

## Spring注解驱动开发第40讲——你晓得@EventListener这个注解的原理吗？

### 写在前面

在上一讲中，我们讲了一下 **事件监听** 机制的内部原理，当然了，在一过程中，我们也看到了事件的整个发布流程。再回顾一下的话，你会发现之前咱们编写的监听器都是来实现ApplicationListener这个接口的，其实，除此之外，还有另外一种方式。因此，这一讲，我们就来着重讲述这种方式。

这里我先提一下这种方式，即使用@EventListener注解，我们就可以让任意方法都能 **监听事件** 。这样的话，我们在一个普通的业务逻辑组件中，就可以直接来使用这个注解了，而不是让它去实现ApplicationListener这个接口。

### @EventListener注解的用法

首先，编写一个普通的业务逻辑组件，例如UserService，并在该组件上标注一个@Service注解。

```
1 package com.meimeixia.ext;
2
3 import org.springframework.stereotype.Service;
4
5 @Service
6 public class UserService {
7
8 }
```

AI写代码java运行

在该组件内，我们肯定会写一些很多的方法，但这里就略去了。那么问题来了，如果我们希望该组件能监听到事件，那么该怎么办呢？我们可以在该组件内写一个listen方法，以便让该方法来监听事件。这时，我们只需要简单地给该方法上标注一个@EventListener注解，就可以让它来监听事件了。那么，到底要监听哪些事件呢？我们可以通过@EventListener注解中的classes属性来指定，例如，我们可以让listen方法监听ApplicationEvent及其下面的子事件。

```
1 package com.meimeixia.ext;
2
3 import org.springframework.context.ApplicationEvent;
4 import org.springframework.context.event.EventListener;
5 import org.springframework.stereotype.Service;
6
7 @Service
8 public class UserService {
9
10     // 一些其他的方法...
11
12     @EventListener(classes=ApplicationEvent.class)
13     public void listen() {
14         System.out.println("UserService...");
15     }
16
17 }
```

AI写代码java运行



当然了，我们还可以通过@EventListener注解中的classes属性来指定监听多个事件。

```
1 package com.meimeixia.ext;
2
3 import org.springframework.context.ApplicationEvent;
4 import org.springframework.context.event.EventListener;
5 import org.springframework.stereotype.Service;
6
7 @Service
8 public class UserService {
9
10     // 一些其他的方法...
11
12     // @EventListener(classes=ApplicationEvent.class)
13     @EventListener(classes={ApplicationEvent.class})
14     public void listen() {
15         System.out.println("UserService...");
16     }
17
18 }
```

AI写代码java运行



如果ApplicationEvent及其下面的子事件发生了，那么我们应该怎么办呢？想都不用想，肯定是拿到这个事件，因此我们就要在listen方法的参数位置上写一个ApplicationEvent参数来接收该事件。

```
1 package com.meimeixia.ext;
2
3 import org.springframework.context.ApplicationEvent;
4 import org.springframework.context.event.EventListener;
5 import org.springframework.stereotype.Service;
6
7 @Service
8 public class UserService {
9
10     // 一些其他的方法...
11
12     // @EventListener(classes=ApplicationEvent.class)
13     @EventListener(classes={ApplicationEvent.class})
14     public void listen(ApplicationEvent event) {
15         System.out.println("UserService...监听到的事件: " + event);
16     }
17 }
18 }
```

AI写代码java运行



以上就是我们自己编写的一个普通的业务逻辑组件，该组件就能监听事件，这跟实现ApplicationListener接口的效果是一模一样的。

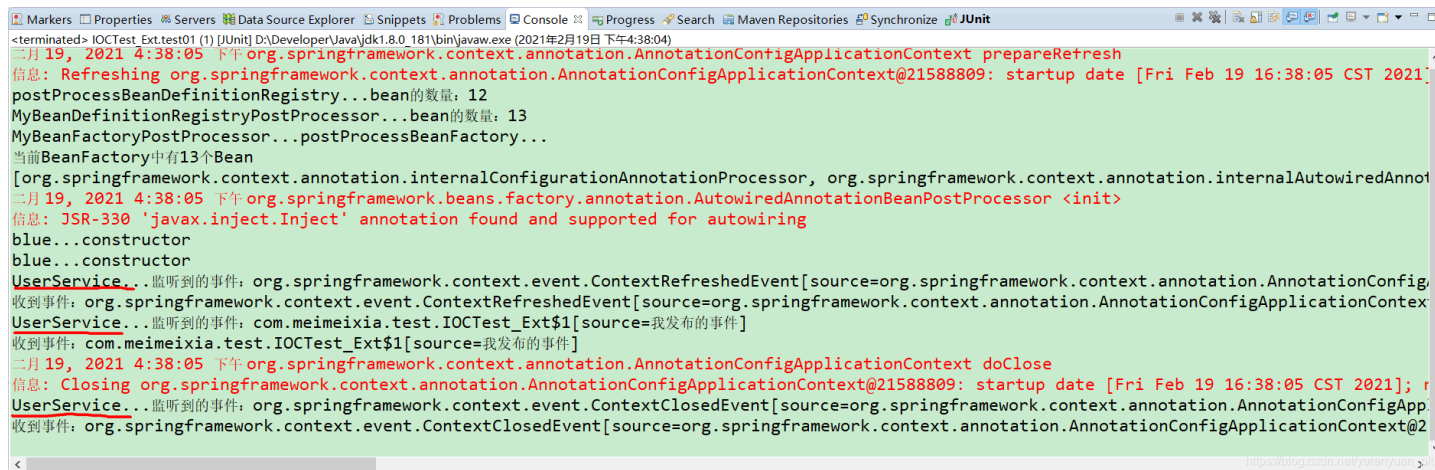
然后，我们就要来进行测试了，就是运行一下以下IOCTest\_Ext测试类中的test01方法。

```
1 package com.meimeixia.test;
2
3 import org.junit.Test;
4 import org.springframework.context.ApplicationEvent;
5 import org.springframework.context.annotation.AnnotationConfigApplicationContext;
6
7 import com.meimeixia.ext.ExtConfig;
8
9 public class IOCTest_Ext {
10
11     @Test
12     public void test01() {
13         AnnotationConfigApplicationContext applicationContext = new AnnotationConfigApplicationContext(ExtConfig.class);
14
15         // 发布一个事件
16         applicationContext.publishEvent(new ApplicationEvent(new String("我发布的事件"))) {
17             };
18
19         // 关闭容器
20         applicationContext.close();
21     }
22 }
23 }
```

AI写代码java运行



你会发现Eclipse 控制台打印出了如下内容，可以清晰地看到，不仅我们之前编写的 **监听器**（例如MyApplicationListener）收到了事件，而且UserService组件也收到了事件。也就是说，每一个都能正确地收到事件。



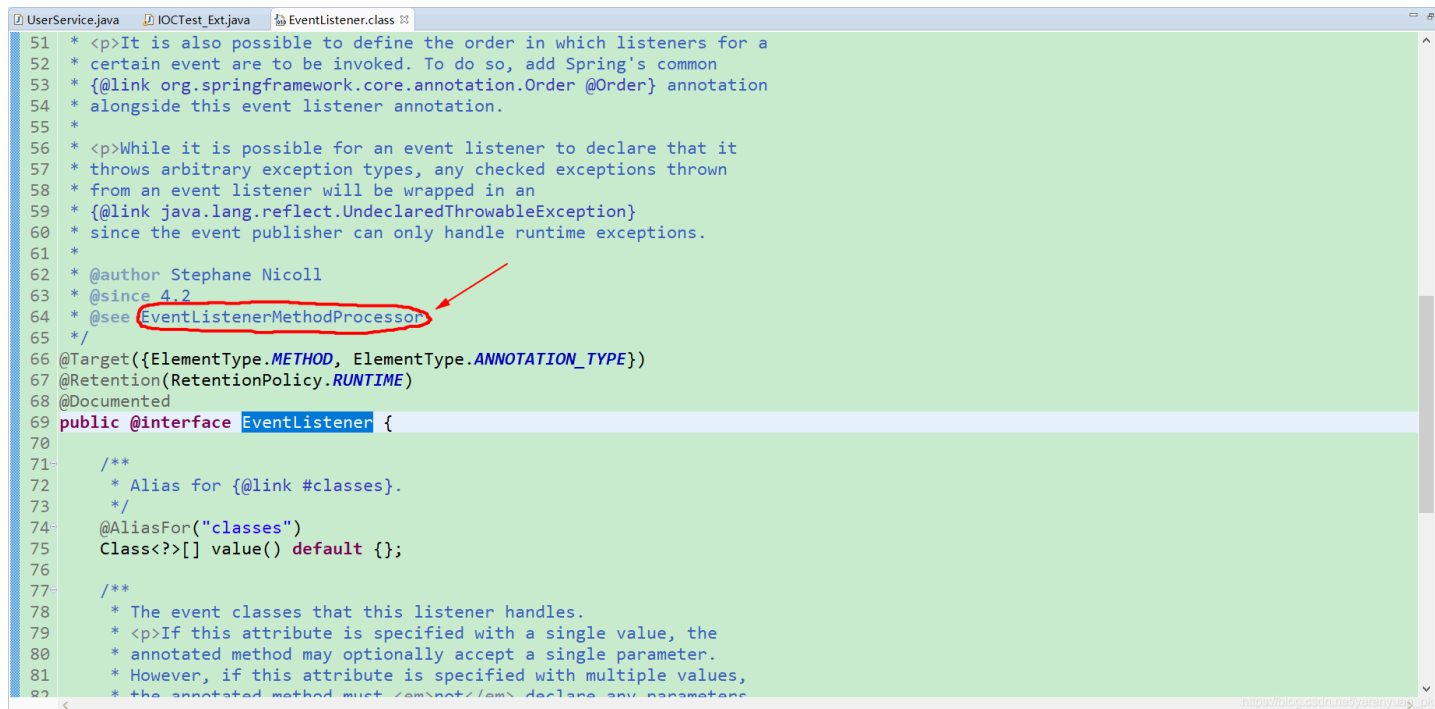
```
<terminated> IOCTest Ext.test01 (1) [JUnit] D:\Developer\Java\jdk1.8.0_181\bin\javaw.exe (2021年2月19日 下午4:38:04)
二月 19, 2021 4:38:05 下午 org.springframework.context.annotation.AnnotationConfigApplicationContext prepareRefresh
信息: Refreshing org.springframework.context.annotation.AnnotationConfigApplicationContext@21588809: startup date [Fri Feb 19 16:38:05 CST 2021]; root of context hierarchy
postProcessBeanDefinitionRegistry...bean的数量: 12
MyBeanDefinitionRegistryPostProcessor...bean的数量: 13
MyBeanFactoryPostProcessor...postProcessBeanFactory...
当前BeanFactory中有13个Bean
[org.springframework.context.annotation.internalConfigurationAnnotationProcessor, org.springframework.context.annotation.internalAutowiredAnnotationProcessor, org.springframework.context.annotation.internalRequiredAnnotationProcessor]
二月 19, 2021 4:38:05 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
UserService...监听到的事件: org.springframework.context.event.ContextRefreshedEvent[source=org.springframework.context.annotation.AnnotationConfigApplicationContext@21588809]
收到事件: org.springframework.context.event.ContextRefreshedEvent[source=org.springframework.context.annotation.AnnotationConfigApplicationContext@21588809]
UserService...监听到的事件: com.meimeixia.test.IOCTest_Ext$1[source=我发布的事件]
收到事件: com.meimeixia.test.IOCTest_Ext$1[source=我发布的事件]
二月 19, 2021 4:38:05 下午 org.springframework.context.annotation.AnnotationConfigApplicationContext doClose
信息: Closing org.springframework.context.annotation.AnnotationConfigApplicationContext@21588809: startup date [Fri Feb 19 16:38:05 CST 2021]; root of context hierarchy
UserService...监听到的事件: org.springframework.context.event.ContextClosedEvent[source=org.springframework.context.annotation.AnnotationConfigApplicationContext@21588809]
收到事件: org.springframework.context.event.ContextClosedEvent[source=org.springframework.context.annotation.AnnotationConfigApplicationContext@21588809]
```

这里我得说一嘴，以后咱们对@EventListener这个注解的使用会比较多，因为它使用起来非常方便。

接下来，我们就得说说这个注解背后的原理了。

## @EventListener注解的原理

我们可以点进去@EventListener这个注解里面去看一看，如下图所示，可以看到这个注解上面有一大堆的描述，从描述中我们是否可以猜到到这个注解的内部工作原理呢？答案是可以的。



```
51 * <p>It is also possible to define the order in which listeners for a
52 * certain event are to be invoked. To do so, add Spring's common
53 * {@link org.springframework.core.annotation.Order @Order} annotation
54 * alongside this event listener annotation.
55 *
56 * <p>While it is possible for an event listener to declare that it
57 * throws arbitrary exception types, any checked exceptions thrown
58 * from an event listener will be wrapped in an
59 * {@link java.lang.reflect.UndeclaredThrowableException}
60 * since the event publisher can only handle runtime exceptions.
61 *
62 * @author Stephane Nicoll
63 * @since 4.2
64 * @see EventListenerMethodProcessor
65 */
66 @Target({ElementType.METHOD, ElementType.ANNOTATION_TYPE})
67 @Retention(RetentionPolicy.RUNTIME)
68 @Documented
69 public @interface EventListener {
70
71     /**
72      * Alias for {@link #classes}.
73      */
74     @AliasFor("classes")
75     Class<?>[] value() default {};
76
77     /**
78      * The event classes that this listener handles.
79      * <p>If this attribute is specified with a single value, the
80      * annotated method may optionally accept a single parameter.
81      * However, if this attribute is specified with multiple values,
82      * the annotated method must (explicitly or implicitly) declare any parameters
```

描述中有一个醒目的字眼，即参考EventListenerMethodProcessor。意思可能是说，如果你想搞清楚@EventListener注解的内部工作原理，那么可以参考EventListenerMethodProcessor这个类。

EventListenerMethodProcessor是啥呢？它就是一个处理器，其作用是来解析方法上的@EventListener注解的。这也就是说，Spring会使用EventListenerMethodProcessor这个处理器来解析方法上的@EventListener注解。因此，接下来，我们就要将关注点放在这个处理器上，搞清楚这个处理器是怎样工作的。搞清楚了这个问题，自然地我们就搞清楚了@EventListener注解的内部工作原理。

我们点进去EventListenerMethodProcessor这个类里面去看一看，如下图所示，发现它实现了一个接口，叫SmartInitializingSingleton。这时，要想搞清楚EventListenerMethodProcessor这个处理器是怎样工作的，那就得先搞清楚SmartInitializingSingleton这个接口的原理了。

```

UserService.java  IOCTest_Ext.java  EventListener.class  EventListenerMethodProcessor.class
47: /**
48:  * Register {@link EventListener} annotated method as individual {@link ApplicationListener}
49:  * instances.
50:  *
51:  * @author Stephane Nicoll
52:  * @author Juergen Hoeller
53:  * @since 4.2
54:  */
55: public class EventListenerMethodProcessor implements SmartInitializingSingleton, ApplicationContextAware {
56:
57:     protected final Log logger = LoggerFactory.getLog(getClass());
58:
59:     private ConfigurableApplicationContext applicationContext;
60:
61:     private final EventExpressionEvaluator evaluator = new EventExpressionEvaluator();
62:
63:     private final Set<Class<?>> nonAnnotatedClasses =
64:         Collections.newSetFromMap(new ConcurrentHashMap<Class<?>, Boolean>(64));
65:
66:
67:     @Override
68:     public void setApplicationContext(ApplicationContext applicationContext) throws BeansException {
69:         Assert.isTrue(applicationContext instanceof ConfigurableApplicationContext,
70:             "ApplicationContext does not implement ConfigurableApplicationContext");
71:         this.applicationContext = (ConfigurableApplicationContext) applicationContext;
72:     }
73:
74:     @Override
75:     public void afterSingletonsInstantiated() {
76:         List<EventListenerFactory> factories = getEventListenerFactories();
77:         String[] beanNames = this.applicationContext.getBeanNamesForType(Object.class);
78:         for (String beanName : beanNames) {

```

不妨点进去SmartInitializingSingleton这个接口里面去看一看，你会发现它里面定义了一个叫afterSingletonsInstantiated的方法，如下图所示。

```

* Copyright 2002-2014 the original author or authors.
package org.springframework.beans.factory;

/**
 * Callback interface triggered at the end of the singleton pre-instantiation phase
 * during {@link BeanFactory} bootstrap. This interface can be implemented by
 * singleton beans in order to perform some initialization after the regular
 * singleton instantiation algorithm, avoiding side effects with accidental early
 * initialization (e.g. from {@link ListableBeanFactory#getBeansOfType} calls).
 * In that sense, it is an alternative to {@link InitializingBean} which gets
 * triggered right at the end of a bean's local construction phase.
 *
 * <p>This callback variant is somewhat similar to
 * {@link org.springframework.context.event.ContextRefreshedEvent} but doesn't
 * require an implementation of {@link org.springframework.context.ApplicationListener},
 * with no need to filter context references across a context hierarchy etc.
 * It also implies a more minimal dependency on just the {@code beans} package
 * and is being honored by standalone {@link ListableBeanFactory} implementations,
 * not just in an {@link org.springframework.context.ApplicationContext} environment.
 *
 * <p><b>NOTE:</b> If you intend to start/manage asynchronous tasks, preferably
 * implement {@link org.springframework.context.Lifecycle} instead which offers
 * a richer model for runtime management and allows for phased startup/shutdown.
 *
 * @author Juergen Hoeller
 * @since 4.1
 * @see org.springframework.beans.factory.config.ConfigurableListableBeanFactory#preInstantiateSingletons()
 */
public interface SmartInitializingSingleton {

    /**
     * Invoked right at the end of the singleton pre-instantiation phase,
     * with a guarantee that all regular singleton beans have been created
     * already. {@link ListableBeanFactory#getBeansOfType} calls within
     * this method won't trigger accidental side effects during bootstrap.
     * <p><b>NOTE:</b> This callback won't be triggered for singleton beans
     * lazily initialized on demand after {@link BeanFactory} bootstrap,
     * and not for any other bean scope either. Carefully use it for beans
     * with the intended bootstrap semantics only.
     */
    void afterSingletonsInstantiated();
}

```

接下来，我们就要搞清楚到底是什么时候开始触发执行afterSingletonsInstantiated方法的。

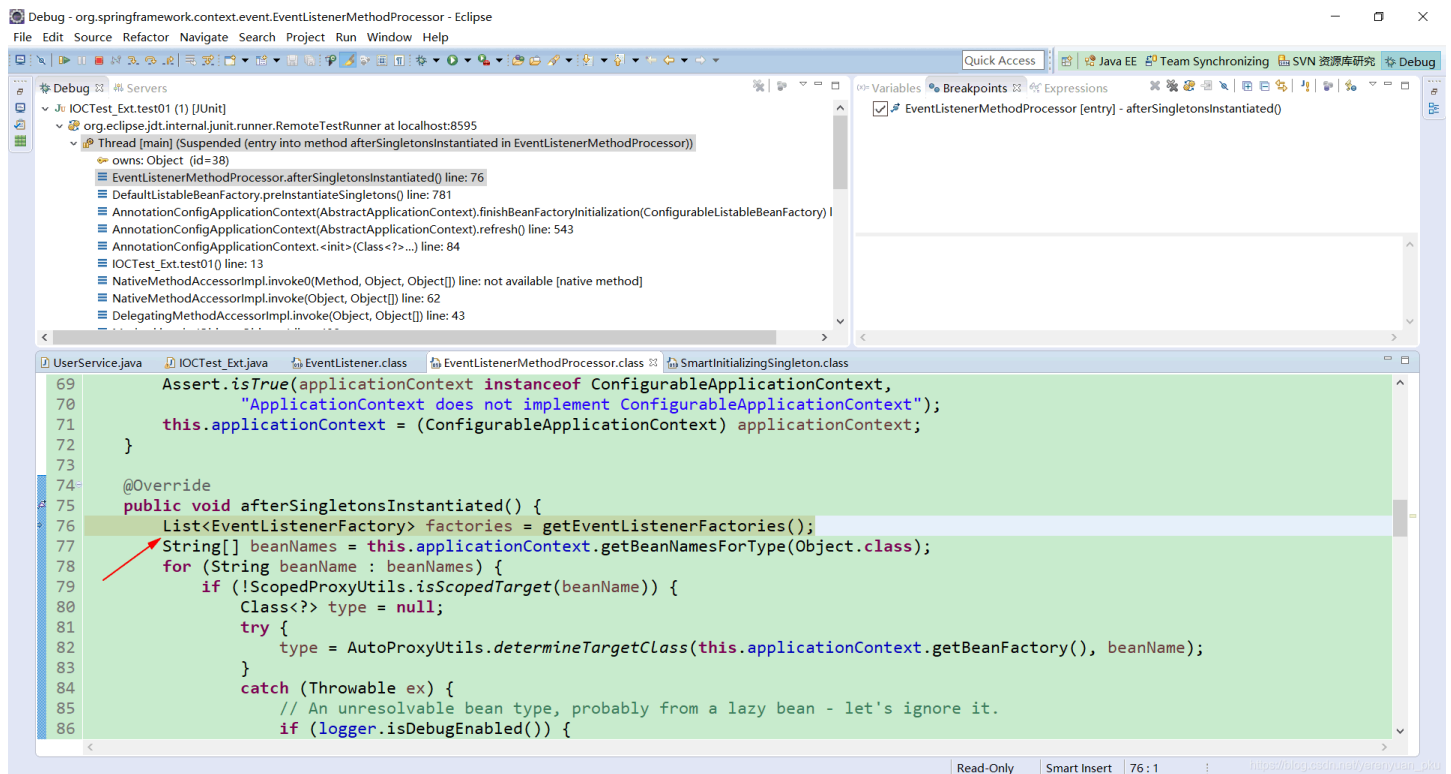
仔细看一下SmartInitializingSingleton接口中afterSingletonsInstantiated方法上面的描述信息，不难看出该方法是在所有的单实例bean已经全部被创建完了以后才会被执行。

其实，在介绍SmartInitializingSingleton接口的时候，我们也能从描述信息中知道，在所有的单实例bean已经全部被创建完成以后才会触发该接口。紧接着下面一段的描述还说了，该接口的调用时机有点类似于ContextRefreshedEvent事件，即在容器刷新完成以后，便会回调该接口。也就是说，这个时候容器已经创建完了。

好吧，回到主题，我们来看看afterSingletonsInstantiated方法的触发时机。首先，我们得在EventListenerMethodProcessor类里面的afterSingletonsInstantiated方法处打上一个断点，如下图所示。

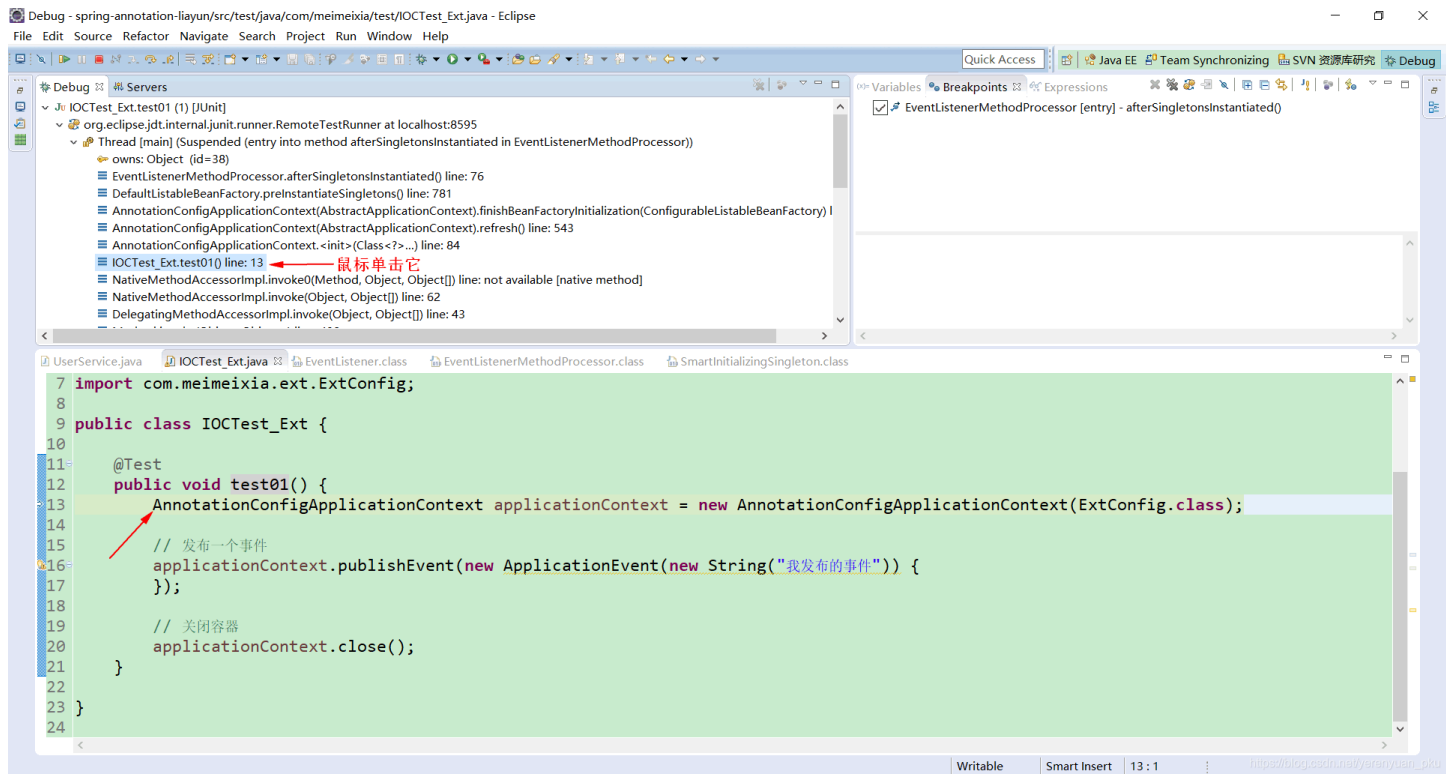


然后，以debug的方式运行IOCTest\_Ext测试类中的test01方法，这时程序停留在了EventListenerMethodProcessor类里面的afterSingletonsInstantiated方法中，如下图所示。

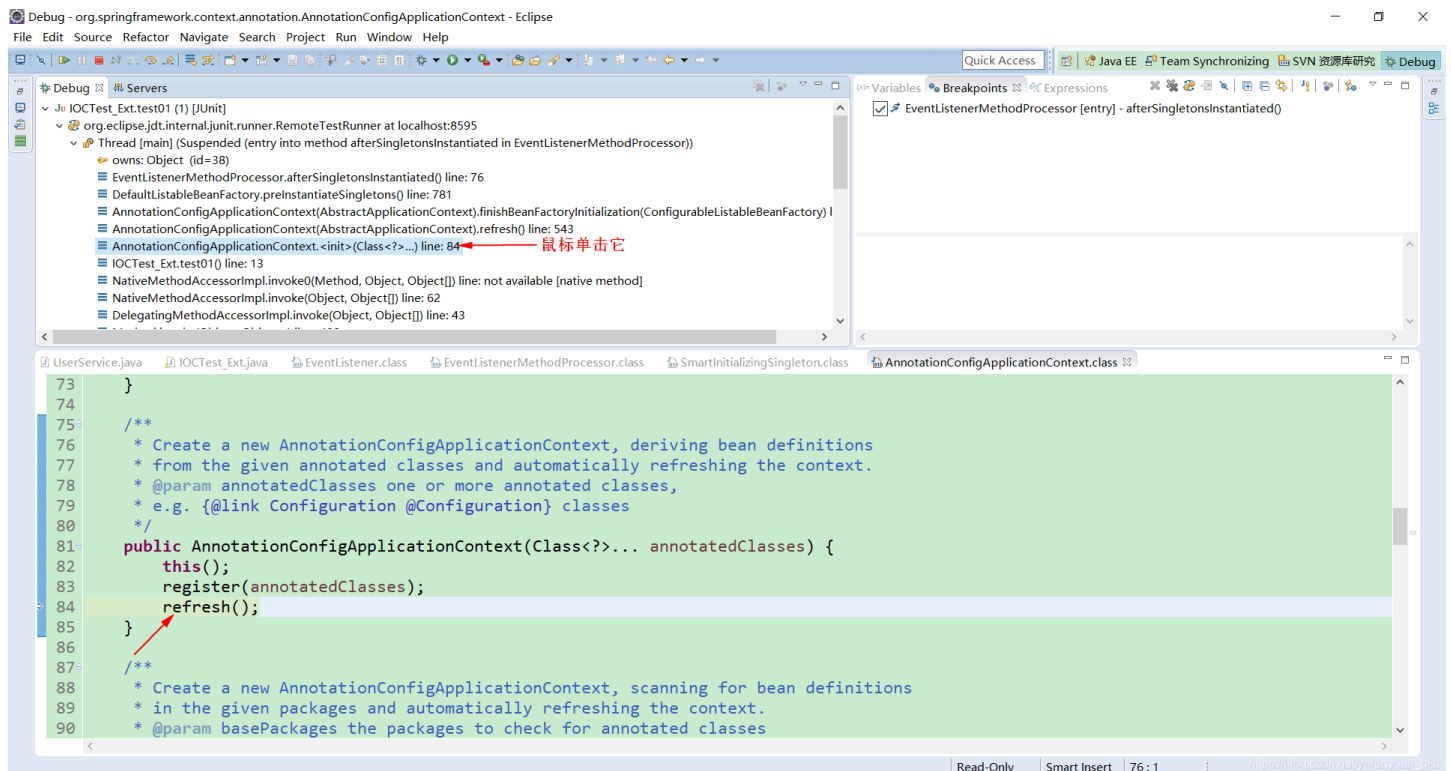


此时，你是不是很想知道是什么时候开始触发执行afterSingletonsInstantiated这个方法的呢？我们不妨从IOCTest\_Ext测试类中的test01方法开始，来梳理一遍整个流程。

鼠标单击Eclipse左上角方法调用栈中的 IOCTest\_Ext.test01() line:13，这时程序来到了IOCTest\_Ext测试类的test01方法中，如下图所示。

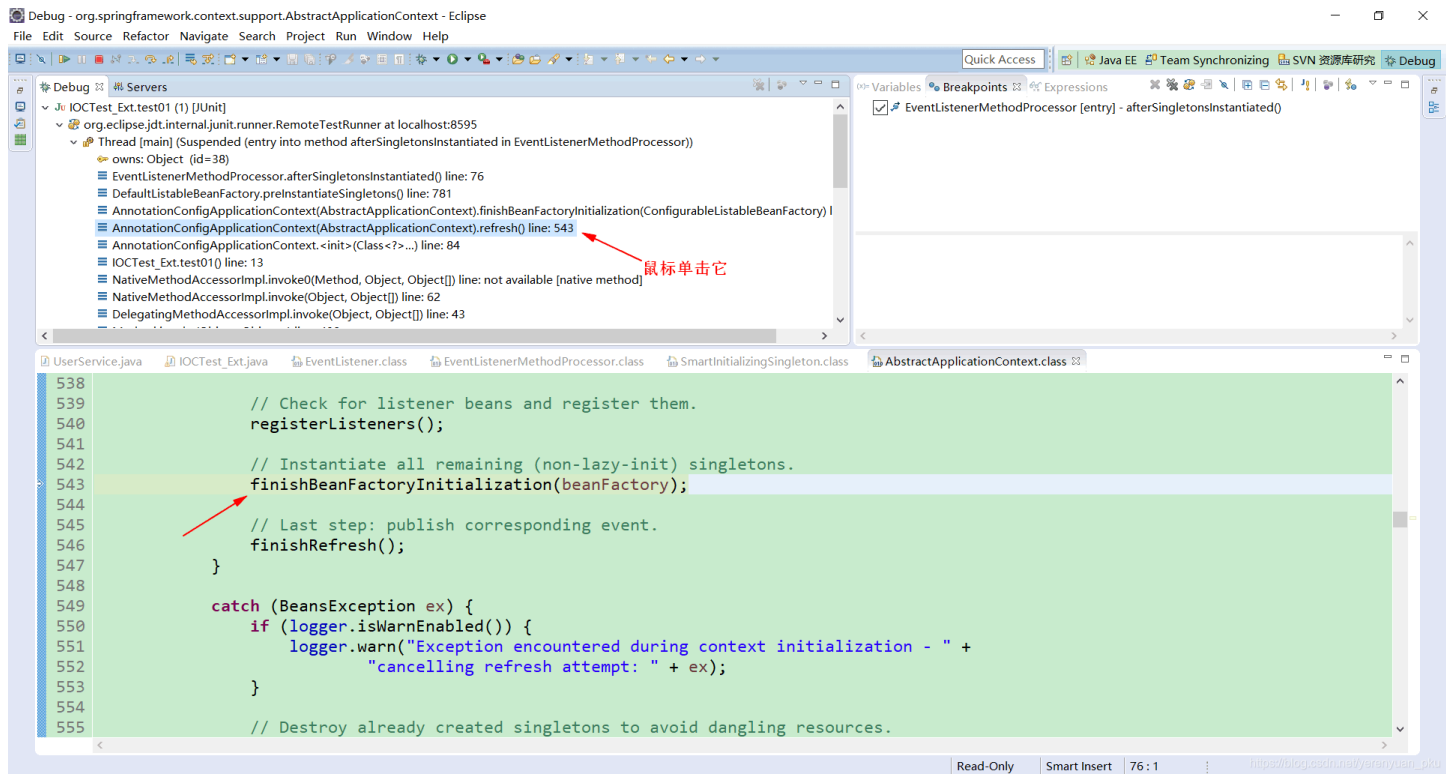


可以看到第一步是要来创建IOC容器的。继续跟进代码，可以看到在创建容器的过程中，还会调用一个refresh方法来刷新容器，刷新容器其实就是创建容器里面的所有bean。



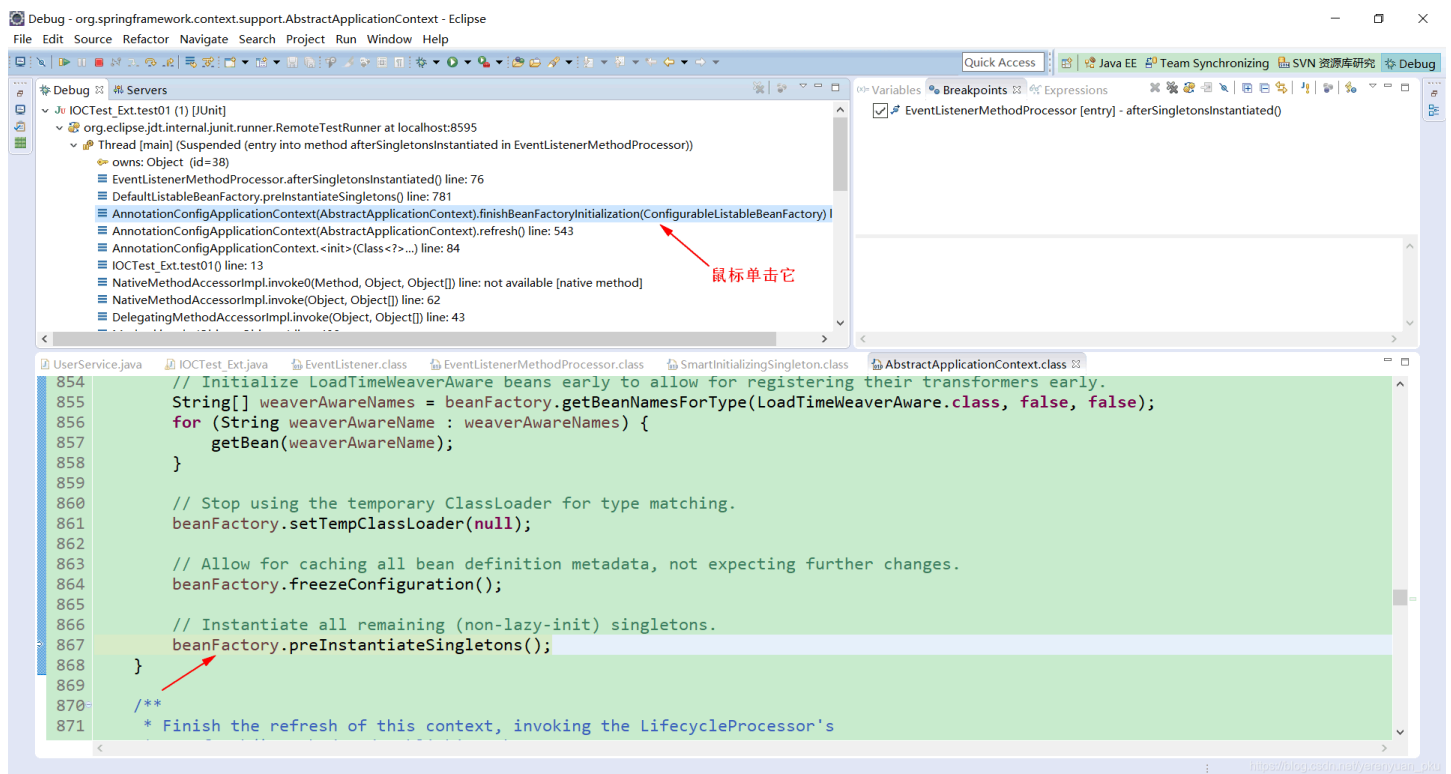
继续跟进代码，看这个refresh方法里面具体都做了啥，如下图所示，可以看到它里面调用了如下一个finishBeanFactoryInitialization方法，顾名思义，该方法就是来完成BeanFactory的初始化工作的。



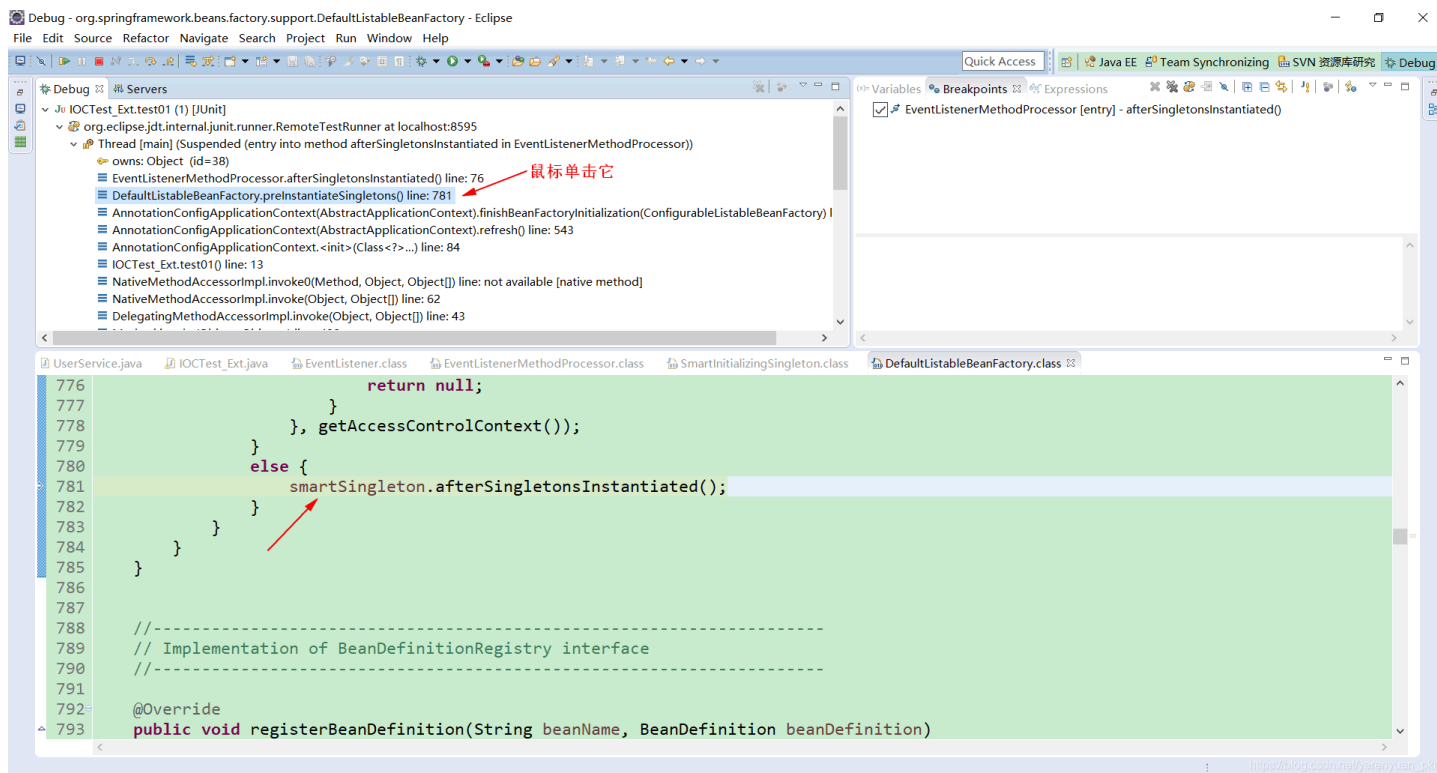


对于以上这个方法，我相信大家都不会陌生，因为我们之前就看过好多遍了，它其实就是来初始化所有剩下的那些单实例bean的。也就是说，如果还有一些单实例bean还没被初始化，即还没创建对象，那么便会在这一步进行（初始化）。

继续跟进代码，如下图所示，可以看到在finishBeanFactoryInitialization方法里面执行了如下一行代码，依旧还是来初始化所有剩下的单实例bean。



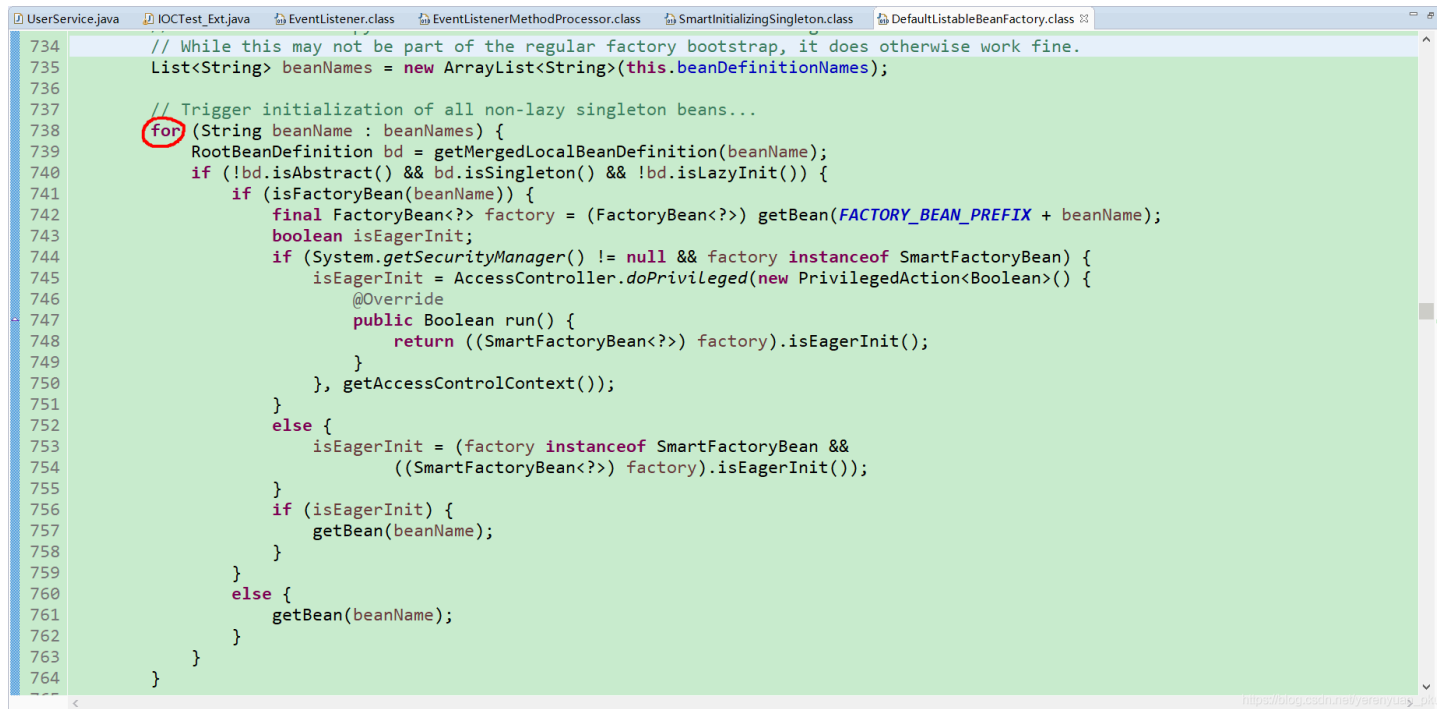
继续跟进代码，如下图所示，可以看到现在程序停留在了如下这行代码处。



这不就是我们要讲的afterSingletonsInstantiated方法吗？它原来是在这儿调用的啊！接下来，咱们就得好好看看在调用该方法之前，具体都做了哪些事。

由于afterSingletonsInstantiated方法位于DefaultListableBeanFactory类的preInstantiateSingletons方法里面，所以我们就得来仔细看看preInstantiateSingletons方法里面具体都做了些啥了。

进入眼帘的首先是一个for循环，在该for循环里面，beanNames里面存储的都是即将要创建的所有bean的名字，紧接着会做一个判断，即判断bean是不是抽象的，是不是单实例的，等等等等。最后，不管怎样，都会调用getBean方法来创建对象。



总结一下就是，先利用一个for循环拿到所有我们要创建的单实例bean，然后挨个调用getBean方法来创建对象。也即，创建所有的单实例bean。

再来往下翻阅preInstantiateSingletons方法，发现它下面还有一个for循环，在该for循环里面，beanNames里面依旧存储的是即将要创建的所有bean的名字。那么，在该for循环中所做的事情又是什么呢？很显然，在最上面的那个for循环中，所有的单实例bean都已经全部创建完了。因此，在下面这个for循环中，咱们所要做的事就是获取所有创建好的单实例bean，然后判断每一个bean对象是否是SmartInitializingSingleton这个接口类型的，如果是，那么便调用它里面的afterSingletonsInstantiated方法，而该方法就是SmartInitializingSingleton接口中定义的方法。



```
UserService.java  IOCTest_Ext.java  EventListener.class  EventListenerMethodProcessor.class  SmartInitializingSingleton.class  DefaultListableBeanFactory.class
760
761     else {
762         getBean(beanName);
763     }
764 }
765
766 // Trigger post-initialization callback for all applicable beans...
767 for (String beanName : beanNames) {
768     Object singletonInstance = getSingleton(beanName);
769     if (singletonInstance instanceof SmartInitializingSingleton) {
770         final SmartInitializingSingleton smartSingleton = (SmartInitializingSingleton) singletonInstance;
771         if (System.getSecurityManager() != null) {
772             AccessController.doPrivileged(new PrivilegedAction<Object>() {
773                 @Override
774                 public Object run() {
775                     smartSingleton.afterSingletonsInstantiated();
776                     return null;
777                 }
778             }, getAccessControlContext());
779         }
780     }
781     smartSingleton.afterSingletonsInstantiated();
782 }
783 }
784 }
785 }
786
787
788 //-----
789 // Implementation of BeanDefinitionRegistry interface
790 //-----
791
```

至此，你该搞清楚afterSingletonsInstantiated方法是什么时候开始触发执行了吧😁！就是在所有单实例bean全部创建完成以后。

最后，我还得说一嘴。如果所有的单实例bean都已经创建完了，也就是说下面这一步都执行完了，那么说明IOC容器已经创建完成了。

Debug - org.springframework.context.support.AbstractApplicationContext - Eclipse

File Edit Source Refactor Navigate Search Project Run Window Help

Quick Access Java EE Team Synchronizing SVN 资源库研究 Debug

Debug Servers

- IOCTest\_Ext.test01 (1) [JUnit]
- org.eclipse.jdt.internal.junit.runner.RemoteTestRunner at localhost:8595
  - Thread [main] (Suspended (entry into method afterSingletonsInstantiated in EventListenerMethodProcessor))
    - owns: Object (id=38)
      - EventListenerMethodProcessor.afterSingletonsInstantiated() line: 76
      - DefaultListableBeanFactory.preInstantiateSingletons() line: 781
      - AnnotationConfigApplicationContext(AbstractApplicationContext).finishBeanFactoryInitialization(ConfigurableListableBeanFactory) line: 543
      - AnnotationConfigApplicationContext(AbstractApplicationContext).refresh() line: 543
      - AnnotationConfigApplicationContext.<init>(Class<?>...) line: 84
      - IOCTest\_Ext.test01() line: 13
      - NativeMethodAccessorImpl.invoke0(Method, Object, Object[]) line: not available [native method]
      - NativeMethodAccessorImpl.invoke(Object, Object[]) line: 62
      - DelegatingMethodAccessorImpl.invoke(Object, Object[]) line: 43

Variables Breakpoints Expressions

✓ EventListenerMethodProcessor (entry) - afterSingletonsInstantiated()

```
UserService.java  IOCTest_Ext.java  EventListener.class  EventListenerMethodProcessor.class  SmartInitializingSingleton.class  AbstractApplicationContext.class
534
535     initApplicationEventMulticaster();
536
537     // Initialize other special beans in specific context subclasses.
538     onRefresh();
539
540     // Check for listener beans and register them.
541     registerListeners();
542
543     // Instantiate all remaining (non-lazy-init) singletons.
544     finishBeanFactoryInitialization(beanFactory);
545
546     // Last step: publish corresponding event.
547     finishRefresh();
548 }
549
550 catch (BeansException ex) {
551     if (logger.isWarnEnabled()) {
552         logger.warn("Exception encountered during context initialization - " +
553
```

那么，紧接着便会来调用finishRefresh方法，容器已经创建完了，此时就会来发布容器已经刷新完成的事件。这就呼应了开头的那句话，即SmartInitializingSingleton接口的调用时机有点类似于ContextRefreshedEvent事件，即在容器刷新完成以后，便会回调该接口。

以上就是@EventListener注解的内部工作原理，在讲解该原理时，我们顺道说了一下SmartInitializingSingleton接口的原理。