

## Spring注解驱动开发第48讲——Spring IOC容器创建源码解析(八)之完成BeanFactory的初始化创建工作，最终完成容器创建

快看我

写在前面

完成BeanFactory的初始化创建工作

初始化和生命周期有关的后置处理器

获取BeanFactory

看容器中是否有id为lifecycleProcessor，类型是LifecycleProcessor的组件

若有，则赋值给`this.lifecycleProcessor`

若没有，则创建一个DefaultLifecycleProcessor类型的组件，并把创建好的组件注册在容器中

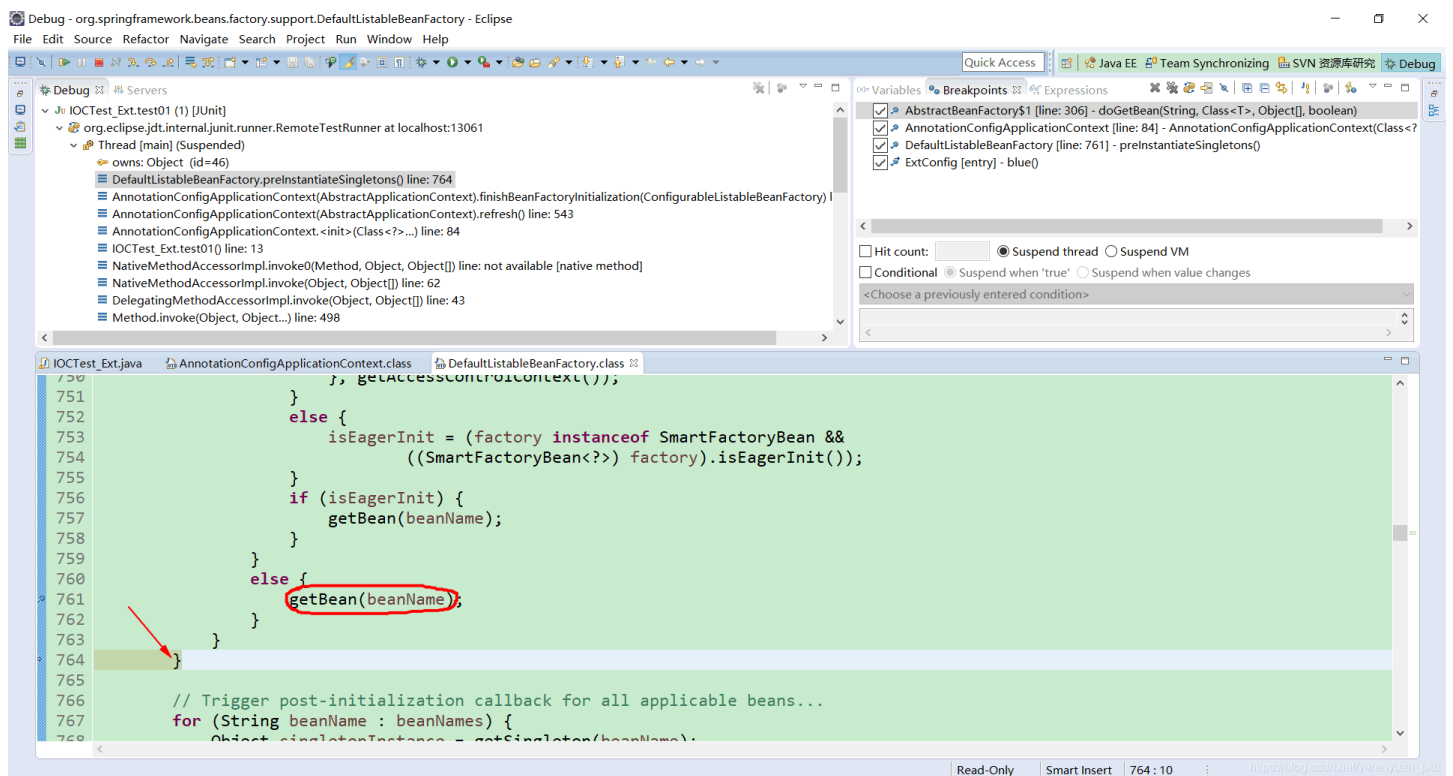
回调生命周期处理器的onRefresh方法

发布容器刷新完成事件

### 写在前面

经过前面两讲的学习，我们就把单实例bean的创建流程完完整整地过了一遍，其所牵扯到的步骤还是非常繁琐的，大家一定要耐心跟踪一下Spring的源码，亲自走一遍这个单实例bean的创建流程，相信你理解Spring的内部原理是极其有帮助的。

在上一讲末尾，我们让程序停留在了下面这行代码处，此时，getBean方法就完全是执行完了，这样，我们单实例bean就被创建出来了。



而且，创建我们单实例bean的流程牵扯到了很多很多东西，你得亲自跟踪一下Spring的源码才能有所体会，不然说再多也是白费口舌，当然了，你也可以在我上一讲中找到答案。

我们的bean创建出来之后，继续让程序往下运行，可以看到接下来就是通过以下for循环来将所有的bean都创建完。

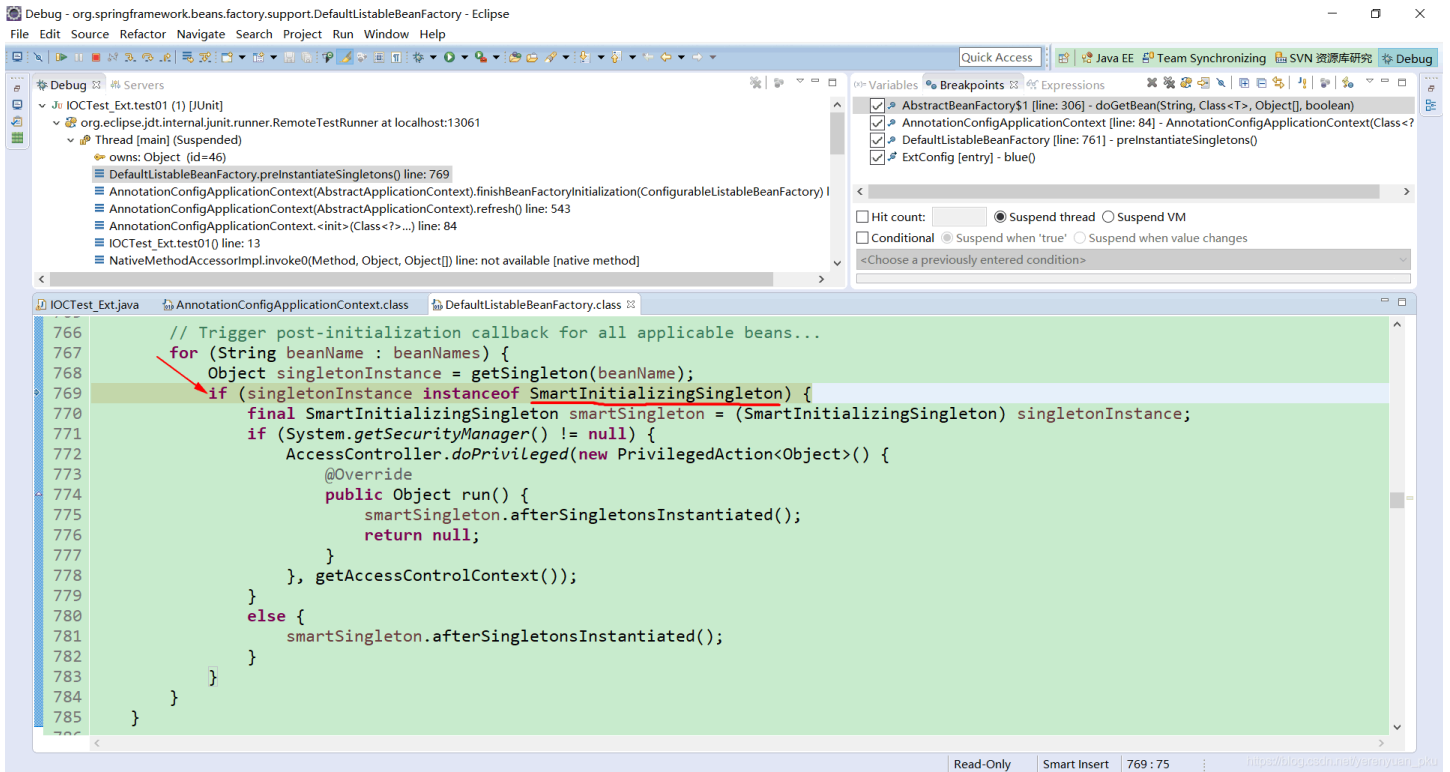
```
734 // While this may not be part of the regular factory bootstrap, it does otherwise work fine.
735 List<String> beanNames = new ArrayList<String>(this.beanDefinitionNames);
736
737 // Trigger initialization of all non-lazy singleton beans...
738 for (String beanName : beanNames) {
739     RootBeanDefinition bd = getMergedLocalBeanDefinition(beanName);
740     if (!bd.isAbstract() && bd.isSingleton() && !bd.isLazyInit()) {
741         if (isFactoryBean(beanName)) {
742             final FactoryBean<?> factory = (FactoryBean<?>) getBean(FACTORY_BEAN_PREFIX + beanName);
743             boolean isEagerInit;
744             if (System.getSecurityManager() != null && factory instanceof SmartFactoryBean) {
745                 isEagerInit = AccessController.doPrivileged(new PrivilegedAction<Boolean>() {
746                     @Override
747                     public Boolean run() {
748                         return ((SmartFactoryBean<?>) factory).isEagerInit();
749                     }
750                 }, getAccessControlContext());
751             }
752             else {
753                 isEagerInit = (factory instanceof SmartFactoryBean &&
754                     ((SmartFactoryBean<?>) factory).isEagerInit());
755             }
756             if (isEagerInit) {
757                 getBean(beanName);
758             }
759         }
760         else {
761             getBean(beanName);
762         }
763     }
764 }
765 }
```

那就让它创建其他的bean呗！创建流程不用我再详述一遍吧，跟我们单实例bean（即Blue对象）的创建流程是一模一样的，我们不停地按下 **F6** 快捷键让程序不停地往下运行，快速地过一遍就行了，这一过程我也不再详细地记录了。

当程序运行到下面这行代码处时，上面的那个for循环就整个地执行完了，也就是说，所有的bean都创建完成了。

```
755     }
756     if (isEagerInit) {
757         getBean(beanName);
758     }
759 }
760 else {
761     getBean(beanName);
762 }
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         else {
781             smartSingleton.afterSingletonsInstantiated();
782         }
783     }
784 }
785 }
```

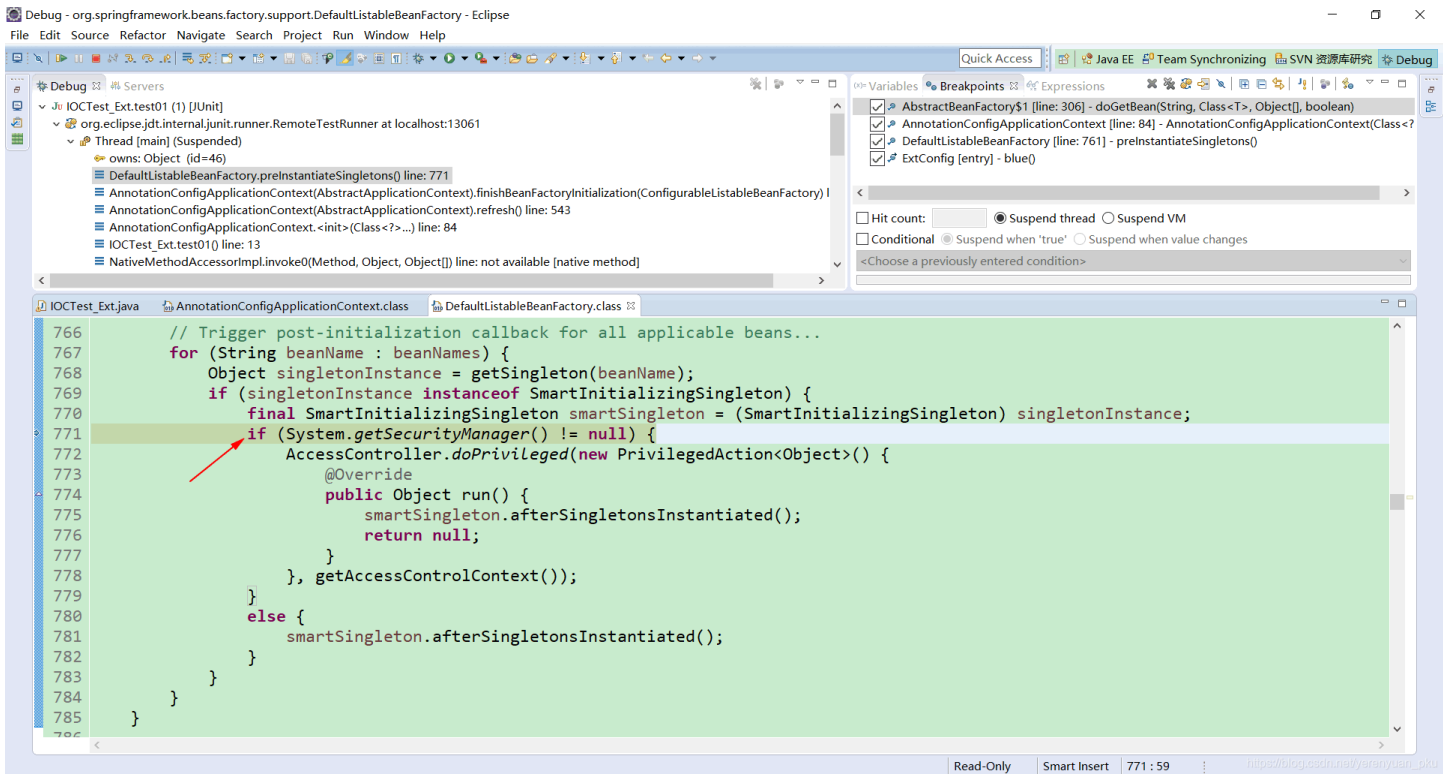
让程序继续往下运行，直至运行到下面这行代码处为止。



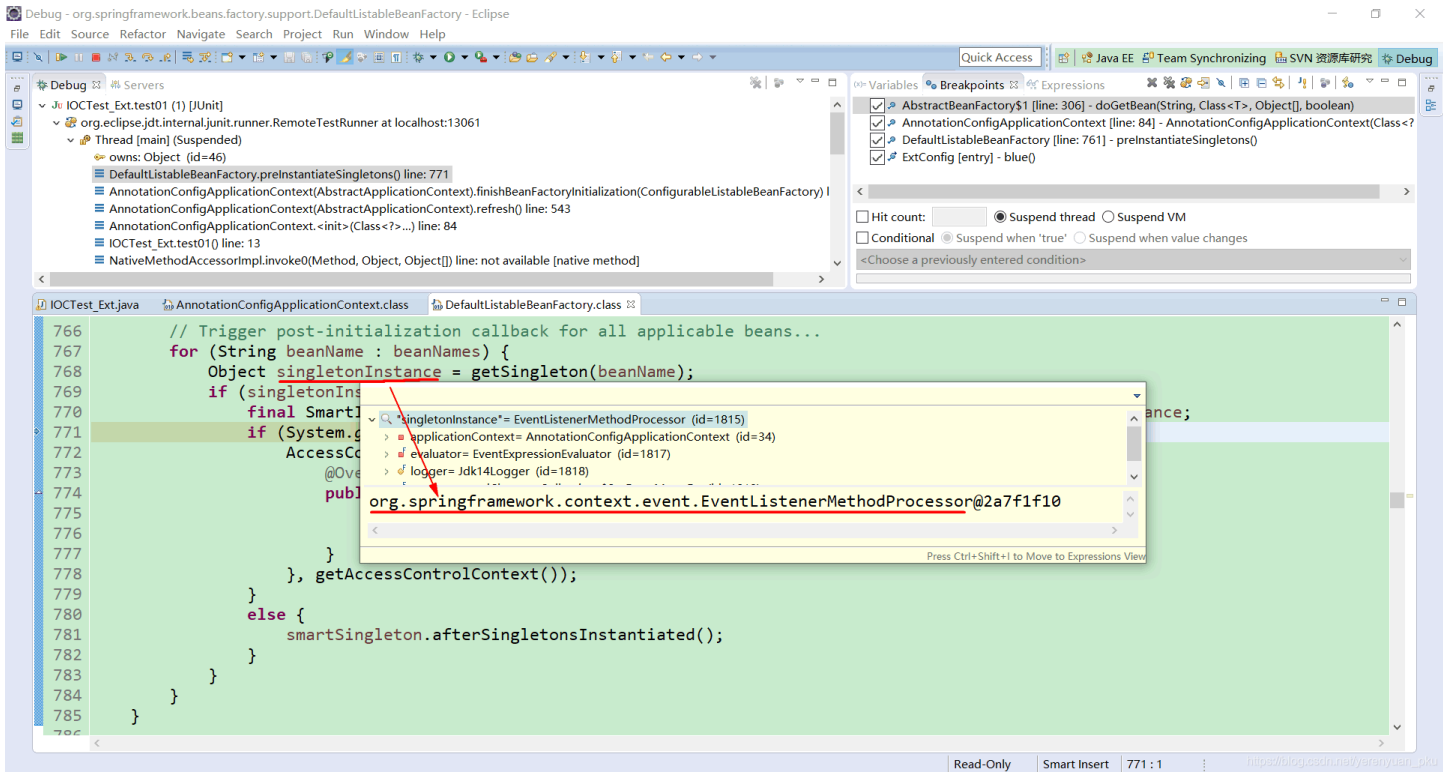
可以看到，这儿是来遍历所有的bean，并来判断遍历出来的每一个bean是否实现了SmartInitializingSingleton接口的。哎，你对SmartInitializingSingleton接口还有印象吗？在讲解@EventListener注解的内部原理时，我们就讲解过它，你还记得吗？要是你不记得了，那么可以回顾回顾《Spring注解驱动开发第40讲——你晓得@EventListener这个注解的原理吗？》这一讲。

OK，所有的bean都利用getBean方法创建完成以后，接下来要做的事情就是检查所有的bean中是否有实现SmartInitializingSingleton接口的，如果有的话，那么便会来执行该接口中的afterSingletonsInstantiated方法。

那我们不妨让程序继续往下运行，来验证上面这段话，当程序运行至下面这行代码处时，发现有一个bean实现了SmartInitializingSingleton接口，不然程序是不会进入到if判断语句中的。

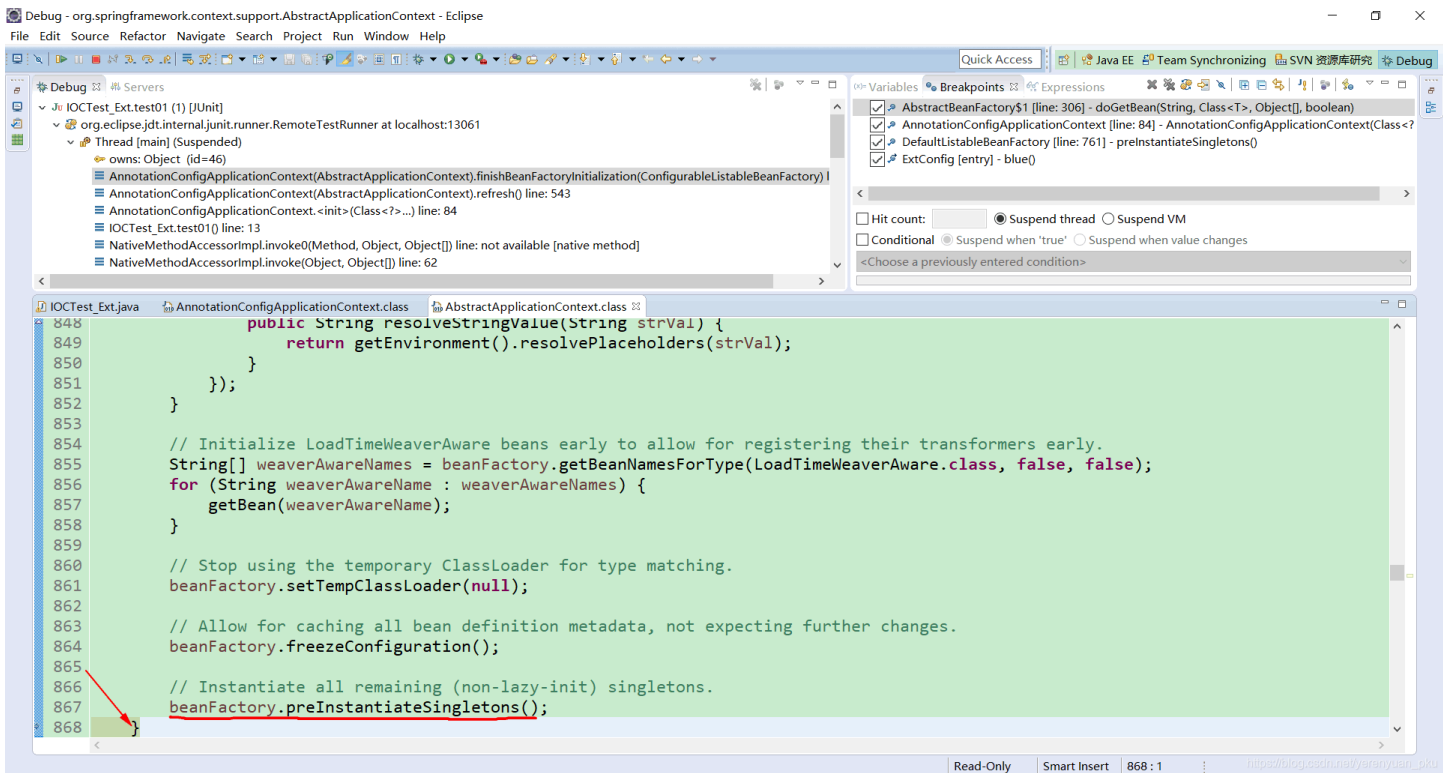


那么，到底是哪一个bean实现了SmartInitializingSingleton接口呢？我们不妨Inspect一下 singletonInstance 变量的值，这样很快就能知道该bean了，它就是EventListenerMethodProcessor，如下图所示。



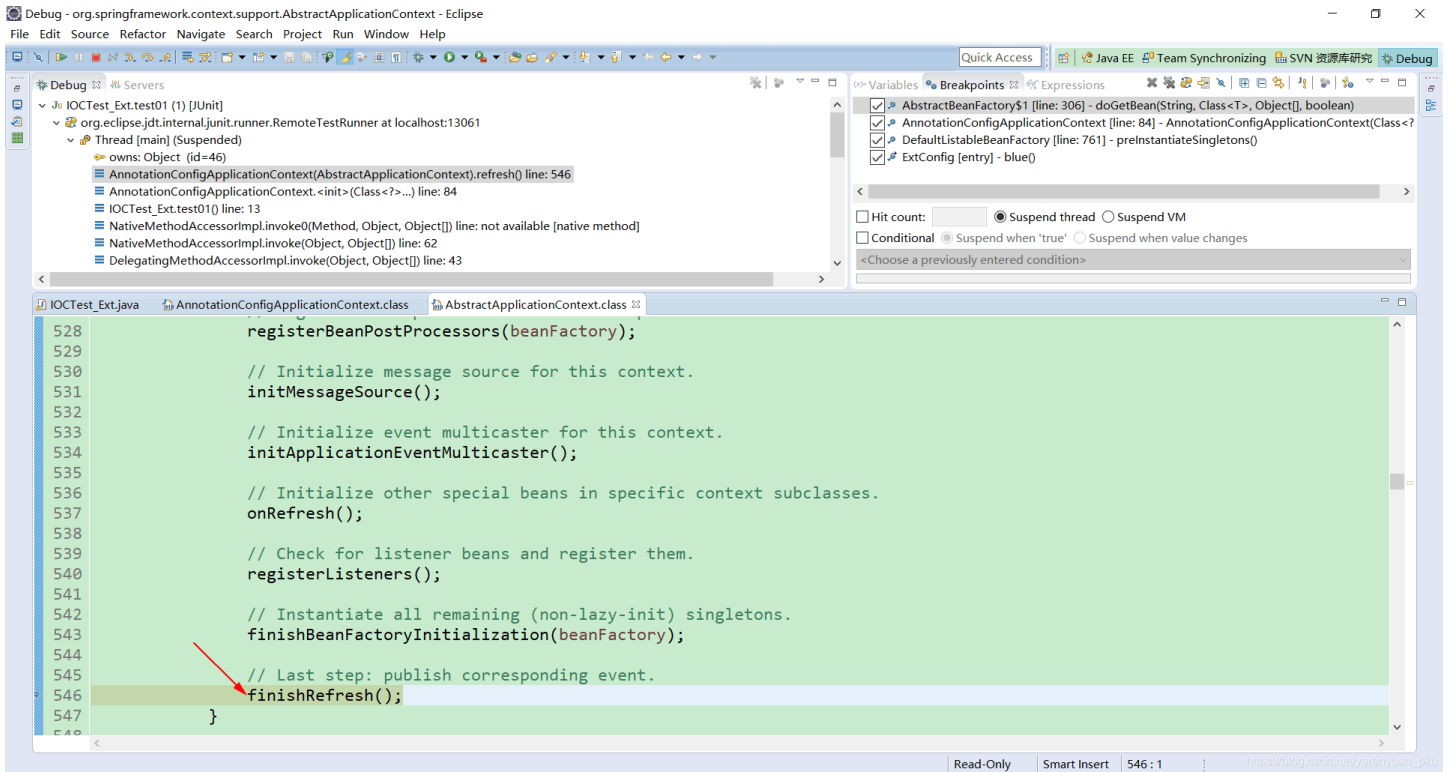
于是，接下来便会执行该bean中的`afterSingletonsInstantiated`方法，也就是`SmartInitializingSingleton`接口中定义的方法。

我们继续让程序往下运行，直至执行完整整个for循环，由于IOC容器中的bean还是蛮多的，所以要执行完整整个for循环，你不得不停地按下 **F6** 快捷键。当程序运行至下面这行代码处时，我们发现 `beanFactory.preInstantiateSingletons()` 这行代码总算是执行完了。



还记得这行代码是来干嘛的吗？它是来初始化所有剩下的单实例bean的。

接着，我们继续让程序往下运行，直至运行至下面这行代码处为止，此时，程序来到了Spring IOC容器创建的最后一步了，即完成BeanFactory的初始化创建工作。



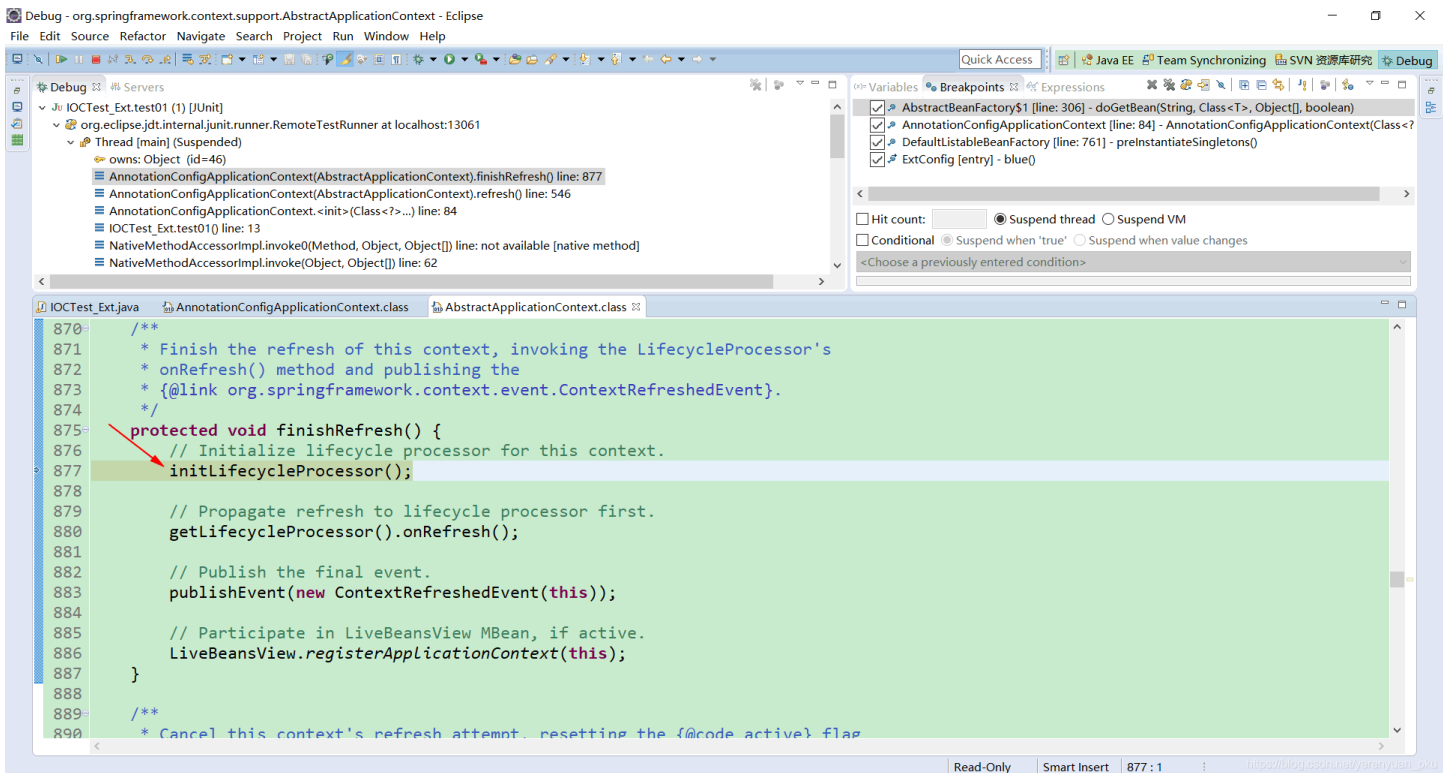
以上该方法一旦执行完，那么Spring IOC容器就创建完成了。接下来，我们就看看`finishRefresh`方法里面都做了些啥事。

## 完成BeanFactory的初始化创建工作

上面也说了，一旦`finishRefresh`方法执行完，就意味着完成了BeanFactory的初始化创建工作，顺带脚地，我们Spring IOC容器就创建完成了。

其实，IOC容器在前一步（即 `finishBeanFactoryInitialization(beanFactory)`）就已经创建完成了，而且所有的单实例bean也都已经加载完了。这个不用我再叙述一遍了吧😁，不懂的同学，请看前面两讲。

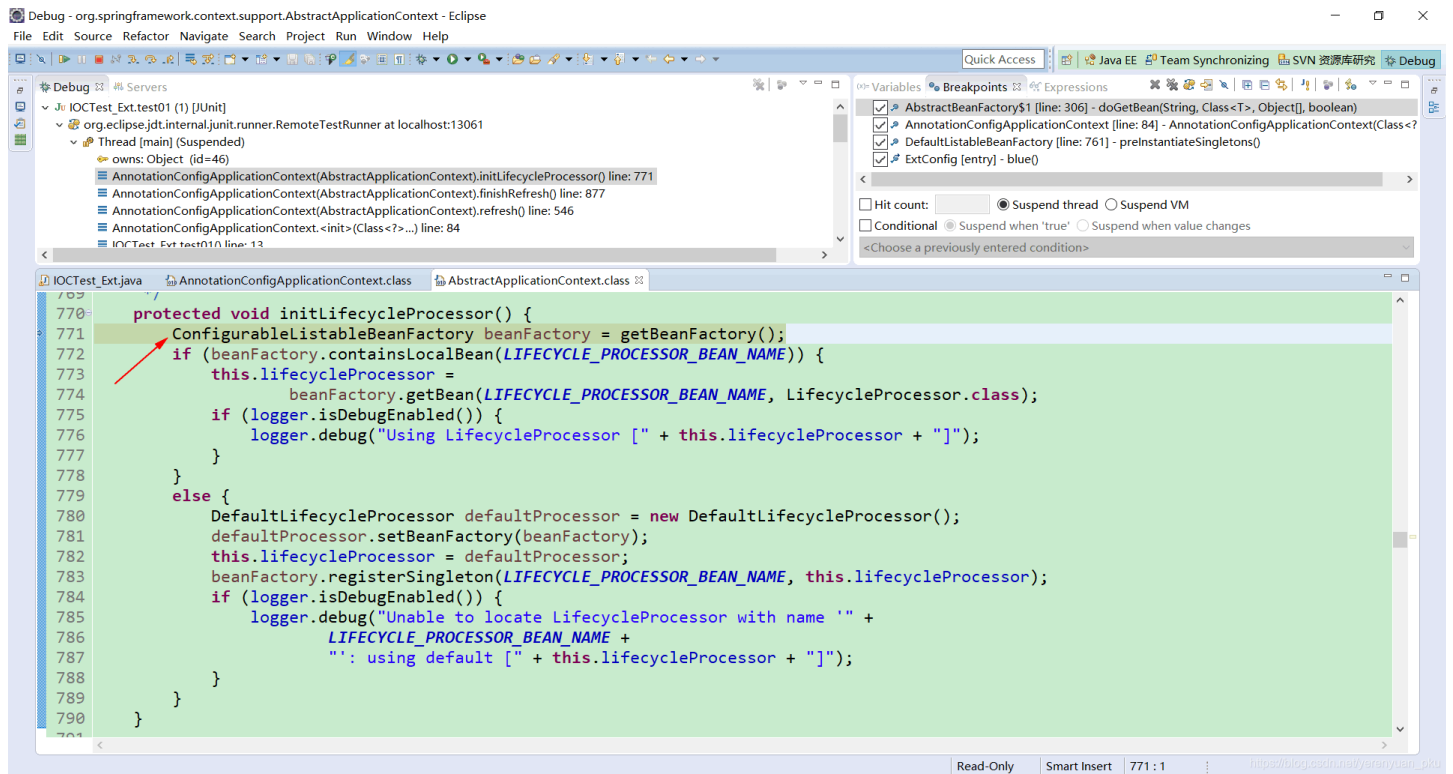
那么，`finishRefresh`方法里面究竟都做了些啥事呢？我们不妨按下 **F5** 快捷键进入该方法里面去看一下，如下图所示，是不是很熟悉啊！这里的逻辑，我们应该以前都瞟过一眼，只是过去一段时间了，我们似乎都快忘记了。



## 初始化和生命周期有关的后置处理器

可以看到`finishRefresh`方法里面首先会调用一个`initLifecycleProcessor`方法，该方法是来初始化和生命周期有关的后置处理器的。我们不妨按下 **F5** 快捷键进入该方法里面去看一下，如下图所示。



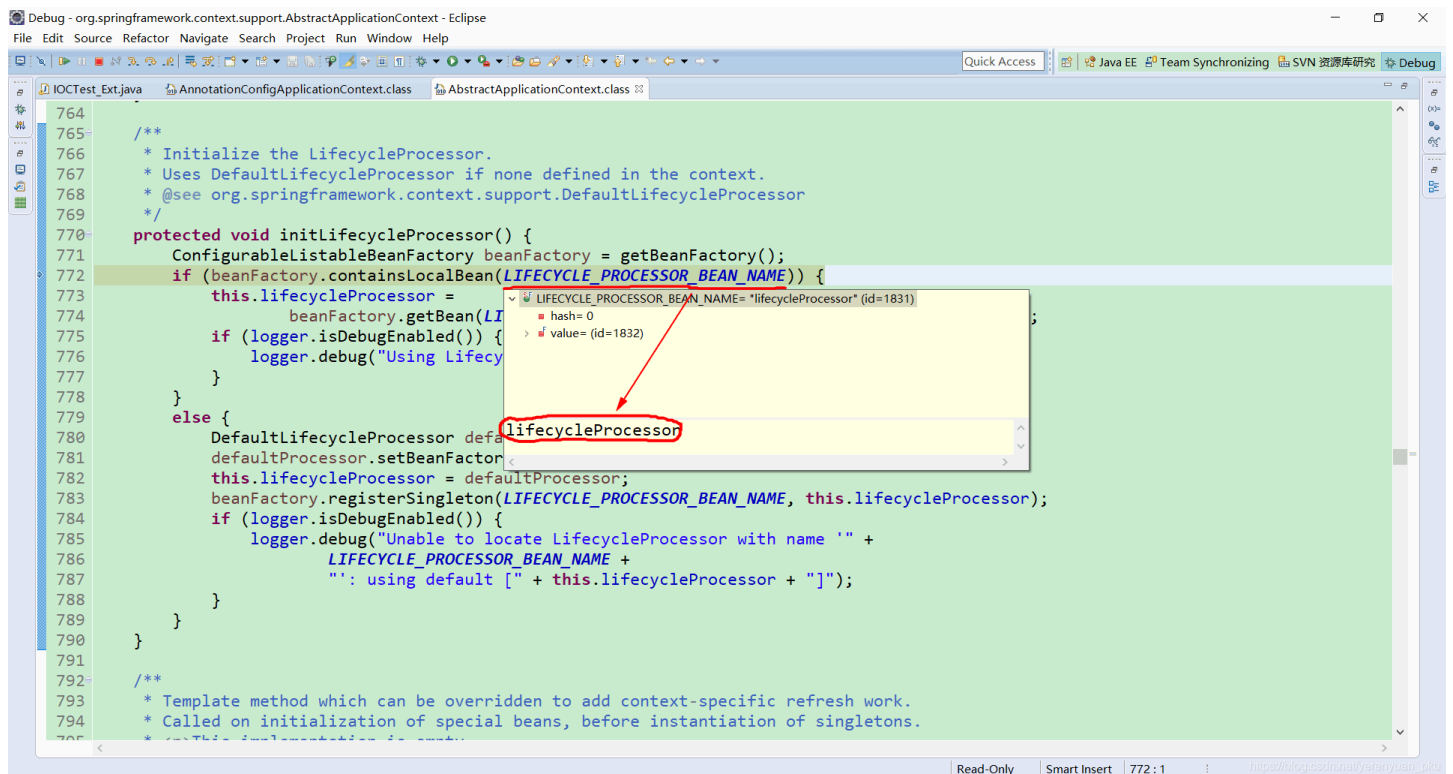


#### 获取BeanFactory

从上图中可以知道，在`initLifecycleProcessor`方法里面一开始就是来获取`BeanFactory`的，而这个`BeanFactory`，我们之前早就准备好了。

#### 看容器中是否有id为`lifecycleProcessor`，类型是`LifecycleProcessor`的组件

按下 **F6** 快捷键让程序继续往下运行，会发现有一个判断，即判断`BeanFactory`中是否有一个id为`lifecycleProcessor`的组件。我为什么会这么说呢，你只要看一下常量 `LIFECYCLE_PROCESSOR_BEAN_NAME` 的值就知道了，如下图所示，该常量的值就是`lifecycleProcessor`。



#### 若有，则赋值给 `this.lifecycleProcessor`

如果有的话，那么会从`BeanFactory`中获取到id为`lifecycleProcessor`，类型是`LifecycleProcessor`的组件，并将其赋值给 `this.lifecycleProcessor`。这可以从下面这行代码看出。

```
765-  /**
766-   * Initialize the LifecycleProcessor.
767-   * Uses DefaultLifecycleProcessor if none defined in the context.
768-   * @see org.springframework.context.support.DefaultLifecycleProcessor
769-   */
770-  protected void initLifecycleProcessor() {
771-      ConfigurableListableBeanFactory beanFactory = getBeanFactory();
772-      if (beanFactory.containsLocalBean(LIFECYCLE_PROCESSOR_BEAN_NAME)) {
773-          this.lifecycleProcessor =
774-              beanFactory.getBean(LIFECYCLE_PROCESSOR_BEAN_NAME, LifecycleProcessor.class);
775-          if (logger.isDebugEnabled()) {
776-              logger.debug("Using LifecycleProcessor [" + this.lifecycleProcessor + "]");
777-          }
778-      }
779-      else {
780-          DefaultLifecycleProcessor defaultProcessor = new DefaultLifecycleProcessor();
781-          defaultProcessor.setBeanFactory(beanFactory);
782-          this.lifecycleProcessor = defaultProcessor;
783-          beanFactory.registerSingleton(LIFECYCLE_PROCESSOR_BEAN_NAME, this.lifecycleProcessor);
784-          if (logger.isDebugEnabled()) {
785-              logger.debug("Unable to locate LifecycleProcessor with name '" +
786-                  LIFECYCLE_PROCESSOR_BEAN_NAME +
787-                  "': using default [" + this.lifecycleProcessor + "]");
788-          }
789-      }
790-  }
791-
792-  /**
793-   * Template method which can be overridden to add context-specific refresh work.
794-   * Called on initialization of special beans, before instantiation of singletons.
795-   * <p>This implementation is empty.
796-   * @throws BeansException in case of errors
```

不难发现，首先默认会从BeanFactory中寻找LifecycleProcessor这种类型的组件，即生命周期组件。由于我们是初次与LifecycleProcessor见面，对其还不是很熟悉，所以我们可以点过去看一看它的源码，如下图所示，发现它是一个接口。

```
2+ * Copyright 2002-2009 the original author or authors.
16
17 package org.springframework.context;
18
19 /**
20  * Strategy interface for processing Lifecycle beans within the ApplicationContext.
21  *
22  * @author Mark Fisher
23  * @author Juergen Hoeller
24  * @since 3.0
25  */
26 public interface LifecycleProcessor extends Lifecycle {
27
28     /**
29      * Notification of context refresh, e.g. for auto-starting components.
30      */
31     void onRefresh();
32
33     /**
34      * Notification of context close phase, e.g. for auto-stopping components.
35      */
36     void onClose();
37
38 }
39
```

而且，可以看到该接口中还定义了两个方法，一个是onRefresh方法，一个是onClose方法，它们能够在BeanFactory的生命周期期间进行回调哟🤔

如此一来，我们就可以自己来编写LifecycleProcessor接口的一个实现类了，该实现类的作用就是可以在BeanFactory的生命周期期间进行拦截，即在BeanFactory刷新完成以及关闭的时候，回调其里面的onRefresh和onClose这两方法。

当程序继续往下运行时，很显然，它并不会进入到if判断语句中，而是来到了下面的else分支语句中，这是因为容器在刚开始创建的时候，肯定还没有生命周期组件的。

若没有，则创建一个DefaultLifecycleProcessor类型的组件，并把创建好的组件注册在容器中

如果没有的话，那么Spring自己会创建一个DefaultLifecycleProcessor类型的对象，即默认的生命周期组件。

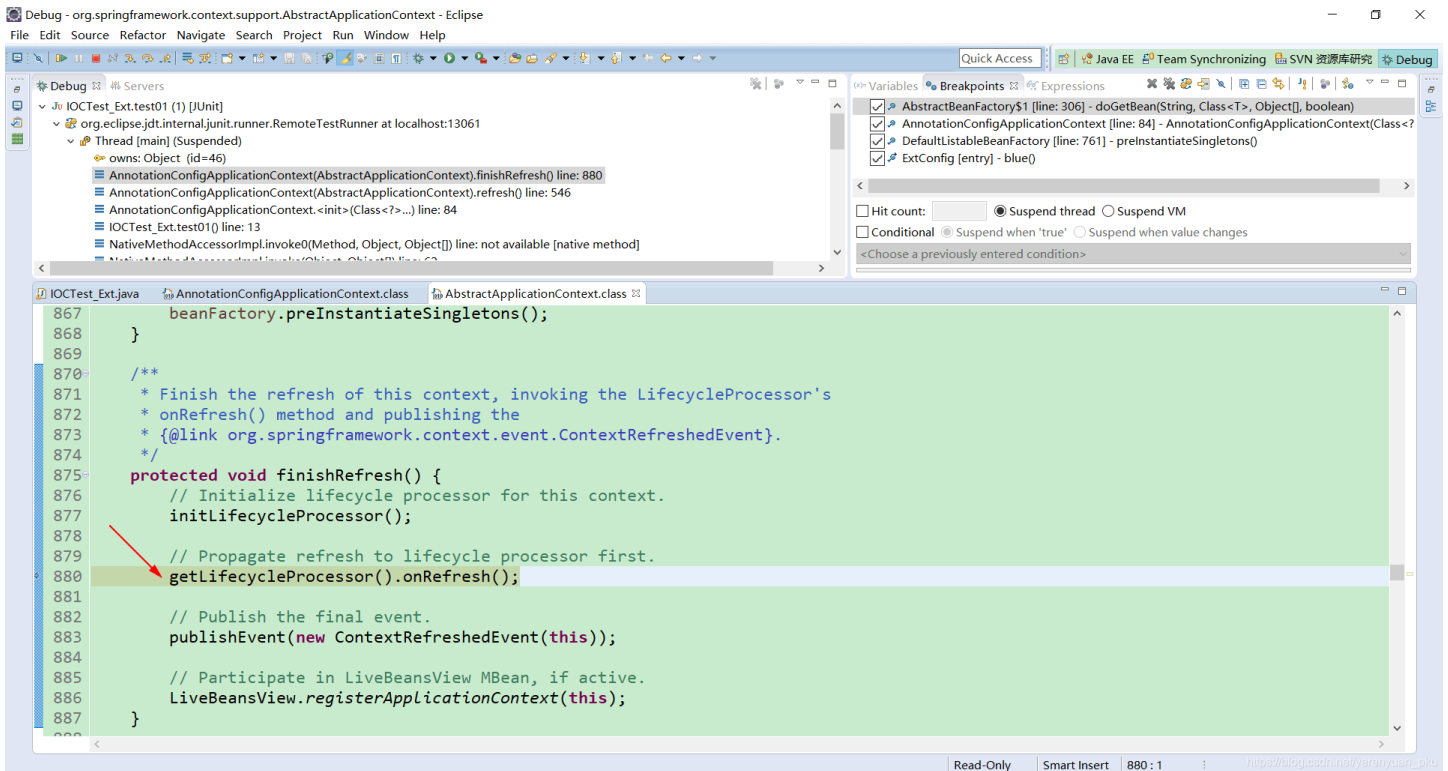
然后，把创建好的DefaultLifecycleProcessor类型的组件注册到容器中，所执行的是下面这行代码。

```
764
765 /**
766  * Initialize the LifecycleProcessor.
767  * Uses DefaultLifecycleProcessor if none defined in the context.
768  * @see org.springframework.context.support.DefaultLifecycleProcessor
769  */
770 protected void initLifecycleProcessor() {
771     ConfigurableListableBeanFactory beanFactory = getBeanFactory();
772     if (beanFactory.containsLocalBean(LIFECYCLE_PROCESSOR_BEAN_NAME)) {
773         this.lifecycleProcessor =
774             beanFactory.getBean(LIFECYCLE_PROCESSOR_BEAN_NAME, LifecycleProcessor.class);
775         if (logger.isDebugEnabled()) {
776             logger.debug("Using LifecycleProcessor [" + this.lifecycleProcessor + "]");
777         }
778     }
779     else {
780         DefaultLifecycleProcessor defaultProcessor = new DefaultLifecycleProcessor();
781         defaultProcessor.setBeanFactory(beanFactory);
782         this.lifecycleProcessor = defaultProcessor;
783         beanFactory.registerSingleton(LIFECYCLE_PROCESSOR_BEAN_NAME, this.lifecycleProcessor);
784         if (logger.isDebugEnabled()) {
785             logger.debug("Unable to locate LifecycleProcessor with name '" +
786                 LIFECYCLE_PROCESSOR_BEAN_NAME +
787                 ": using default [" + this.lifecycleProcessor + "]");
788         }
789     }
790 }
791
792 /**
793  * Template method which can be overridden to add context-specific refresh work.
794  * Called on initialization of special beans, before instantiation of singletons.
795  */
```

也就是说，容器中会有一个默认的生命周期组件。这样，我们以后其他组件想要使用生命周期组件，直接自动注入这个生命周期组件即可。

这里，我得多说一嘴，你也不要嫌我烦，所有Spring创建的组件，基本上都是这个逻辑，它把组件创建过来以后，就会添加到容器中，这样就能方便我们程序员来使用了。

最后，让程序继续往下运行，直至运行到下面这行代码处为止。



```
867     beanFactory.preInstantiateSingletons();
868 }
869
870 /**
871  * Finish the refresh of this context, invoking the LifecycleProcessor's
872  * onRefresh() method and publishing the
873  * {@link org.springframework.context.event.ContextRefreshedEvent}.
874  */
875 protected void finishRefresh() {
876     // Initialize lifecycle processor for this context.
877     initLifecycleProcessor();
878
879     // Propagate refresh to lifecycle processor first.
880     getLifecycleProcessor().onRefresh();
881
882     // Publish the final event.
883     publishEvent(new ContextRefreshedEvent(this));
884
885     // Participate in LiveBeansView MBean, if active.
886     LiveBeansView.registerApplicationContext(this);
887 }
```

可以看到，这儿会拿到生命周期组件，然后再回调其onRefresh方法。

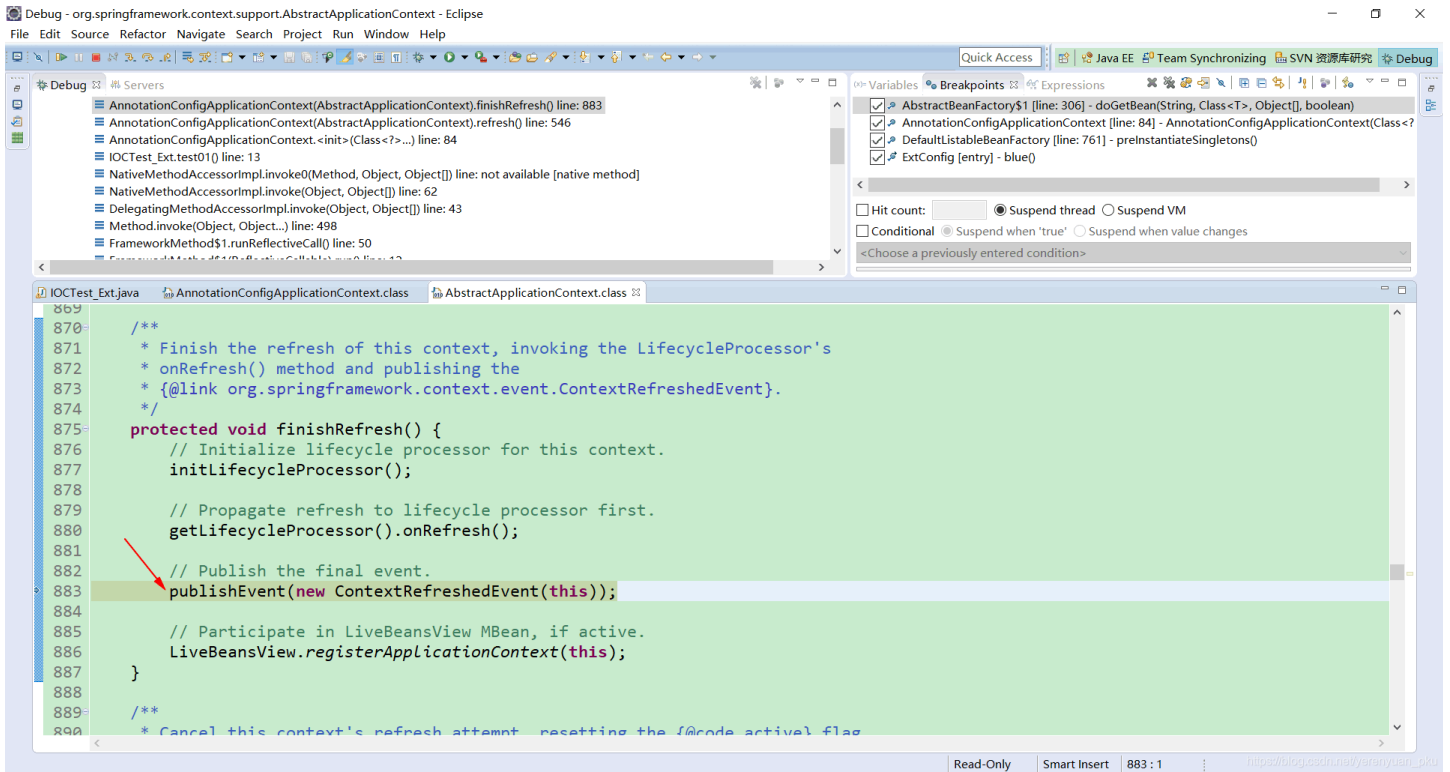
### 回调生命周期处理器的onRefresh方法

从上图我们可以看到，当程序运行到 `getLifecycleProcessor().onRefresh();` 这行代码处时，会先拿到我们前面定义的生命周期处理器（即监听BeanFactory生命周期的处理器），然后再回调其onRefresh方法，也就是容器刷新完成的方法。

### 发布容器刷新完成事件

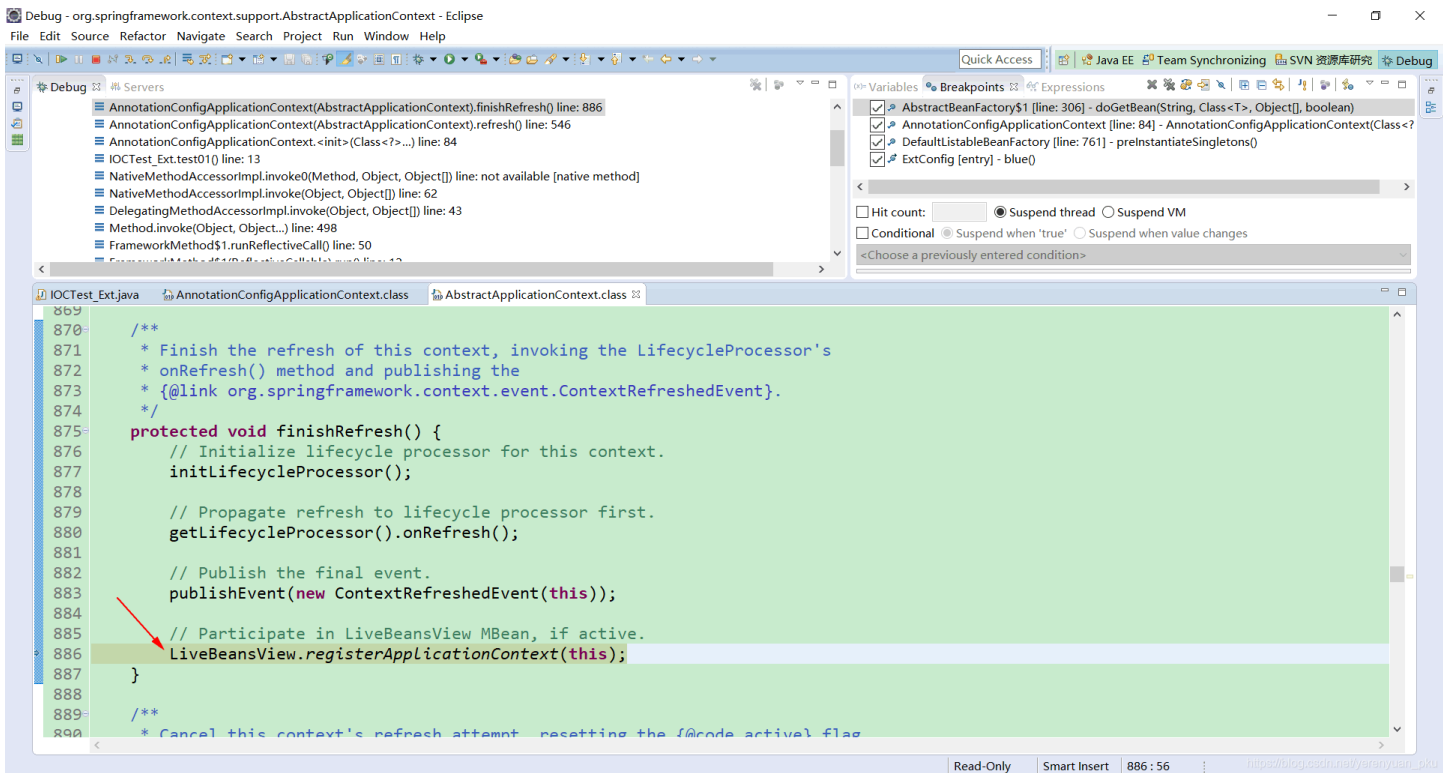
我们让程序继续往下运行，运行一步即可，这时，程序来到了下面这行代码处。





很明显，这儿是来发布容器刷新完成事件的。如何来发布容器刷新完成事件，想必不用我来说了吧！我之前就已经详细讲述过了，要是你还不知道的话，那么可以参考《Spring注解驱动开发第39讲——你不知道的ApplicationListener的原理》这一讲中的事件发布流程这一小节。

接着，继续让程序往下运行，运行一步即可，这时，程序来到了下面这行代码处。



这是finishRefresh方法里面的最后一步了，这儿是来干嘛的呢？我也说不清，好像是暴露一些什么MBean的，搞不清就不必深究了，直接略过。

至此，Spring IOC容器的整个创建过程，我就帮大家大致地过了一遍。如有我写的不对的地方，欢迎指出，我一定竭力修改。