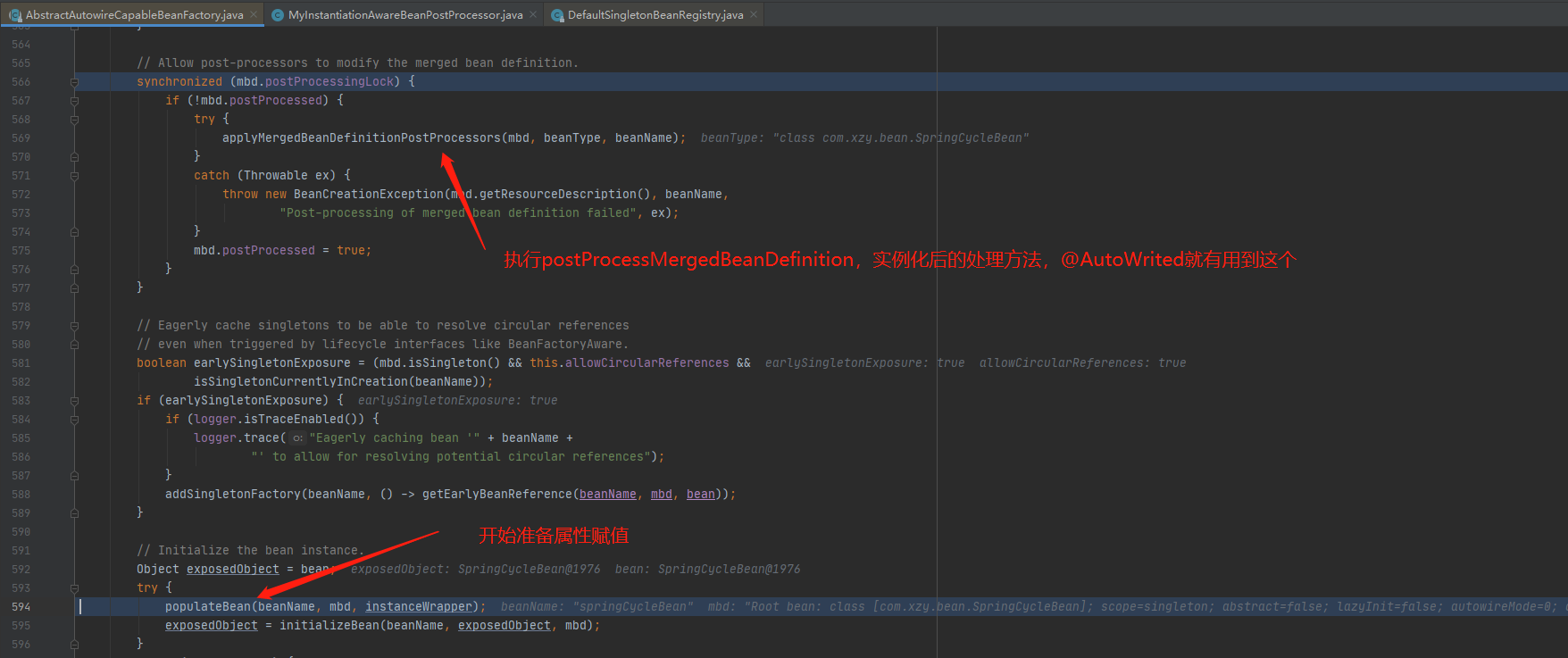
AOP的原理就是基于这个处理器方法，在他之前做了一层代理



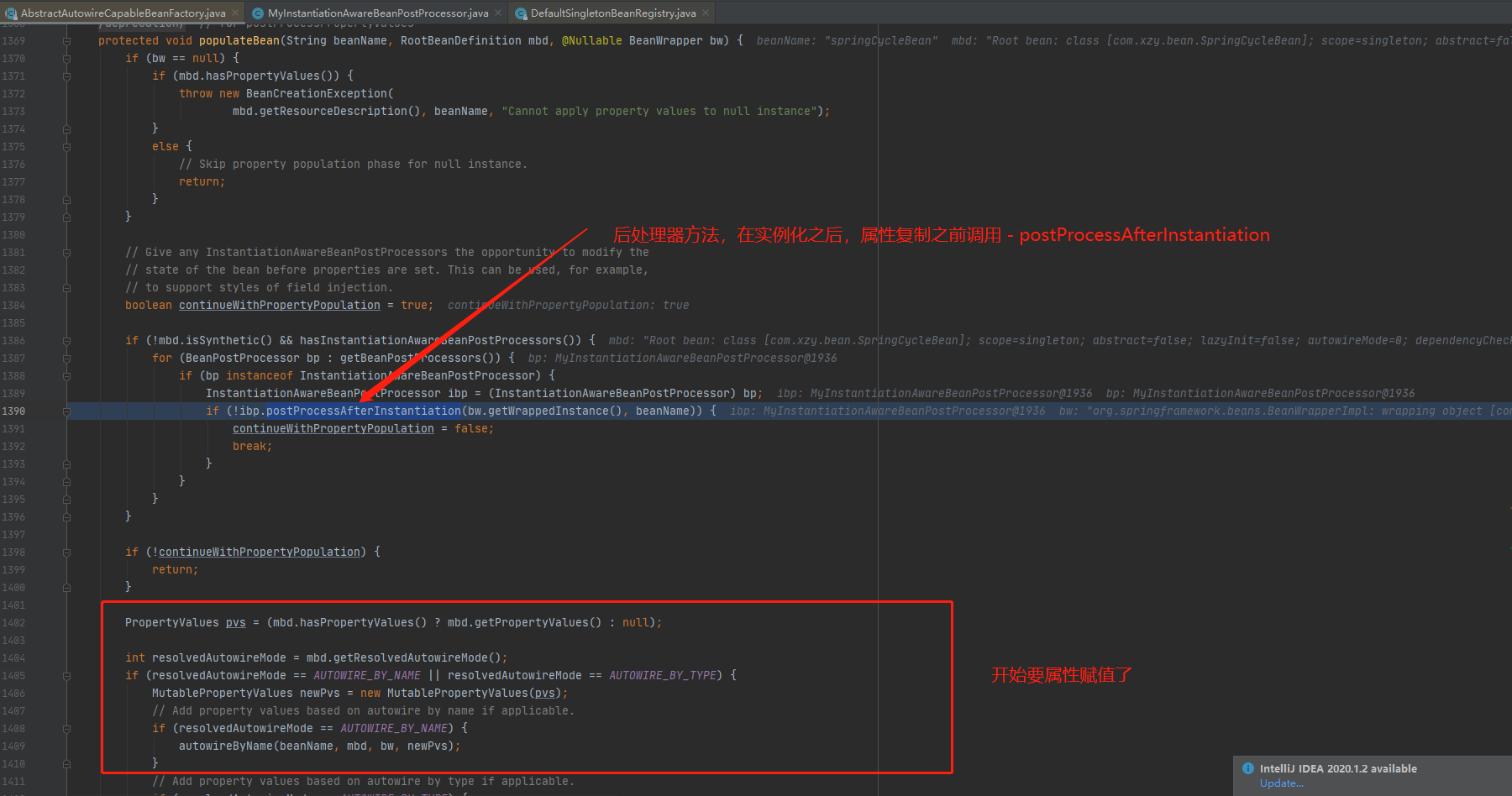
#### postProcessMergedBeanDefinition

org.springframework.beans.factory.support.AbstractAutowireCapableBeanFactory

postProcessMergedBeanDefinition和postProcessAfterInstantiation比对，两个都是在实例化之后，属性赋值之前调用，且前者早于后者，目前不清楚之间的区别



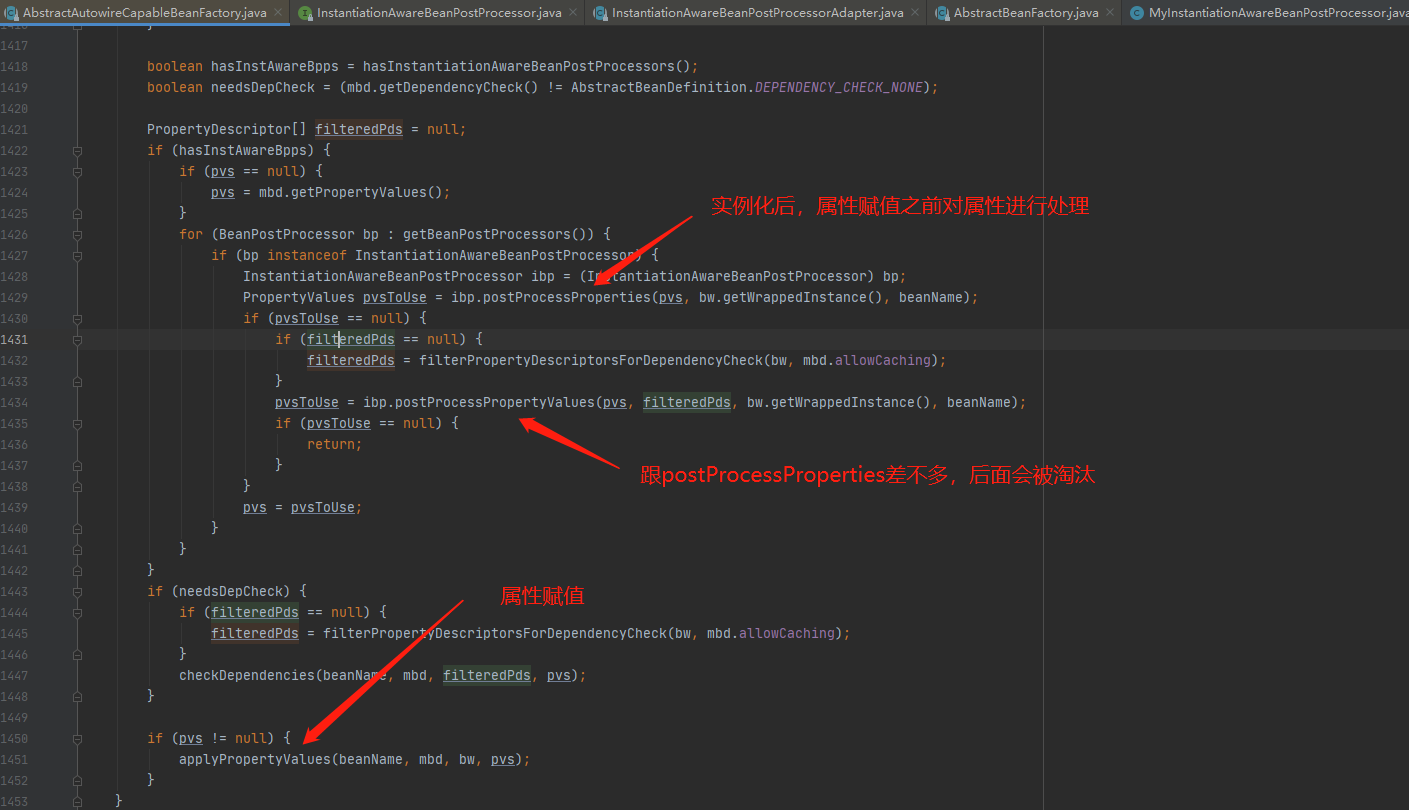
#### postProcessAfterInstantiation

org.springframework.beans.factory.support.AbstractAutowireCapableBeanFactory

#### postProcessProperties

org.springframework.beans.factory.support.AbstractAutowireCapableBeanFactory

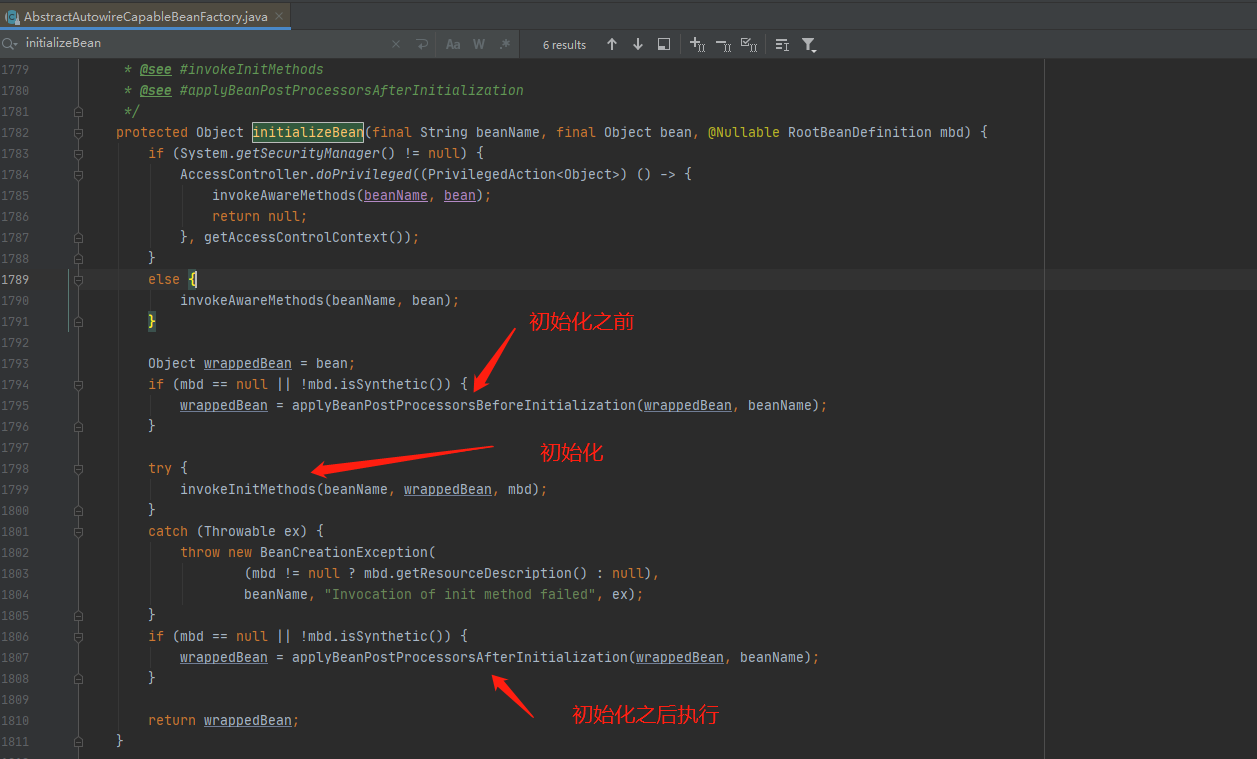
用于在属性赋值之前对属性做一些改动



#### postProcessBeforeInitialization

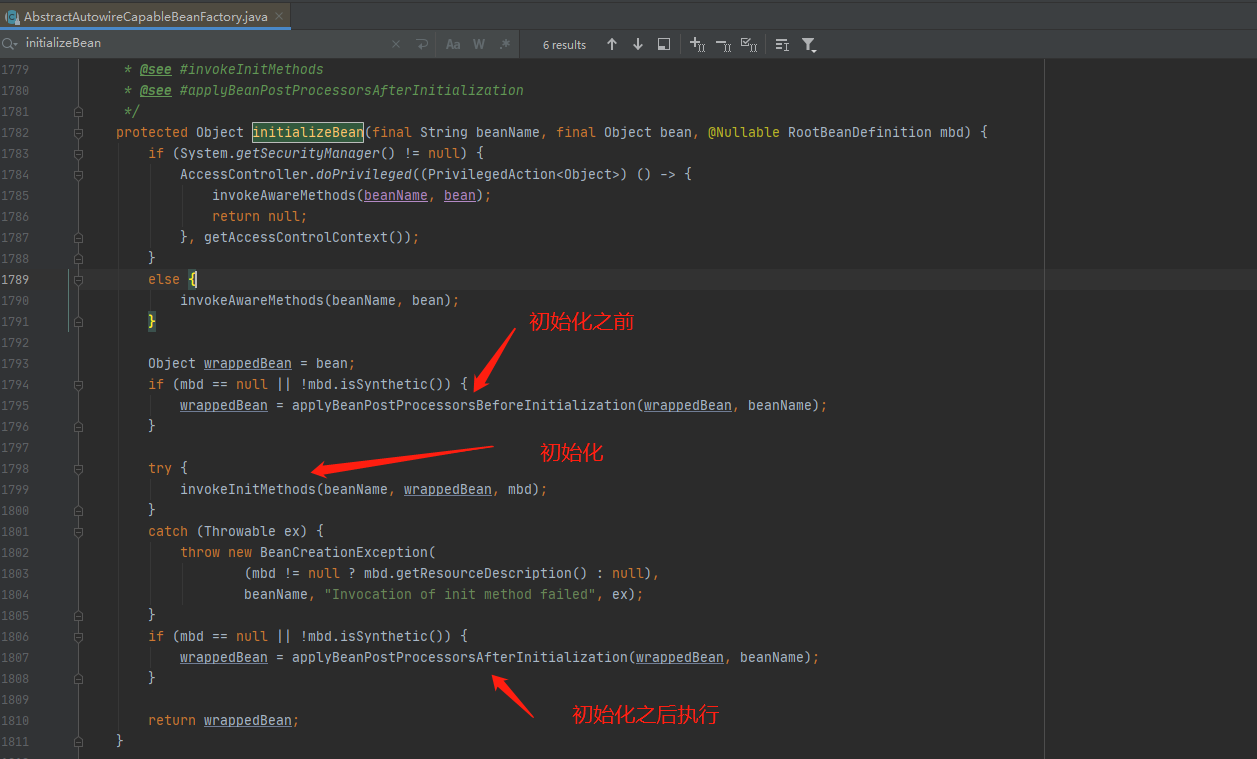
org.springframework.beans.factory.support.AbstractAutowireCapableBeanFactory

在初始化之前执行



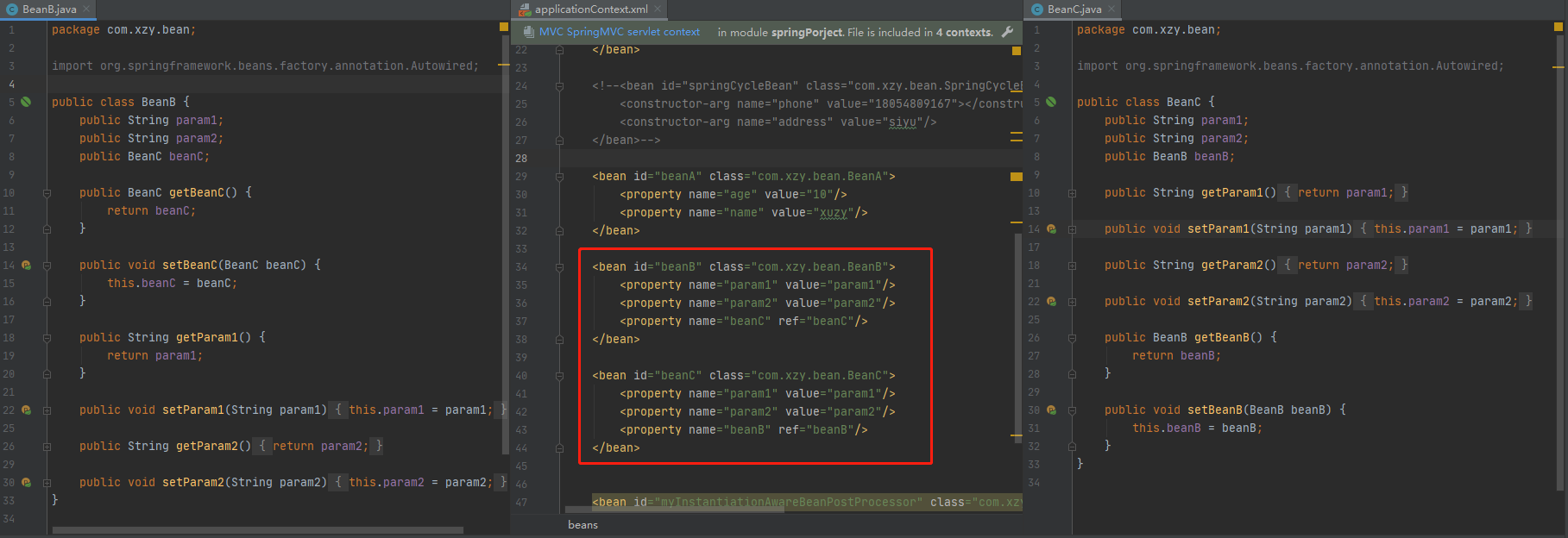
#### postProcessAfterInitialization

org.springframework.beans.factory.support.AbstractAutowireCapableBeanFactory

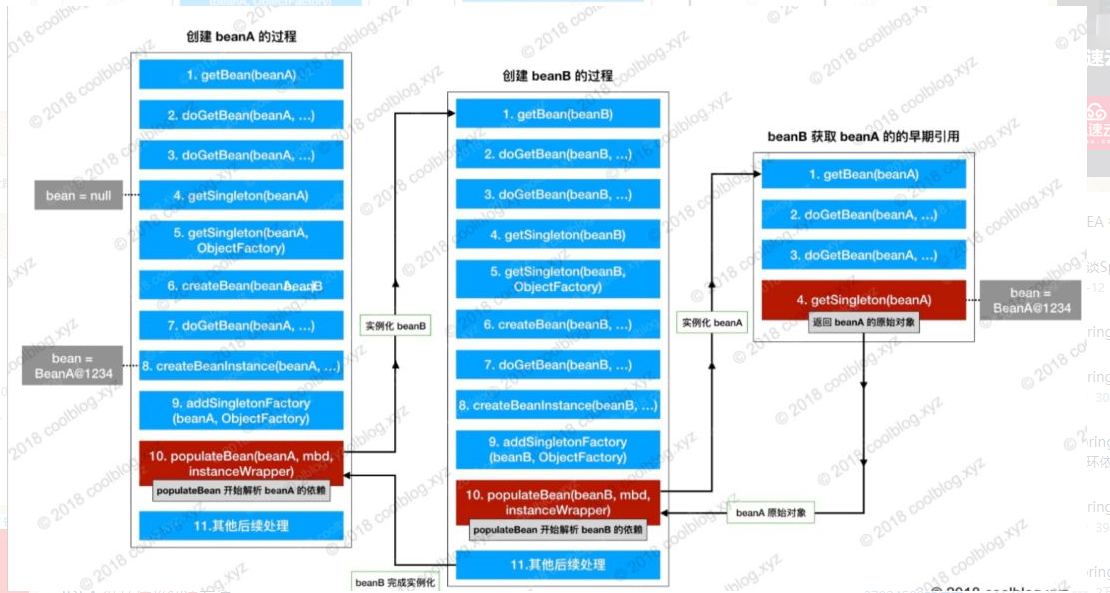


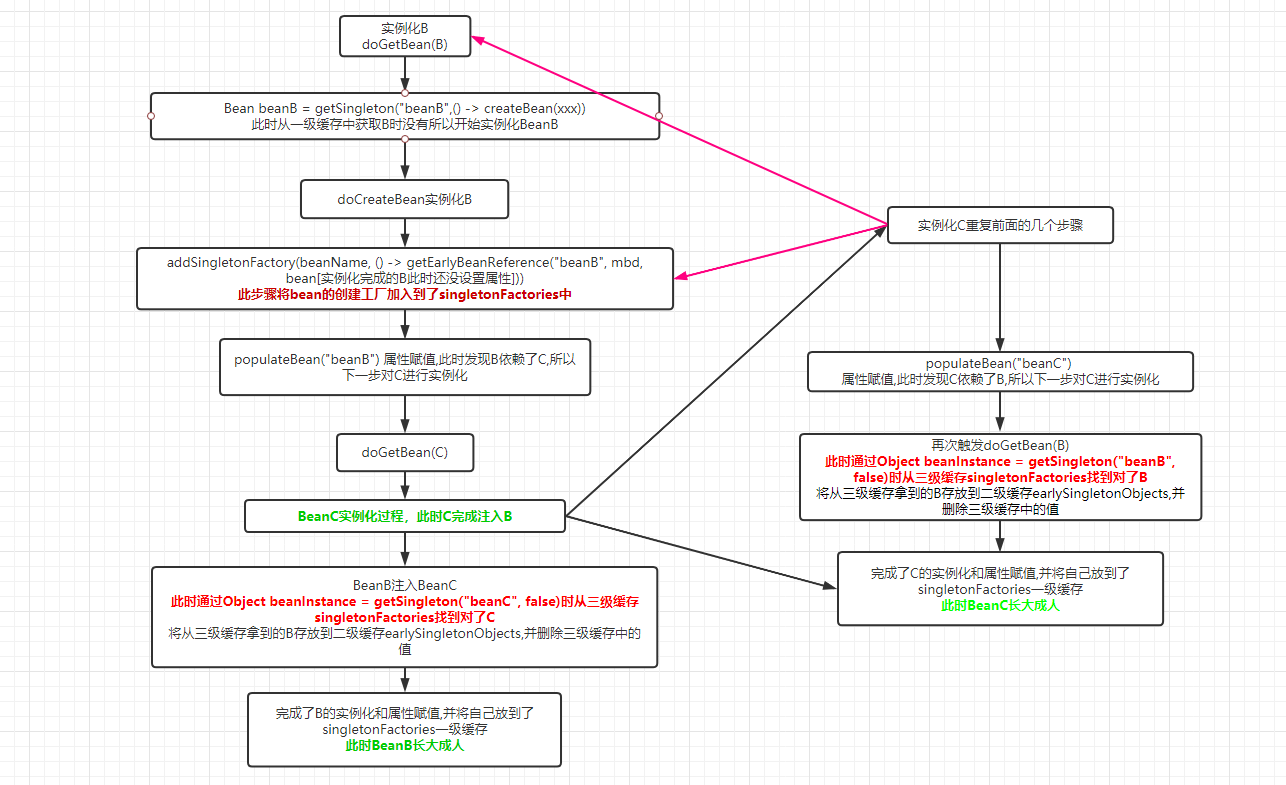
### Spring如何解决循环依赖

#### 循环依赖类设置



#### 相关流程图



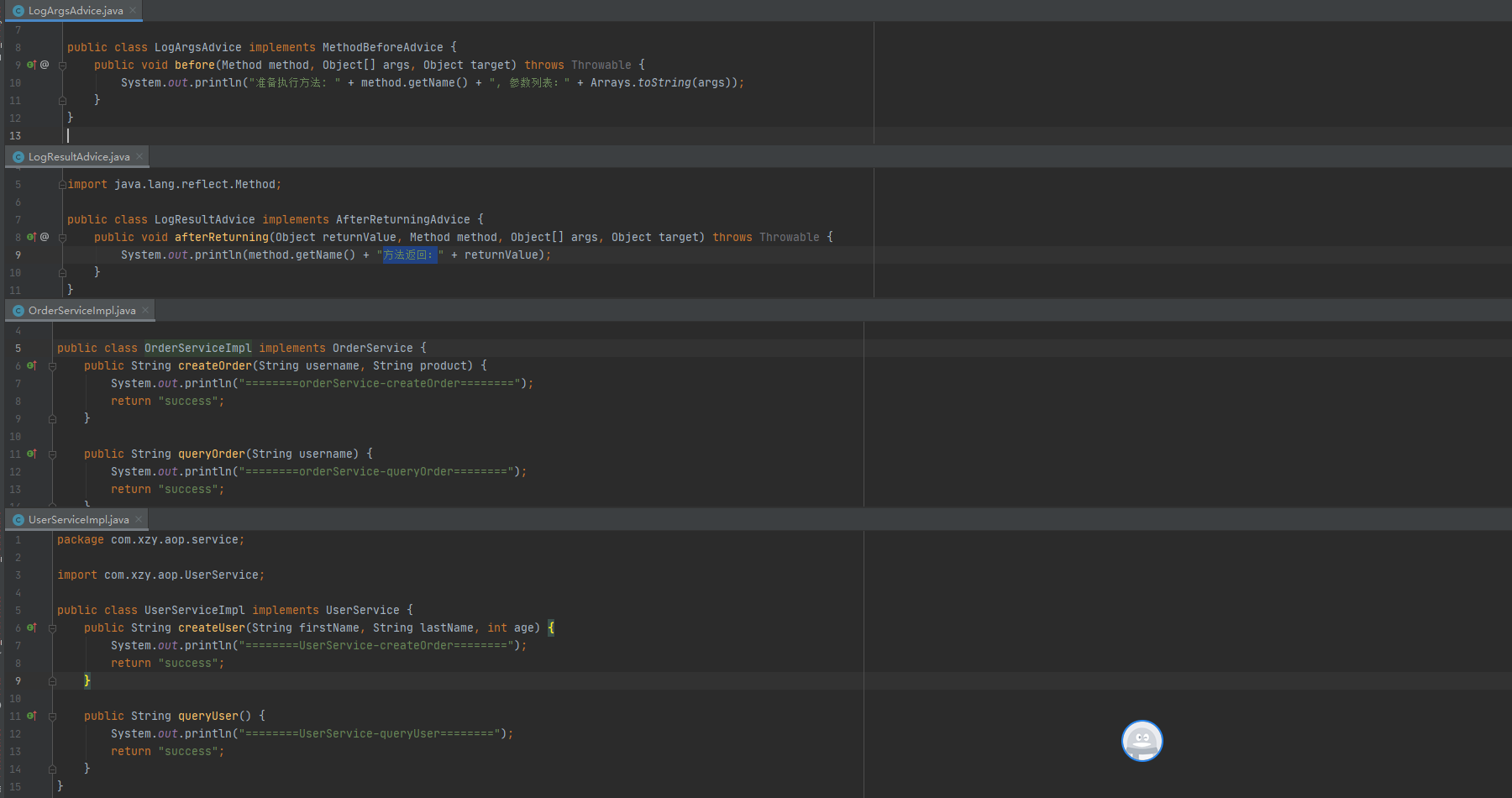


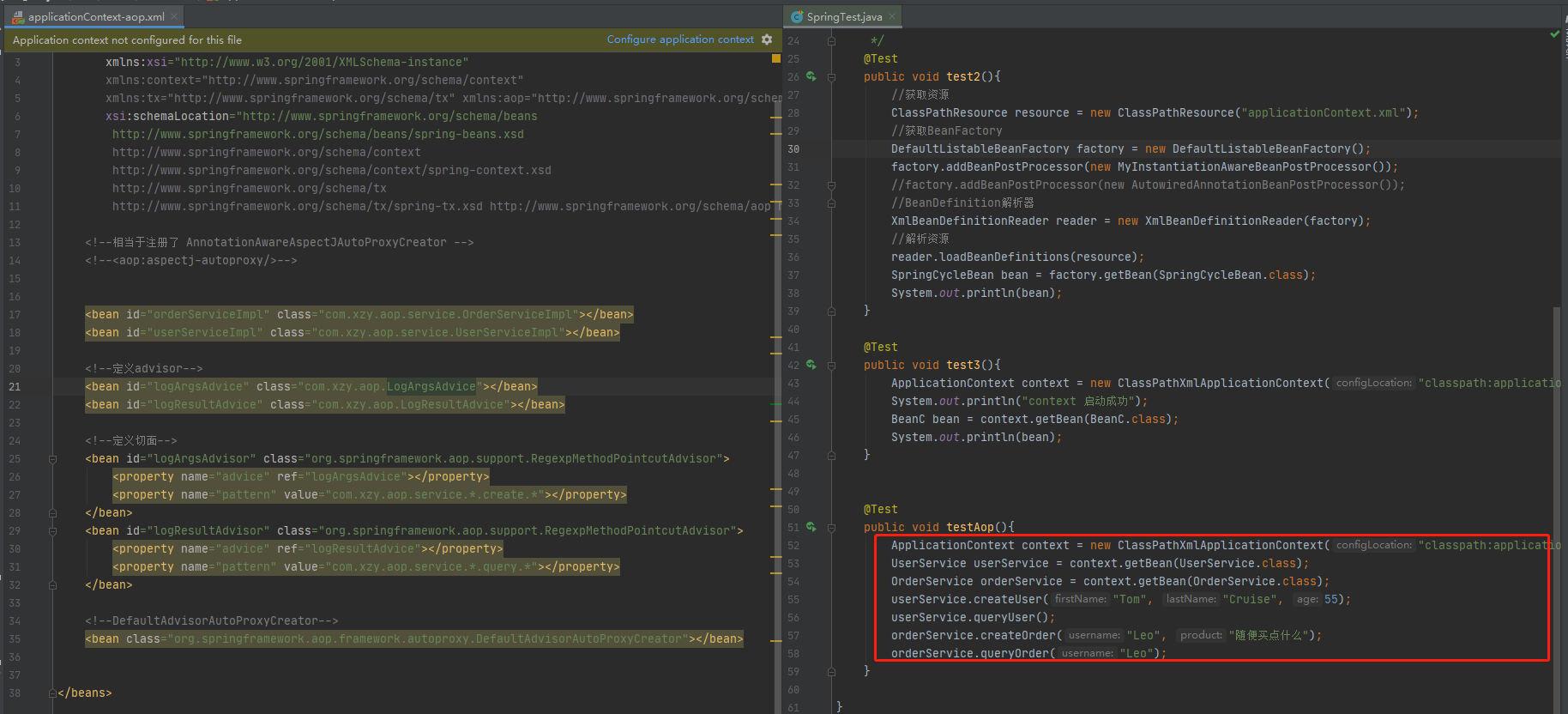
### SrpingAOP原理

#### SrpingAOP的原理是什么？

AOP(面向切面编程)实现的目标就是可以在方法执行前后插入自己的逻辑。他的实现原理它基于动态代理来实现。默认地，如果使用接口的，用 JDK 提供的动态代理实现，如果没有接口，使用 CGLIB 实现。

#### 具体测试代码类





#### 具体原理解析

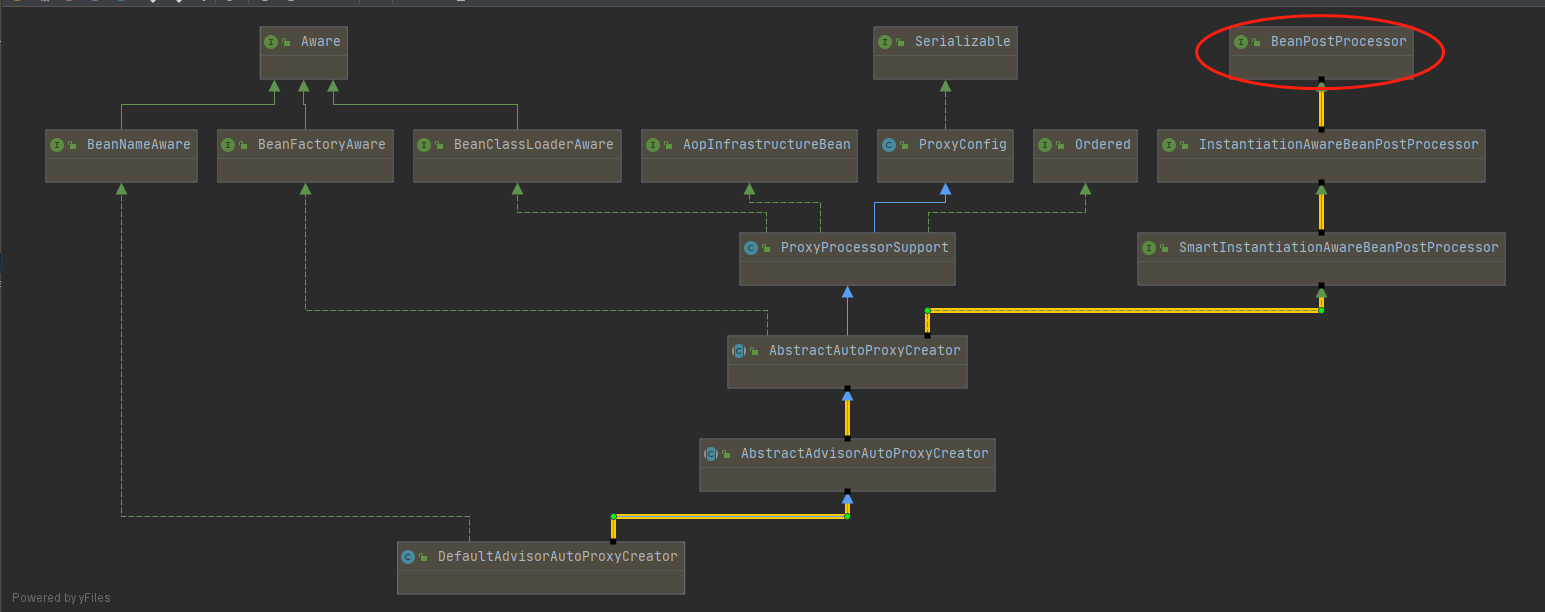
我们知道在执行createUser方法前后被植入了方法是用了代理的方式，如果没有AOP，在Spring获取生成orderServiceImpl实例时只是普通的实例，那到底SpringAOP是如何将orderServiceImpl生成的普通对象转成代理对象呢？

原理在于配置文件上的

**<bean class="org.springframework.aop.framework.autoproxy.DefaultAdvisorAutoProxyCreator"></bean>**

我们先看下这个类的继承关系

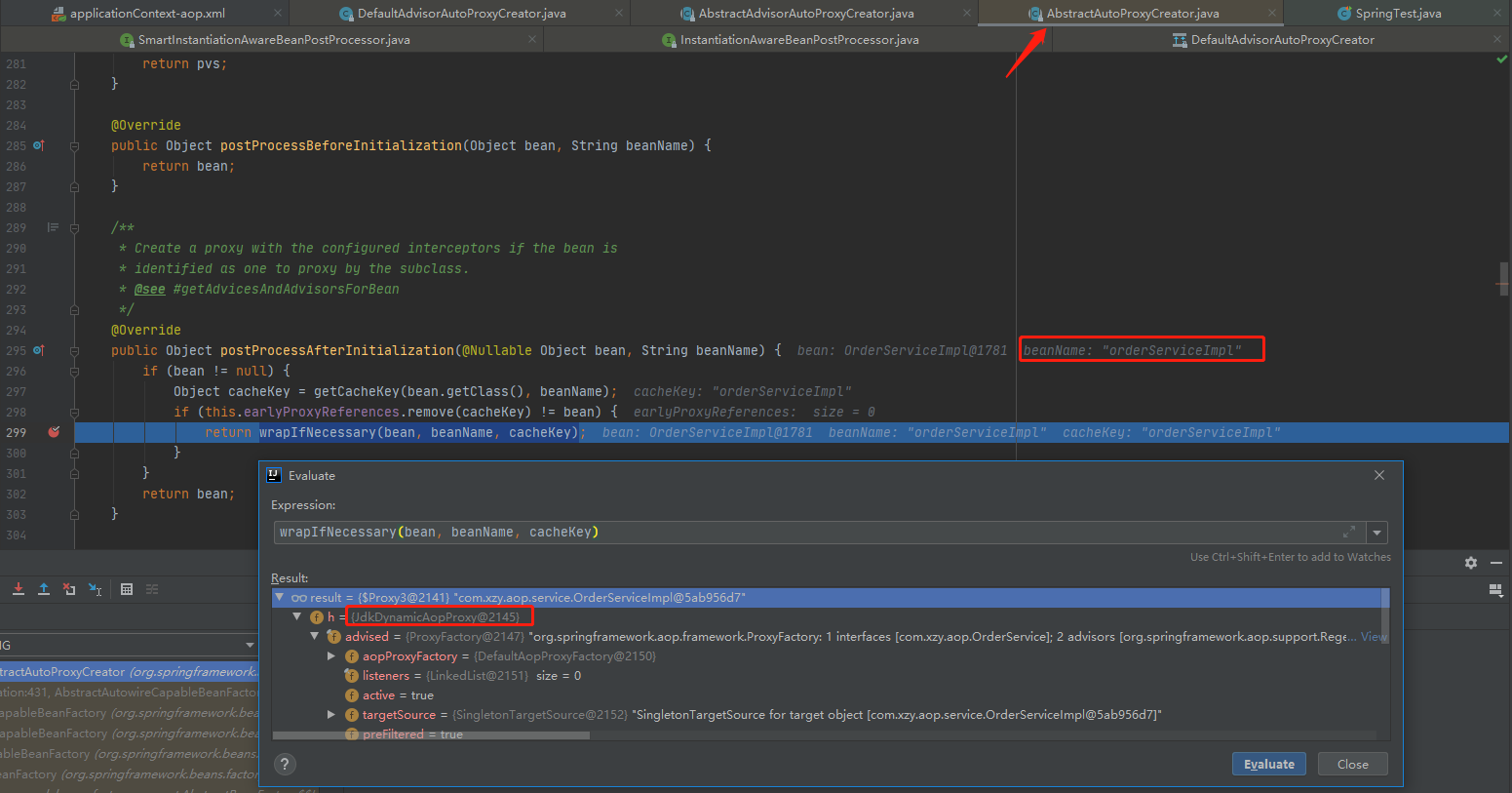
可以看到这个类实现了接口BeanPostProcessor后处理器，实现了这处理器后，就可以在每次bean初始化之前做一些操作，**所以可以猜想他是通过postProcessAfterInitialization方法在每个bean初始化之后将其用JDK代理包装了**。



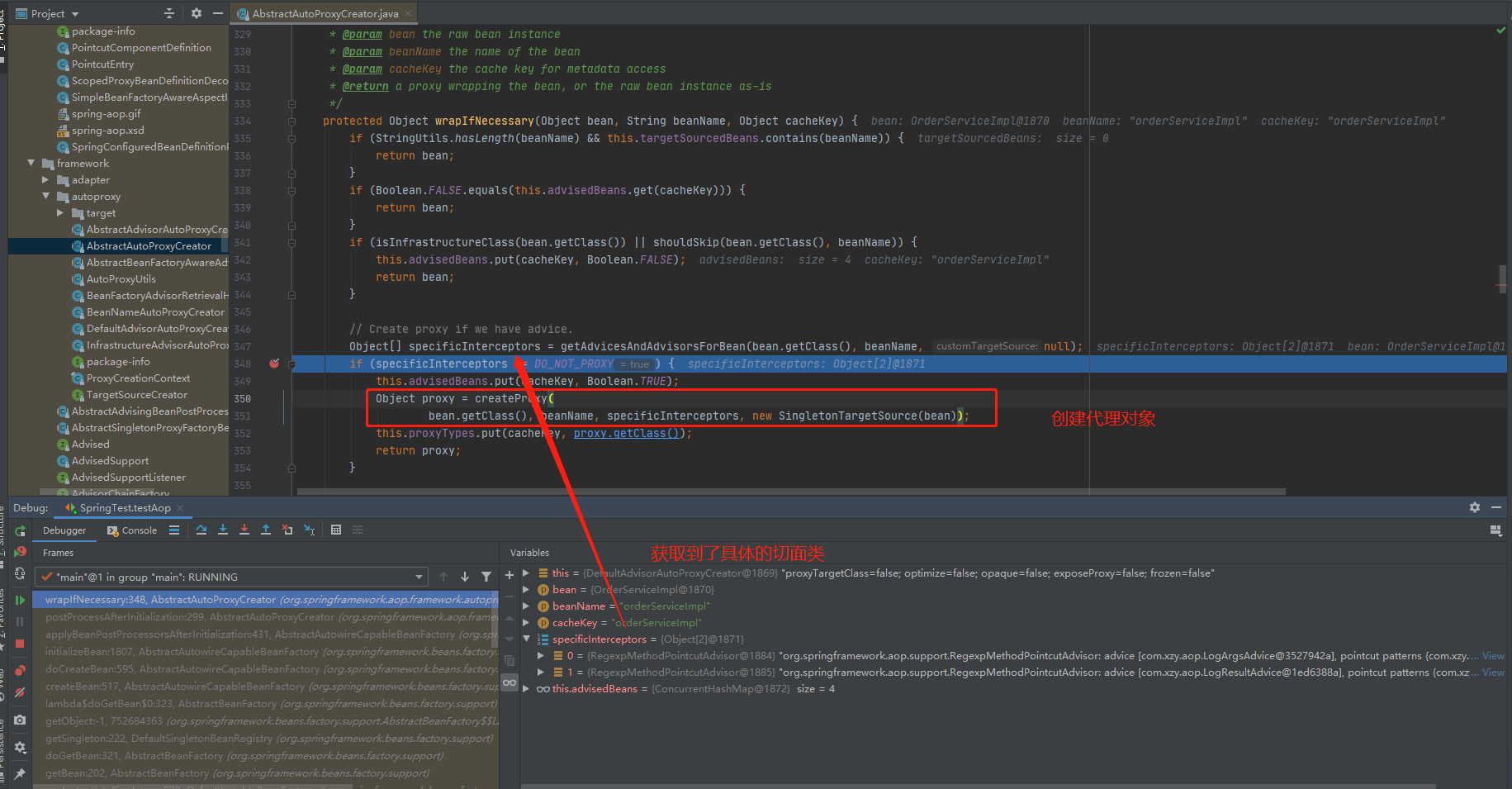
具体我们找到了，在其父类

org.springframework.aop.framework.autoproxy.AbstractAutoProxyCreator.java中，通过

postProcessAfterInitialization方法将初始化后的bean转成代理对象

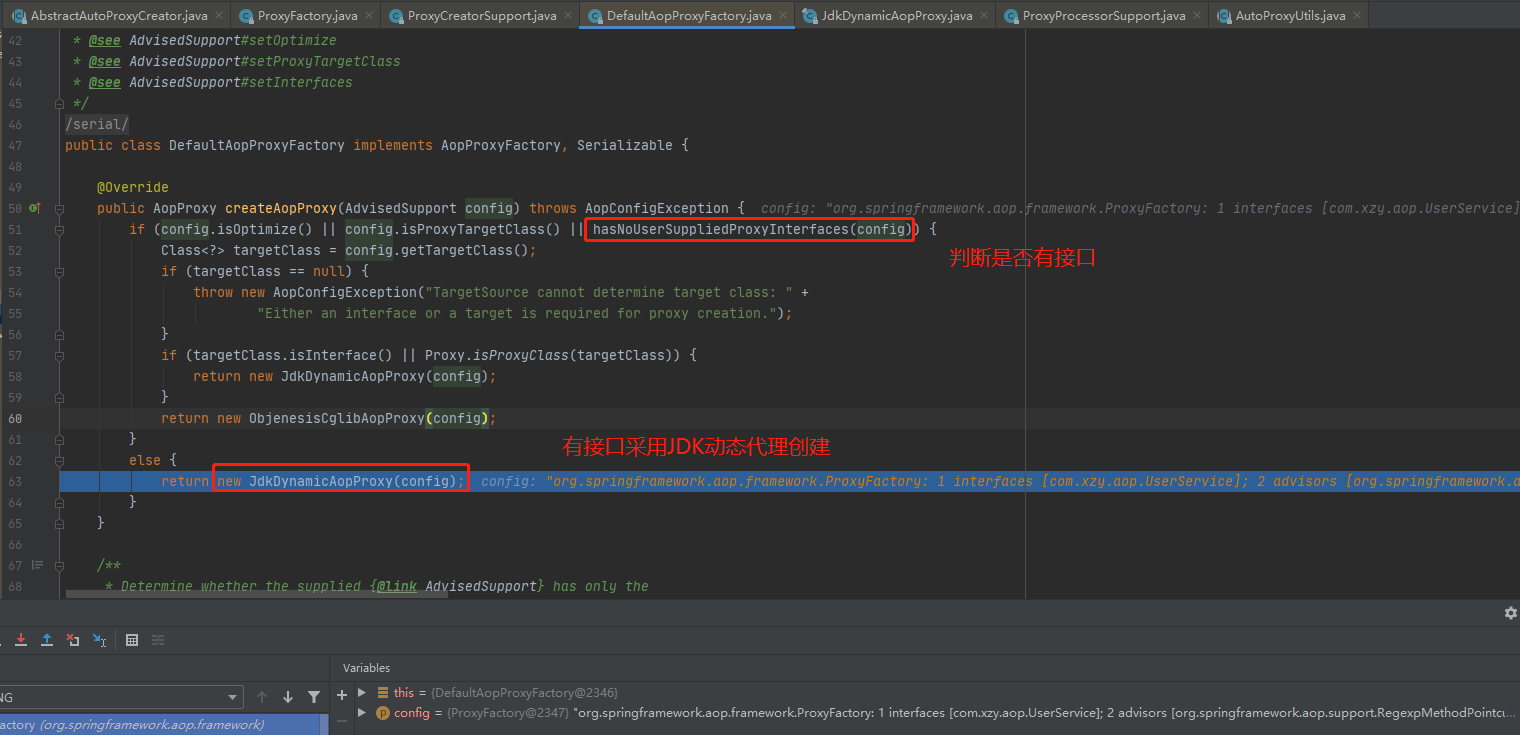


具体分析wrapIfNecessary方法



最后到了是使用cglib还是JDK创建代理的方法中，通过判断是否有接口来决定按照那种方式创建代理

org\springframework\aop\framework\DefaultAopProxyFactory.java



### <aop:aspectj-autoproxy/>是如何实现将其转成代理的呢?



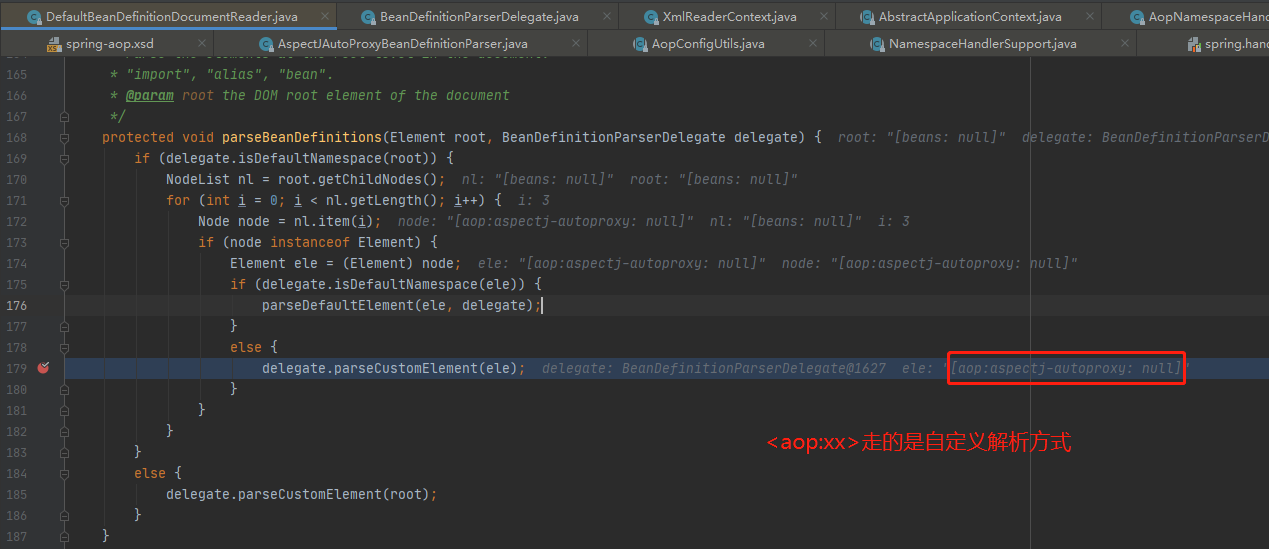
#### 具体原理

类似我们上面讲的

**<bean class="org.springframework.aop.framework.autoproxy.DefaultAdvisorAutoProxyCreator"></bean>**

这个类，<aop:aspectj-autoproxy/>的原理也是通过这个自定义的标签帮我们注入了AnnotationAwareAspectJAutoProxyCreator类，功能与DefaultAdvisorAutoProxyCreator类似，只是他是通过注解来扫描。

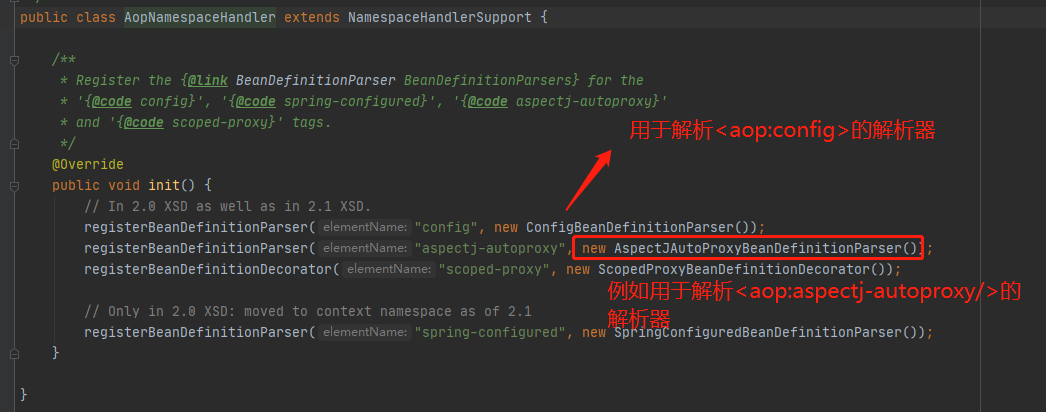
因为<aop>是属于自定义标签，所以在解析时走自定义解析方法



由于自定义标签一般包括如下几个过程:

1. 创建一个需要扩展的组件
2. 定义一个 XSD 文件，用于描述组件内容
3. 创建一个实现 AbstractSingleBeanDefinitionParser 接口的类，用来解析 XSD 文件中的定义和组件定义
4. 创建一个 Handler，继承 NamespaceHandlerSupport ，用于将组件注册到 Spring 容器
5. 编写 Spring.handlers 和 Spring.schemas 文件

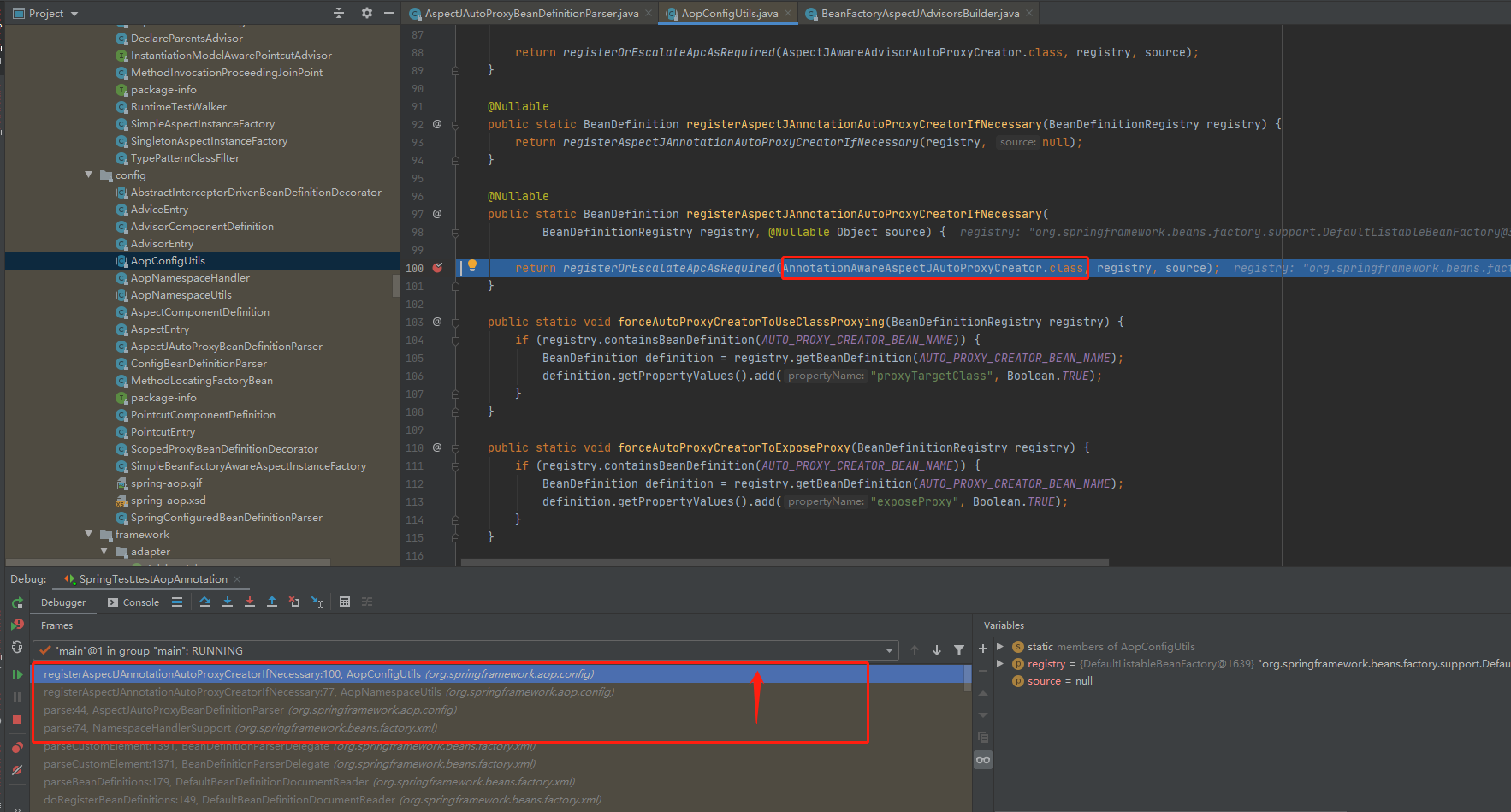
所以根据spring.handlers中可以知道，对于aop的标签在一开始启动时采用了org.springframework.aop.config.AopNamespaceHandler注册了标签的解析器。



然后根据代码org.springframework.aop.config.AspectJAutoProxyBeanDefinitionParser，**这个解析器帮我们自动注入了**

**org.springframework.aop.aspectj.annotation.AnnotationAwareAspectJAutoProxyCreator类，这个类跟我们将AOP原理时候的下面这个**

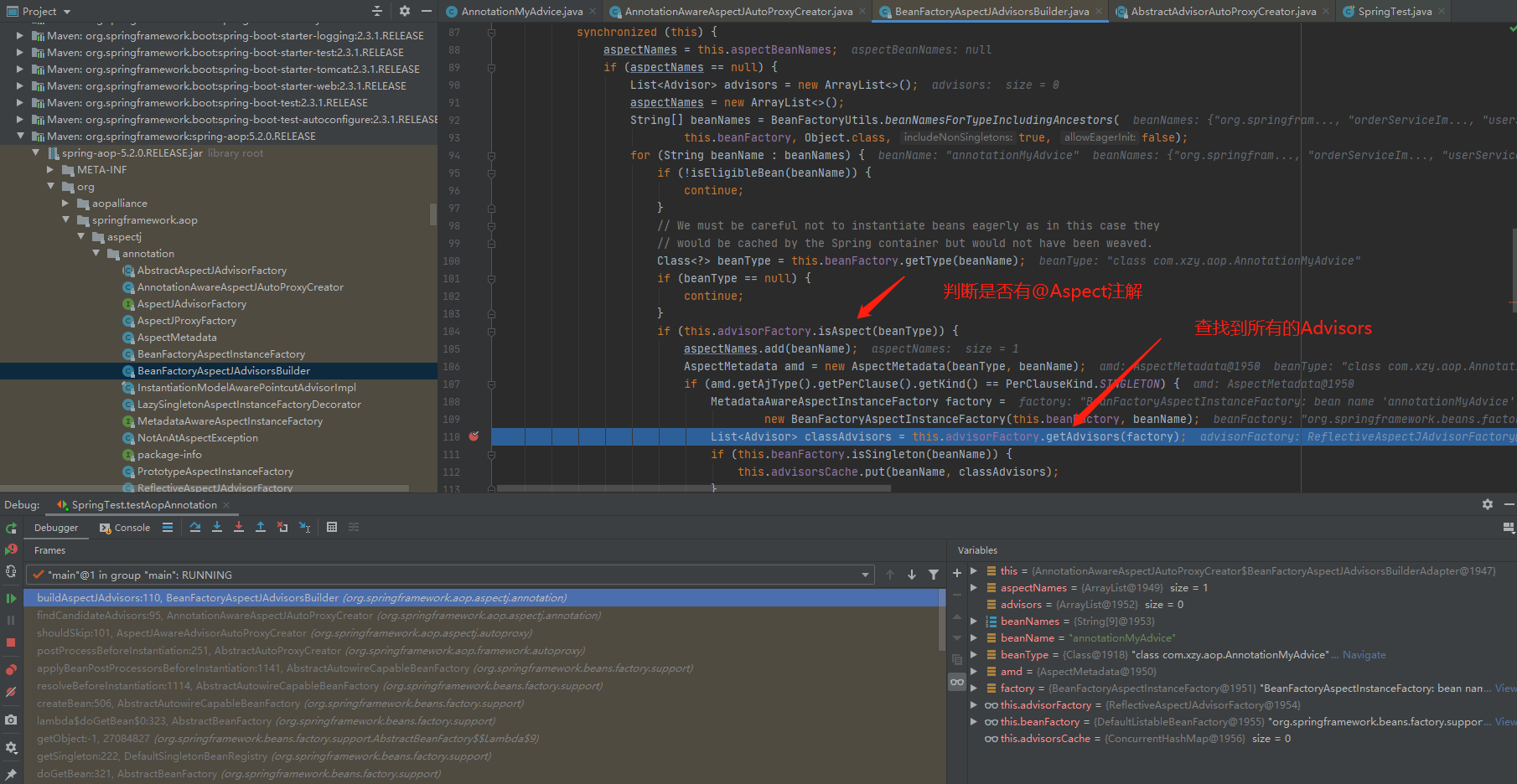
**org.springframework.aop.framework.autoproxy.DefaultAdvisorAutoProxyCreator类似，就是一个后处理器**



那具体AnnotationAwareAspectJAutoProxyCreator是如何解析注解@Aspect来获取对应的切面的呢?

具体方法在

org\springframework\aop\aspectj\annotation\BeanFactoryAspectJAdvisorsBuilder.buildAspectJAdvisors()方法中



### 事务是如何通过AOP来实现的

#### 概述

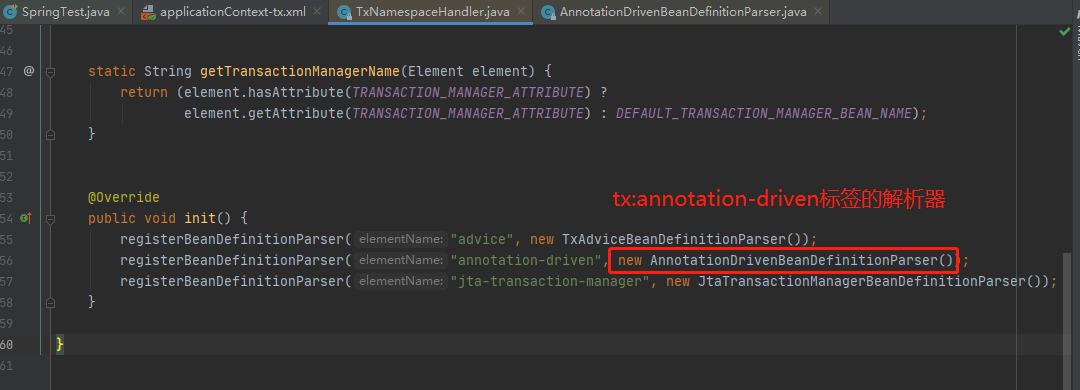
事务也是通过AOP的方式，使用代理的方式在方法前后注入增强来实现事务的回滚等操作

#### 具体原理

首先配置文件有<tx:annotation-driven/>，这个标签是事务的关键。在解析配置文件时，这个标签是自定义标签，所以在TxNamespaceHandler.java中初始化了annotation-driven标签使用

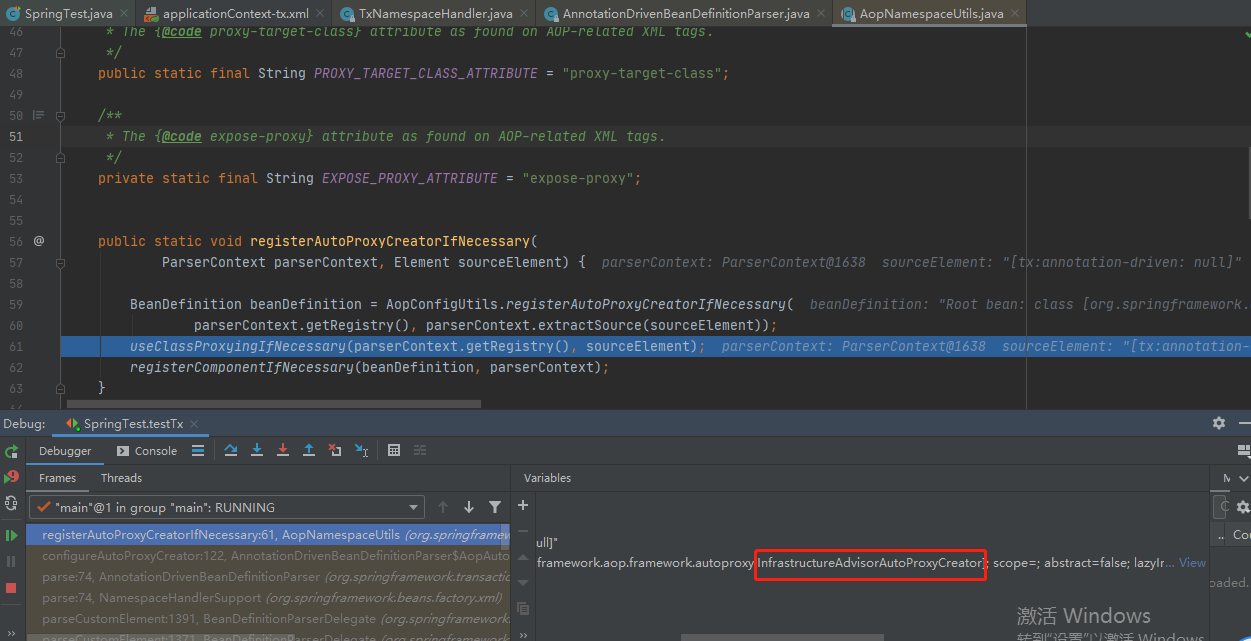
AnnotationDrivenBeanDefinitionParser.java进行解析

**在遇到诸如tx:annotation-driven为开头的配置后，Spring都会使用AnnotationDrivenBeanDefinitionParser类的parse方法进行解析**

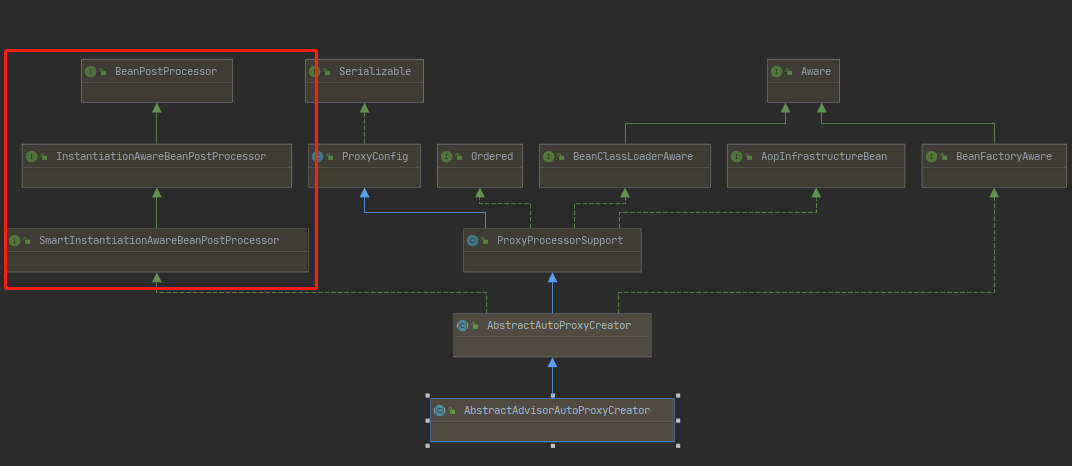


接下去看下面两图，可以发现，解析时注册了**InfrastructureAdvisorAutoProxyCreator.java**





XxxProxyCreator.java这种后缀的应该有印象吧，就跟DefaultAdvisorAutoProxyCreator.java功能一样，不懂可以看看上面AOP的原理，这里再说一次，无非他就是一个处理器，在初始化后将bean加上Advices(增强-也可以说是拦截，这里就是事务的怎强)后形成一个代理对象



回到刚AnnotationDrivenBeanDefinitionParser.java中的

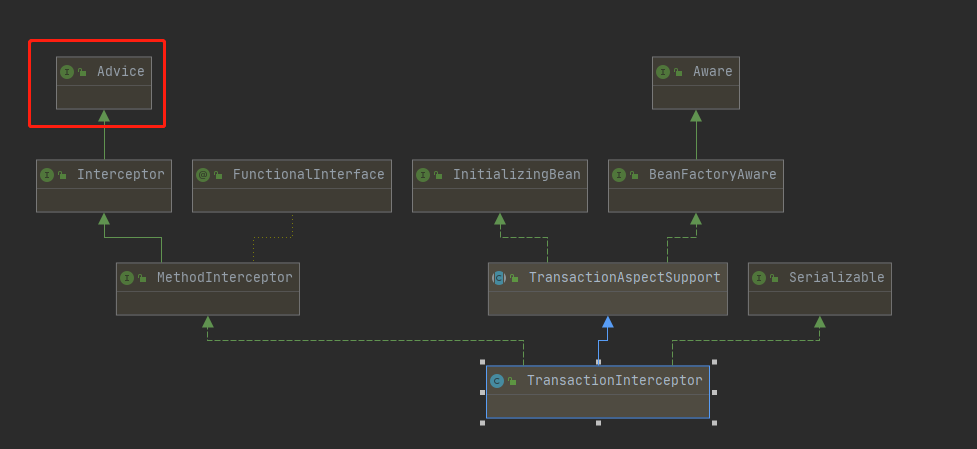
AopAutoProxyConfigurer#configureAutoProxyCreator方法，除了注入了

InfrastructureAdvisorAutoProxyCreator.java，我们想一下，现在已经可以将其转成代理类了，但是还缺少Advisor。所以接下去还注册了3个bean.

BeanFactoryTransactionAttributeSourceAdvisor.java -- 切点

TransactionInterceptor.java --- 上面的Advisor需要用到的 advice-增强

AnnotationTransactionAttributeSource.java - 它是基于注解驱动的事务管理的事务属性源。它的基本作用为：它遇上比如@Transaction标注的方法时，此类会分析此事务注解，最终组织形成一个TransactionAttribute供随后的调用

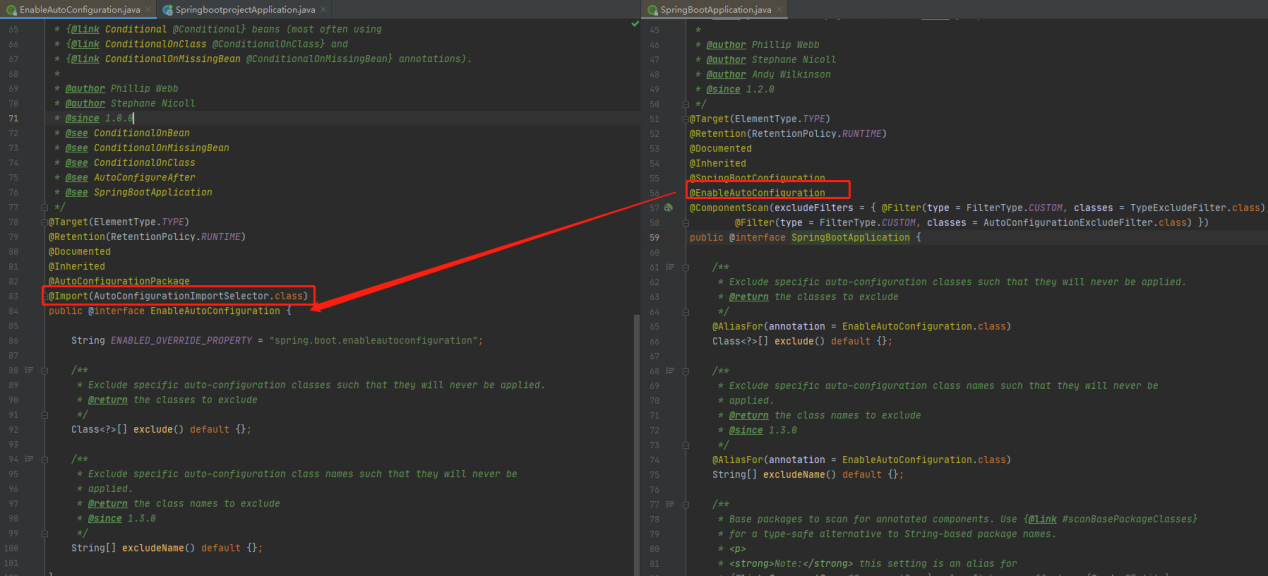


### SpringBoot自动配置原理

#### 原理

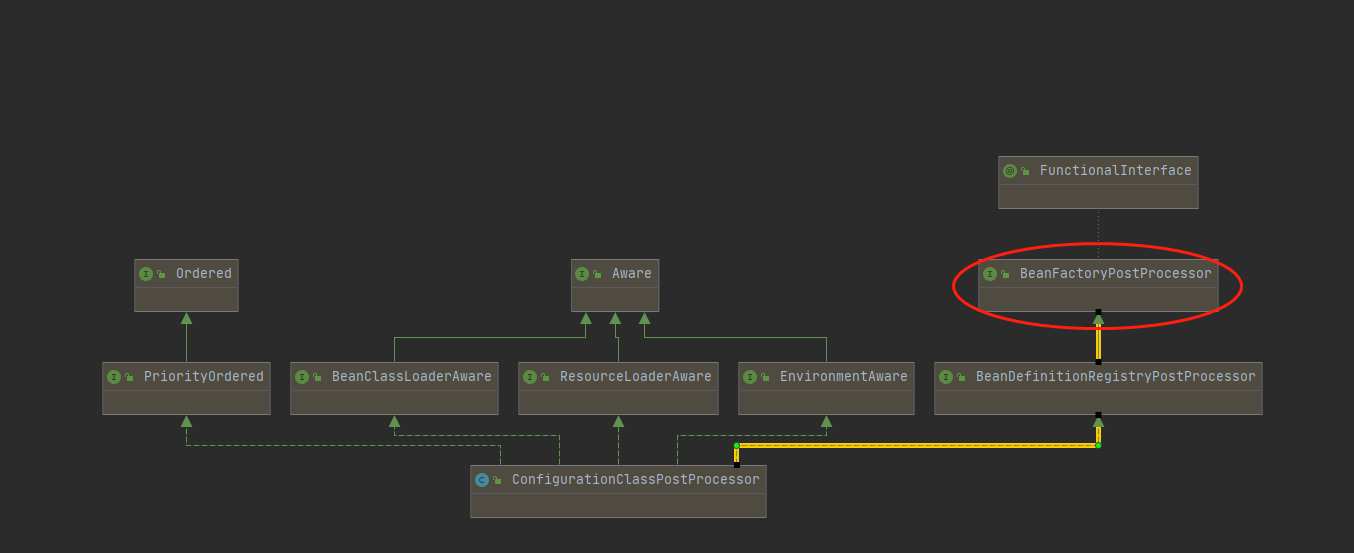
Spring [Boot启动的时候会通过@EnableAutoConfiguration注解找到META-INF/spring.factories配置文件中的所有自动配置类，并对其进行加载，而这些自动配置类都是以AutoConfiguration结尾来命名的，它实际上就是一个JavaConfig形式的Spring容器配置类。](mailto:Boot启动的时候会通过@EnableAutoConfiguration注解找到META-INF/spring.factories配置文件中的所有自动配置类，并对其进行加载，而这些自动配置类都是以AutoConfiguration结尾来命名的，它实际上就是一个JavaConfig形式的Spring容器配置类。)

首先在启动类上有注释@SpringBootApplication，这是一个复合注释类，里面包含了@EnableAutoConfiguration，借助这个注解，SpringBoot用来收集和注册特定场景相关bean的定义。@EnableAutoConfiguration借助@Import的帮助，将所有符合自动配置条件的bean定义加载到IoC容器



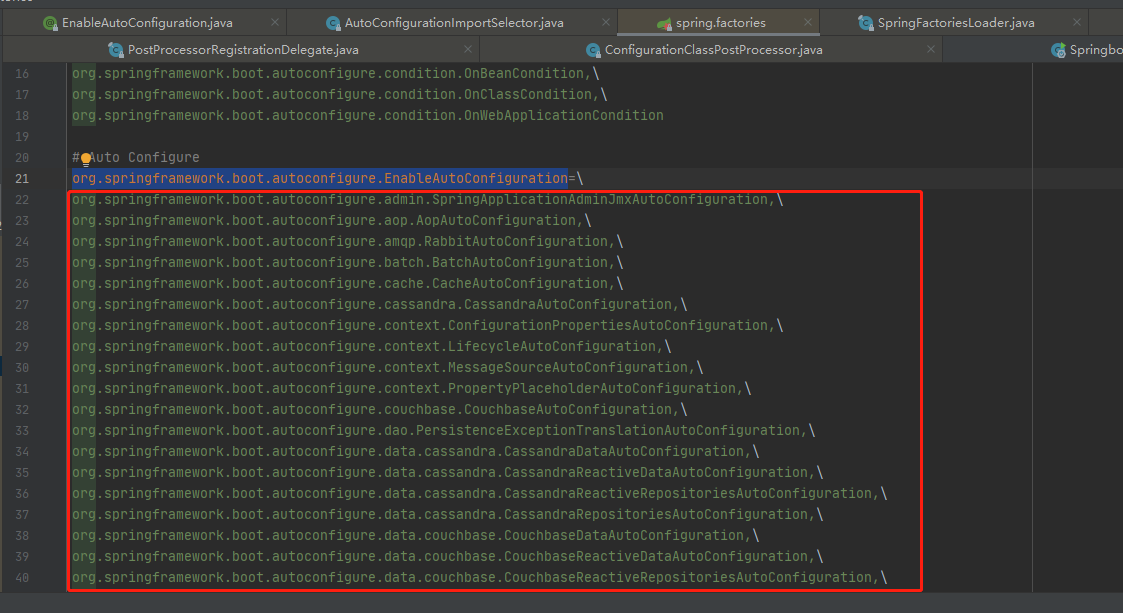
在SpringBoot中有一个关键的类ConfigurationClassPostProcessor.java，实现了

BeanFactoryPostProcessor后工厂处理器

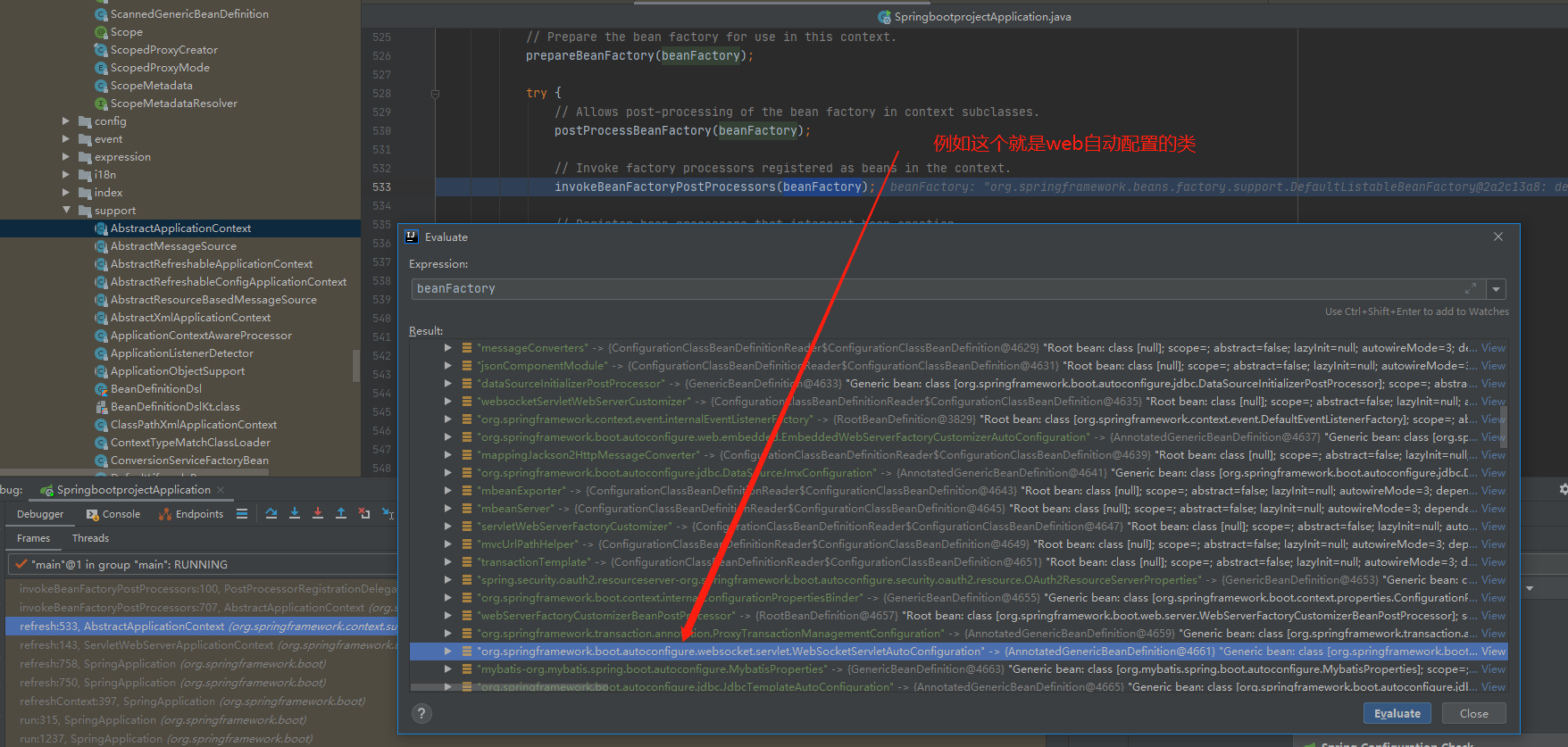


所以在beanFactory初始化后会调用其方法postProcessBeanDefinitionRegistry，最终会执行到

org.springframework.boot.autoconfigure.AutoConfigurationImportSelector#getCandidateConfigurations方法中，从META-INF/spring.factories文件中获取org.springframework.boot.autoconfigure.EnableAutoConfiguration对应的类，这些**类的特点就是文件名后缀都是XXXAutoConfiguration，且包含了注解@Configuration，说明他们也是一个配置类**



获取到后就将springboot给我们提供的这100多个类注册到beanFactory中，之后Spring在加载bean的时候会根据这些类去加载对应的bean。



#### 究竟是怎么实现加一个start就自动加载对应的类到spring中呢？

分析类ServletWebServerFactoryAutoConfiguration.java.首先他是一个配置类，因为里面有注解@Configuration(proxyBeanMethods = false)

[[@Configuration会通过ConfigurationClassPostProcessor.java来处理注入]](mailto:[@Configuration会通过ConfigurationClassPostProcessor.java来处理注入])

而且springboot提供了多个条件依赖注解，只要在特定情况下还会帮我们注入，而不是META-INF/spring.factorie有就一定注入

@ConditionalOnBean，仅在当前上下文中存在某个bean时，才会实例化这个Bean。

@ConditionalOnClass，某个class位于类路径上，才会实例化这个Bean。

@ConditionalOnExpression，当表达式为true的时候，才会实例化这个Bean。

@ConditionalOnMissingBean，仅在当前上下文中不存在某个bean时，才会实例化这个Bean。

@ConditionalOnMissingClass，某个class在类路径上不存在的时候，才会实例化这个Bean。

@ConditionalOnNotWebApplication，不是web应用时才会实例化这个Bean。

@AutoConfigureAfter，在某个bean完成自动配置后实例化这个bean。

@AutoConfigureBefore，在某个bean完成自动配置前实例化这个bean。

到此，解释了为啥在pom.xml加了一个spring-boot-starter-xxx后能自动注入的原理。

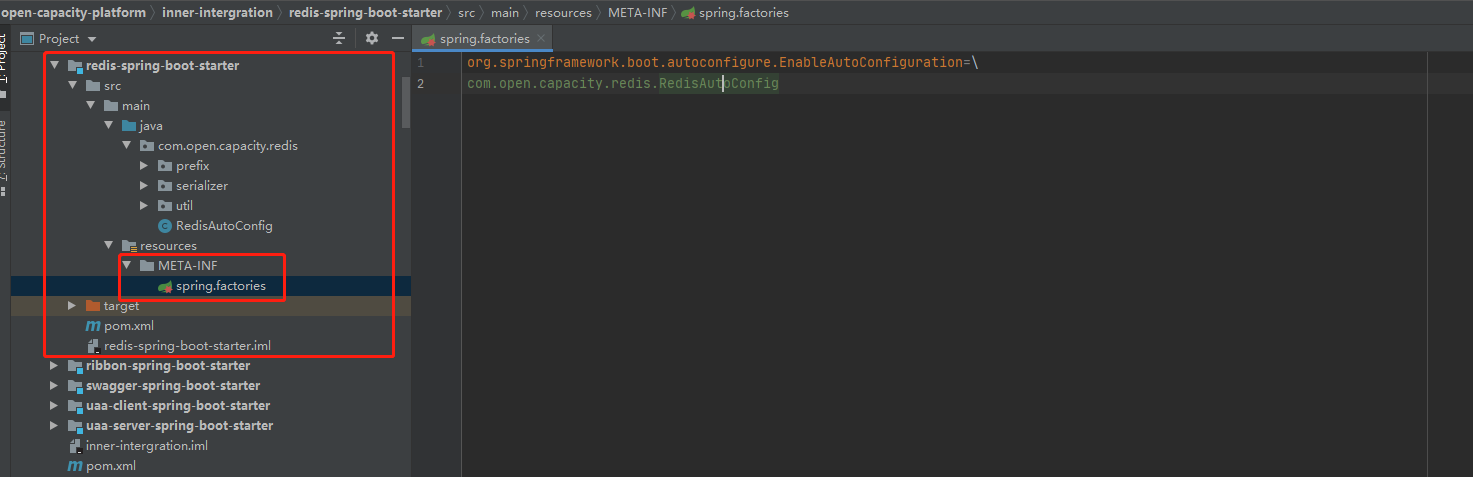
#### 如何自定义开发一个starter

假设现在主要开发一个redis工具类的starter，在pom.xml中定义后能自动注入redis的相关bean。

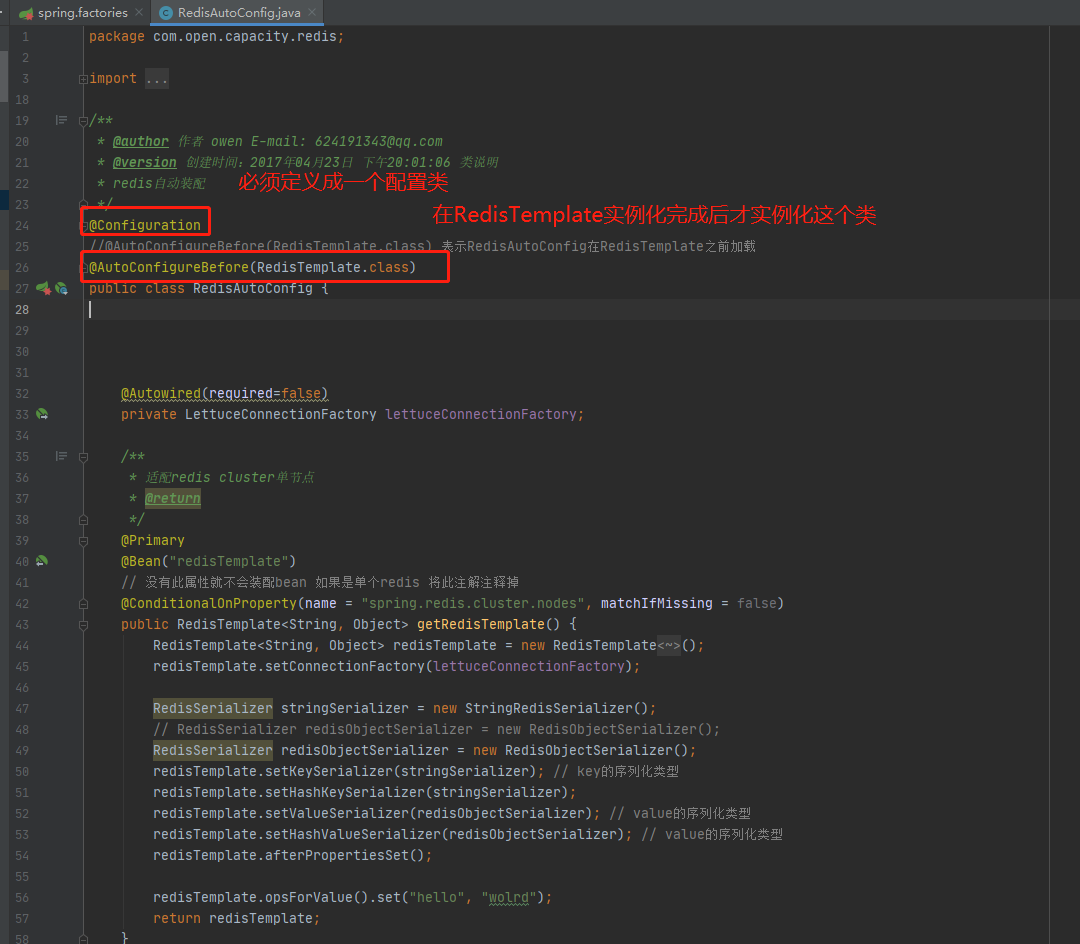
新建一个maven项目，结构如下，且添加一个spring.factories。且设置

org.springframework.boot.autoconfigure.EnableAutoConfiguration=\

com.open.capacity.redis.RedisAutoConfig



编写RedisAutoConfig.java类



### Spring中@Configuration和@Bean

Lite模式：在没有标注@Configuration的类里面有@Bean方法就称为Lite模式的配置。以下情况都是属于Lite模式，这个模式的优点是：运行时不再需要给对应类生成CGLIB子类，提高了运行性能，降低了启动时间。缺点是：不能声明@Bean之间的依赖，也就是说不能通过方法调用来依赖其它Bean。

类上标注有@Component注解

类上标注有@ComponentScan注解

类上标注有@Import注解

类上标注有@ImportResource注解

若类上没有任何注解，但类内存在@Bean方法

Full模式：标注有@Configuration注解的类被称为full模式的配置类。优点：可以支持通过常规Java调用相同类的@Bean方法而保证是容器内的Bean，所以这种方式是单例。缺点：运行时会给该类生成一个CGLIB子类放进容器，有一定的性能、时间开销。