

Spring注解驱动开发第29讲——注册完AnnotationAwareAspectJAutoProxyCreator后置处理器之后，就得完成BeanFactory的初始化工作了

文章目录

写在前面

完成BeanFactory的初始化工作

完成BeanFactory的初始化工作的第一步

完成BeanFactory的初始化工作的第二步

写在前面

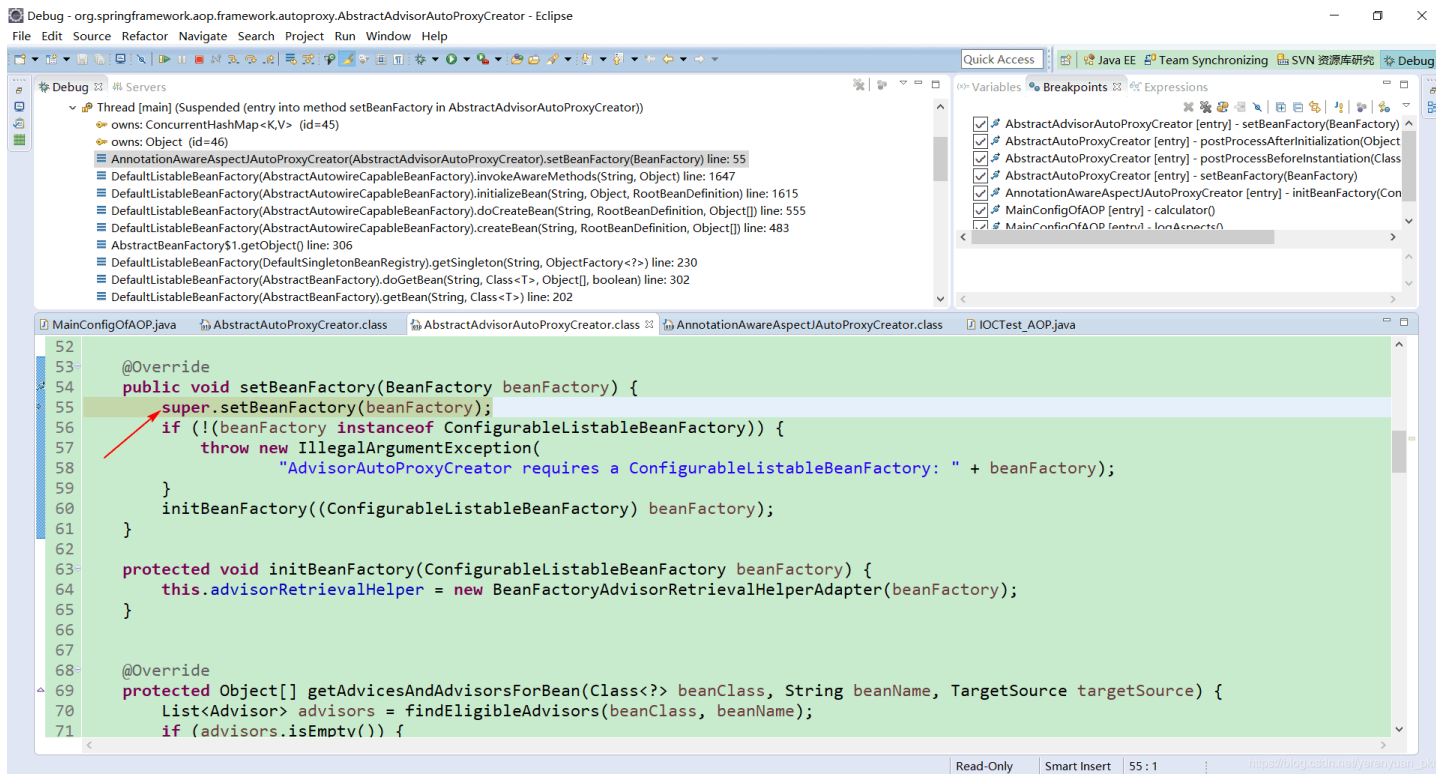
经过前面的研究，我们发现，**IOC容器** 在创建对象的时候，会注册这一些后置处理器，而在上一讲中，就已经把AnnotationAwareAspectJAutoProxyCreator这个后置处理器创建出来了，它呢，就是@EnableAspectJAutoProxy注解利用AspectJAutoProxyRegistrar给容器中创建出的一个bean的配置信息。

当然了，在注册后置处理器的时候，这个bean肯定就已经提前创建出来了。而且，它呢，我们也都知道是一个后置处理器，只要这个后置处理器已经创建出来并且放在容器中，那么以后在创建其他组件的时候，它就可以拦截到这些组件的创建过程了。因为我们知道，任何组件在创建bean的实例时，都会经历给bean中的各种属性赋值、**初始化** bean（并且在初始化bean前后都会有后置处理器的作用）等过程。

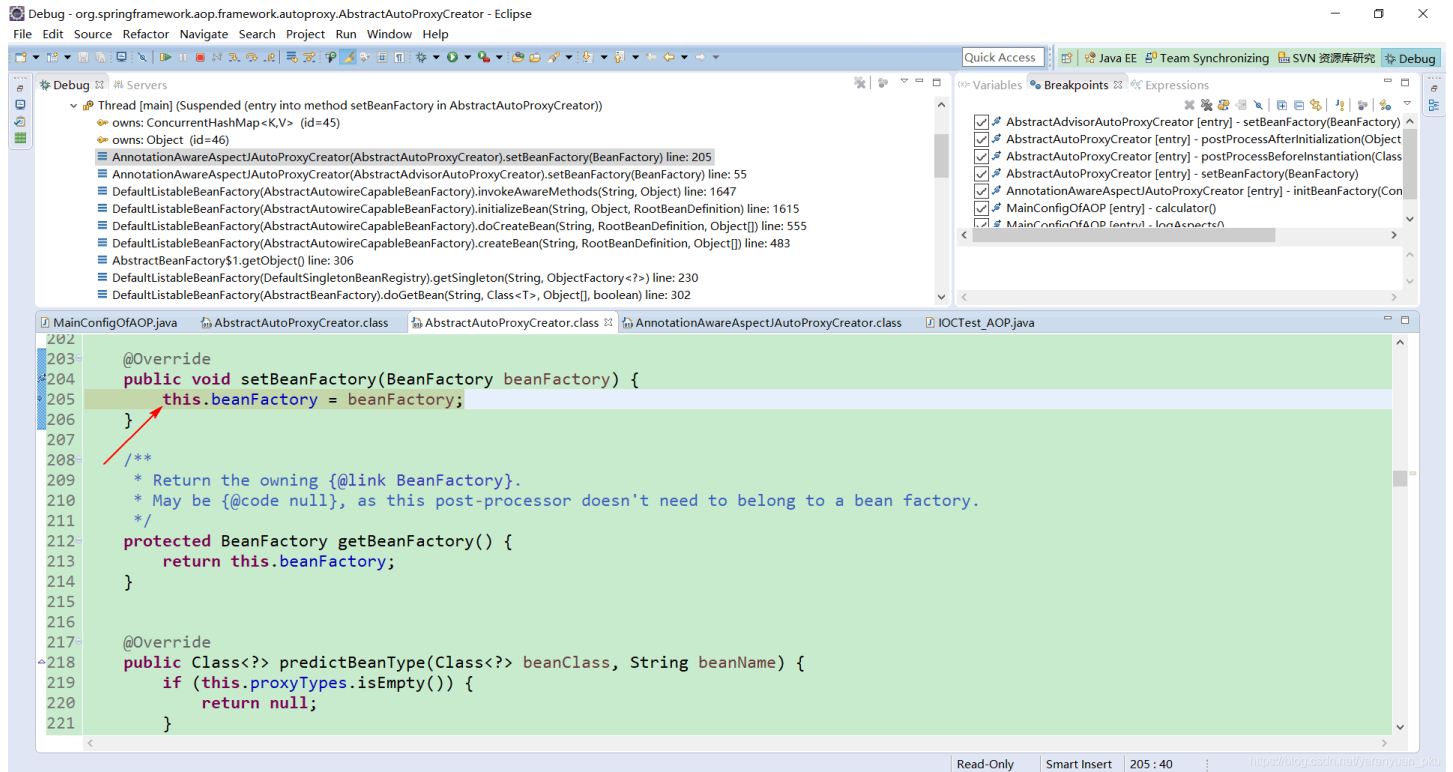
在这一讲中，我们就来看一下AnnotationAwareAspectJAutoProxyCreator作为后置处理器，被注册完之后，接下来又会做些什么事？算了，这里我就明说了吧！注册完AnnotationAwareAspectJAutoProxyCreator后置处理器之后，接下来就得完成BeanFactory的初始化工作了。

完成BeanFactory的初始化工作

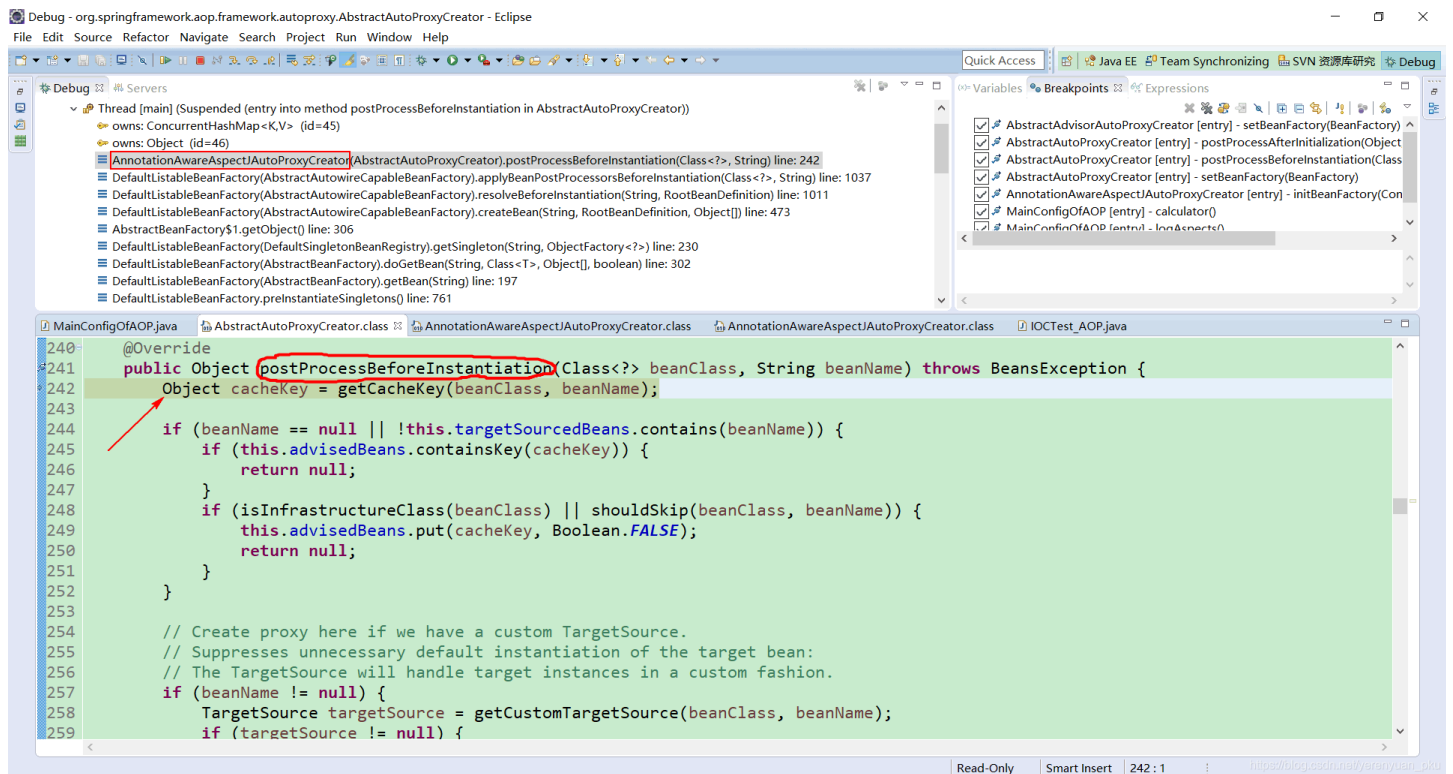
我们还是以debug模式来运行IOCTest_AOP测试类，这时，应该还是会来到AbstractAdvisorAutoProxyCreator类的setBeanFactory()方法中，如下图所示。



在上一讲中，我们是从test01()方法开始一步一步研究慢慢分析到这儿的，所以这儿就不再重复地讲一遍了。那接下来该怎么办呢？我们按下 **F8** 快捷键直接运行到下一个断点，如下图所示，可以看到现在是定位到了AbstractAutoProxyCreator抽象类的setBeanFactory()方法中。



然后继续按下 **F8** 快捷键运行直到下一个断点，一直运行到如下图所示的这行代码处。



可以看到程序现在是停留在了AbstractAutoProxyCreator类的postProcessBeforeInstantiation()方法中，不过从方法调用栈中我们可以清楚地看到现在其实调用的是AnnotationAwareAspectJAutoProxyCreator的postProcessBeforeInstantiation()方法。

这个方法大家一定要引起注意，它跟我们之前经常讲到的后置处理器中的方法是有区别的。你不妨看一下BeanPostProcessor接口的源码，如下图所示，它里面有一个postProcessBeforeInitialization()方法。

```
41 */
42 public interface BeanPostProcessor {
43
44     /**
45      * Apply this BeanPostProcessor to the given new bean instance <i>before</i> any bean
46      * initialization callbacks (like InitializingBean's {@code afterPropertiesSet}
47      * or a custom init-method). The bean will already be populated with property values.
48      * The returned bean instance may be a wrapper around the original.
49      * @param bean the new bean instance
50      * @param beanName the name of the bean
51      * @return the bean instance to use, either the original or a wrapped one;
52      *         if {@code null}, no subsequent BeanPostProcessors will be invoked
53      * @throws org.springframework.beans.BeansException in case of errors
54      * @see org.springframework.beans.factory.InitializingBean#afterPropertiesSet
55      */
56     Object postProcessBeforeInitialization(Object bean, String beanName) throws BeansException;
57
58     /**
59      * Apply this BeanPostProcessor to the given new bean instance <i>after</i> any bean
60      * initialization callbacks (like InitializingBean's {@code afterPropertiesSet}
```

而现在这个方法是叫postProcessBeforeInstantiation，大家可一定要分清哟！

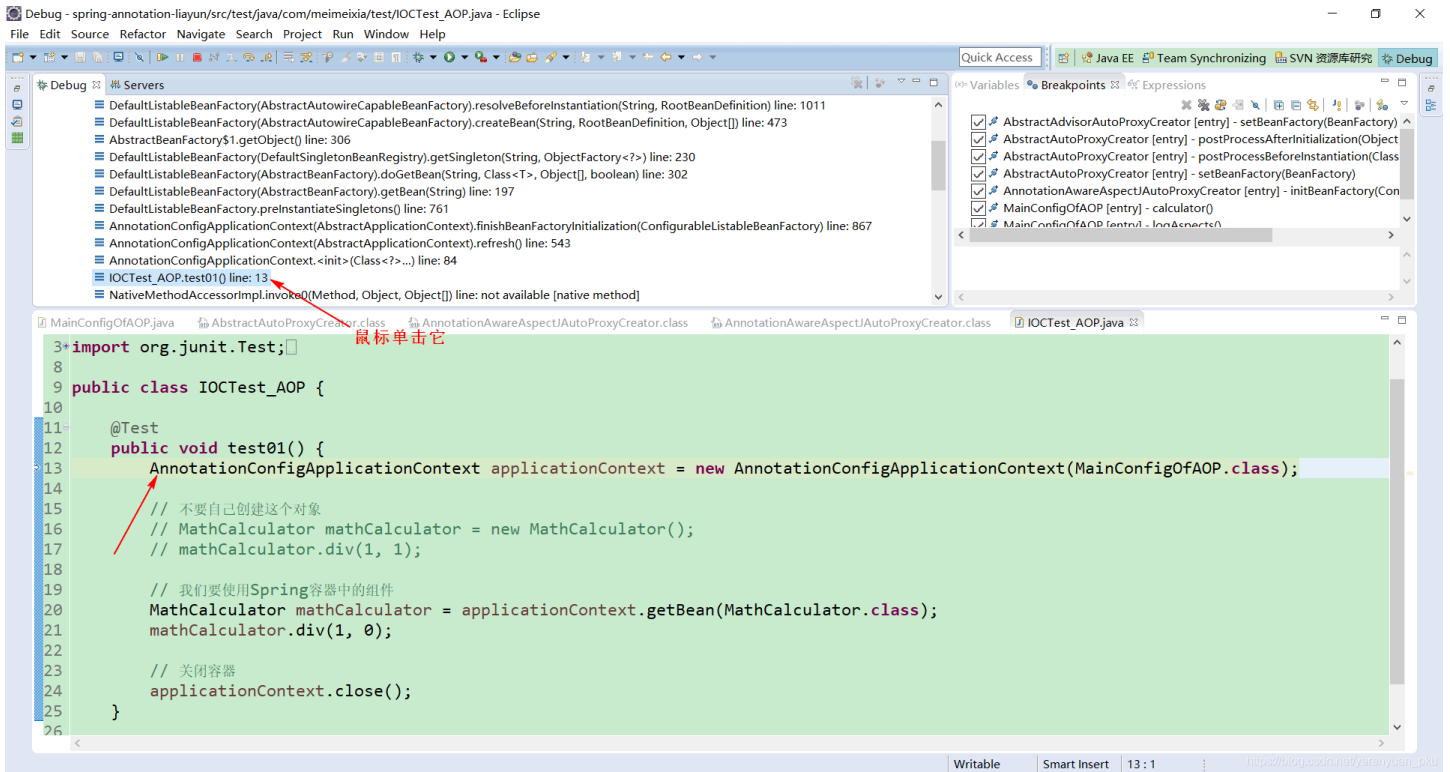
你可能要问了，AnnotationAwareAspectJAutoProxyCreator它本身就是一个后置处理器，为何其中的方法叫postProcessBeforeInstantiation，而不是叫postProcessBeforeInitialization呢？因为后置处理器跟为后置处理器是不一样的，当前我们要用到的这个后置处理器（即AnnotationAwareAspectJAutoProxyCreator）实现的是一个叫SmartInstantiationAwareBeanPostProcessor的接口，而该接口继承的是InstantiationAwareBeanPostProcessor接口（它又继承了BeanPostProcessor接口），也就是说，AnnotationAwareAspectJAutoProxyCreator虽然是一个BeanPostProcessor，但是它却是InstantiationAwareBeanPostProcessor这种类型的，而InstantiationAwareBeanPostProcessor接口中声明的方法就叫postProcessBeforeInstantiation。

```
41 * @author Rod Johnson
42 * @since 1.2
43 * @see org.springframework.aop.framework.autoproxy.AbstractAutoProxyCreator#setCustomTargetSourceCreators
44 * @see org.springframework.aop.framework.autoproxy.target.LazyInitTargetSourceCreator
45 */
46 public interface InstantiationAwareBeanPostProcessor extends BeanPostProcessor {
47
48     /**
49      * Apply this BeanPostProcessor <i>before</i> the target bean gets instantiated</i>.
50      * Object postProcessBeforeInstantiation(Class<?> beanClass, String beanName) throws BeansException;
51
52     /**
53      * Perform operations after the bean has been instantiated, via a constructor or factory method,
54      * but before Spring property population (from explicit properties or autowiring) occurs.
55      * <p>This is the ideal callback for performing custom field injection on the given bean
56      * instance, right before Spring's autowiring kicks in.
57      * @param bean the bean instance created, with properties not having been set yet
58      * @param beanName the name of the bean
59      * @return {@code true} if properties should be set on the bean; {@code false}
60      *         if property population should be skipped. Normal implementations should return {@code true}.
61      *         Returning {@code false} will also prevent any subsequent InstantiationAwareBeanPostProcessor
```

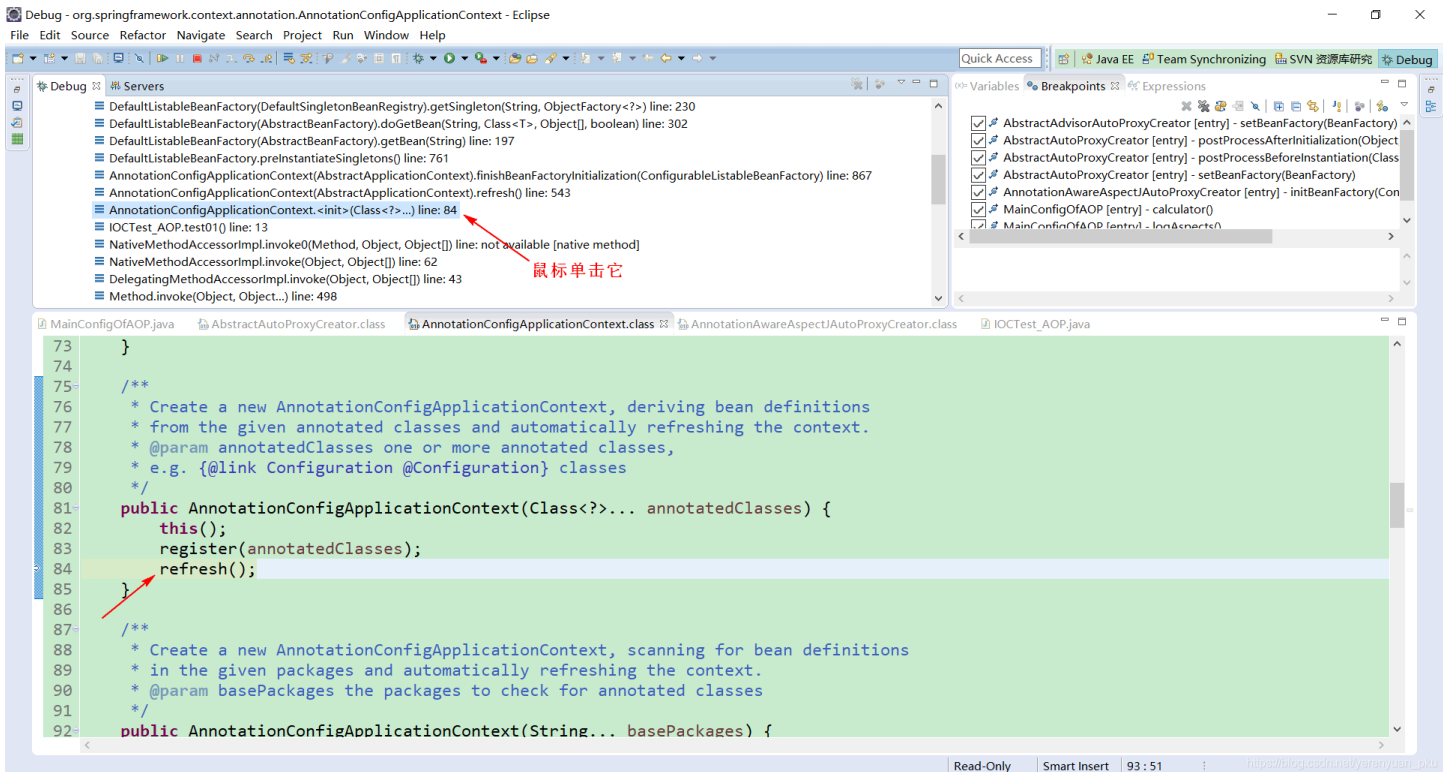
故而程序就停留到了AbstractAutoProxyCreator类的postProcessBeforeInstantiation()方法中。

那么你又问了，为什么会来到这儿呢？我们同样可以仿照前面来大致地来探究一下，在左上角的方法调用栈中，仔细查找，就会在前面找到一个test01()方法，它其实就是IOCTest_AOP测试类中的测试方法，我们就从该方法开始分析。

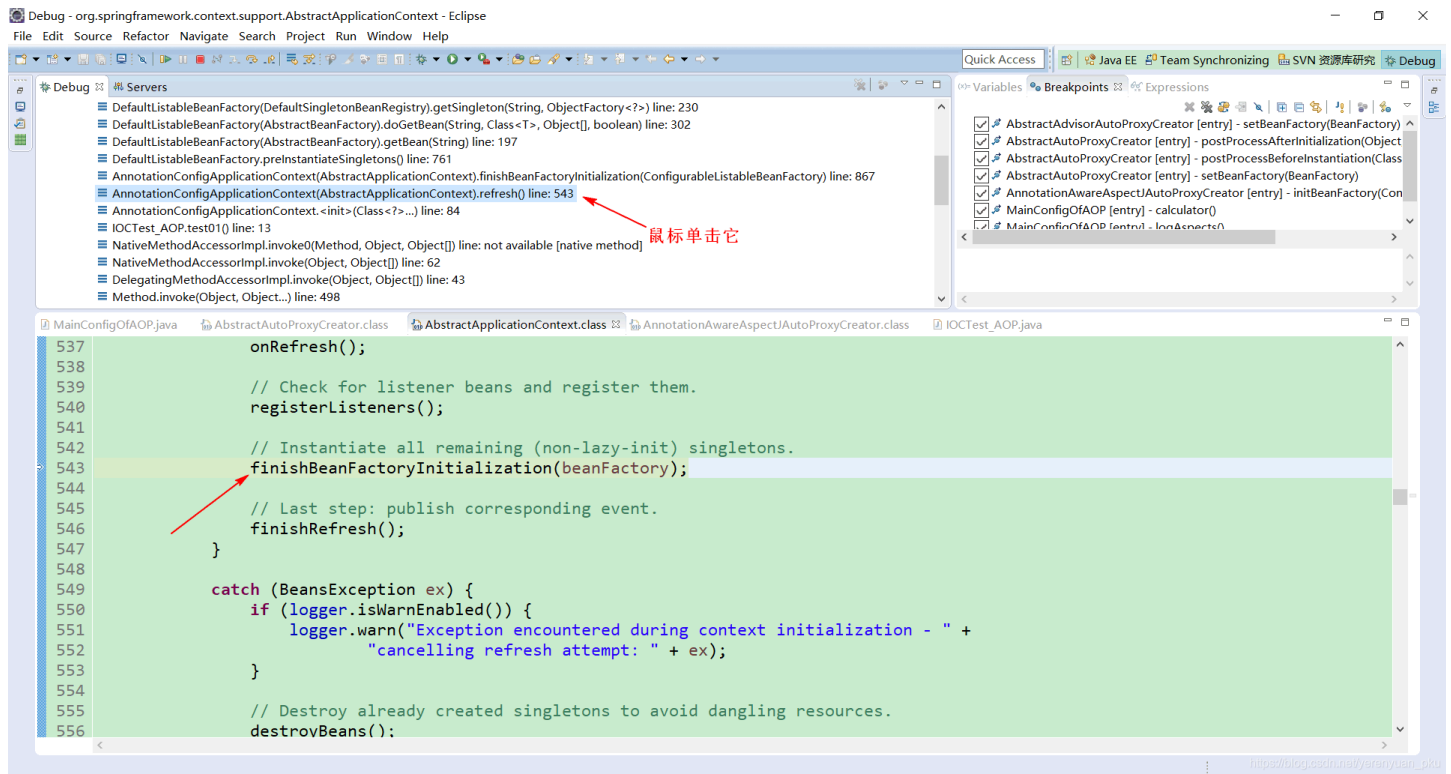
鼠标单击方法调用栈中的那个test01()方法，此时，我们会进入到IOCTest_AOP测试类中的test01()方法中，如下图所示。



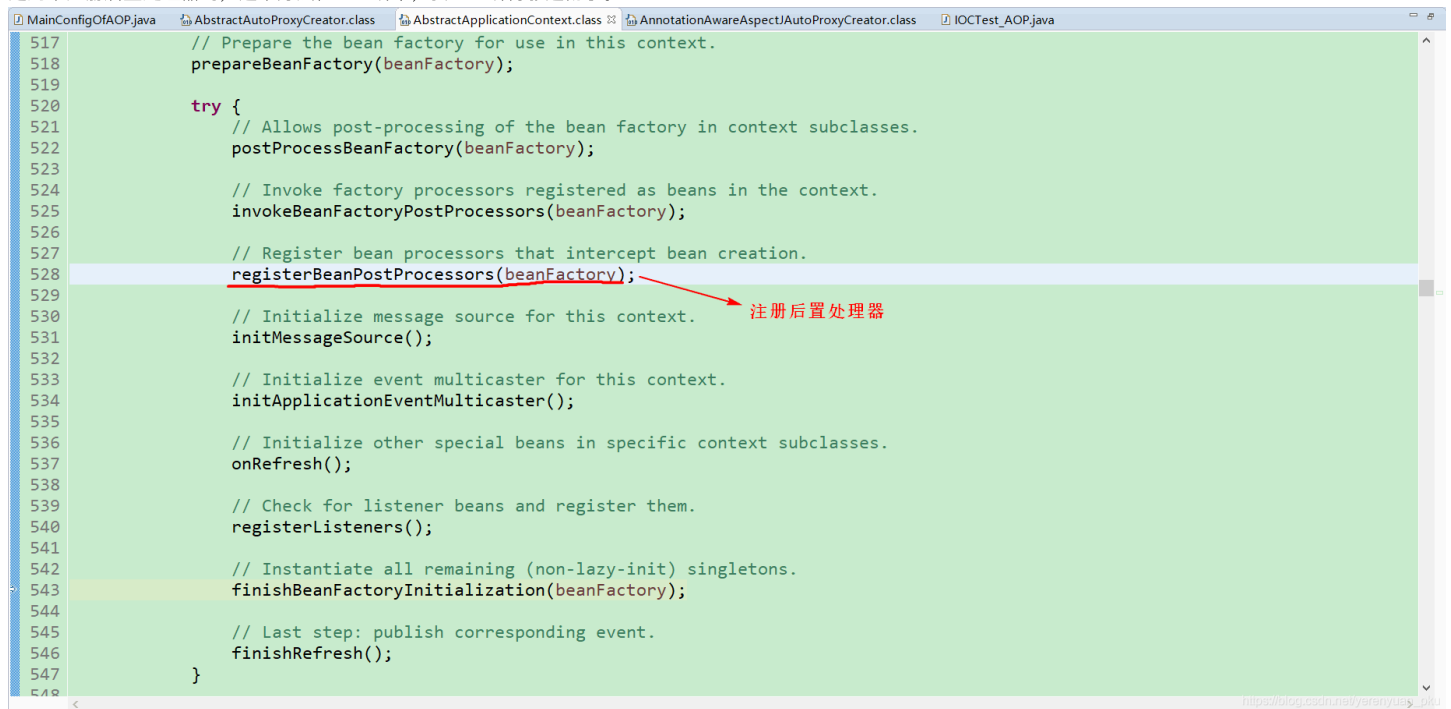
可以看到这一步还是传入主配置类来创建IOC容器，依旧会调用refresh()方法，如下图所示。



我们继续跟进方法调用栈，如下图所示，可以看到现在是定位到了AbstractApplicationContext抽象类的refresh()方法中。

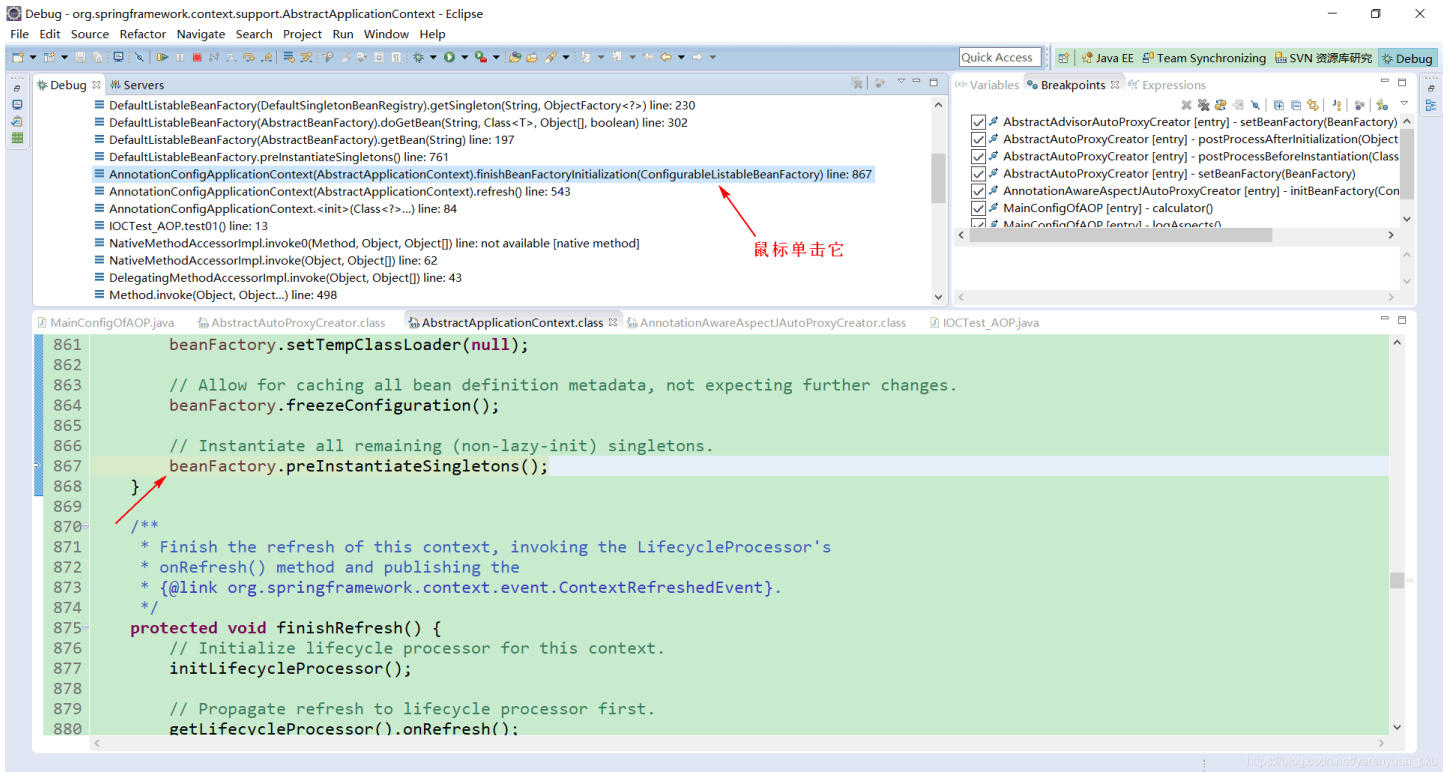


可以看到，在这儿会调用finishBeanFactoryInitialization()方法，这是用来初始化剩下的单实例bean的。而在该方法前面，有一个叫registerBeanPostProcessors的方法，它是用来注册后置处理器的，这个方法在上一讲中，我已经讲得够透彻了。



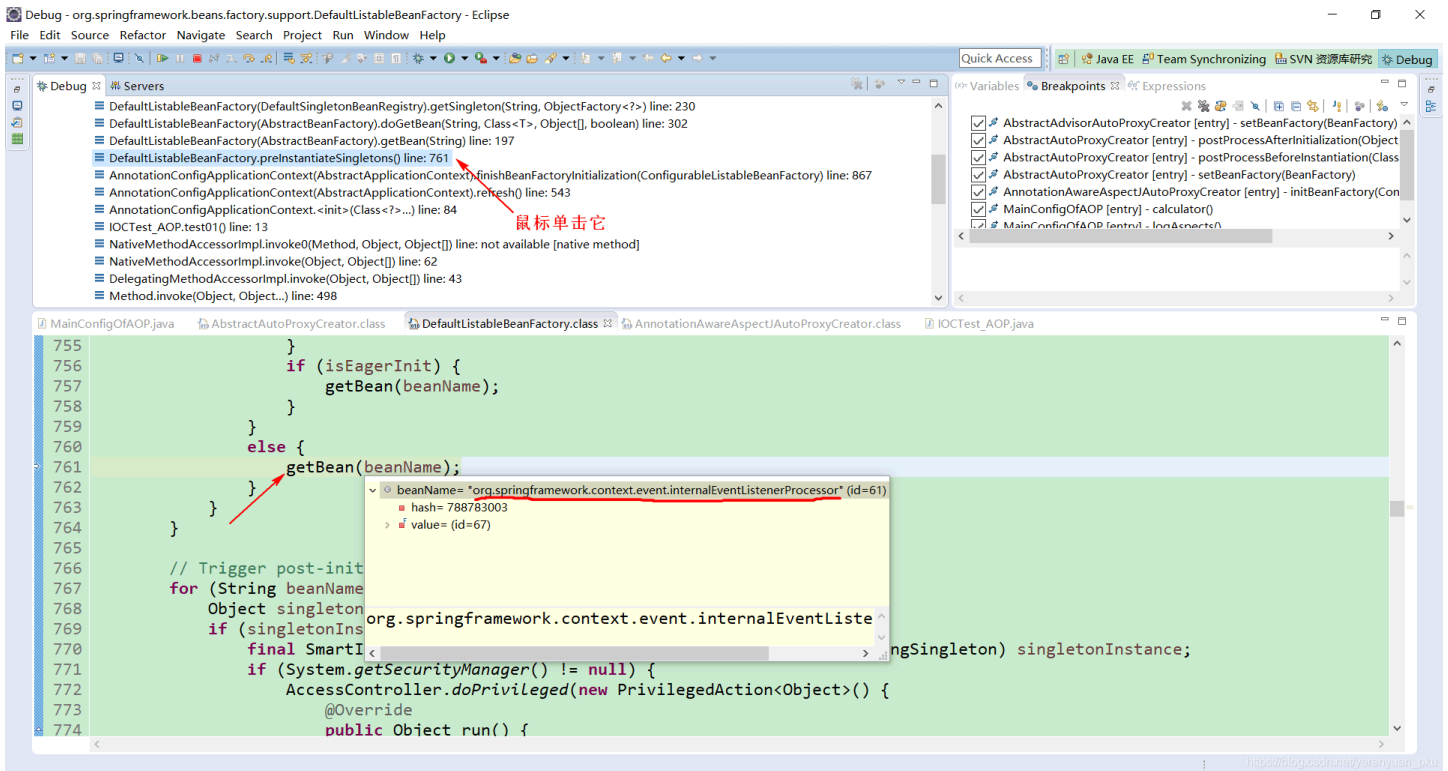
注册完后置处理器之后，接下来就来到了finishBeanFactoryInitialization()方法处，以完成BeanFactory的初始化工作。所谓的完成BeanFactory的初始化工作，其实就是来创建剩下的单实例bean。为什么叫剩下的呢？因为IOC容器中的这些组件，比如一些BeanPostProcessor，早都已经在注册的时候就被创建了，所以会留下一下没被创建的组件，让它们在这儿进行创建。

我们继续跟进方法调用栈，如下图所示，可以看到现在是定位到了AbstractApplicationContext抽象类的finishBeanFactoryInitialization()方法中。



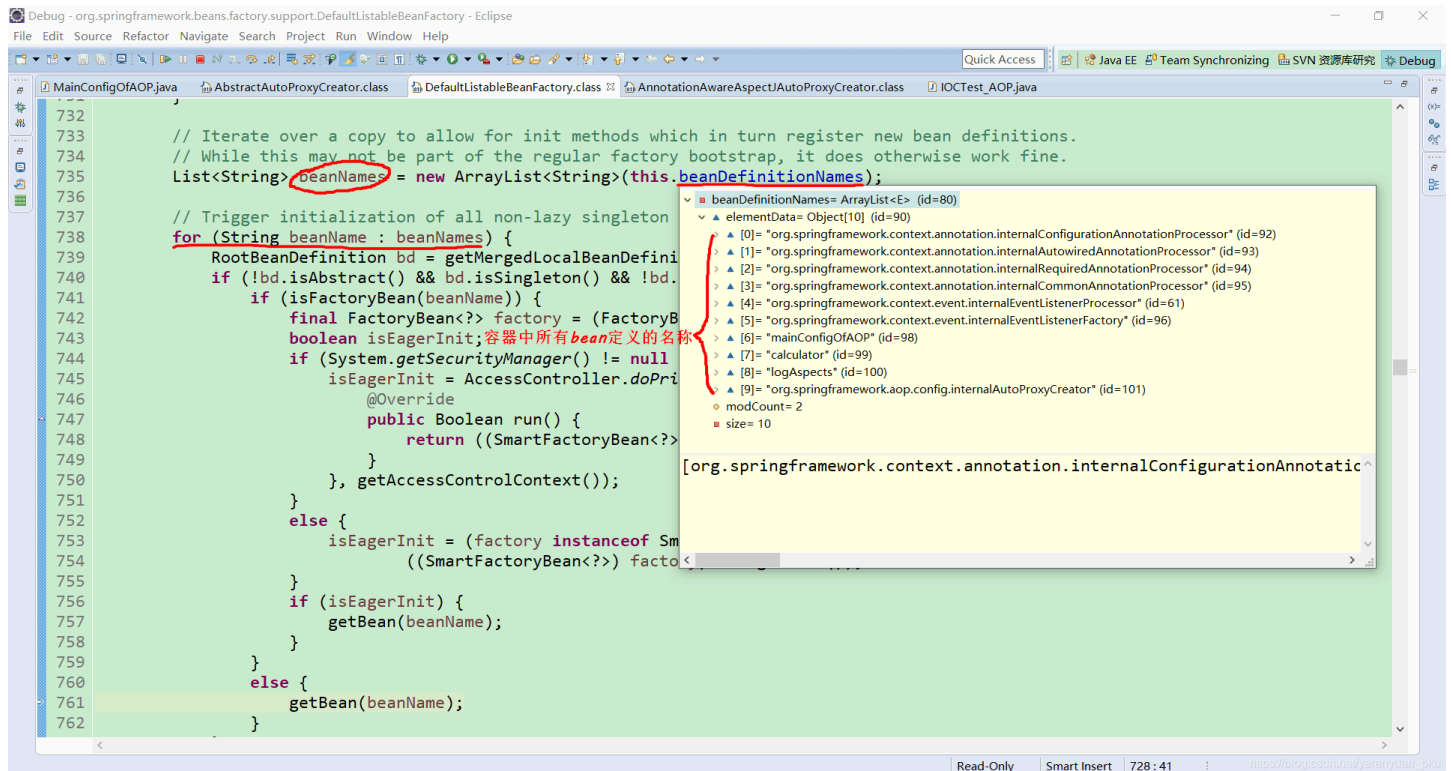
也就是说，这儿会继续调用preInstantiateSingletons()方法来创建剩下的单实例bean。

继续跟进方法调用栈，如下图所示，可以看到现在是定位到了DefaultListableBeanFactory类的preInstantiateSingletons()方法中。



在这儿会调用getBean()方法来获取一个bean，那获取的是哪个bean呢？获取的是名称为 `org.springframework.context.event.internalEventListenerProcessor` 的bean，它跟我们目前的研究没什么关系。

既然没有关系，那为何还要获取这个bean呢？往前翻阅preInstantiateSingletons()方法，可以看到有一个for循环，它是来遍历一个beanNames的List集合的，这个beanNames又是什么呢？很明显它是一个 `List<String>` 集合，它里面保存的是容器中所有bean定义的名称，如下图所示。



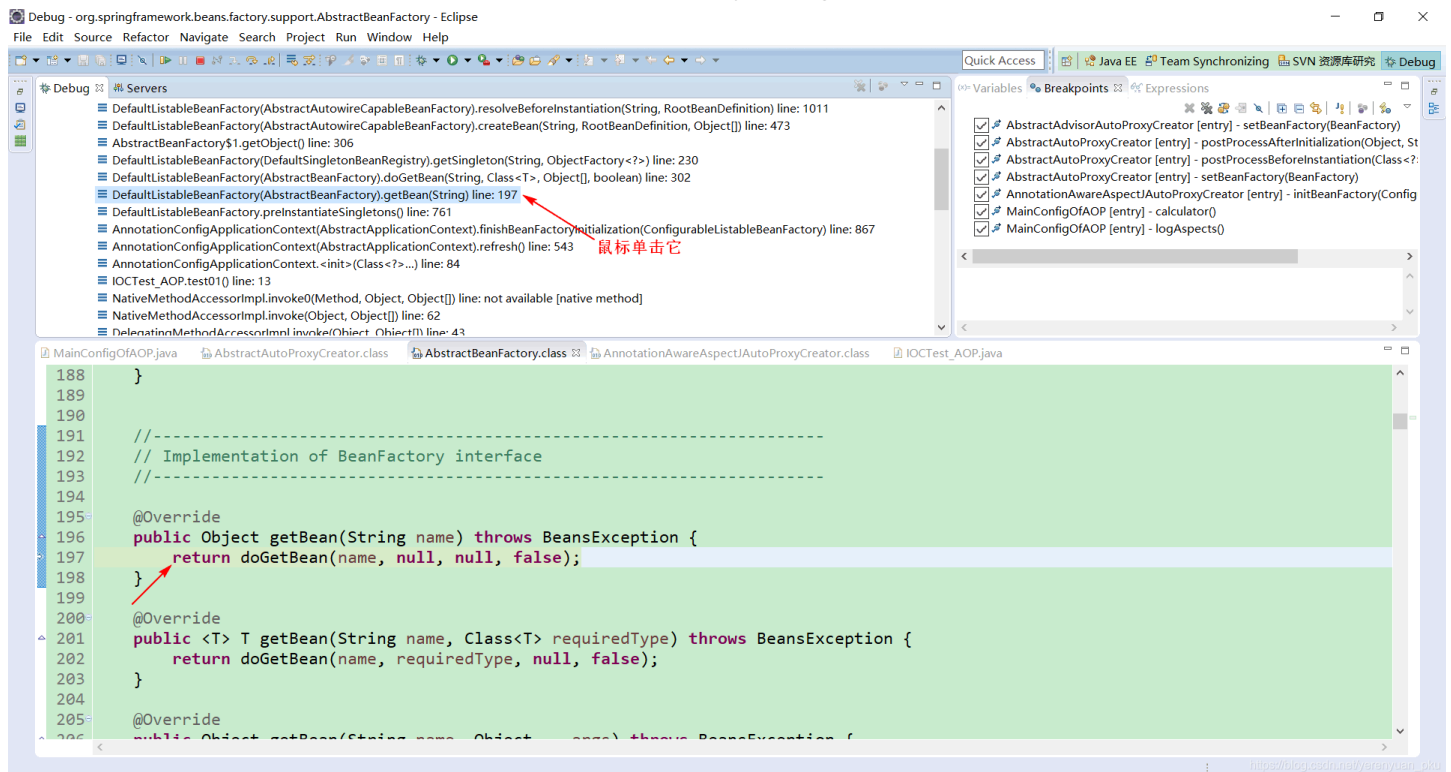
所以，接下来，我们就可以讲讲完成BeanFactory的初始化工作的第一步了。

完成BeanFactory的初始化工作的第一步

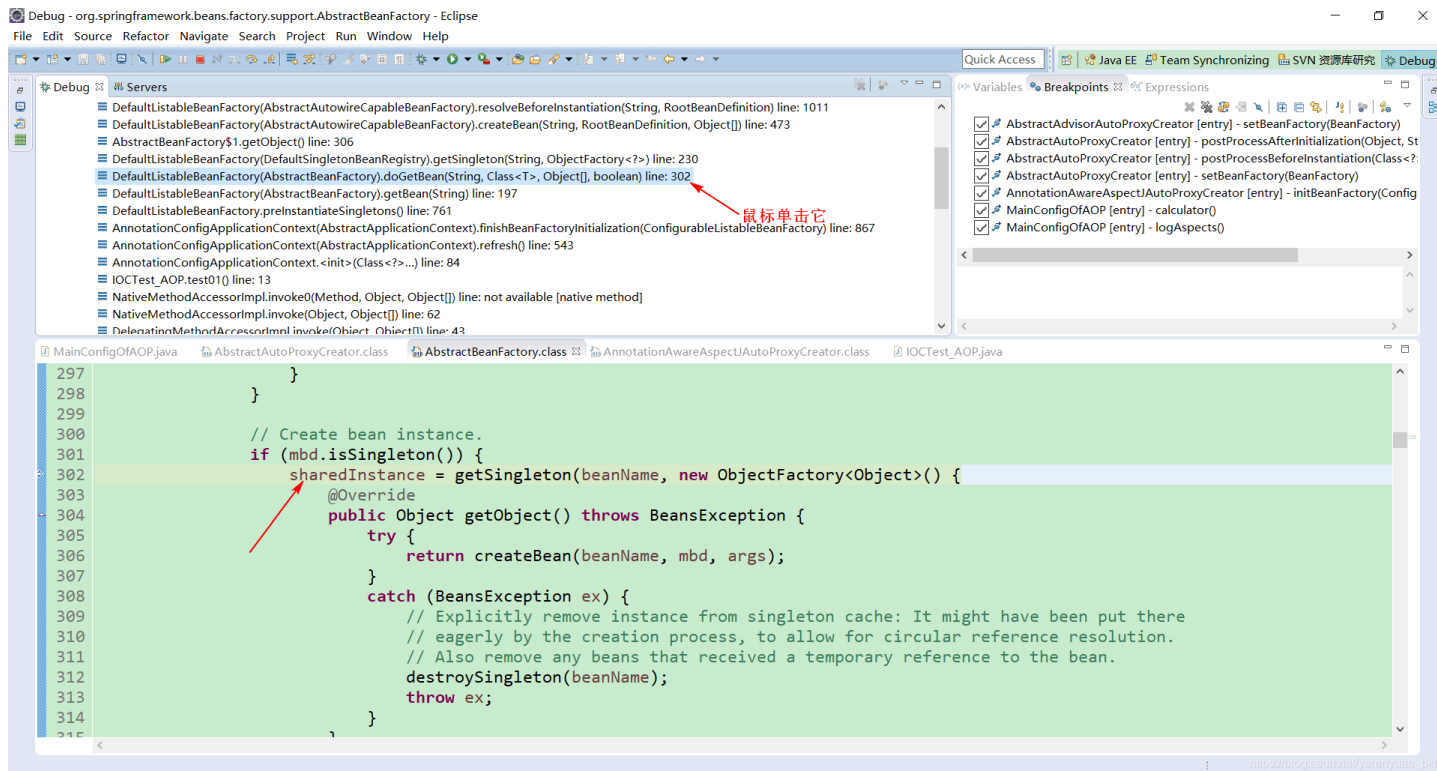
遍历获取容器中的所有bean，并依次创建对象，注意是依次调用getBean()方法来创建对象的。

此刻，咱们是来到了第一个bean的创建，只不过它跟我们目前的研究没什么关系。我们可以以它的创建为例来看一下这个bean到底是怎么来创建的。

我们继续跟进方法调用栈，如下图所示，可以看到现在是定位到了AbstractBeanFactory抽象类的getBean()方法中。



再继续跟进方法调用栈，如下图所示，可以看到现在是定位到了AbstractBeanFactory抽象类的doGetBean()方法中。



可以看到，获取单实例bean调用的是getSingleton()方法，并且会返回一个sharedInstance对象。其实，从该方法上面的注释中也能看出，这儿是来创建bean实例的。

其实呢，在这儿创建之前，sharedInstance变量已经提前声明过了，我们往前翻阅doGetBean()方法，就能看到已声明的sharedInstance变量了。



可以清楚地看到，在如下这行代码处是来第一次获取单实例bean。

```
1 // Eagerly check singleton cache for manually registered singletons.
2 Object sharedInstance = getSingleton(beanName);
AI写代码java运行
```

那到底是怎么获取的呢？其实从注释中可以知道，它会提前先检查单实例的缓存中是不是已经人工注册了一些单实例的bean，若是则获取。

完成BeanFactory的初始化工作的第二步

也就是说，这个bean的创建不是说一下就创建好了的，它得先从缓存中获取当前bean，如果能获取到，说明当前bean之前是被创建过的，那么就直接使用，否则的话再创建。

往上翻阅AbstractBeanFactory抽象类的doGetBean()方法，可以看到有这样的逻辑：


```

242 // Eagerly check singleton cache for manually registered singletons.
243 Object sharedInstance = getSingleton(beanName);
244 if (sharedInstance != null && args == null) {
245     if (logger.isDebugEnabled()) {
246         if (isSingletonCurrentlyInCreation(beanName)) {
247             logger.debug("Returning eagerly cached instance of singleton bean '" + beanName +
248                 "' that is not fully initialized yet - a consequence of a circular reference");
249         }
250         else {
251             logger.debug("Returning cached instance of singleton bean '" + beanName + "'");
252         }
253     }
254     bean = getObjectForBeanInstance(sharedInstance, name, beanName, null);
255 }
256
257 else {
258     // Fail if we're already creating this bean instance:
259     // We're assumably within a circular reference.
260     if (isPrototypeCurrentlyInCreation(beanName)) {
261         throw new BeanCurrentlyInCreationException(beanName);
262     }
263
264     // Check if bean definition exists in this factory.
265     BeanFactory parentBeanFactory = getParentBeanFactory();
266     if (parentBeanFactory != null && !containsBeanDefinition(beanName)) {
267         // Not found -> check parent.
268         String nameToLookup = originalBeanName(name);
269         if (args != null) {
270             // Delegation to parent with explicit args.
271             return (T) parentBeanFactory.getBean(nameToLookup, args);
272         }
273         else {
274             // No args -> delegate to standard getBean method.
275             return parentBeanFactory.getBean(nameToLookup, requiredType);
276         }
277     }
278
279     if (!typeCheckOnly) {
280         markBeanAsCreated(beanName);
281     }
282
283     try {
284         final RootBeanDefinition mbd = getMergedLocalBeanDefinition(beanName);
285         checkMergedBeanDefinition(mbd, beanName, args);
286
287         // Guarantee initialization of beans that the current bean depends on.
288         String[] dependsOn = mbd.getDependsOn();
289         if (dependsOn != null) {
290             for (String dep : dependsOn) {
291                 if (isDependent(beanName, dep)) {
292                     throw new BeanCreationException(mbd.getResourceDescription(), beanName,
293                         "Circular depends-on relationship between '" + beanName + "' and '" + dep + "'");
294                 }
295                 registerDependentBean(dep, beanName);
296                 getBean(dep);
297             }
298         }
299
300         // Create bean instance.
301         if (mbd.isSingleton()) {
302             sharedInstance = getSingleton(beanName, new ObjectFactory<Object>() {
303                 @Override
304                 public Object getObject() throws BeansException {
305                     try {
306                         return createBean(beanName, mbd, args);
307                     }
308                 }
309             });
310         }
311     }
312 }

```

如果sharedInstance不等于null，那么会在这儿做一堆操作，并返回一个bean。

即包装一下，然后就直接结束了

bean = getObjectForBeanInstance(sharedInstance, name, beanName, null);

Fail if we're already creating this bean instance: We're assumably within a circular reference.

throw new BeanCurrentlyInCreationException(beanName);

Check if bean definition exists in this factory.

BeanFactory parentBeanFactory = getParentBeanFactory();

if (parentBeanFactory != null && !containsBeanDefinition(beanName)) {

// Not found -> check parent.

String nameToLookup = originalBeanName(name);

if (args != null) {

// Delegation to parent with explicit args.

return (T) parentBeanFactory.getBean(nameToLookup, args);

}

else {

// No args -> delegate to standard getBean method.

return parentBeanFactory.getBean(nameToLookup, requiredType);

}

if (!typeCheckOnly) {

markBeanAsCreated(beanName);

}

try {

final RootBeanDefinition mbd = getMergedLocalBeanDefinition(beanName);

checkMergedBeanDefinition(mbd, beanName, args);

// Guarantee initialization of beans that the current bean depends on.

String[] dependsOn = mbd.getDependsOn();

if (dependsOn != null) {

for (String dep : dependsOn) {

if (isDependent(beanName, dep)) {

throw new BeanCreationException(mbd.getResourceDescription(), beanName,

"Circular depends-on relationship between '" + beanName + "' and '" + dep + "'");

}

registerDependentBean(dep, beanName);

getBean(dep);

}

}

// Create bean instance.

if (mbd.isSingleton()) {

sharedInstance = getSingleton(beanName, new ObjectFactory<Object>() {

@Override

public Object getObject() throws BeansException {

try {

return createBean(beanName, mbd, args);

}

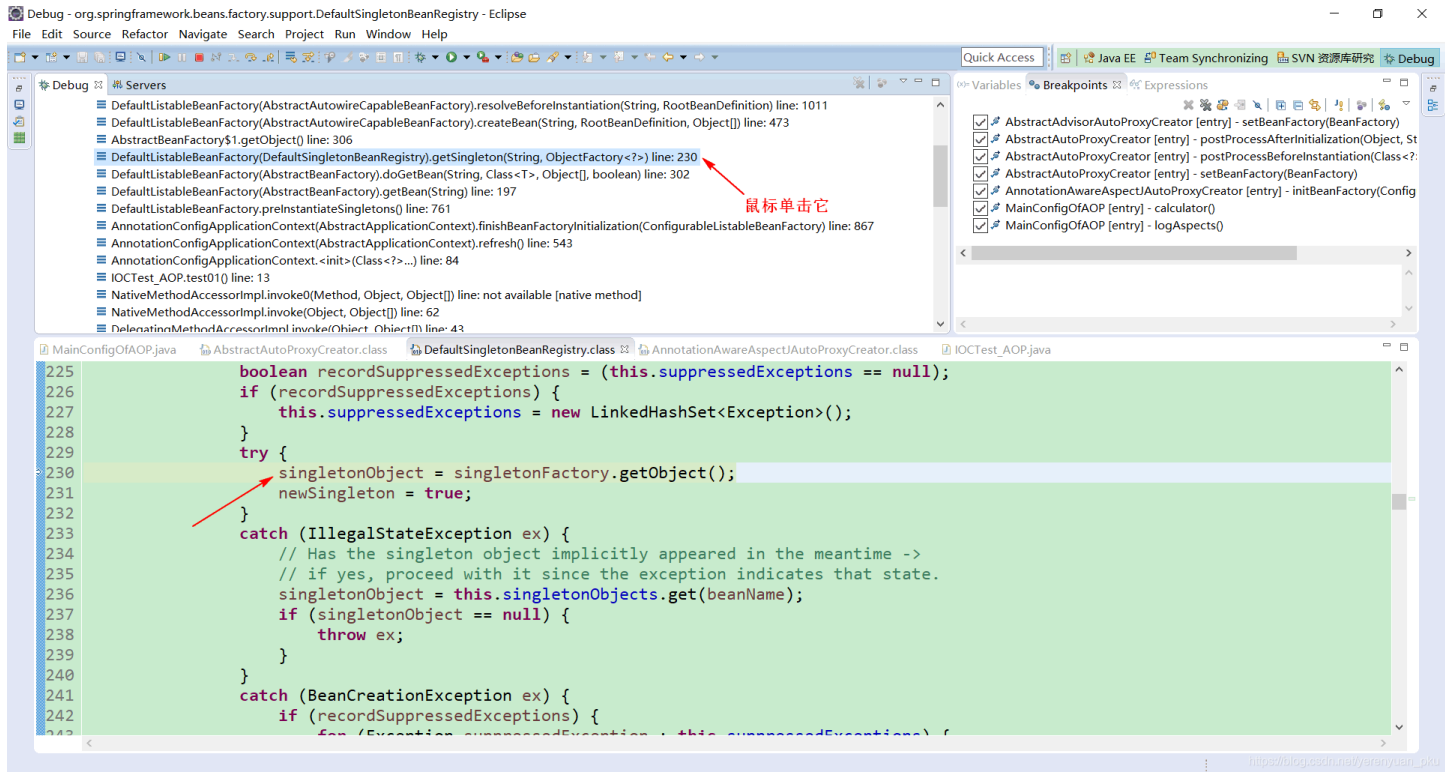
}

});

}

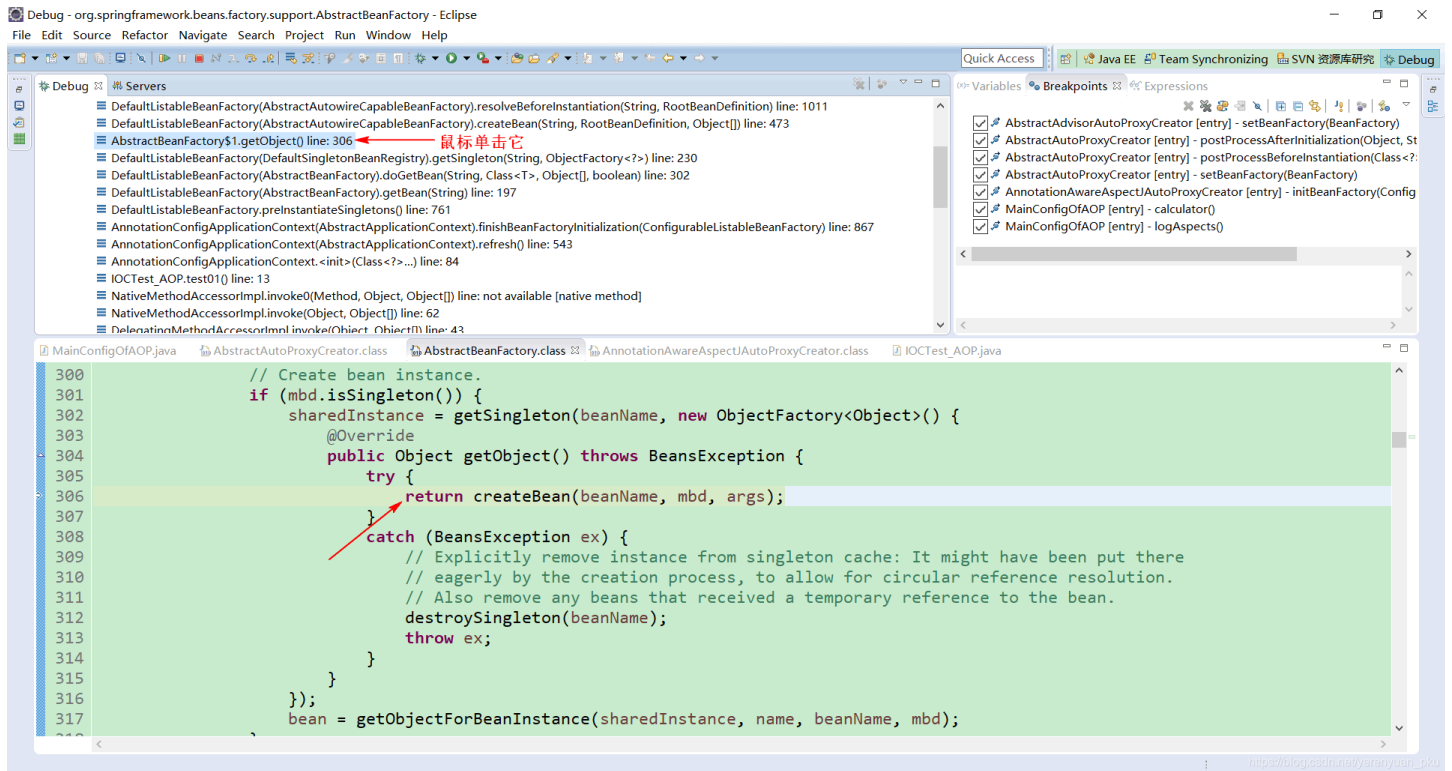
可以看到，单实例bean是能获取就获取，不能获取才创建。Spring就是利用这个机制来保证我们这些单实例bean只会被创建一次，也就是说只要创建好的bean都会被缓存起来。

继续跟进方法调用栈，如下图所示，可以看到现在是定位到了DefaultSingletonBeanRegistry类的getSingleton()方法中。



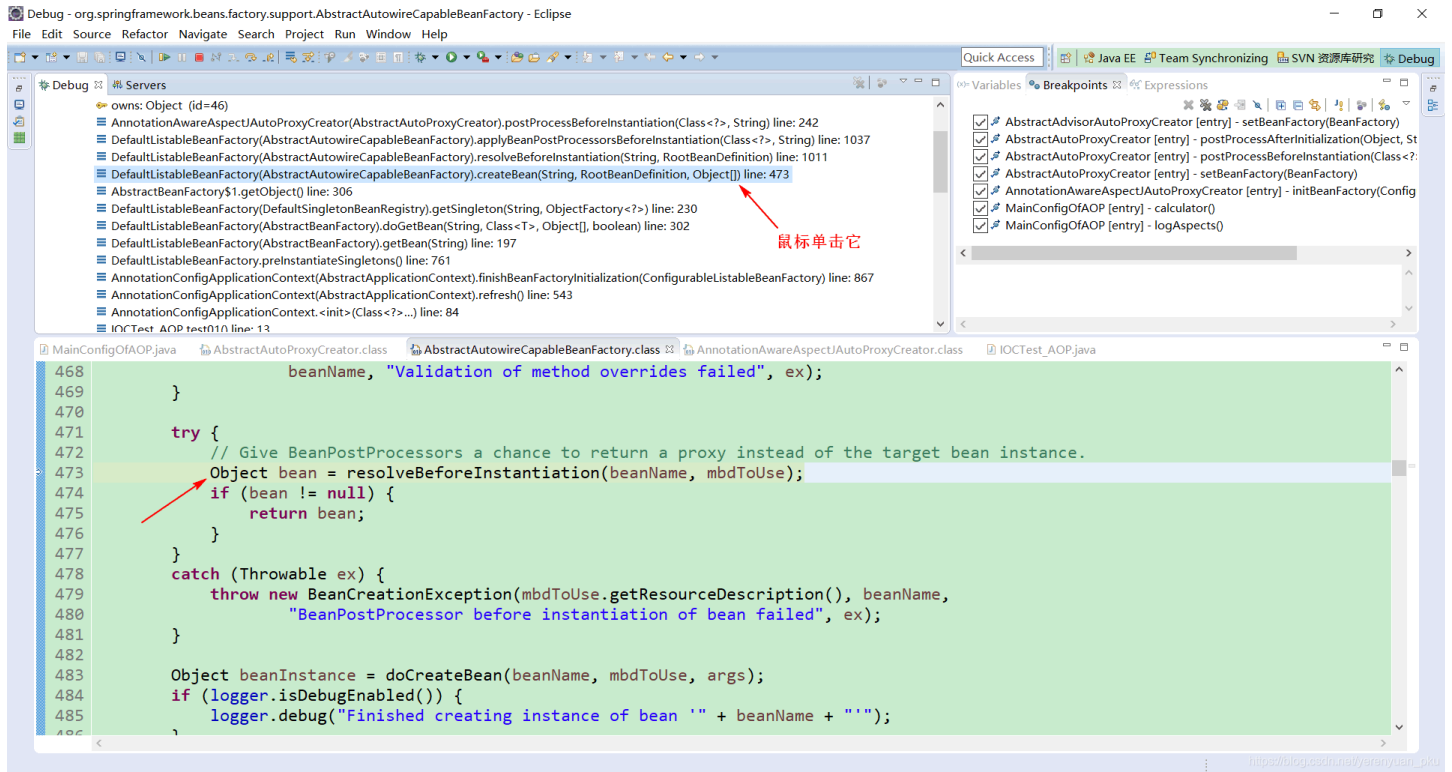
这儿是调用单实例工厂来进行创建单实例bean。

继续跟进方法调用栈，如下图所示，可以看到现在又定位到了AbstractBeanFactory抽象类的doGetBean()方法中。

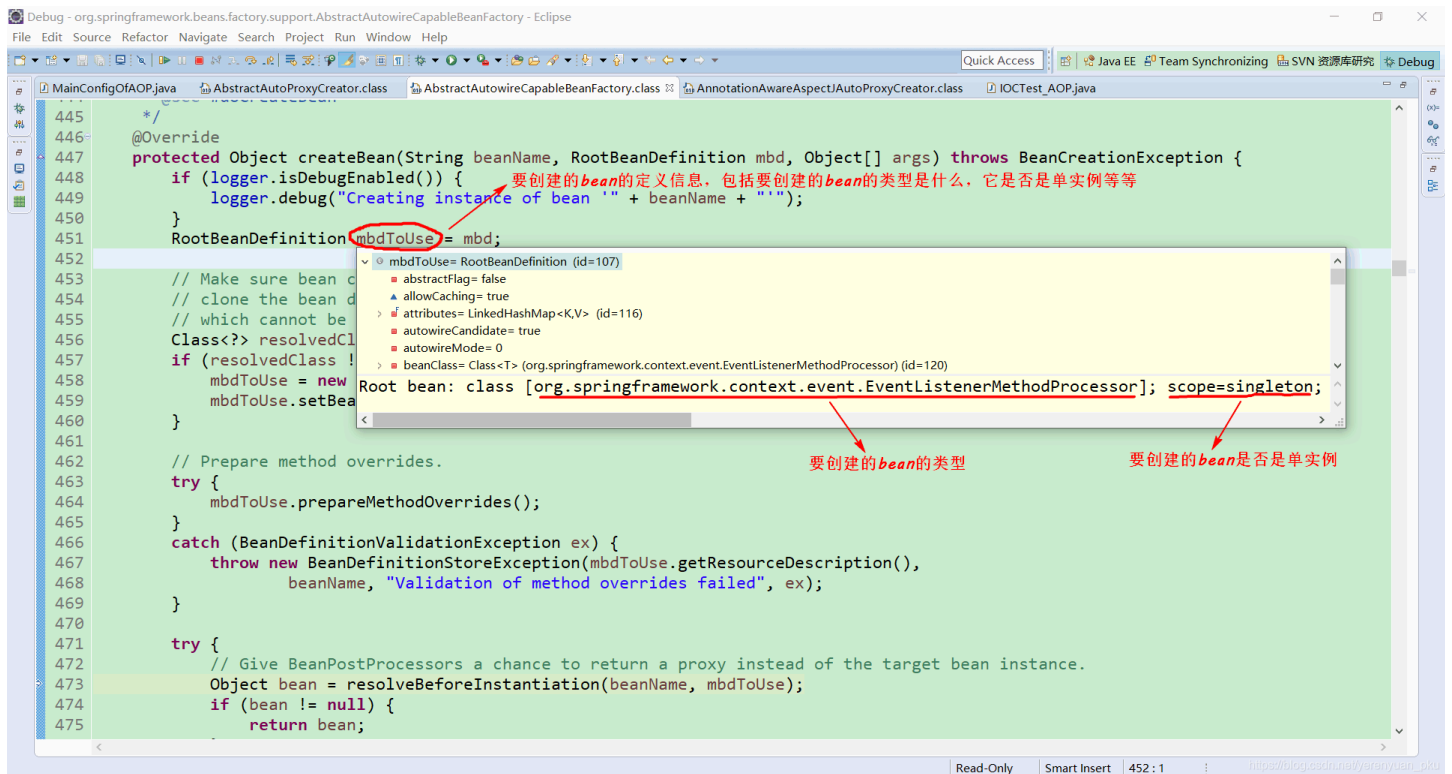


可以看到又会调用createBean()方法来进行创建单实例bean。而在该方法前面是bean能获取到就不会再创建了。接下来，我们来看一下createBean()方法有什么好说的。

继续跟进方法调用栈，如下图所示，可以看到现在是定位到了AbstractAutowireCapableBeanFactory抽象类的createBean()方法中。

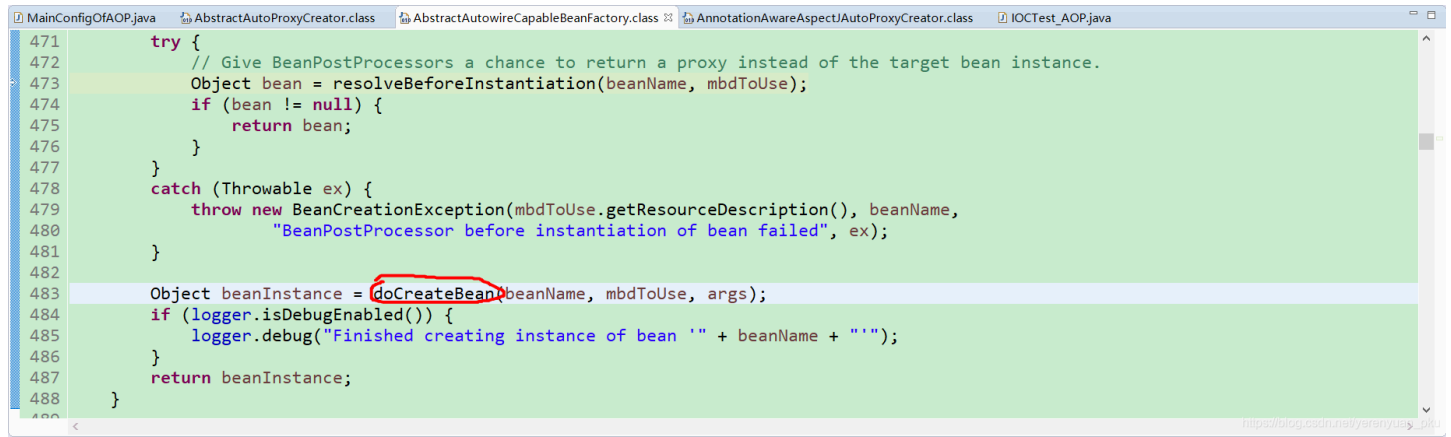


往上翻阅createBean()方法，发现可以拿到要创建的bean的定义信息，包括要创建的bean的类型是什么，它是否是单实例等等，如下图所示。



好，我们还是将关注点放在resolveBeforeInstantiation()方法上，当前程序也是停在了这一行，即473行。

该方法是来解析BeforeInstantiation的，这是啥意思啊？我们可以看一下该方法上的注释，它是说给后置处理器一个机会，来返回一个代理对象，替代我们创建的目标的bean实例。也就是说，我们希望后置处理器在此能返回一个代理对象，如果能返回代理对象那当然就很好了，直接使用就得了，如果不能那么就调用doCreateBean()方法来创建一个bean实例了。



```
471     try {
472         // Give BeanPostProcessors a chance to return a proxy instead of the target bean instance.
473         Object bean = resolveBeforeInstantiation(beanName, mbdToUse);
474         if (bean != null) {
475             return bean;
476         }
477     }
478     catch (Throwable ex) {
479         throw new BeanCreationException(mbdToUse.getResourceDescription(), beanName,
480             "BeanPostProcessor before instantiation of bean failed", ex);
481     }
482
483     Object beanInstance = doCreateBean(beanName, mbdToUse, args);
484     if (logger.isDebugEnabled()) {
485         logger.debug("Finished creating instance of bean '" + beanName + "'");
486     }
487     return beanInstance;
488 }
```

我为什么要说这个方法呢？进入该方法里面看看你自然就懂了，点进去之后会发现该方法真的好长好长，我为了让大家能够更加清楚地看到这个方法的全貌，就截了如下一张图。


```

MainConfigOfAOP.java AbstractAutoProxyCreator.class AbstractAutowiredCapableBeanFactory.class AnnotationAwareAspectJAutoProxyCreator.class IOCTest_AOP.java
504 protected Object doCreateBean(final String beanName, final RootBeanDefinition mbd, final Object[] args)
505     throws BeanCreationException {
506
507     // Instantiate the bean.
508     BeanWrapper instanceWrapper = null;
509     if (mbd.isSingleton()) {
510         instanceWrapper = this.factoryBeanInstanceCache.remove(beanName);
511     }
512     if (instanceWrapper == null) {
513         instanceWrapper = createBeanInstance(beanName, mbd, args);
514     }
515     final Object bean = (instanceWrapper != null ? instanceWrapper.getWrappedInstance() : null);
516     Class<?> beanType = (instanceWrapper != null ? instanceWrapper.getWrappedClass() : null);
517     mbd.resolvedTargetType = beanType;
518
519     // Allow post-processors to modify the merged bean definition.
520     synchronized (mbd.postProcessingLock) {
521         if (!mbd.postProcessed) {
522             try {
523                 applyMergedBeanDefinitionPostProcessors(mbd, beanType, beanName);
524             }
525             catch (Throwable ex) {
526                 throw new BeanCreationException(mbd.getResourceDescription(), beanName,
527                     "Post-processing of merged bean definition failed", ex);
528             }
529             mbd.postProcessed = true;
530         }
531     }
532
533     // Eagerly cache singletons to be able to resolve circular references
534     // even when triggered by lifecycle interfaces like BeanFactoryAware.
535     boolean earlySingletonExposure = (mbd.isSingleton() && this.allowCircularReferences &&
536         isSingletonCurrentlyInCreation(beanName));
537     if (earlySingletonExposure) {
538         if (logger.isDebugEnabled()) {
539             logger.debug("Eagerly caching bean '" + beanName +
540                 "' to allow for resolving potential circular references");
541         }
542         addSingletonFactory(beanName, new ObjectFactory<Object>() {
543             @Override
544             public Object getObject() throws BeansException {
545                 return getEarlyBeanReference(beanName, mbd, bean);
546             }
547         });
548     }
549
550     // Initialize the bean instance.
551     Object exposedObject = bean;
552     try {
553         populateBean(beanName, mbd, instanceWrapper);
554         if (exposedObject != null) {
555             exposedObject = initializeBean(beanName, exposedObject, mbd);
556         }
557     }
558     catch (Throwable ex) {
559         if (ex instanceof BeanCreationException && beanName.equals(((BeanCreationException) ex).getBeanName())) {
560             throw (BeanCreationException) ex;
561         }
562         else {
563             throw new BeanCreationException(
564                 mbd.getResourceDescription(), beanName, "Initialization of bean failed", ex);
565         }
566     }
567
568     if (earlySingletonExposure) {
569         Object earlySingletonReference = getSingleton(beanName, false);
570         if (earlySingletonReference != null) {
571             if (exposedObject == bean) {
572                 exposedObject = earlySingletonReference;
573             }
574             else if (!this.allowRawInjectionDespiteWrapping && hasDependentBean(beanName)) {
575                 String[] dependentBeans = getDependentBeans(beanName);
576                 Set<String> actualDependentBeans = new LinkedHashSet<String>(dependentBeans.length);
577                 for (String dependentBean : dependentBeans) {
578                     if (!removeSingletonIfCreatedForTypeCheckOnly(dependentBean)) {
579                         actualDependentBeans.add(dependentBean);
580                     }
581                 }
582                 if (!actualDependentBeans.isEmpty()) {
583                     throw new BeanCurrentlyInCreationException(beanName,
584                         "Bean with name '" + beanName + "' has been injected into other beans [" +
585                         StringUtils.collectionToCommaDelimitedString(actualDependentBeans) +
586                         "] in its raw version as part of a circular reference, but has eventually been " +
587                         "wrapped. This means that said other beans do not use the final version of the " +
588                         "bean. This is often the result of over-eager type matching - consider using " +
589                         "'getBeanNamesOfType' with the 'allowEagerInit' flag turned off, for example.");
590                 }
591             }
592         }
593     }
594
595     // Register bean as disposable.
596     try {
597         registerDisposableBeanIfNecessary(beanName, bean, mbd);
598     }
599     catch (BeanDefinitionValidationException ex) {
600         throw new BeanCreationException(

```

```

600     throw new BeansException("Invalid destruction signature", ex);
601     }
602     }
603     }
604     return exposedObject;
605 }

```

https://blog.csdn.net/yanyuan_pku

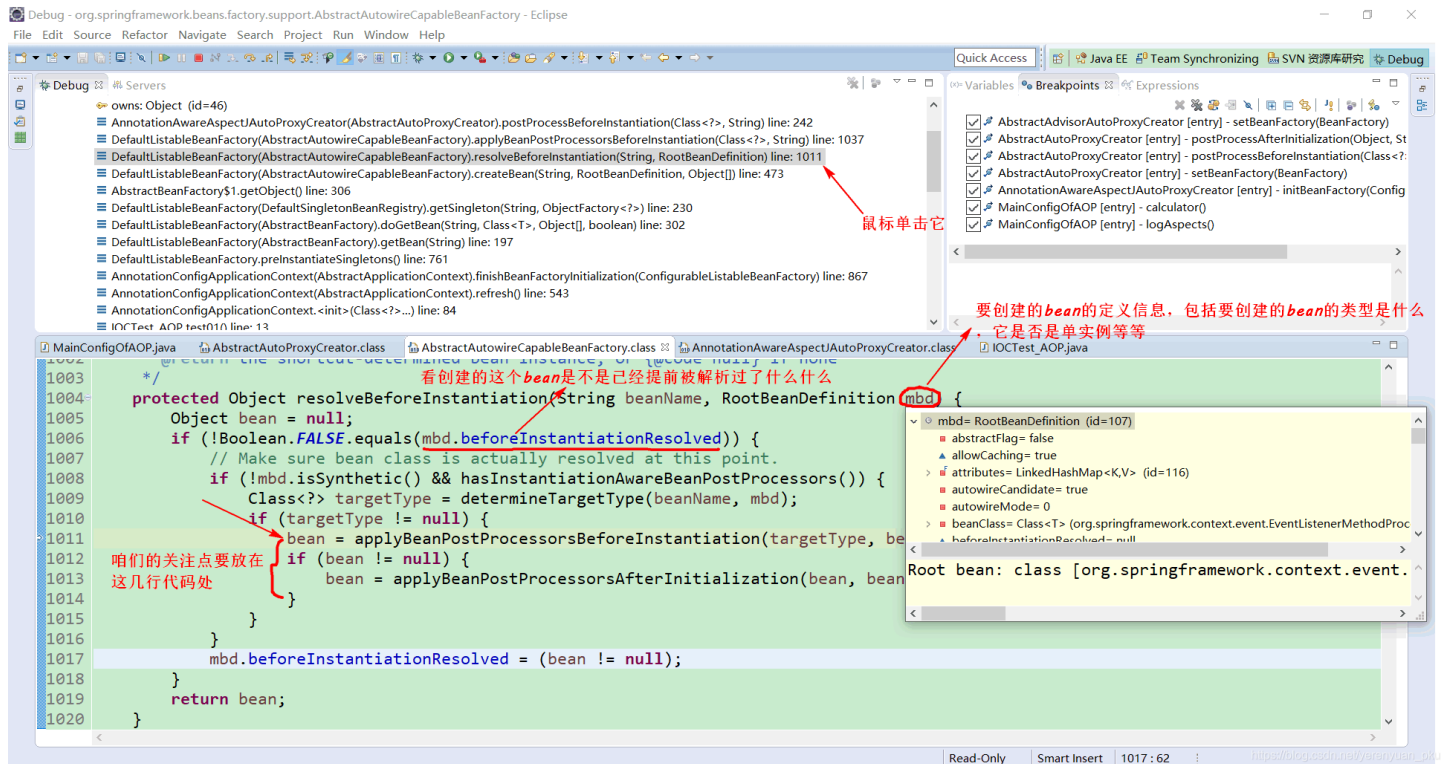
其实，这个doCreateBean()方法我们之前看过很多遍了，所做的事情无非就是：

1. 首先创建bean的实例
2. 然后给bean的各种属性赋值
3. 接着初始化bean
 - o 1) 先执行Aware接口的方法
 - o 2) 应用后置处理器的postProcessBeforeInitialization()方法
 - o 3) 执行自定义的初始化方法
 - o 4) 应用后置处理器的postProcessAfterInitialization()方法

调用doCreateBean()方法才是真正的去创建一个bean实例。

我们还是来到程序停留的地方，即AbstractAutowireCapableBeanFactory抽象类的第473行。我们希望后置处理器在此能返回一个代理对象，如果能返回代理对象那当然就很好了，直接使用就得了。接下来，我们就要看看resolveBeforeInstantiation()方法里面具体是怎么做的了。

继续跟进方法调用栈，如下图所示，可以看到现在是定位到了AbstractAutowireCapableBeanFactory抽象类的resolveBeforeInstantiation()方法中，既然程序是停留在了此处，那说明并没有走后调用doCreateBean()方法创建bean实例的流程，而是先来到这儿，希望后置处理器能返回一个代理对象。



可以看到，在该方法中，首先会拿到要创建的bean的定义信息，包括要创建的bean的类型是什么，它是否是单实例等等，然后看它是不是已经提前被解析过了什么什么，这都不算太重要，我们主要关注如下这几行代码：

```

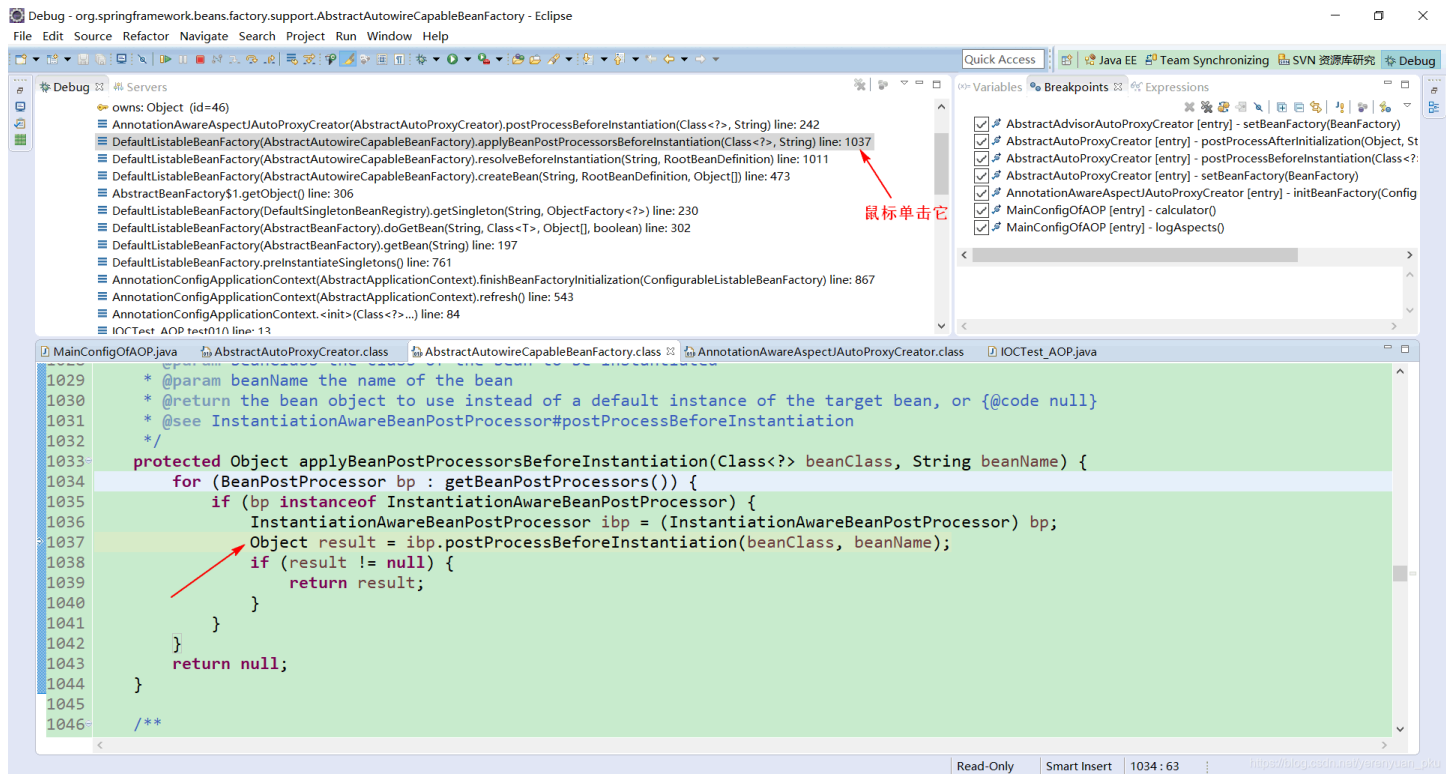
1 bean = applyBeanPostProcessorsBeforeInstantiation(targetType, beanName);
2 if (bean != null) {
3     bean = applyBeanPostProcessorsAfterInitialization(bean, beanName);
4 }

```

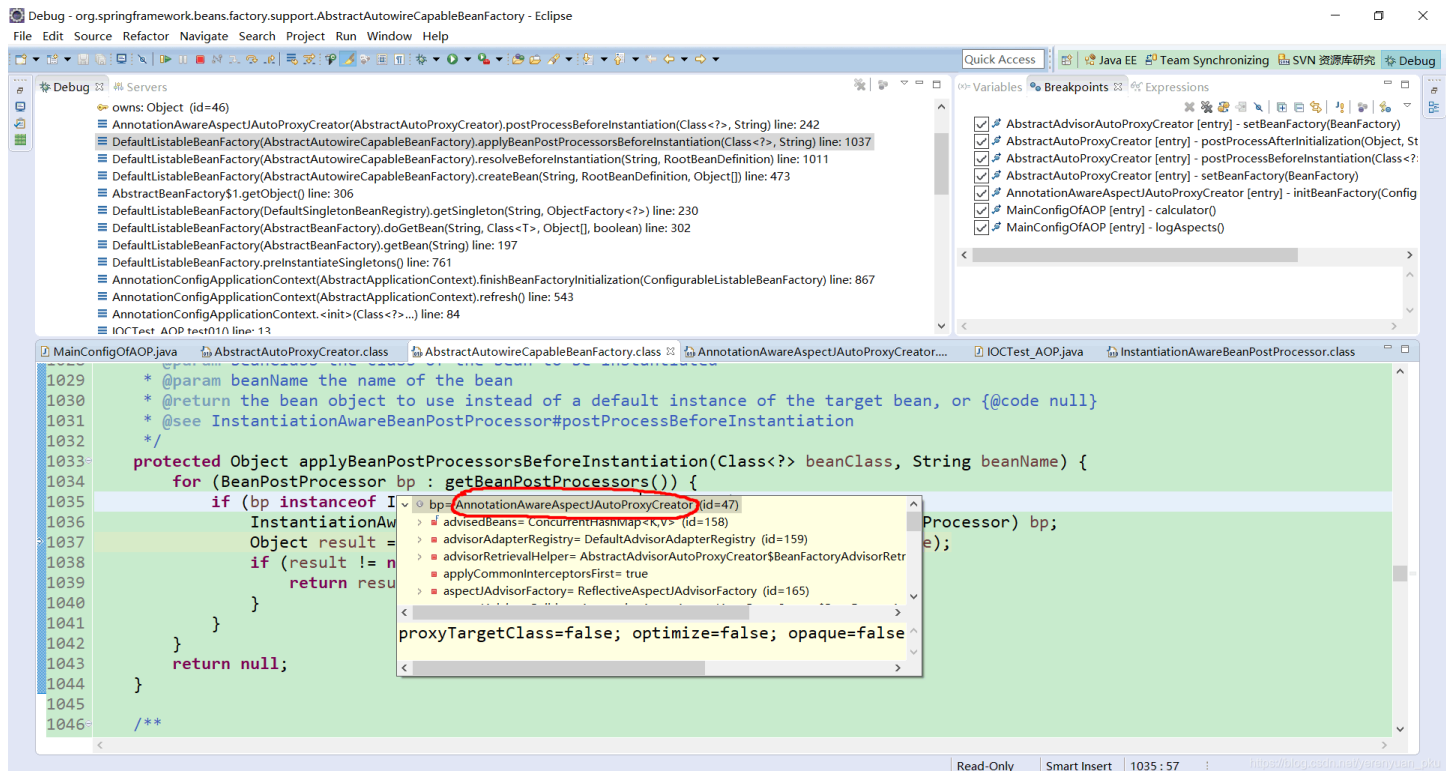
AI写代码java运行

这一块会调用两个方法，一个叫方法叫applyBeanPostProcessorsBeforeInstantiation，另一个方法叫applyBeanPostProcessorsAfterInitialization。

后置处理器会先尝试返回对象，怎么尝试返回呢？可以看到，是调用applyBeanPostProcessorsBeforeInstantiation()方法返回一个对象的，那这个方法是干啥的呢？我们继续跟进方法调用栈，如下图所示，可以看到现在是定位到了AbstractAutowireCapableBeanFactory抽象类的applyBeanPostProcessorsBeforeInstantiation()方法中。



我们发现，它是拿到所有的后置处理器，如果后置处理器是InstantiationAwareBeanPostProcessor这种类型的，那么就执行该后置处理器的postProcessBeforeInstantiation()方法。我为什么要说这个方法呢？因为现在遍历拿到的后置处理器是AnnotationAwareAspectJAutoProxyCreator这种类型的，如下图所示。

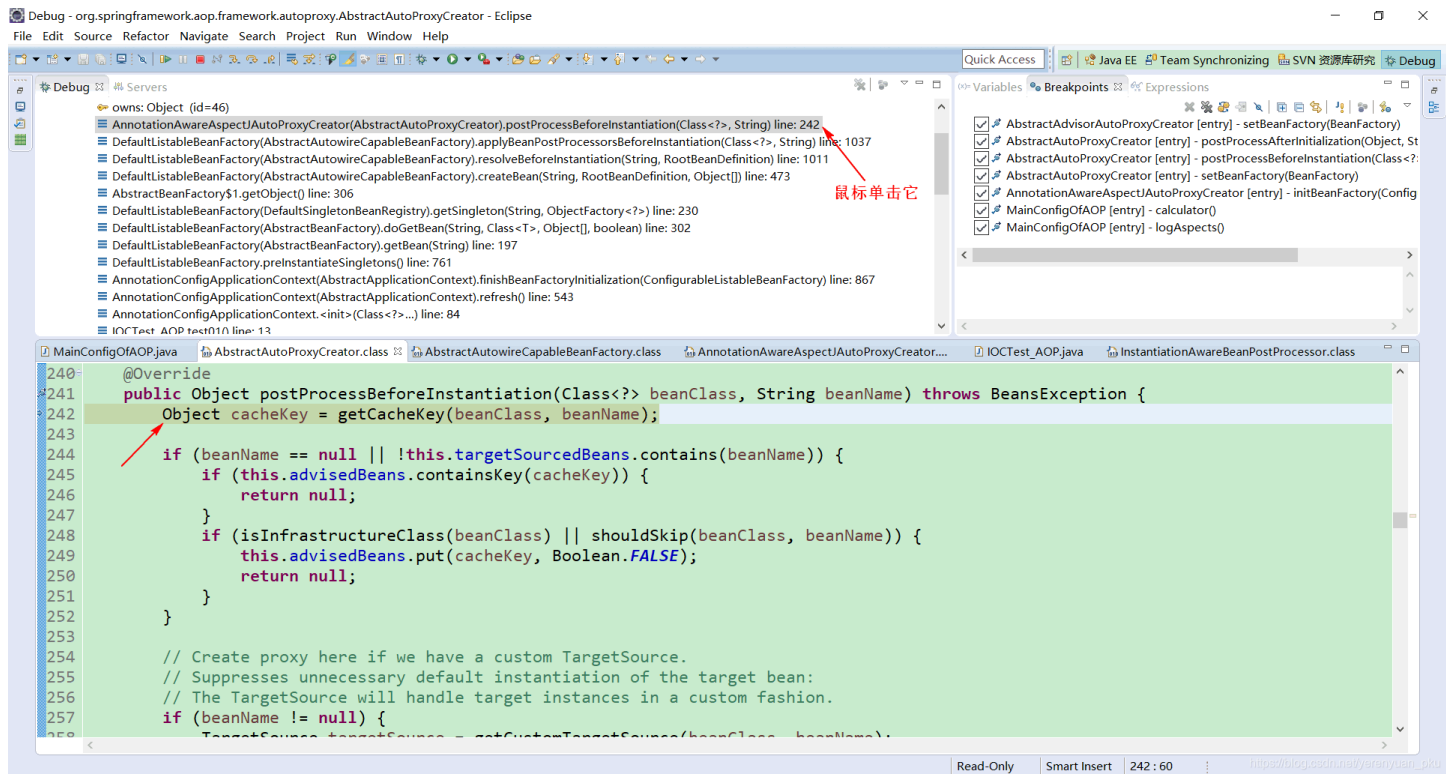


并且前面我也说了，它就是InstantiationAwareBeanPostProcessor这种类型后置处理器，这种类型后置处理器中声明的方法就叫postProcessBeforeInstantiation，而不是我们以前学的后置处理器中的叫postProcessBeforeInitialization的方法，也就是说后置处理器跟后置处理器是不一样的。

我们以前就知道，BeanPostProcessor是在bean对象创建完成初始化前后调用的。而在这儿我们也看到了，首先是会有一个判断，即判断后置处理器是不是InstantiationAwareBeanPostProcessor这种类型的，然后再尝试用后置处理器返回对象（当然了，是在创建bean实例之前）。

总之，我们可以得出一个结论：AnnotationAwareAspectJAutoProxyCreator会在任何bean创建之前，先尝试返回bean的实例。

最后，我们继续跟进方法调用栈，如下图所示，可以看到终于又定位到了AbstractAutoProxyCreator抽象类的postProcessBeforeInstantiation()方法中。



为什么程序会来到这个方法中呢？想必你也非常清楚了，因为判断后置处理器是不是`InstantiationAwareBeanPostProcessor`这种类型时，轮到了`AnnotationAwareAspectJAutoProxyCreator`这个后置处理器，而它正好是`InstantiationAwareBeanPostProcessor`这种类型的，所以程序自然就会来到它的`postProcessBeforeInstantiation()`方法中。

呼应前面，我们现在是终于分析到了`AnnotationAwareAspectJAutoProxyCreator`这个后置处理器的`postProcessBeforeInstantiation()`方法中，也就是知道了程序是怎么到这儿来的。

最终，我们得出这样一个结论：**`AnnotationAwareAspectJAutoProxyCreator`**在所有bean创建之前，会有一个拦截，因为它是`InstantiationAwareBeanPostProcessor`这种类型的后置处理器，然后会调用它的`postProcessBeforeInstantiation()`方法。