

# Spring注解驱动开发第36讲——或许，这是你以前没看过的从源码角度理解BeanFactoryPostProcessor的原理

## 写在前面

在前面，我们学习了一下Spring中的IOC、AOP以及 **声明式事务** 等最核心的知识点，而且我们还掌握了它们的注解驱动开发，不仅如此，我们还从源码的角度分析了一下它们里面的工作原理，相信看过我前面文章的同学，一定对此深有体会。

而从这一讲开始，我们便来学习一下Spring里面一些其他的扩展原理，希望大家通过这些原理的学习，对Spring里面的运行机制，包括其内部的工作原理，能有一个非常深刻的认识，这样的话，对于大家来学习Spring里面的其他框架会有非常好的帮助。

## 从源码角度理解BeanFactoryPostProcessor的原理

根据我们在第一讲中的安排，我们接下来要说的Spring里面的一些扩展原理有：

- BeanFactoryPostProcessor
- BeanDefinitionRegistryPostProcessor
- ApplicationListener
- Spring容器创建过程

不过在这一讲中，我们只是先来说一下BeanFactoryPostProcessor的原理，其他的在后续的文章中都会讲到。

### BeanFactoryPostProcessor的调用时机

BeanFactoryPostProcessor其实就是BeanFactory（创建bean的工厂）的后置处理器。说起这个，你脑海中是不是泛起了回忆，是不是想起了有一个与BeanFactoryPostProcessor的名字极其相似的玩意，它就是BeanPostProcessor。那什么是BeanPostProcessor呢？我们之前早就说过了，它就是bean的后置处理器，并且是在bean创建对象初始化前后进行拦截工作的。

现在我们要讲解的是BeanFactoryPostProcessor，上面也说过了，它是BeanFactory（创建bean的工厂）的后置处理器。接下来，我们就要搞清楚它的内部原理了，想要搞清楚其内部原理，我们需要从它是什么时候工作这一点开始入手研究，也即搞清楚它的调用时机是什么。

我们点进去BeanFactoryPostProcessor的源码里面去看一看，发现它是一个接口，如下图所示。

```
BeanFactoryPostProcessor.class
2+ * Copyright 2002-2012 the original author or authors.
16
17 package org.springframework.beans.factory.config;
18
19 import org.springframework.beans.BeansException;
20
22+ * Allows for custom modification of an application context's bean definitions,
45 public interface BeanFactoryPostProcessor {
46
47     /**
48      * Modify the application context's internal bean factory after its standard
49      * initialization. All bean definitions will have been loaded, but no beans
50      * will have been instantiated yet. This allows for overriding or adding
51      * properties even to eager-initializing beans.
52      * @param beanFactory the bean factory used by the application context
53      * @throws org.springframework.beans.BeansException in case of errors
54      */
55     void postProcessBeanFactory(ConfigurableListableBeanFactory beanFactory) throws BeansException;
56
57 }
```

仔细看一下其内部postProcessBeanFactory方法上的描述，这很重要，因为从这段描述中我们就可以知道BeanFactoryPostProcessor的调用时机。描述中说，我们可以在IOC容器里面的BeanFactory的标准初始化完成之后，修改IOC容器里面的这个BeanFactory。

也就是说，BeanFactoryPostProcessor的调用时机是在BeanFactory标准初始化之后，这样一来，我们就可以来定制和修改BeanFactory里面的一些内容了。那什么叫标准初始化呢？接着看描述，它说的是所有的bean定义已经被加载了，但是还没有bean被初始化。

说人话，就是BeanFactoryPostProcessor的调用时机是在BeanFactory标准初始化之后，这样一来，我们就可以来定制和修改BeanFactory里面的一些内容了，此时，所有的bean定义已经保存加载到BeanFactory中了，但是bean的实例还未创建。

### 案例实践

接下来，我们来编写一个案例来验证一下以上说的内容。

首先，我们来编写一个我们自己的BeanFactoryPostProcessor，例如MyBeanFactoryPostProcessor。要编写这样一个bean工厂的后置处理器，它得需要实现我们上面说的BeanFactoryPostProcessor接口，并且还得添加一个实现方法。由于BeanFactoryPostProcessor接口里面只声明了一个方法，即postProcessBeanFactory，所以咱们自己编写的MyBeanFactoryPostProcessor类中只需要实现其即可。

```
1 package com.meimeixia.ext;
2
3 import java.util.Arrays;
4
5 import org.springframework.beans.BeansException;
6 import org.springframework.beans.factory.config.BeanFactoryPostProcessor;
7 import org.springframework.beans.factory.config.ConfigurableListableBeanFactory;
8 import org.springframework.stereotype.Component;
9
10 @Component
11 public class MyBeanFactoryPostProcessor implements BeanFactoryPostProcessor {
12
13     @Override
14     public void postProcessBeanFactory(ConfigurableListableBeanFactory beanFactory) throws BeansException {
15         System.out.println("MyBeanFactoryPostProcessor...postProcessBeanFactory..."); // 这个时候我们所有的bean还没被创建
16                                             // 但是我们可以看一下通过Spring给我们传过来的这个beanFe
17         int count = beanFactory.getBeanDefinitionCount(); // 我们能拿到有几个bean定义
18         String[] names = beanFactory.getBeanDefinitionNames(); // 除此之外，我们还能拿到每一个bean定义的名字
19         System.out.println("当前BeanFactory中有" + count + "个Bean");
20         System.out.println(Arrays.asList(names));
21     }
22 }
23 }
```

AI写代码java运行



注意，我们自己编写的MyBeanFactoryPostProcessor类要想让Spring知道，并且还要能被使用起来，那么它一定就得被加在容器中，为此，我们可以在其上标注一个@Component注解。

然后，创建一个配置类，例如ExtConfig，记得还要在该配置类上使用@ComponentScan注解来配置包扫描哟！

```
1 package com.meimeixia.ext;
2
3 import org.springframework.context.annotation.ComponentScan;
4 import org.springframework.context.annotation.Configuration;
5
6 /**
7  *
8  * @author liayun
9  *
10 */
11 @ComponentScan("com.meimeixia.ext")
12 @Configuration
13 public class ExtConfig {
14
15 }
```

AI写代码java运行



当然了，我们也可以使用@Bean注解向容器中注入咱自己写的组件，例如，在这里，我们可以向容器中注入一个Blue组件。

```
1 package com.meimeixia.ext;
2
3 import org.springframework.context.annotation.Bean;
4 import org.springframework.context.annotation.ComponentScan;
5 import org.springframework.context.annotation.Configuration;
6
7 import com.meimeixia.bean.Blue;
8
9 /**
10  *
11  * @author liayun
12  *
13 */
14 @ComponentScan("com.meimeixia.ext")
15 @Configuration
16 public class ExtConfig {
17
18     @Bean
19     public Blue blue() {
20         return new Blue();
21     }
22 }
```

```
22 |     }
23 | }
    |
    | AI写代码java运行
```



上面这个Blue组件其实就是一个非常普通的组件，代码如下所示：

```
1 | package com.meimeixia.bean;
2 |
3 | public class Blue {
4 |
5 |     public Blue() {
6 |         System.out.println("blue...constructor");
7 |     }
8 |
9 |     public void init() {
10 |         System.out.println("blue...init...");
11 |     }
12 |
13 |     public void destory() {
14 |         System.out.println("blue...destory...");
15 |     }
16 | }
17 |
    | AI写代码java运行
```



可以看到，在创建Blue对象的时候，无参构造器会有相应打印。

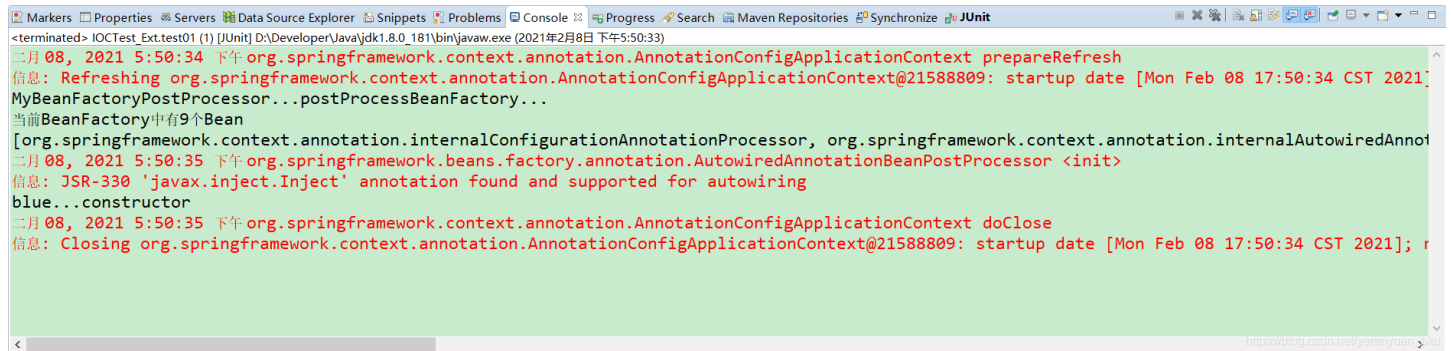
接着，编写一个单元测试类，例如IOCTest\_Ext，来进行测试。

```
1 | package com.meimeixia.test;
2 |
3 | import org.junit.Test;
4 | import org.springframework.context.annotation.AnnotationConfigApplicationContext;
5 |
6 | import com.meimeixia.ext.ExtConfig;
7 |
8 | public class IOCTest_Ext {
9 |
10 |     @Test
11 |     public void test01() {
12 |         AnnotationConfigApplicationContext applicationContext = new AnnotationConfigApplicationContext(ExtConfig.class);
13 |
14 |         // 关闭容器
15 |         applicationContext.close();
16 |     }
17 | }
18 |
    | AI写代码java运行
```



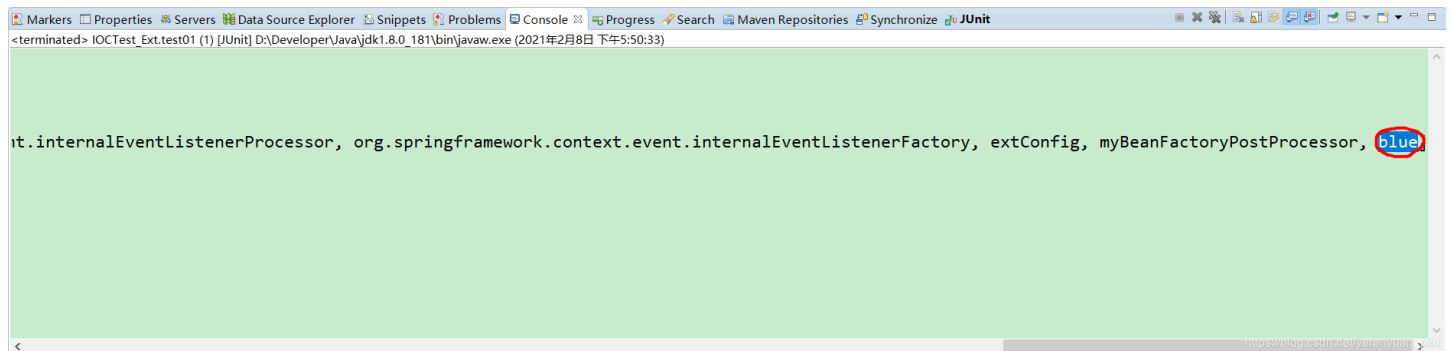
测试啥呢？其实就是来验证一下BeanFactoryPostProcessor的调用时机。说得更具体一点就是，就是看一下咱们自己编写的BeanFactoryPostProcessor究竟是不是在所有的bean定义已经被加载，但是还未创建对象的时候工作？

那咱们接下来就来一探究竟。运行IOCTest\_Ext类中的test01方法，可以看到Eclipse 控制台打印出了如下内容。



```
<terminated> IOCTest_Ext.test01 (1) [JUnit] D:\Developer\Java\jdk1.8.0_181\bin\javaw.exe (2021年2月8日 下午5:50:33)
二月 08, 2021 5:50:34 下午 org.springframework.context.annotation.AnnotationConfigApplicationContext prepareRefresh
信息: Refreshing org.springframework.context.annotation.AnnotationConfigApplicationContext@21588809: startup date [Mon Feb 08 17:50:34 CST 2021]; root of context hierarchy
MyBeanFactoryPostProcessor...postProcessBeanFactory...
当前BeanFactory中有9个Bean
[org.springframework.context.annotation.internalConfigurationAnnotationProcessor, org.springframework.context.annotation.internalAutowiredAnnotationProcessor, org.springframework.context.annotation.internalRequiredAnnotationProcessor, org.springframework.context.annotation.internalRequiredAnnotationProcessor, org.springframework.context.annotation.internalRequiredAnnotationProcessor, org.springframework.context.annotation.internalRequiredAnnotationProcessor, org.springframework.context.annotation.internalRequiredAnnotationProcessor, org.springframework.context.annotation.internalRequiredAnnotationProcessor, org.springframework.context.annotation.internalRequiredAnnotationProcessor]
二月 08, 2021 5:50:35 下午 org.springframework.beans.factory.annotation.AutowiredAnnotationBeanPostProcessor <init>
信息: JSR-330 'javax.inject.Inject' annotation found and supported for autowiring
blue...constructor
二月 08, 2021 5:50:35 下午 org.springframework.context.annotation.AnnotationConfigApplicationContext doClose
信息: Closing org.springframework.context.annotation.AnnotationConfigApplicationContext@21588809: startup date [Mon Feb 08 17:50:34 CST 2021]; root of context hierarchy
```

哎呀！咱们自己编写的BeanFactoryPostProcessor在Blue类的无参构造器创建Blue对象之前就已经工作了。细心一点看的话，从bean的定义信息中还能看到Blue组件注册到容器中的名字，只是此刻还没创建对象，如下图所示。



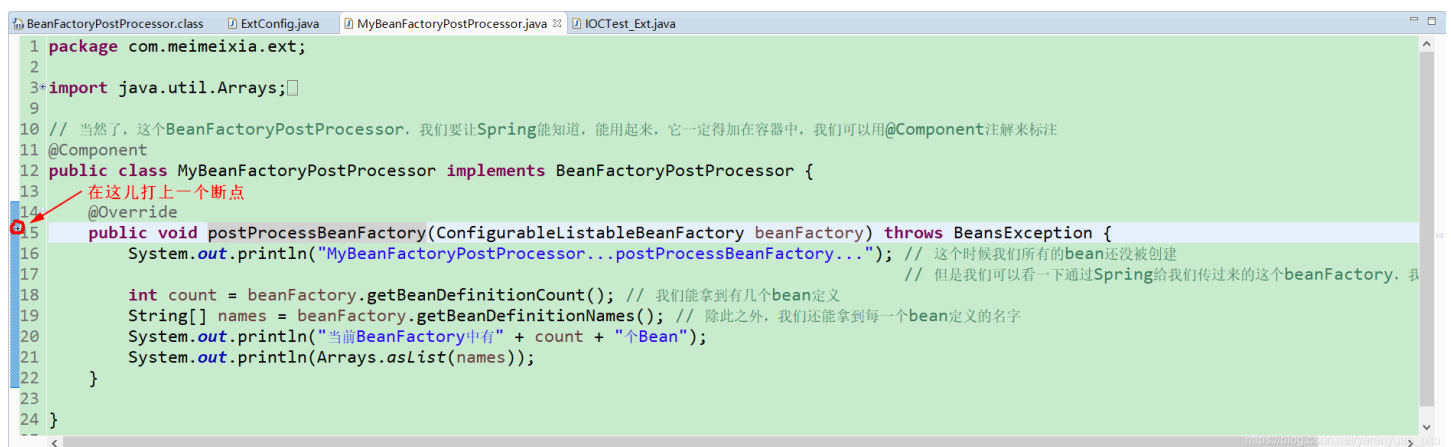
```
rt.internalEventListenerProcessor, org.springframework.context.event.internalEventListenerFactory, extConfig, myBeanFactoryPostProcessor, blue
```

说明BeanFactoryPostProcessor是在所有的bean定义信息都被加载之后才调用的。

## 源码分析

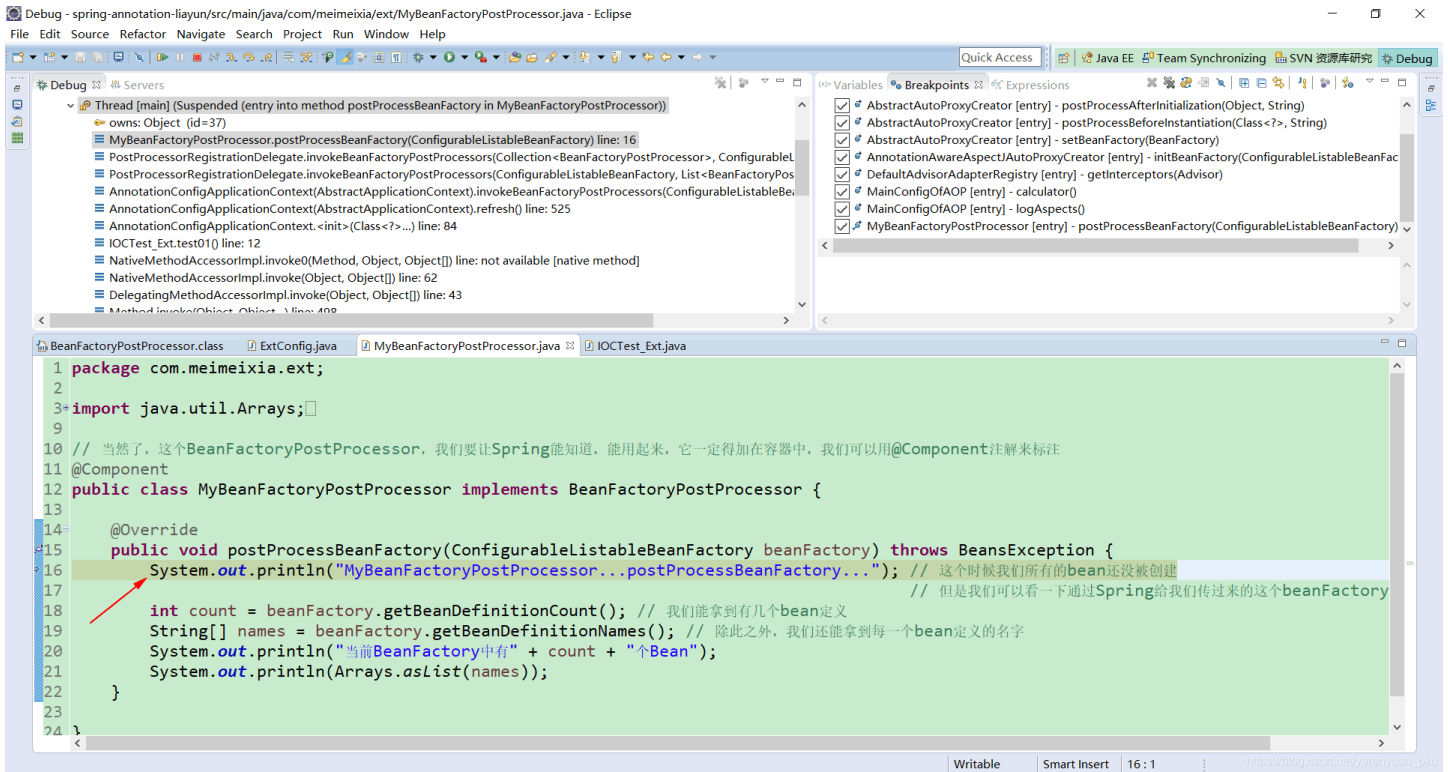
接下来，我们就以debug的方式来看一下BeanFactoryPostProcessor的调用时机。

首先，在咱们自己编写的MyBeanFactoryPostProcessor类里面的postProcessBeanFactory方法处打上一个断点，如下图所示。



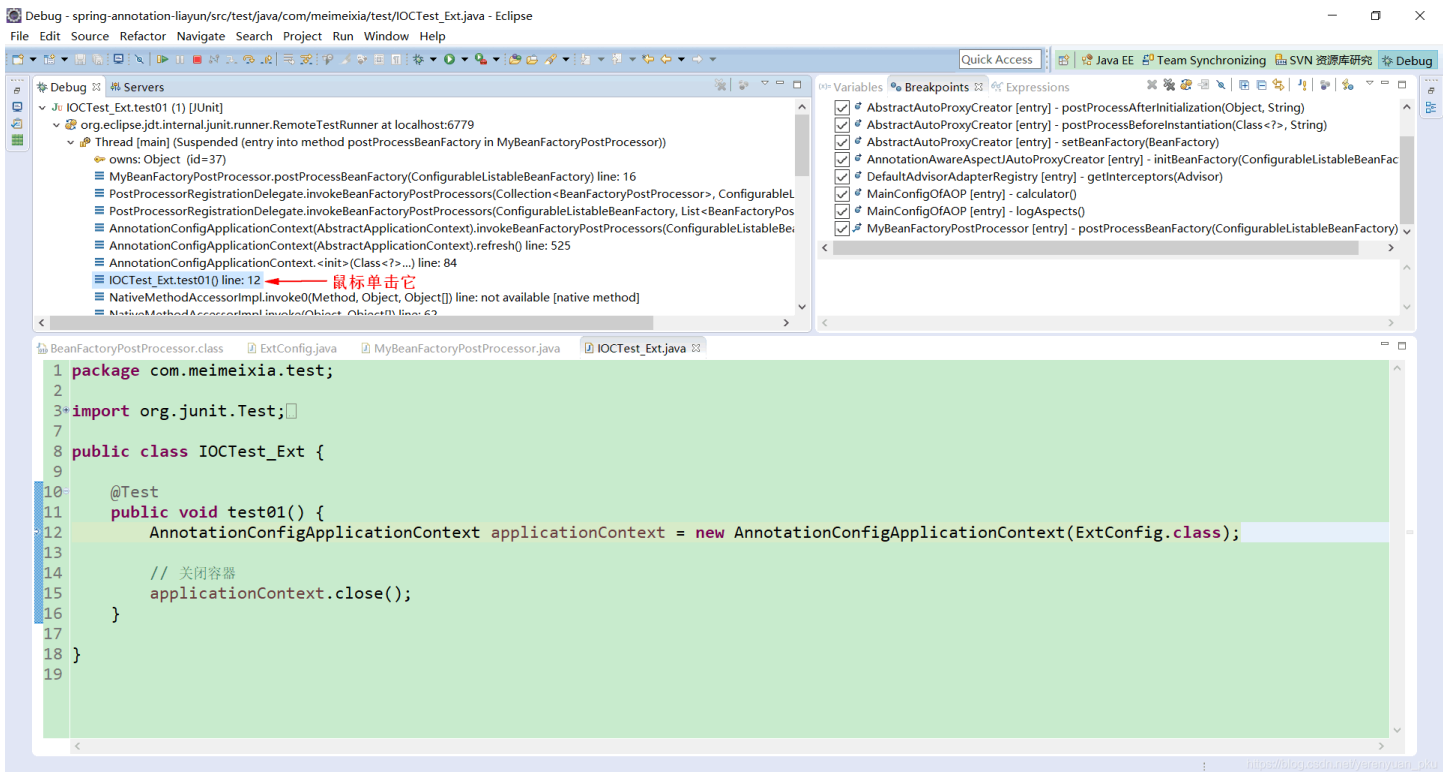
```
1 package com.meimeixia.ext;
2
3 import java.util.Arrays;
4
5 // 当然了，这个BeanFactoryPostProcessor，我们要让Spring能知道，能用起来，它一定得加在容器中，我们可以用@Component注解来标注
6 @Component
7 public class MyBeanFactoryPostProcessor implements BeanFactoryPostProcessor {
8     @Override
9     public void postProcessBeanFactory(ConfigurableListableBeanFactory beanFactory) throws BeansException {
10         System.out.println("MyBeanFactoryPostProcessor...postProcessBeanFactory..."); // 这个时候我们所有的bean还没被创建
11         // 但是我们可以看一下通过Spring给我们传过来的这个beanFactory，我
12
13         int count = beanFactory.getBeanDefinitionCount(); // 我们能拿到有几个bean定义
14         String[] names = beanFactory.getBeanDefinitionNames(); // 除此之外，我们还能拿到每一个bean定义的名字
15         System.out.println("当前BeanFactory中有" + count + "个Bean");
16         System.out.println(Arrays.asList(names));
17     }
18 }
```

然后，以debug的方式来运行IOCTest\_Ext类中的test01方法，如下图所示，程序现在停到了MyBeanFactoryPostProcessor类里面的postProcessBeanFactory方法处。



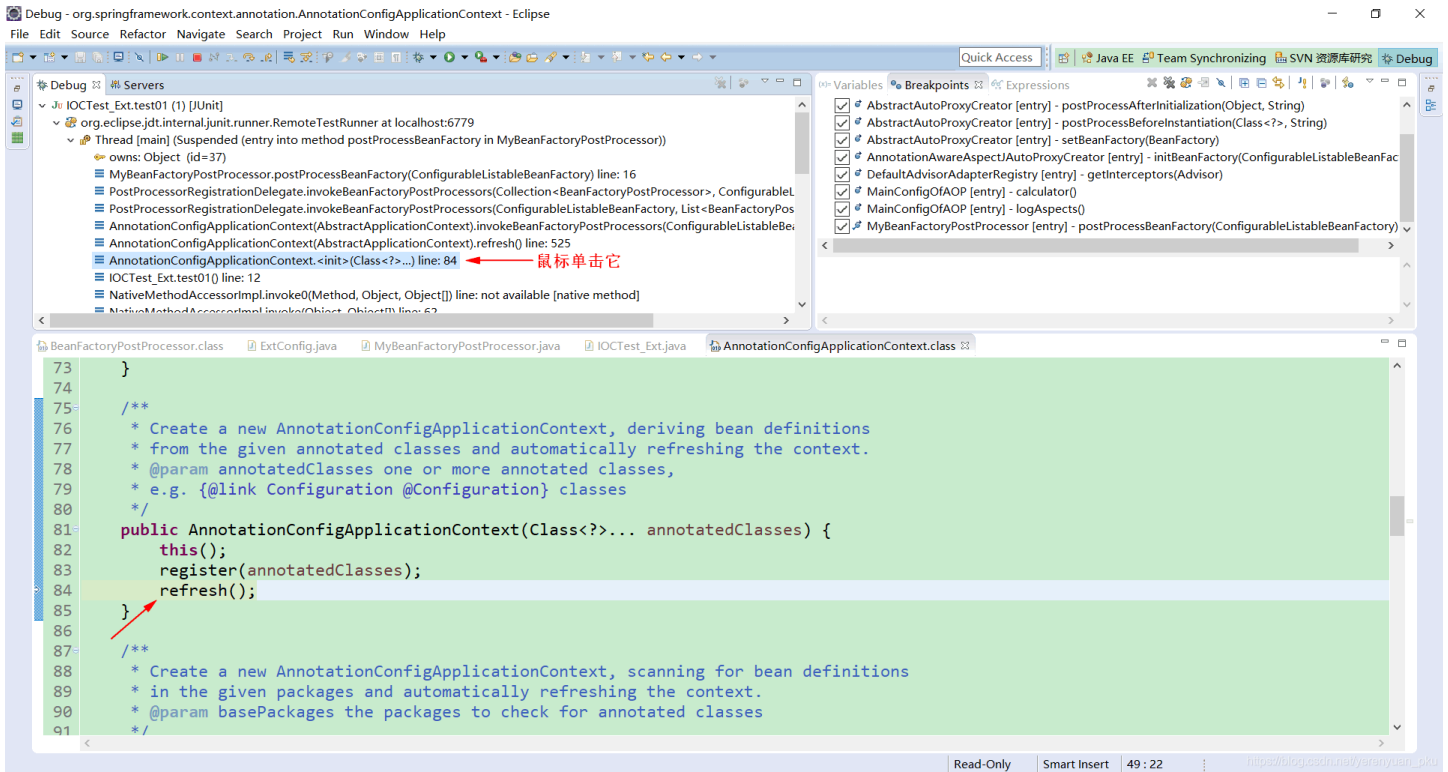
那么程序是怎么运行到这儿的呢？我们不妨从IOCTest\_Ext类中的test01方法开始，来梳理一遍整个流程。

鼠标单击Eclipse左上角方法调用栈中的 `IOCTest_Ext.test01()` line:12，这时程序来到了IOCTest\_Ext类的test01方法中，如下图所示。

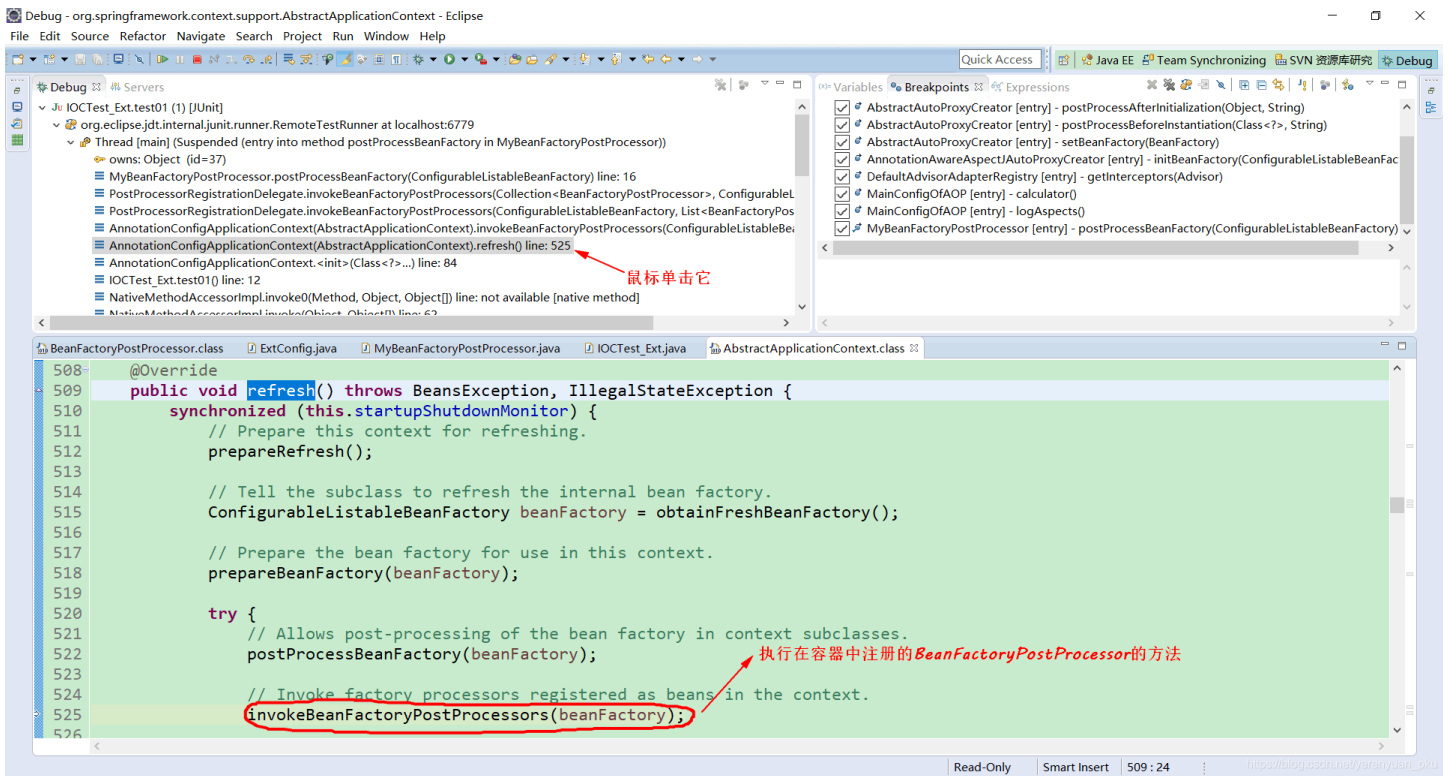


可以看到现在是要来创建IOC容器的。

继续跟进代码，可以看到创建IOC容器时，最后还得刷新容器，如下图所示。

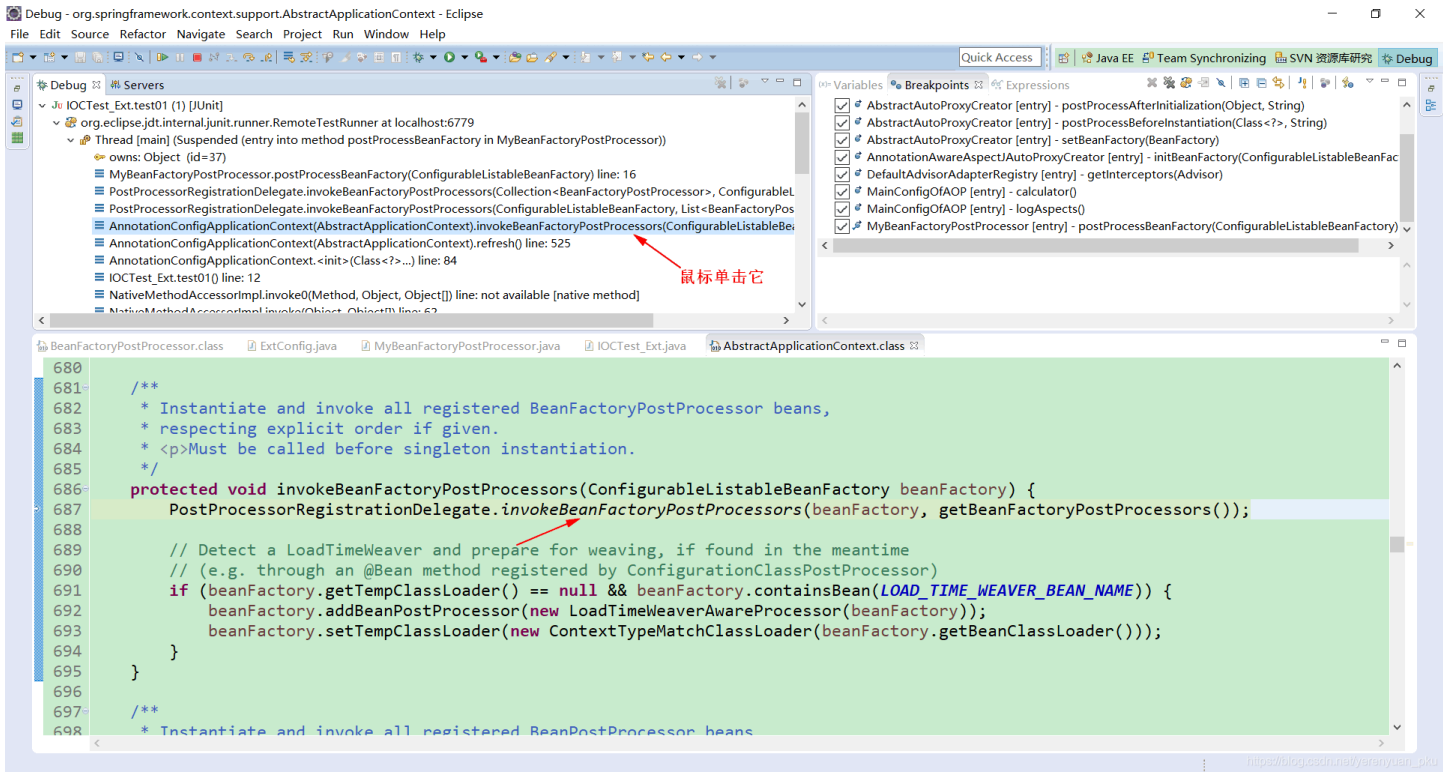


继续跟进代码，可以看到在刷新容器的过程中，还得执行在容器中注册的BeanFactoryPostProcessor（BeanFactory的后置处理器）的方法。

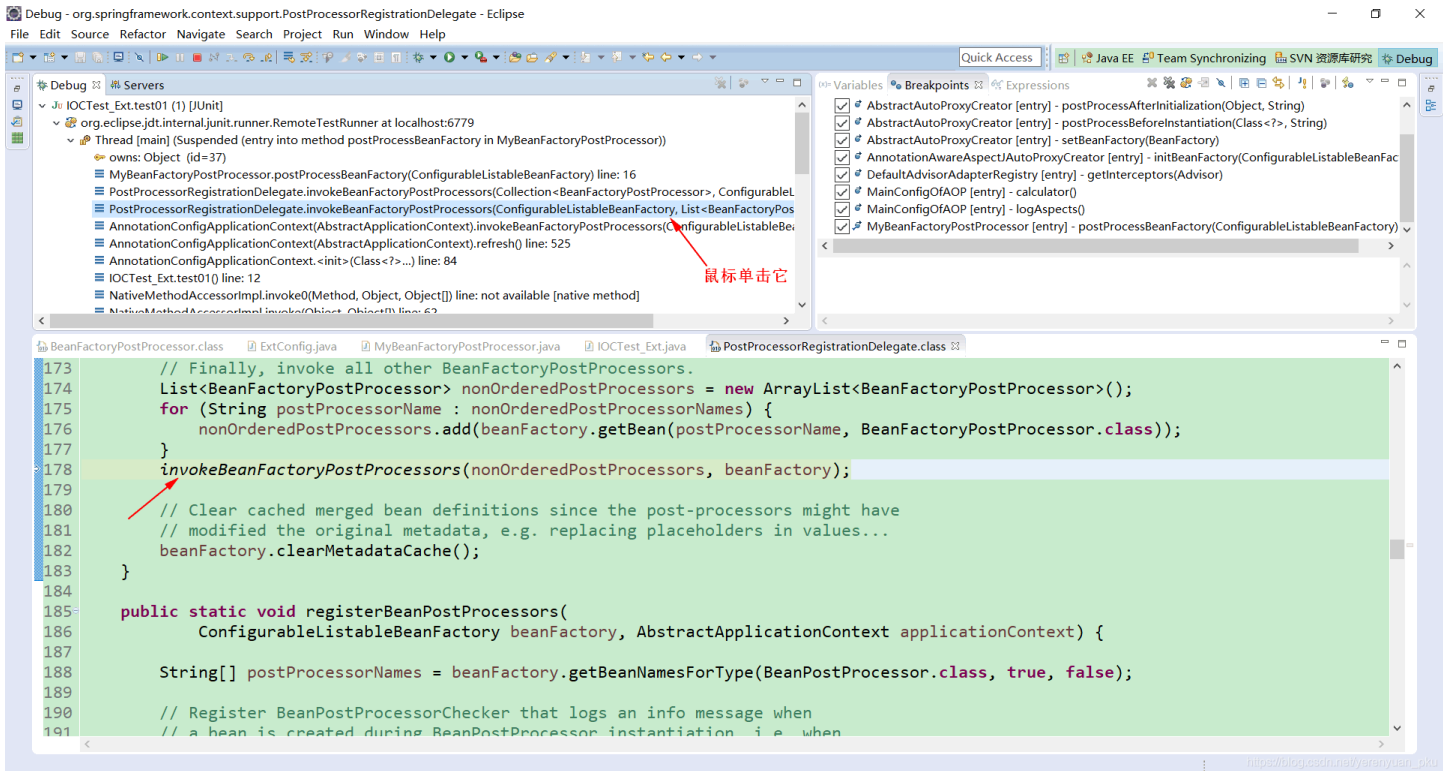


那具体是怎么来执行BeanFactoryPostProcessor的呢？我们继续跟进代码，发现又调用了invokeBeanFactoryPostProcessors方法，如下图所示。





继续跟进代码，可以看到又调用了如下一个invokeBeanFactoryPostProcessors方法。



跟进程到这里，你有没有想过这样一个问题，此时要执行哪些BeanFactoryPostProcessor呢？从以上invokeBeanFactoryPostProcessors方法的参数中，我们可以看到第一个参数代表的是一个List<BeanFactoryPostProcessor>集合，它里面保存的就是那些要执行的BeanFactoryPostProcessor。

```
166 List<BeanFactoryPostProcessor> orderedPostProcessors = new ArrayList<BeanFactoryPostProcessor>();
167 for (String postProcessorName : orderedPostProcessorNames) {
168     orderedPostProcessors.add(beanFactory.getBean(postProcessorName, BeanFactoryPostProcessor.class));
169 }
170 sortPostProcessors(orderedPostProcessors, beanFactory);
171 invokeBeanFactoryPostProcessors(orderedPostProcessors, beanFactory);
172
173 // Finally, invoke all other BeanFactoryPostProcessors.
174 List<BeanFactoryPostProcessor> nonOrderedPostProcessors = new ArrayList<BeanFactoryPostProcessor>();
175 for (String postProcessorName : nonOrderedPostProcessorNames) {
176     nonOrderedPostProcessors.add(beanFactory.getBean(postProcessorName, BeanFactoryPostProcessor.class));
177 }
178 invokeBeanFactoryPostProcessors(nonOrderedPostProcessors, beanFactory);
179
180 // Clear cached merged bean definitions since the post-processors might have
181 // modified the original metadata, e.g. replacing placeholders in values...
182 beanFactory.clearMetadataCache();
183 }
184
```

也就是说现在要执行的BeanFactoryPostProcessor从名为nonOrderedPostProcessors的List<BeanFactoryPostProcessor>集合中拿就可以了。

下面我们来仔细分析一下PostProcessorRegistrationDelegate类中的invokeBeanFactoryPostProcessors方法具体都做了哪些操作。

首先，来看一下如下图所示的这行代码，这行代码说的是拿到所有BeanFactoryPostProcessor组件的名字。

```
135 // Do not initialize FactoryBeans here: We need to leave all regular beans
136 // uninitialized to let the bean factory post-processors apply to them!
137
138 String[] postProcessorNames =
139     beanFactory.getBeanNamesForType(BeanFactoryPostProcessor.class, true, false);
140
141 // Separate between BeanFactoryPostProcessors that implement PriorityOrdered,
142 // Ordered, and the rest.
143 List<BeanFactoryPostProcessor> priorityOrderedPostProcessors = new ArrayList<BeanFactoryPostProcessor>();
144 List<String> orderedPostProcessorNames = new ArrayList<String>();
145 List<String> nonOrderedPostProcessorNames = new ArrayList<String>();
146 for (String ppName : postProcessorNames) {
147     if (processedBeans.contains(ppName)) {
148         // skip - already processed in first phase above
149     }
150     else if (beanFactory.isTypeMatch(ppName, PriorityOrdered.class)) {
151         priorityOrderedPostProcessors.add(beanFactory.getBean(ppName, BeanFactoryPostProcessor.class));
152     }
153     else if (beanFactory.isTypeMatch(ppName, Ordered.class)) {

```

从容器中获取所有BeanFactoryPostProcessor组件的名字

然后，来挨个看相应名字的BeanFactoryPostProcessor组件，哪些是实现了PriorityOrdered接口的，哪些是实现了Ordered接口的，以及哪些是什么接口都没有实现的，说的简单一点就是将不同的BeanFactoryPostProcessor组件给分离出来。

```
136 // Do not initialize FactoryBeans here: We need to leave all regular beans
137 // uninitialized to let the bean factory post-processors apply to them!
138 String[] postProcessorNames =
139     beanFactory.getBeanNamesForType(BeanFactoryPostProcessor.class, true, false);
140
141 // Separate between BeanFactoryPostProcessors that implement PriorityOrdered,
142 // Ordered, and the rest.
143 List<BeanFactoryPostProcessor> priorityOrderedPostProcessors = new ArrayList<BeanFactoryPostProcessor>();
144 List<String> orderedPostProcessorNames = new ArrayList<String>();
145 List<String> nonOrderedPostProcessorNames = new ArrayList<String>();
146 for (String ppName : postProcessorNames) {
147     if (processedBeans.contains(ppName)) {
148         // skip - already processed in first phase above
149     }
150     else if (beanFactory.isTypeMatch(ppName, PriorityOrdered.class)) {
151         priorityOrderedPostProcessors.add(beanFactory.getBean(ppName, BeanFactoryPostProcessor.class));
152     }
153     else if (beanFactory.isTypeMatch(ppName, Ordered.class)) {
154         orderedPostProcessorNames.add(ppName);
155     }
156     else {
157         nonOrderedPostProcessorNames.add(ppName);
158     }
159 }
```

如果BeanFactoryPostProcessor组件实现了PriorityOrdered接口，那么就将其保存在名为priorityOrderedPostProcessors的List<BeanFactoryPostProcessor>集合中

如果BeanFactoryPostProcessor组件实现了Ordered接口，那么就将其名字保存在名为orderedPostProcessorNames的List<String>集合中

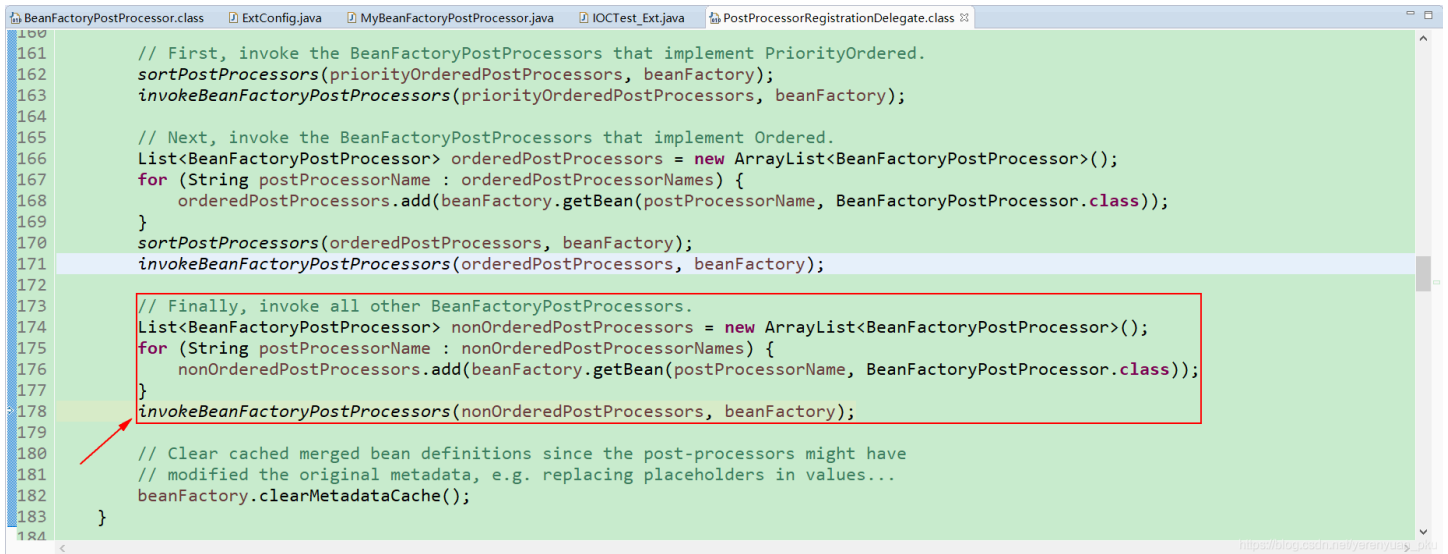
如果BeanFactoryPostProcessor组件什么接口都没有实现，那么就将其名字保存在名为nonOrderedPostProcessorNames的List<String>集合中

接着，分别按不同的执行顺序来处理三种不同的BeanFactoryPostProcessor组件。

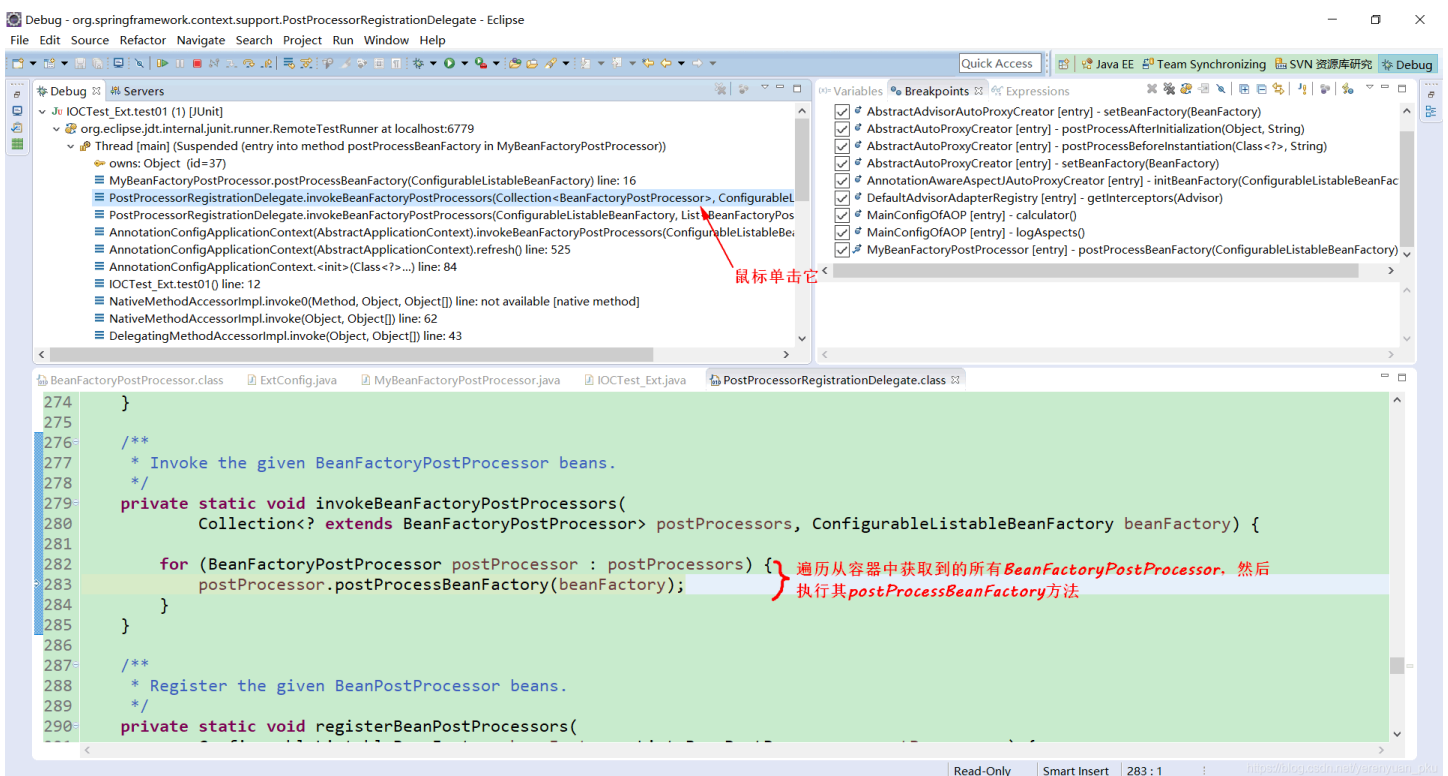




由于咱们自己编写的BeanFactoryPostProcessor既没有实现PriorityOrdered接口，也没有实现Ordered接口，所以就按照最后一种顺序来执行。



以上就是对PostProcessorRegistrationDelegate类中的invokeBeanFactoryPostProcessors方法大致分析。分析完之后，我们继续跟进代码，会发现其遍历了所有的BeanFactoryPostProcessor组件，我们自己编写的实现了BeanFactoryPostProcessor接口的MyBeanFactoryPostProcessor类肯定也属于其中，所以会被遍历到，然后便会执行其postProcessBeanFactory方法。



以上就是我们从源码的角度分析了一下BeanFactoryPostProcessor的整个执行顺序以及原理。

## 小结

经过以上源码分析，我们可以得出这样一个简单结论：首先从IOC容器中找到所有类型是BeanFactoryPostProcessor的组件，然后再来执行它们其中的方法，而且是在初始化创建其他组件前面执行。

我为什么可以这么肯定地说呢？如果我们大家还有回忆的话，那么你一定记得在我讲解 **AOP原理** 的时候，bean的创建与初始化还在很后面，如下图所示。

