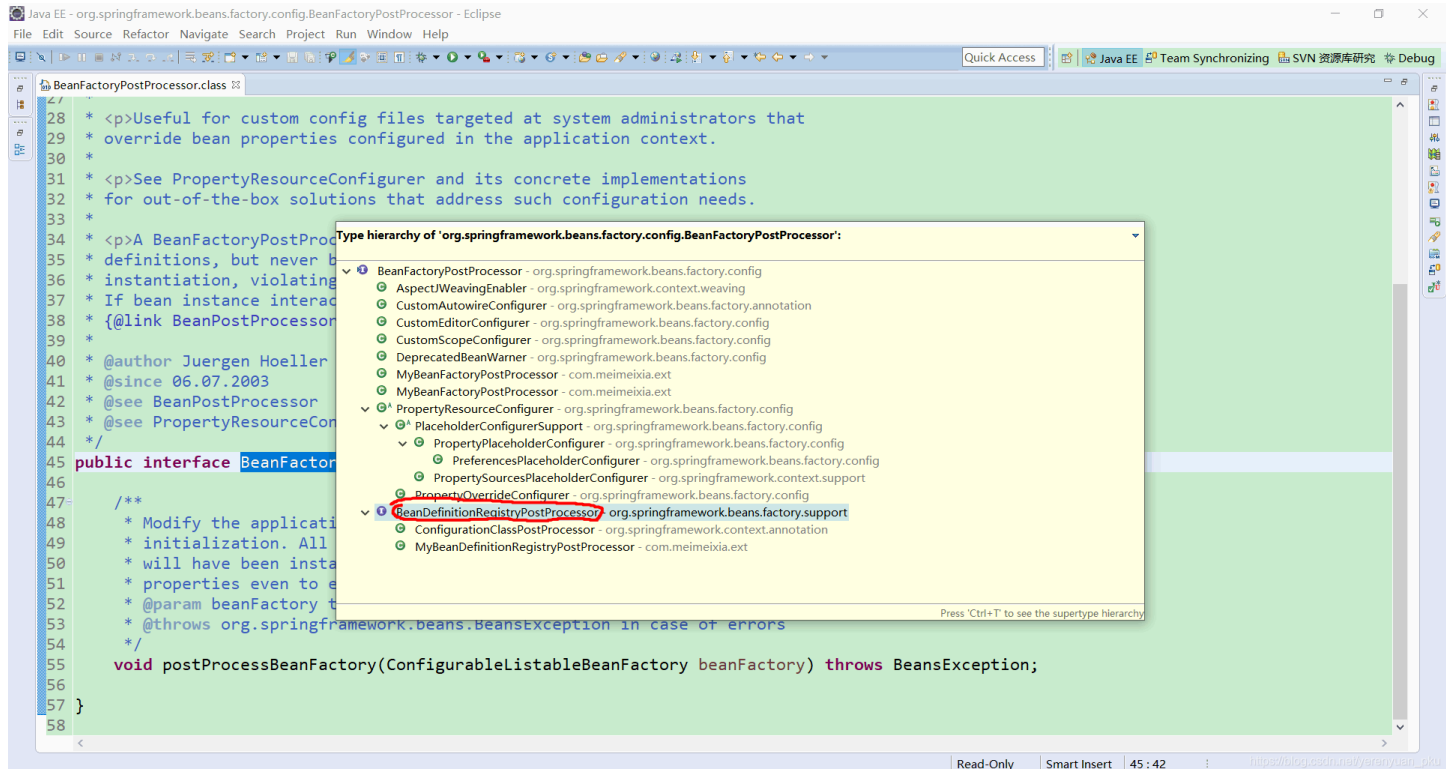


Spring注解驱动开发第37讲——你知道Spring中BeanDefinitionRegistryPostProcessor是如何执行的吗？

写在前面

在上一讲中，我们学习了一下BeanFactoryPostProcessor接口，了解了一下它是怎样使用的，以及其内部原理，我们知道，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。

接下来，我们就要学习一下BeanFactoryPostProcessor的一个子接口，即BeanDefinitionRegistryPostProcessor。



从上图中可以看到BeanDefinitionRegistryPostProcessor是BeanFactoryPostProcessor旗下的一个子接口。

从源码角度理解BeanDefinitionRegistryPostProcessor的原理

初步认识一下BeanDefinitionRegistryPostProcessor

首先，咱们来看一下BeanDefinitionRegistryPostProcessor的源码，如下图所示。



从该接口的名字中，我们大概能知道个一二，说它是 **bean定义** 注册中心的后置处理器并不过分。而且，从该接口的源码中我们也可以看出，它是 BeanFactoryPostProcessor 旗下的一个子接口。

我们还能看到，它里面定义了一个方法，叫postProcessBeanDefinitionRegistry，那么问题来了，它是什么时候执行的呢？我们可以看一下它上面的详细描述，说的是啥呢，说的是在IOC容器标准初始化之后，允许我们来修改IOC容器里面的bean定义注册中心。此时，所有合法的bean定义将要被加载，但是这些bean还没有初始化完成。

说人话就是，**postProcessBeanDefinitionRegistry**方法的执行时机是在所有bean定义信息将要被加载，但是bean实例还未创建的时候。这句话听起来，总感觉 BeanDefinitionRegistryPostProcessor是在BeanFactoryPostProcessor前面执行的，真的是这样吗？确实是这样。为什么呢？BeanFactoryPostProcessor的执行时机是在所有的bean定义信息已经保存加载到BeanFactory中之后，而BeanDefinitionRegistryPostProcessor却是在所有的bean定义信息将要被加载的时候，所以，BeanDefinitionRegistryPostProcessor就应该要先来执行。接下来，我们就写一个实践案例来验证一番。

案例实践

首先，编写一个类，例如MyBeanDefinitionRegistryPostProcessor，它应要实现BeanDefinitionRegistryPostProcessor这个接口。

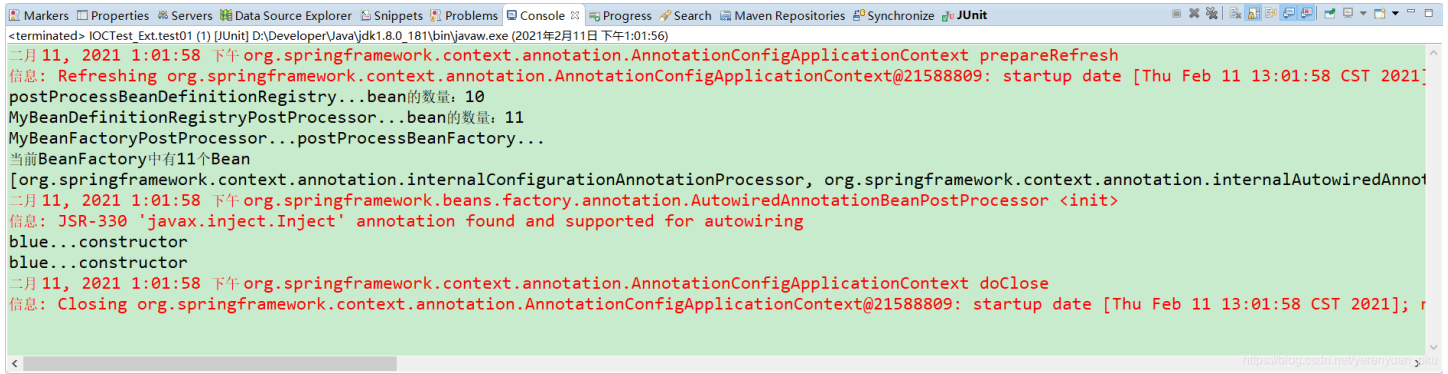
```
1 package com.meimeixia.ext;
2
3 import org.springframework.beans.BeansException;
4 import org.springframework.beans.factory.config.ConfigurableListableBeanFactory;
5 import org.springframework.beans.factory.support.AbstractBeanDefinition;
6 import org.springframework.beans.factory.support.BeanDefinitionBuilder;
7 import org.springframework.beans.factory.support.BeanDefinitionRegistry;
8 import org.springframework.beans.factory.support.BeanDefinitionRegistryPostProcessor;
9 import org.springframework.beans.factory.support.RootBeanDefinition;
10 import org.springframework.stereotype.Component;
11
12 import com.meimeixia.bean.Blue;
13
14 // 记住，我们这个组件写完之后，一定别忘了给它加在容器中
15 @Component
16 public class MyBeanDefinitionRegistryPostProcessor implements BeanDefinitionRegistryPostProcessor {
17
18     @Override
19     public void postProcessBeanFactory(ConfigurableListableBeanFactory beanFactory) throws BeansException {
20         // TODO Auto-generated method stub
21         System.out.println("MyBeanDefinitionRegistryPostProcessor...bean的数量: " + beanFactory.getBeanDefinitionCount());
22     }
23
24     /**
25      * 这个BeanDefinitionRegistry就是Bean定义信息的保存中心，这个注册中心里面存储了所有的bean定义信息，
26      * 以后，BeanFactory就是按照BeanDefinitionRegistry里面保存的每一个bean定义信息来创建bean实例的。
27      *
28      * bean定义信息包括有哪些呢？有这些，这个bean是单例的还是多例的、bean的类型是什么以及bean的id是什么。
29      * 也就是说，这些信息都是存在BeanDefinitionRegistry里面的。
30      */
31     @Override
32     public void postProcessBeanDefinitionRegistry(BeanDefinitionRegistry registry) throws BeansException {
33         // TODO Auto-generated method stub
34         System.out.println("postProcessBeanDefinitionRegistry...bean的数量: " + registry.getBeanDefinitionCount());
35         // 除了查看bean的数量之外，我们还可以给容器里面注册一些bean，我们以前也简单地用过
36         /**
37          * 第一个参数：我们将要给容器中注册的bean的名字
38          * 第二个参数：BeanDefinition对象
39          */
40         // RootBeanDefinition beanDefinition = new RootBeanDefinition(Blue.class); // 现在我准备给容器中添加一个Blue对象
41         // 咱们也可以用另外一种办法，即使用BeanDefinitionBuilder这个构建器生成一个BeanDefinition对象，很显然，这两种方法的效果都是一样的
42         AbstractBeanDefinition beanDefinition = BeanDefinitionBuilder.rootBeanDefinition(Blue.class).getBeanDefinition();
43         registry.registerBeanDefinition("hello", beanDefinition);
44     }
45
46 }
```

AI写代码java运行



咱们编写的类实现BeanDefinitionRegistryPostProcessor接口之后，还得来实现两个方法，第一个方法，即postProcessBeanFactory，它来源于 BeanFactoryPostProcessor接口里面定义的方法；第二个方法，即postProcessBeanDefinitionRegistry，它来源于BeanDefinitionRegistryPostProcessor接口里面定义的方法。

接下来，我们就来测试一下以上类里面的两个方法是什么时候执行的。运行IOCTest_Ext测试类中的test01方法，可以看到Eclipse 控制台打印了如下内容。



```
<terminated> IOCTest_Ext.test01 (1) [JUnit] D:\Developer\Java\jdk1.8.0_181\bin\javaw.exe (2021年2月11日 下午1:01:56)
二月 11, 2021 1:01:58 下午 org.springframework.context.annotation.AnnotationConfigApplicationContext prepareRefresh
信息: Refreshing org.springframework.context.annotation.AnnotationConfigApplicationContext@21588809: startup date [Thu Feb 11 13:01:58 CST 2021]
postProcessBeanDefinitionRegistry...bean的数量: 10
MyBeanDefinitionRegistryPostProcessor...bean的数量: 11
MyBeanFactoryPostProcessor...postProcessBeanFactory...
当前BeanFactory中有11个Bean
[org.springframework.context.annotation.internalConfigurationAnnotationProcessor, org.springframework.context.annotation.internalAutowiredAnnotationProcessor]
二月 11, 2021 1:01:58 下午 org.springframework.beans.factory.annotation.AutowiredAnnotationBeanPostProcessor <init>
信息: JSR-330 'javax.inject.Inject' annotation found and supported for autowiring
blue...constructor
blue...constructor
二月 11, 2021 1:01:58 下午 org.springframework.context.annotation.AnnotationConfigApplicationContext doClose
信息: Closing org.springframework.context.annotation.AnnotationConfigApplicationContext@21588809: startup date [Thu Feb 11 13:01:58 CST 2021]; root of context hierarchy
```

可以看到，是我们自己写的MyBeanDefinitionRegistryPostProcessor类里面的postProcessBeanDefinitionRegistry方法先执行，该方法具体都做了哪些事呢？它先是拿到IOC容器中bean的数量（即10），再是向IOC容器中注册一个组件。接着，是我们自己写的MyBeanDefinitionRegistryPostProcessor类里面的postProcessBeanFactory方法再执行，该方法只是打印了一下IOC容器中bean的数量。你不仅要问了，为什么打印出的IOC容器中bean的数量是11，而不是10呢？这是因为我们之前已经向IOC容器中注册了一个组件。

除此之外，从Eclipse控制台输出的结果中我们还能看到，我们自己写的MyBeanDefinitionRegistryPostProcessor类里面的方法都执行完了以后，才轮到外面那些BeanFactoryPostProcessor来执行，执行的时候，不仅输出了IOC容器中bean的数量，而且还输出了每一个bean定义的名字。

现在是不是可以得出这样一个结论，**BeanDefinitionRegistryPostProcessor**是优先于**BeanFactoryPostProcessor**执行的，而且我们可以利用它给容器中再额外添加一些组件。

源码分析

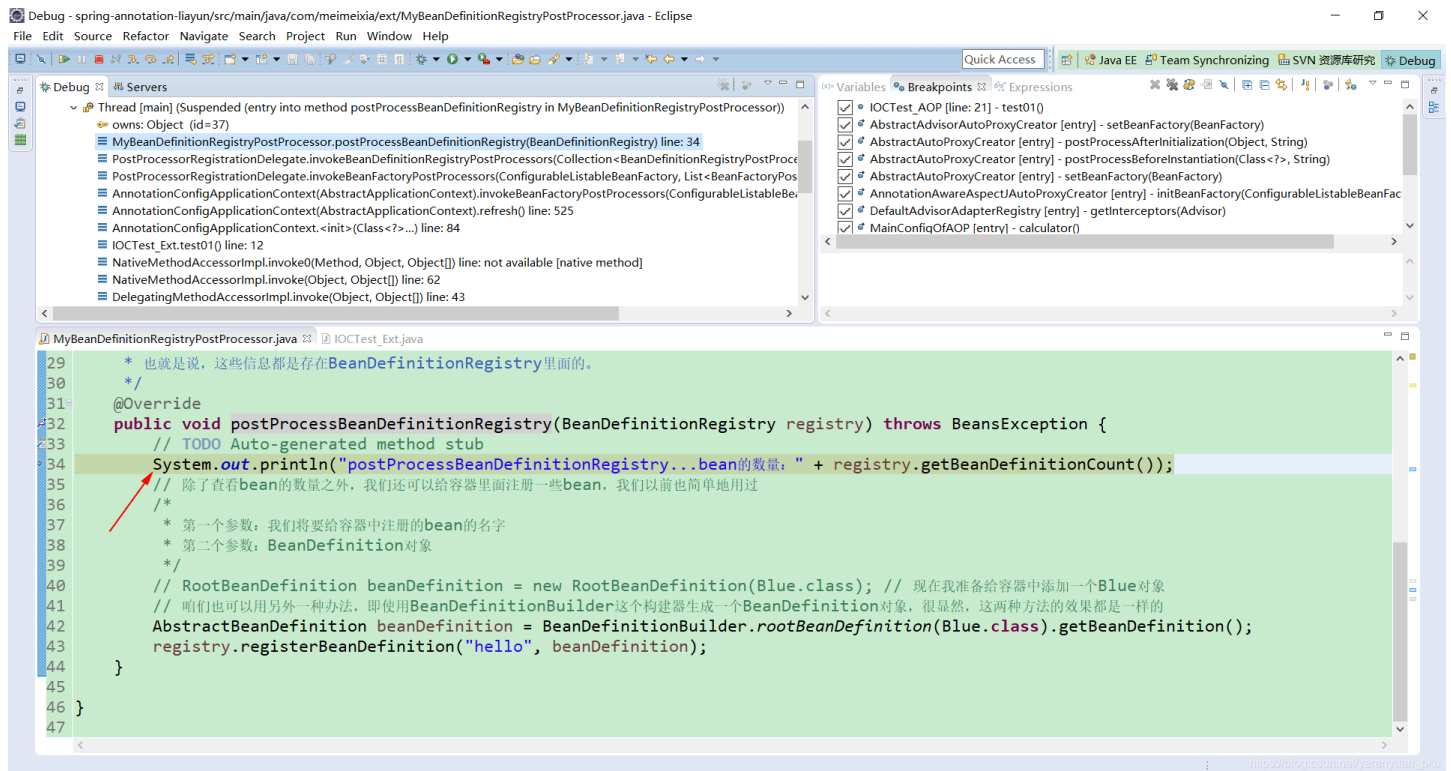
为什么BeanDefinitionRegistryPostProcessor是优先于BeanFactoryPostProcessor执行的呢？我们可以从源码的角度来深入分析一下。

首先，在我们自己写的MyBeanDefinitionRegistryPostProcessor类里面的两个方法上都打上一个断点，如下图所示。



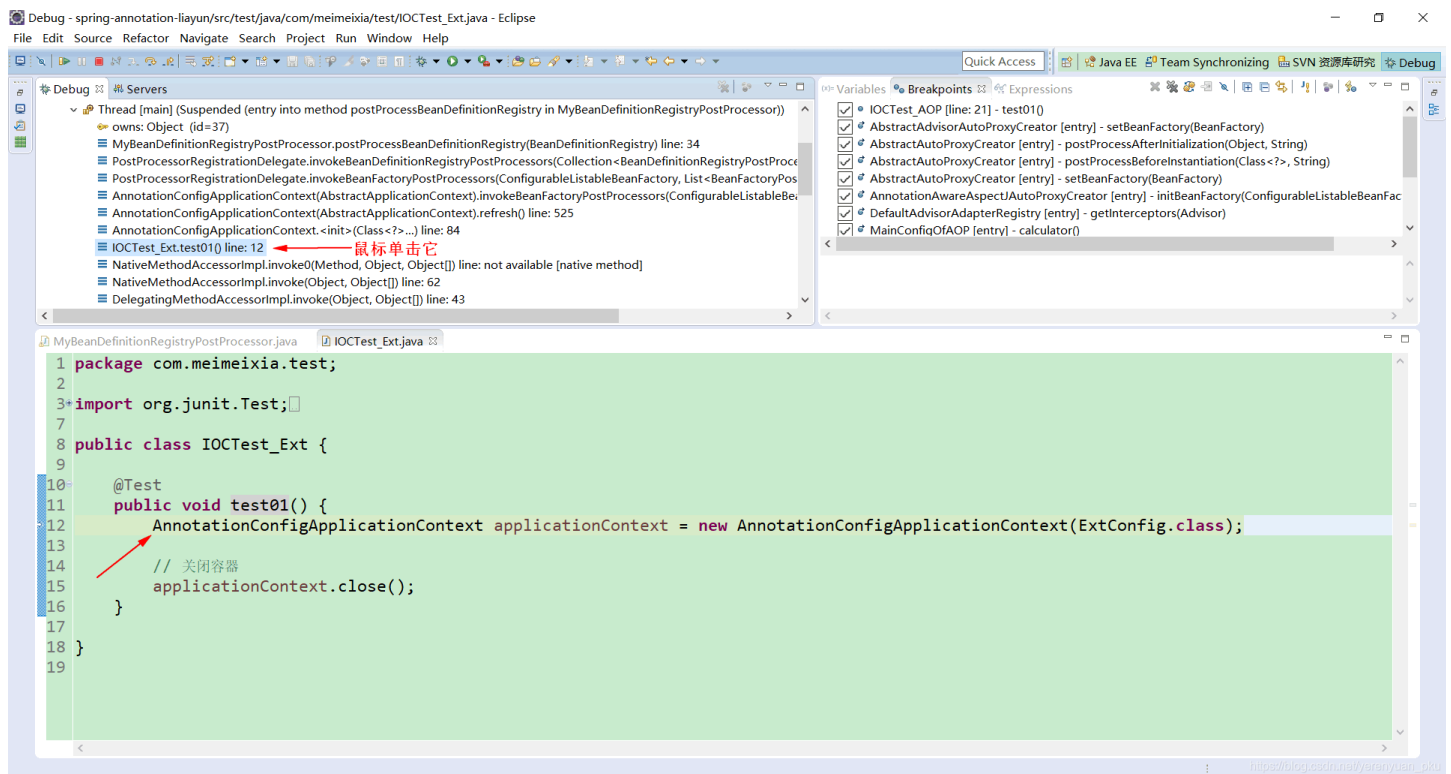
```
MyBeanDefinitionRegistryPostProcessor.java  IOCTest_Ext.java
14 // 记住，我们这个组件写完之后，一定别忘了给它加在容器中
15 @Component
16 public class MyBeanDefinitionRegistryPostProcessor implements BeanDefinitionRegistryPostProcessor {
17     // 先在postProcessBeanFactory方法处打上一个断点
18     @Override
19     public void postProcessBeanFactory(ConfigurableListableBeanFactory beanFactory) throws BeansException {
20         // TODO Auto-generated method stub
21         System.out.println("MyBeanDefinitionRegistryPostProcessor...bean的数量: " + beanFactory.getBeanDefinitionCount());
22     }
23
24     /**
25      * 这个BeanDefinitionRegistry就是Bean定义信息的保存中心，这个注册中心里面存储了所有的bean定义信息，
26      * 以后，BeanFactory就是按照BeanDefinitionRegistry里面保存的每一个bean定义信息来创建bean实例的。
27      * bean定义信息包括有哪些呢？有这些，这个bean是单例的还是多例的、bean的类型是什么以及bean的id是什么。
28      * 也就是说，这些信息都是存在BeanDefinitionRegistry里面的。
29      */
30     // 再在postProcessBeanDefinitionRegistry方法处打上一个断点
31     @Override
32     public void postProcessBeanDefinitionRegistry(BeansDefinitionRegistry registry) throws BeansException {
33         // TODO Auto-generated method stub
34         System.out.println("postProcessBeanDefinitionRegistry...bean的数量: " + registry.getBeanDefinitionCount());
35         // 除了查看bean的数量之外，我们还可以给容器里面注册一些bean，我们以前也简单地用过
36         /*
37          * 第一个参数：我们要给容器中注册的bean的名字
38          * 第二个参数：BeanDefinition对象
39          */
40         // RootBeanDefinition beanDefinition = new RootBeanDefinition(Blue.class); // 现在我准备给容器中添加一个Blue对象
41         // 咱们也可以用另外一种办法，即使用BeanDefinitionBuilder这个构建器生成一个BeanDefinition对象，很显然，这两种方法的效果都是一样的
42         AbstractBeanDefinition beanDefinition = BeanDefinitionBuilder.rootBeanDefinition(Blue.class).getBeanDefinition();
43         registry.registerBeanDefinition("hello", beanDefinition);
44     }
45 }
```

然后，以debug的方式运行IOCTest_Ext测试类中的test01方法，如下图所示，程序现在停到了MyBeanDefinitionRegistryPostProcessor类里面的postProcessBeanDefinitionRegistry方法处。



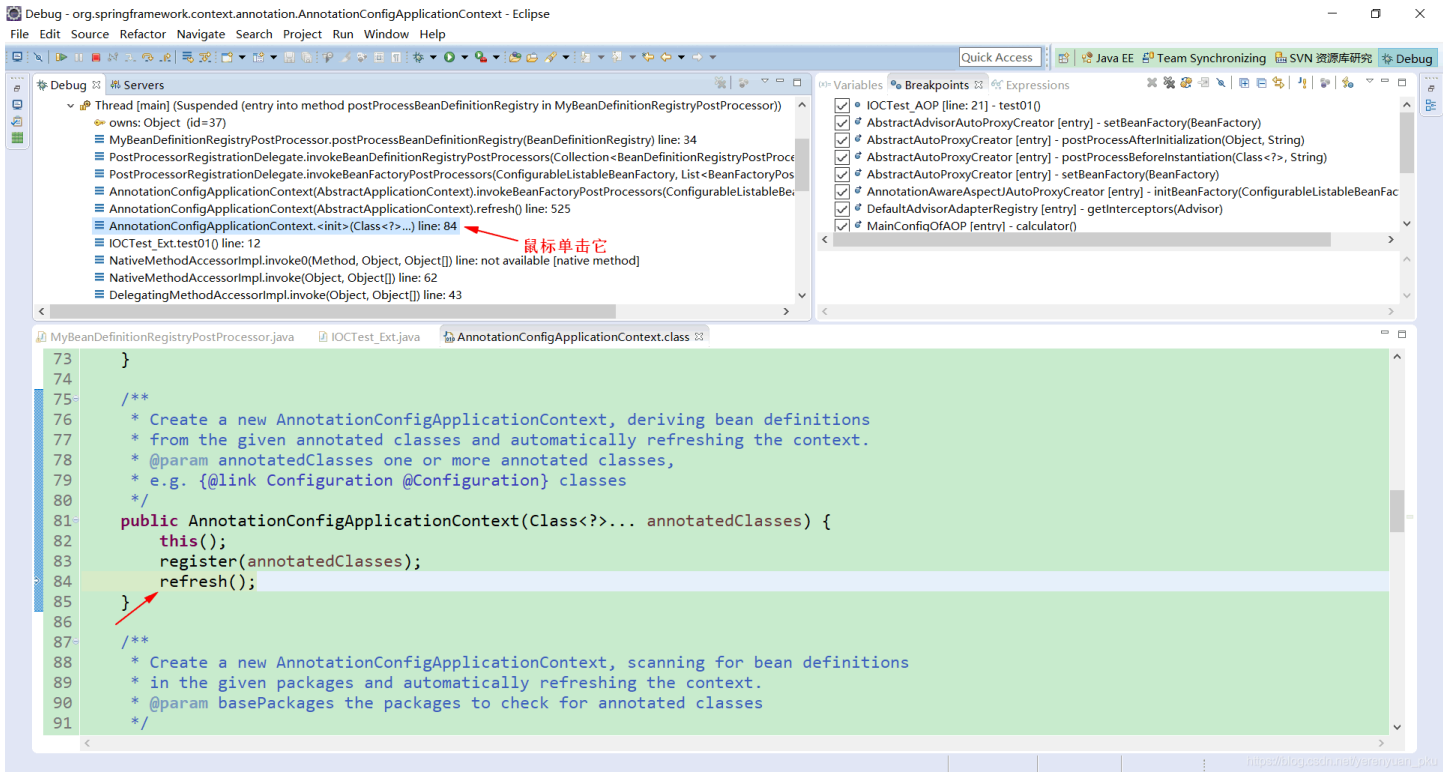
那么程序是怎么运行到这儿的呢？我们不妨从IOCTest_Ext测试类中的test01方法开始，来梳理一遍整个流程。

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测试类的test01方法中，如下图所示。

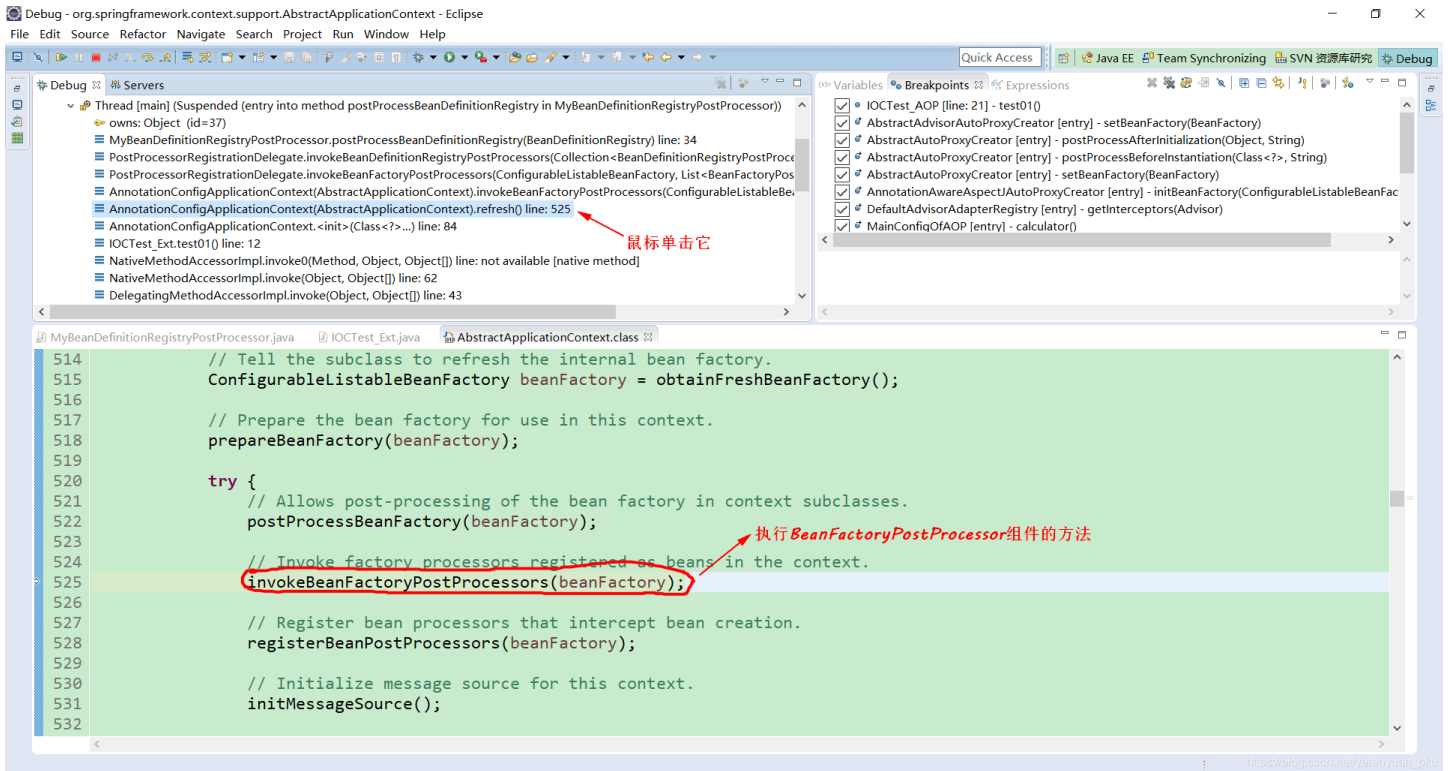


可以看到现在是要来创建IOC容器的。

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。

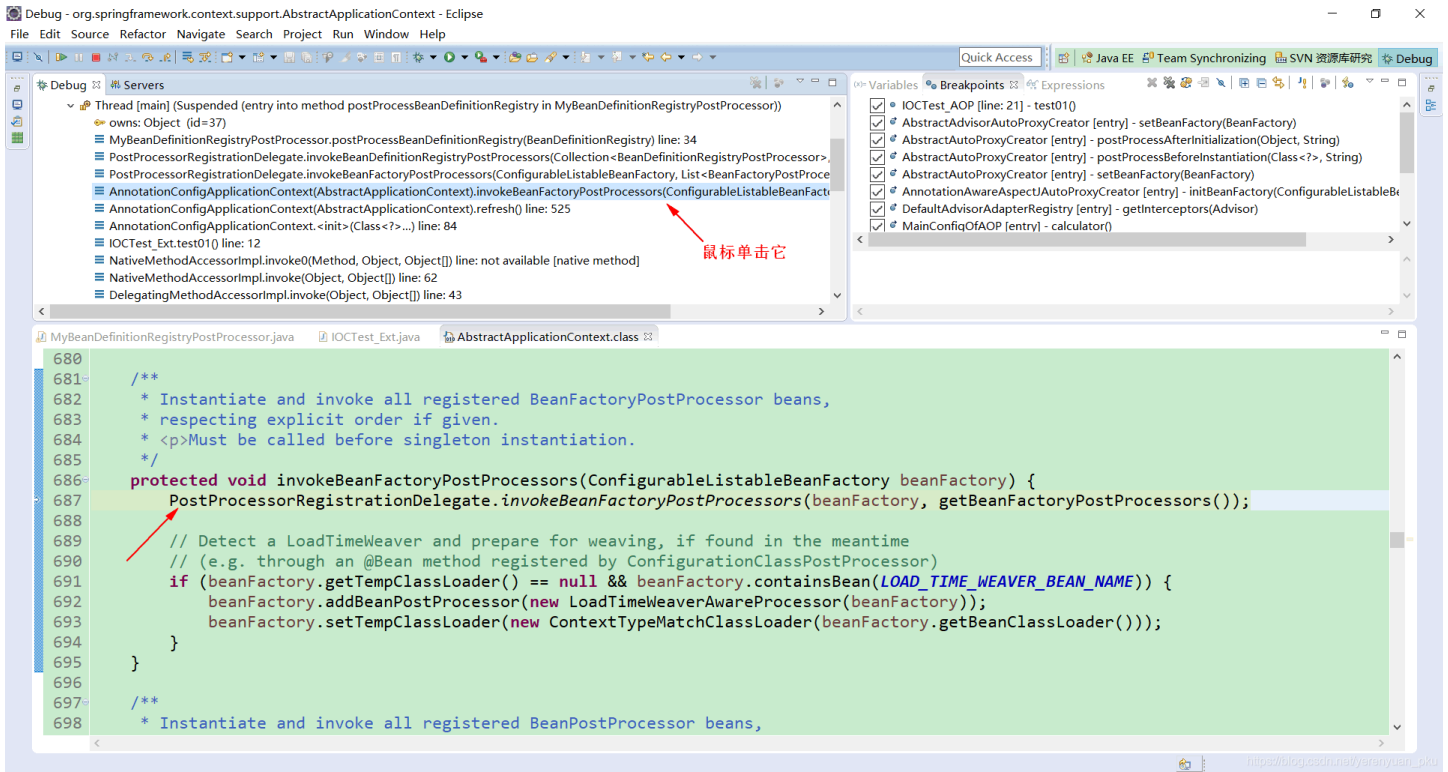


继续跟进代码，看这个`refresh` 方法里面具体都做了些啥，如下图所示，可以看到它里面调用了如下一个`invokeBeanFactoryPostProcessors`方法。

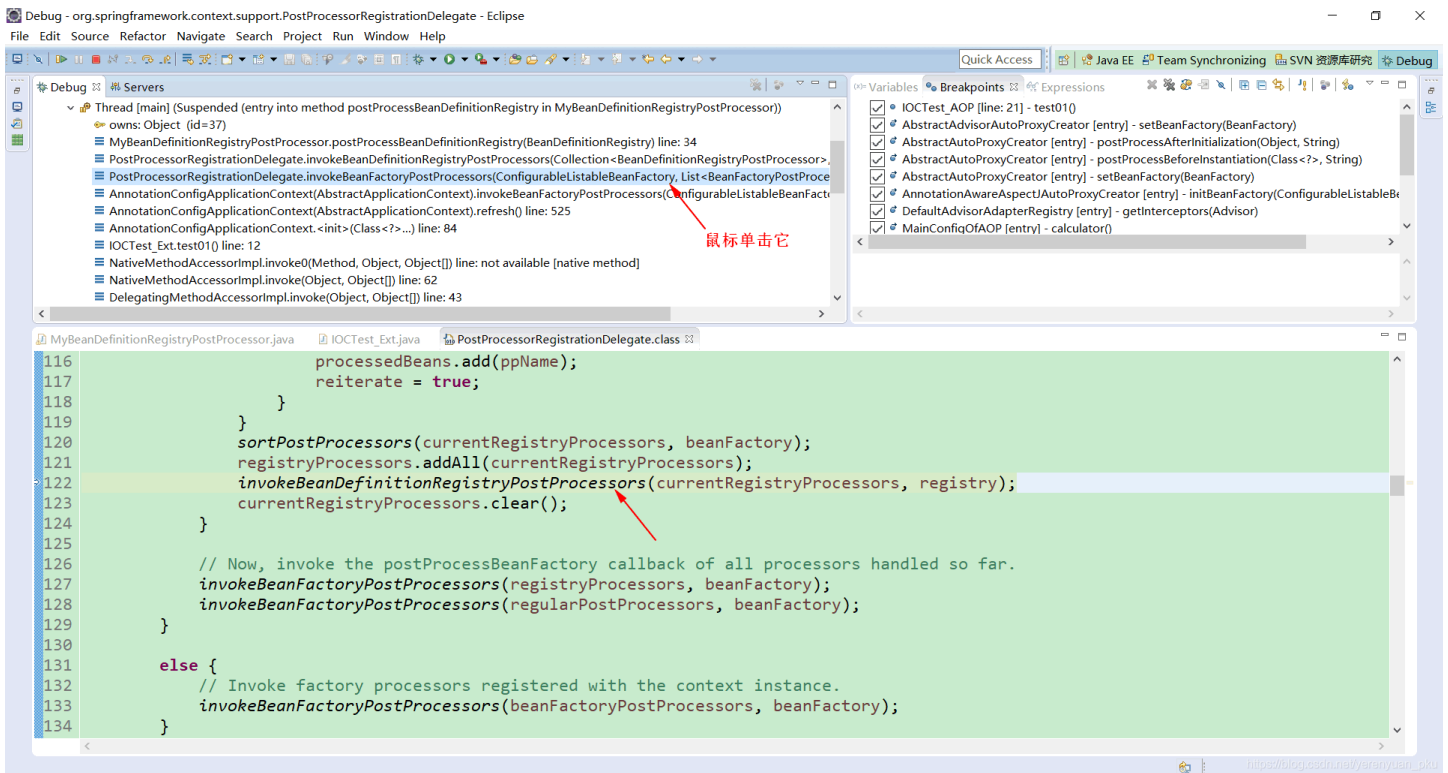


其实这跟我们上一讲中分析`BeanFactoryPostProcessor`的原理是一模一样的，它也是在IOC容器创建对象的时候，会来调用`invokeBeanFactoryPostProcessors`这个方法。既然都是调用这个方法，那怎么能说`BeanDefinitionRegistryPostProcessor`就要优先于`BeanFactoryPostProcessor`执行呢？

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invokeBeanDefinitionRegistryPostProcessors方法。



大家一定要注意这个方法哟！看清楚，这个方法的名字叫invokeBeanDefinitionRegistryPostProcessors。此外，你还能看到传递进该方法的第一个参数是currentRegistryProcessors，那它又是在哪儿定义的呢？这就不得不好好看看PostProcessorRegistrationDelegate类中的invokeBeanFactoryPostProcessors方法了。

我们仔细查看该方法，会发现刚进入该方法时，就说明了不少时候都会优先调用BeanDefinitionRegistryPostProcessor。由于我们自己写的MyBeanDefinitionRegistryPostProcessor类实现了这个接口，所以它肯定会被先调用。

```
MyBeanDefinitionRegistryPostProcessor.java  IOCTest_Ext.java  PostProcessorRegistrationDelegate.class
50 class PostProcessorRegistrationDelegate {
51
52     public static void invokeBeanFactoryPostProcessors(
53         ConfigurableListableBeanFactory beanFactory, List<BeanFactoryPostProcessor> beanFactoryPostProcessors) {
54
55         // Invoke BeanDefinitionRegistryPostProcessors first, if any.
56         Set<String> processedBeans = new HashSet<String>();
57
58         if (beanFactory instanceof BeanDefinitionRegistry) {
59             BeanDefinitionRegistry registry = (BeanDefinitionRegistry) beanFactory;
60             List<BeanFactoryPostProcessor> regularPostProcessors = new LinkedList<BeanFactoryPostProcessor>();
61             List<BeanDefinitionRegistryPostProcessor> registryProcessors = new LinkedList<BeanDefinitionRegistryPostProcessor>();
62
63             for (BeanFactoryPostProcessor postProcessor : beanFactoryPostProcessors) {
64                 if (postProcessor instanceof BeanDefinitionRegistryPostProcessor) {
65                     BeanDefinitionRegistryPostProcessor registryProcessor =
66                         (BeanDefinitionRegistryPostProcessor) postProcessor;
67                     registryProcessor.postProcessBeanDefinitionRegistry(registry);
68                     registryProcessors.add(registryProcessor);
69                 }
70                 else {
71                     regularPostProcessors.add(postProcessor);
72                 }
73             }
74
75             // Do not initialize FactoryBeans here: We need to leave all regular beans
76             // uninitialized to let the bean factory post-processors apply to them!
```

继续向下看，可以看到会取出所有实现了BeanDefinitionRegistryPostProcessor接口的类，即从容器中获取到所有的BeanDefinitionRegistryPostProcessor组件。然后，优先调用实现了PriorityOrdered接口的BeanDefinitionRegistryPostProcessor组件。

```
MyBeanDefinitionRegistryPostProcessor.java  IOCTest_Ext.java  PostProcessorRegistrationDelegate.class
74
75     // Do not initialize FactoryBeans here: We need to leave all regular beans
76     // uninitialized to let the bean factory post-processors apply to them!
77     // Separate between BeanDefinitionRegistryPostProcessors that implement
78     // PriorityOrdered, Ordered, and the rest.
79     List<BeanDefinitionRegistryPostProcessor> currentRegistryProcessors = new ArrayList<BeanDefinitionRegistryPostProcessor>()
80
81     // First, invoke the BeanDefinitionRegistryPostProcessors that implement PriorityOrdered.
82     String[] postProcessorNames =
83         beanFactory.getBeanNamesForType(BeanDefinitionRegistryPostProcessor.class, true, false);
84     for (String ppName : postProcessorNames) {
85         if (beanFactory.isTypeMatch(ppName, PriorityOrdered.class)) {
86             currentRegistryProcessors.add(beanFactory.getBean(ppName, BeanDefinitionRegistryPostProcessor.class));
87             processedBeans.add(ppName);
88         }
89     }
90     sortPostProcessors(currentRegistryProcessors, beanFactory);
91     registryProcessors.addAll(currentRegistryProcessors);
92     invokeBeanDefinitionRegistryPostProcessors(currentRegistryProcessors, registry);
93     currentRegistryProcessors.clear();
94
```

现在你该知道参数currentRegistryProcessors是在哪儿定义了吧，它其实就是一个保存BeanDefinitionRegistryPostProcessor组件的List集合。

调用完实现了PriorityOrdered接口的BeanDefinitionRegistryPostProcessor组件之后，接着会再调用实现了Ordered接口的BeanDefinitionRegistryPostProcessor组件。

```
MyBeanDefinitionRegistryPostProcessor.java  IOCTest_Ext.java  PostProcessorRegistrationDelegate.class
95     // Next, invoke the BeanDefinitionRegistryPostProcessors that implement Ordered
96     postProcessorNames = beanFactory.getBeanNamesForType(BeanDefinitionRegistryPostProcessor.class, true, false);
97     for (String ppName : postProcessorNames) {
98         if (!processedBeans.contains(ppName) && beanFactory.isTypeMatch(ppName, Ordered.class)) {
99             currentRegistryProcessors.add(beanFactory.getBean(ppName, BeanDefinitionRegistryPostProcessor.class));
100             processedBeans.add(ppName);
101         }
102     }
103     sortPostProcessors(currentRegistryProcessors, beanFactory);
104     registryProcessors.addAll(currentRegistryProcessors);
105     invokeBeanDefinitionRegistryPostProcessors(currentRegistryProcessors, registry);
106     currentRegistryProcessors.clear();
107
108     // Finally, invoke all other BeanDefinitionRegistryPostProcessors until no further ones appear.
```

最后再来调用剩余其他的BeanDefinitionRegistryPostProcessor组件，例如我们自己编写的MyBeanDefinitionRegistryPostProcessor类。

```

104 registryProcessors.addAll(currentRegistryProcessors);
105 invokeBeanDefinitionRegistryPostProcessors(currentRegistryProcessors, registry);
106 currentRegistryProcessors.clear();
107
108 // Finally, invoke all other BeanDefinitionRegistryPostProcessors until no further ones appear.
109 boolean reiterate = true;
110 while (reiterate) {
111     reiterate = false;
112     postProcessorNames = beanFactory.getBeanNamesForType(BeaDefinitionRegistryPostProcessor.class, true, false);
113     for (String ppName : postProcessorNames) {
114         if (!processedBeans.contains(ppName)) {
115             currentRegistryProcessors.add(beanFactory.getBean(ppName, BeanDefinitionRegistryPostProcessor.class));
116             processedBeans.add(ppName);
117             reiterate = true;
118         }
119     }
120     sortPostProcessors(currentRegistryProcessors, beanFactory);
121     registryProcessors.addAll(currentRegistryProcessors);
122     invokeBeanDefinitionRegistryPostProcessors(currentRegistryProcessors, registry);
123     currentRegistryProcessors.clear();
124 }
125

```

OK, 现在程序停留在了上图所示的地方, 很显然, 此时已经从容器中获取到了所有的BeanDefinitionRegistryPostProcessor组件, 说是所有, 但实际上现在就只获取到了一个, 即我们自己编写的MyBeanDefinitionRegistryPostProcessor类, 如下图所示。

```

118     }
119     sortPostProcessors(currentRegistryProcessors, beanFactory);
120     registryProcessors.addAll(currentRegistryProcessors);
121     invokeBeanDefinitionRegistryPostProcessors(currentRegistryProcessors, registry);
122     currentRegistryProcessors.clear();
123 }
124
125 // Now, invoke the postProcessBeanFactory callb
126 invokeBeanFactoryPostProcessors(registryProcess
127 invokeBeanFactoryPostProcessors(regularPostProc
128 }
129
130 else {
131     // Invoke factory processors registered with the context instance.
132     invokeBeanFactoryPostProcessors(beanFactoryPostProcessors, beanFactory);
133 }
134
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 String[] postProcessorNames =
138     beanFactory.getBeanNamesForType(BeaDefinitionRegistryPostProcessor.class, true, false);
139

```

继续往下跟进代码, 可以看到现在所做的事情就是从容器中获取到所有的BeanDefinitionRegistryPostProcessor组件之后, 再来依次调用它们的postProcessBeanDefinitionRegistry方法。

```

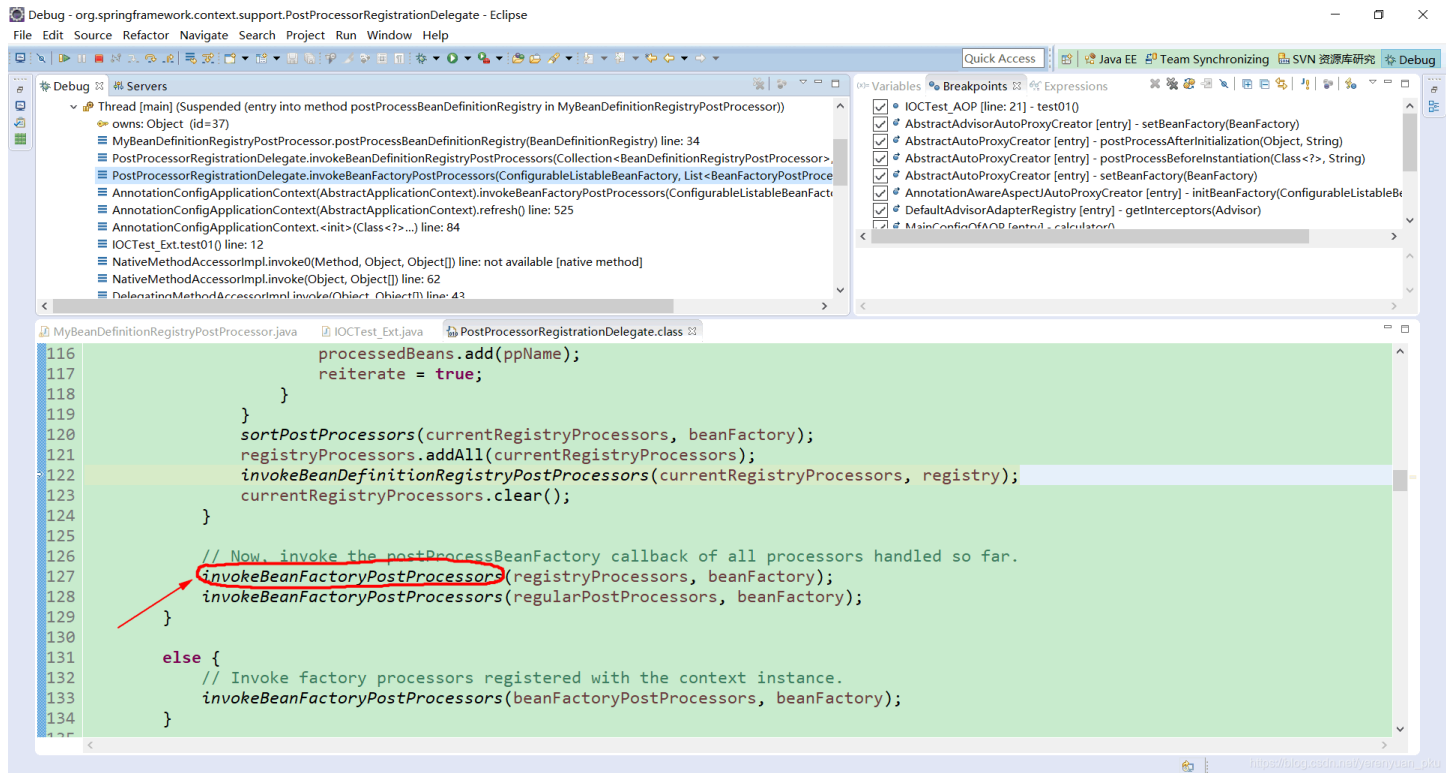
261     }
262     Collections.sort(postProcessors, comparatorToUse);
263 }
264
265 /**
266  * Invoke the given BeanDefinitionRegistryPostProcessor beans.
267  */
268 private static void invokeBeanDefinitionRegistryPostProcessors(
269     Collection<? extends BeanDefinitionRegistryPostProcessor> postProcessors, BeanDefinitionRegistry registry) {
270
271     for (BeanDefinitionRegistryPostProcessor postProcessor : postProcessors) {
272         postProcessor.postProcessBeanDefinitionRegistry(registry);
273     }
274 }
275
276 /**
277  * Invoke the given BeanFactoryPostProcessor beans.
278  */
279 private static void invokeBeanFactoryPostProcessors(
280     Collection<? extends BeanFactoryPostProcessor> postProcessors, ConfigurableListableBeanFactory beanFactory) {
281

```

所以, BeanDefinitionRegistryPostProcessor组件里面的postProcessBeanDefinitionRegistry方法会最优先被调用。

大家一定要注意哟😏, 这儿是先来调用BeanDefinitionRegistryPostProcessor组件里面的postProcessBeanDefinitionRegistry方法的哟! 想必你已经猜到了, 接下来就应该调用BeanDefinitionRegistryPostProcessor组件里面的postProcessBeanFactory方法了。那这是从何处知道的呢?

回到PostProcessorRegistrationDelegate类的invokeBeanFactoryPostProcessors方法中，还是回到程序停留在如下图所示的地方，你会发现它下面紧挨着的地方有一个叫invokeBeanFactoryPostProcessors的方法。



点进去这个方法里面一看究竟，你就会恍然大悟了，原来是先调用完BeanDefinitionRegistryPostProcessor组件里面的postProcessBeanDefinitionRegistry方法，然后再来调用它里面的postProcessBeanFactory方法。你也应该要知道，该方法实际上是BeanFactoryPostProcessor接口里面定义的方法。



说了这么多，也只是针对BeanDefinitionRegistryPostProcessor组件来说的，我们知道了，它里面的方法，postProcessBeanDefinitionRegistry方法会先被调用，postProcessBeanFactory方法会后被调用。

这时，你不禁要问了，那些BeanFactoryPostProcessor组件又该是何时调用的呢？也就是说，这些BeanFactoryPostProcessor组件里面的postProcessBeanFactory方法又是什么时候被调用的呢？其实，我在上面已经说过了，BeanDefinitionRegistryPostProcessor组件是要优先于BeanFactoryPostProcessor组件执行的，也就是说，BeanFactoryPostProcessor组件（例如在上一讲中，我们自己写的MyBeanFactoryPostProcessor类）是之后才会被执行的，为什么我会这样说呢？

我们再来仔细看一下PostProcessorRegistrationDelegate类中的invokeBeanFactoryPostProcessors方法，只不过这时是从程序停留的地方（即第122行代码处）往下看，如下图所示。

```
MyBeanDefinitionRegistryPostProcessor.java  IOCTest_Ext.java  PostProcessorRegistrationDelegate.class
122     invokeBeanDefinitionRegistryPostProcessors(currentRegistryProcessors, registry);
123     currentRegistryProcessors.clear();
124 }
125
126 // Now, invoke the postProcessBeanFactory callback of all processors handled so far.
127 invokeBeanFactoryPostProcessors(registryProcessors, beanFactory);
128 invokeBeanFactoryPostProcessors(regularPostProcessors, beanFactory);
129 }
130
131 else {
132     // Invoke factory processors registered with the context instance.
133     invokeBeanFactoryPostProcessors(beanFactoryPostProcessors, beanFactory);
134 }
135
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(BeenFactoryPostProcessor.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)) {
```

看到这些代码，你是不是觉得很熟悉，这不就是咱们上一讲中已经看过的内容吗？不用我再讲一遍了吧😂，其实这不就是上一讲中我们讲解的BeanFactoryPostProcessor的整个执行顺序以及原理吗！

由此可见，这时会再来从容器中找到所有的BeanFactoryPostProcessor组件，然后依次触发其postProcessBeanFactory方法。

小结

分析了这么多，我们是不是可以来小结一下BeanDefinitionRegistryPostProcessor组件执行时的原理呢？

1. 创建IOC容器
2. 创建IOC容器时，要调用一个刷新方法，即refresh方法
3. 从IOC容器中获取到所有的BeanDefinitionRegistryPostProcessor组件，并依次触发它们的postProcessBeanDefinitionRegistry方法，然后再来触发它们的postProcessBeanFactory方法
4. 再来从IOC容器中获取到所有的BeanFactoryPostProcessor组件，并依次触发它们的postProcessBeanFactory方法