

Spring注解驱动开发第41讲——Spring IOC容器创建源码解析(一)之BeanFactory的创建以及预准备工作

文章目录

写在前面

BeanFactory的创建以及预准备工作

prepareRefresh(): 刷新容器前的预处理工作

initPropertySources(): 子类自定义个性化的属性设置的方法

getEnvironment().validateRequiredProperties(): 获取其环境变量, 然后校验属性的合法性

保存容器中早期的事件

obtainFreshBeanFactory(): 获取BeanFactory对象

refreshBeanFactory(): 创建BeanFactory对象, 并为其设置一个序列化id

getBeanFactory(): 返回设置了序列化id后的BeanFactory对象

prepareBeanFactory(beanFactory): BeanFactory的预准备工作, 即对BeanFactory进行一些预处理

postProcessBeanFactory(beanFactory): BeanFactory准备工作完成后进行的后置处理工作

写在前面

在前面, 我们已经学会了怎样来使用ApplicationListener, 也研究了一下其内部原理。而从这一讲开始, 我们就要结合我们以前学过的所有内容, 来梳理一下Spring整个容器的创建以及 **初始化** 过程。我是希望通过对Spring源码的整个分析, 令大家对Spring内部的工作原理以及运行机制能有一个更深刻的理解。

接下来, 我们来分析并详细记录一下Spring容器的创建以及初始化过程。

BeanFactory的创建以及预准备工作

我们先来看一下如下的一个单元测试类 (例如IOCTest_Ext)。

```
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



我们知道如下这样一行代码是来new一个IOC容器的, 而且还可以看到传入了一个配置类。

```
1 AnnotationConfigApplicationContext applicationContext = new AnnotationConfigApplicationContext(ExtConfig.class);
AI写代码java运行
```

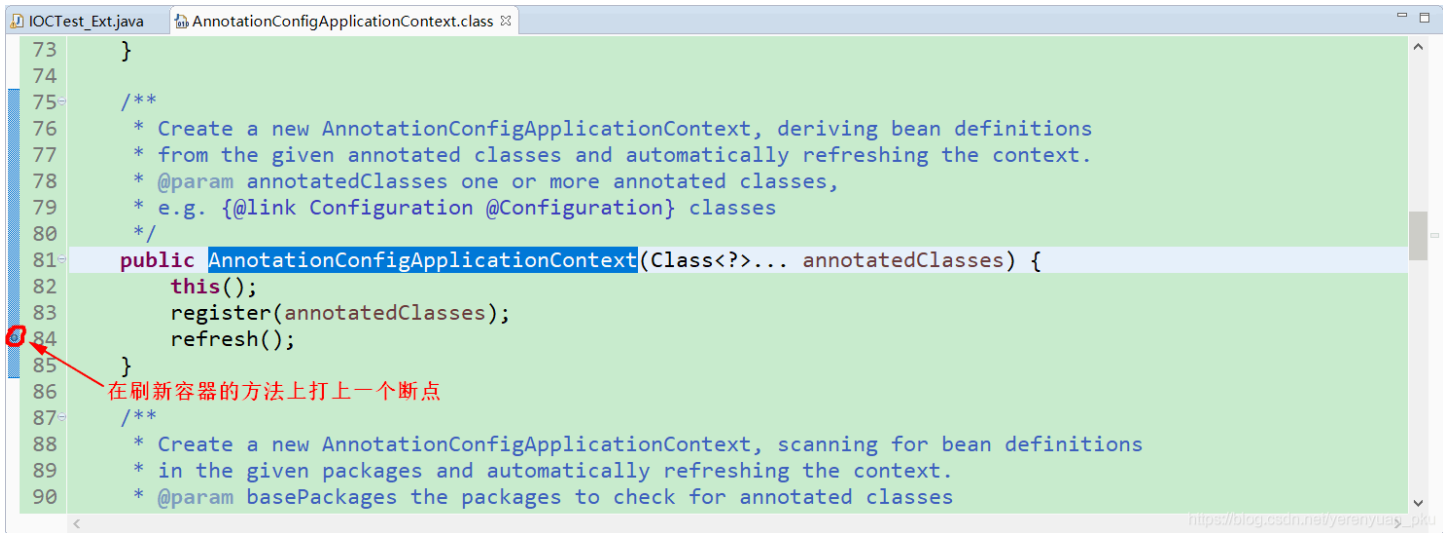
我们不妨点进去AnnotationConfigApplicationContext类的有参构造方法里面去看一看, 如下图所示, 相信大家对该有参构造方法是再熟悉不过了。



```
73 }
74
75 /**
76  * Create a new AnnotationConfigApplicationContext, deriving bean definitions
77  * from the given annotated classes and automatically refreshing the context.
78  * @param annotatedClasses one or more annotated classes,
79  * e.g. {@link Configuration @Configuration} classes
80  */
81 public AnnotationConfigApplicationContext(Class<?>... annotatedClasses) {
82     this();
83     register(annotatedClasses);
84     refresh();
85 }
86
87 /**
88  * Create a new AnnotationConfigApplicationContext, scanning for bean definitions
89  * in the given packages and automatically refreshing the context.
90  * @param basePackages the packages to check for annotated classes
```

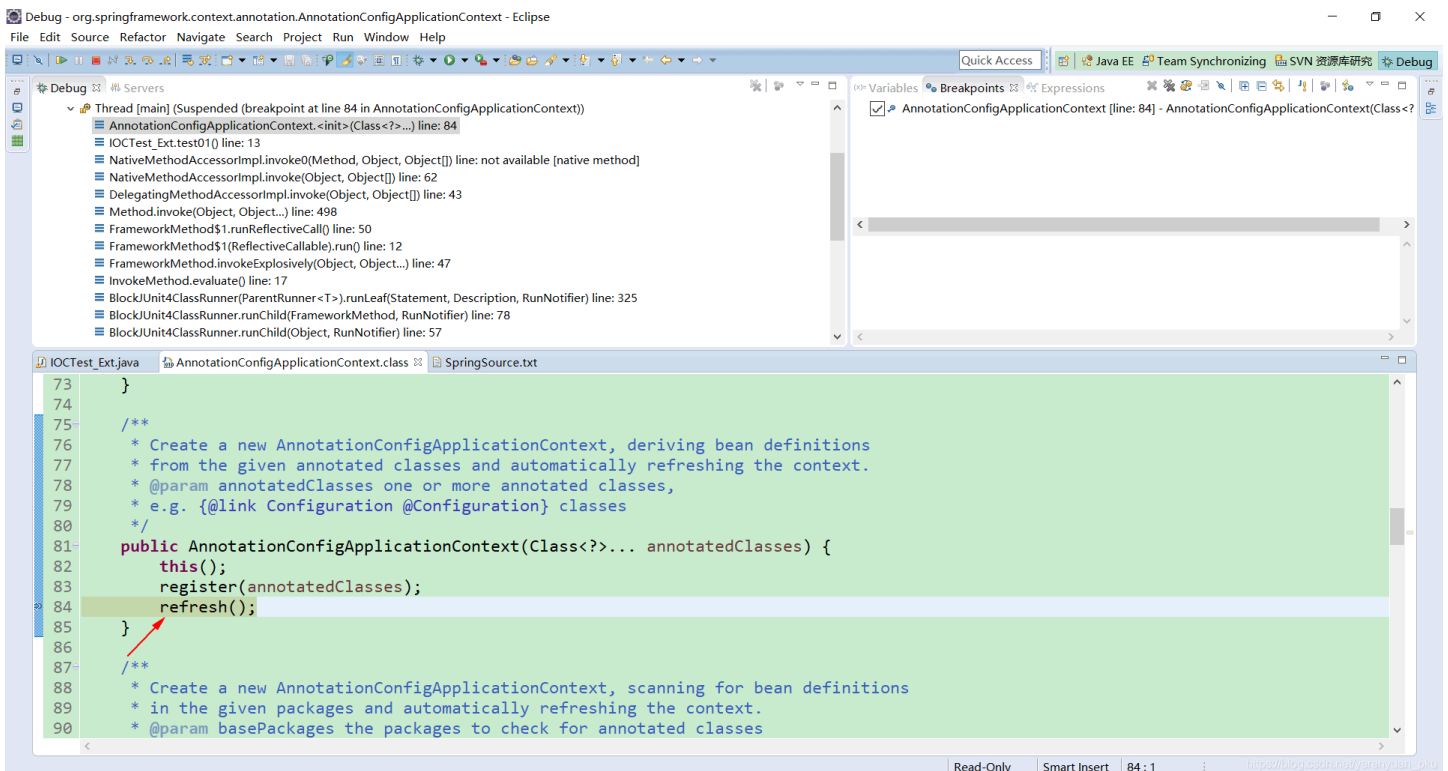
由于我们现在是来分析Spring容器的创建以及初始化过程，所以我们将核心的关注点放在refresh 方法上，也即刷新容器。该方法运行完以后，容器就创建完成了，包括所有的bean对象也都创建和初始化完成了。

接下来，我们在刷新容器的方法上打上一个断点，如下图所示，重点分析一下刷新容器这个方法里面到底做了些什么事。



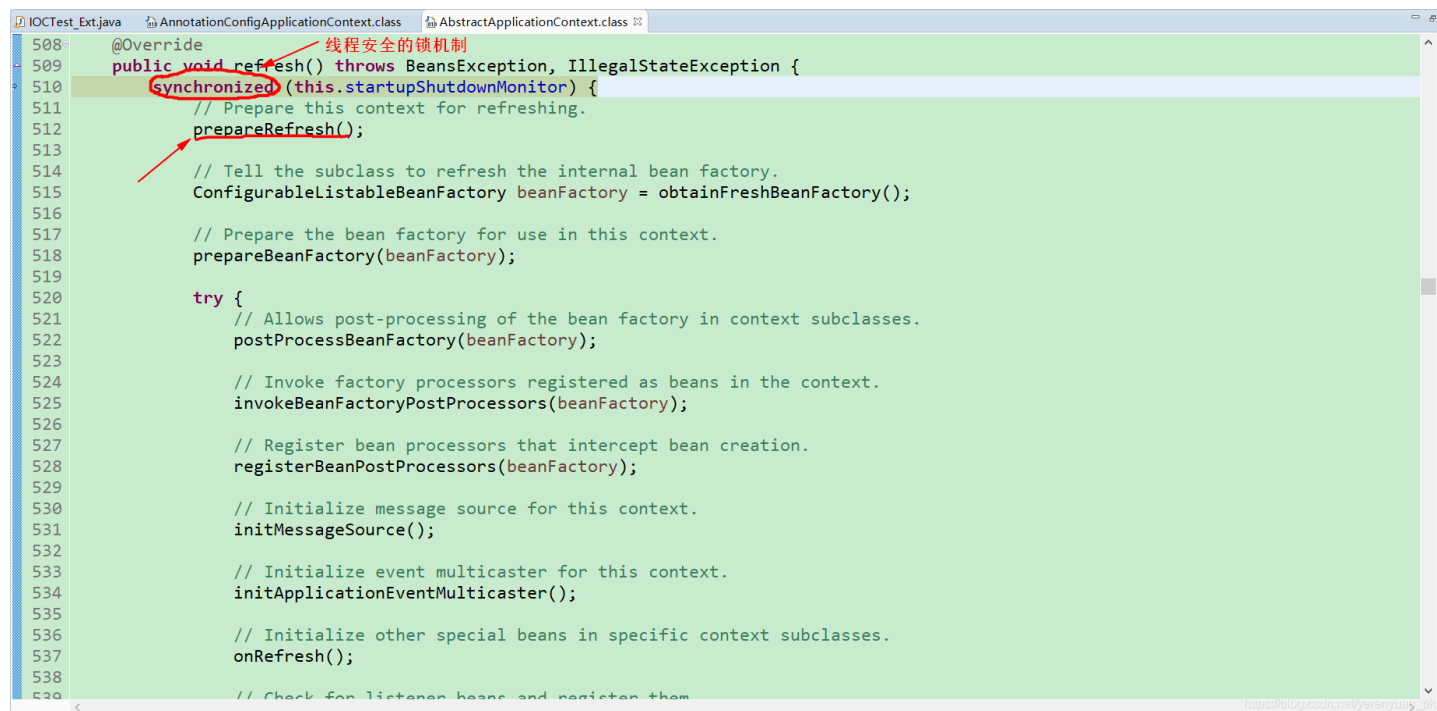
```
73 }
74
75 /**
76  * Create a new AnnotationConfigApplicationContext, deriving bean definitions
77  * from the given annotated classes and automatically refreshing the context.
78  * @param annotatedClasses one or more annotated classes,
79  * e.g. {@link Configuration @Configuration} classes
80  */
81 public AnnotationConfigApplicationContext(Class<?>... annotatedClasses) {
82     this();
83     register(annotatedClasses);
84     refresh();
85 }
86
87 /**
88  * Create a new AnnotationConfigApplicationContext, scanning for bean definitions
89  * in the given packages and automatically refreshing the context.
90  * @param basePackages the packages to check for annotated classes
```

我们以debug的方式运行IOCTest_Ext测试类中的test01方法，如下图所示，程序现在停到了标注断点的refresh方法处。



```
73 }
74
75 /**
76  * Create a new AnnotationConfigApplicationContext, deriving bean definitions
77  * from the given annotated classes and automatically refreshing the context.
78  * @param annotatedClasses one or more annotated classes,
79  * e.g. {@link Configuration @Configuration} classes
80  */
81 public AnnotationConfigApplicationContext(Class<?>... annotatedClasses) {
82     this();
83     register(annotatedClasses);
84     refresh();
85 }
86
87 /**
88  * Create a new AnnotationConfigApplicationContext, scanning for bean definitions
89  * in the given packages and automatically refreshing the context.
90  * @param basePackages the packages to check for annotated classes
```

按下 **F5** 快捷键进入refresh方法里面，如下图所示，可以看到映入眼帘的是一个线程安全的锁机制，除此之外，你还能看到第一个方法，即prepareRefresh方法，顾名思义，它是来执行刷新容器前的预处理工作的。



```
508: @Override
509: public void refresh() throws BeansException, IllegalStateException {
510:     synchronized (this.startupShutdownMonitor) {
511:         // Prepare this context for refreshing.
512:         prepareRefresh();
513:
514:         // Tell the subclass to refresh the internal bean factory.
515:         ConfigurableListableBeanFactory beanFactory = obtainFreshBeanFactory();
516:
517:         // Prepare the bean factory for use in this context.
518:         prepareBeanFactory(beanFactory);
519:
520:         try {
521:             // Allows post-processing of the bean factory in context subclasses.
522:             postProcessBeanFactory(beanFactory);
523:
524:             // Invoke factory processors registered as beans in the context.
525:             invokeBeanFactoryPostProcessors(beanFactory);
526:
527:             // Register bean processors that intercept bean creation.
528:             registerBeanPostProcessors(beanFactory);
529:
530:             // Initialize message source for this context.
531:             initMessageSource();
532:
533:             // Initialize event multicaster for this context.
534:             initApplicationEventMulticaster();
535:
536:             // Initialize other special beans in specific context subclasses.
537:             onRefresh();
538:
539:             // Check for listener beans and register them
```

那么问题来了，刷新容器前的这个预处理工作它到底都做了哪些事呢？下面我们就来详细说说。

prepareRefresh(): 刷新容器前的预处理工作

按下 **F6** 快捷键让程序往下运行，运行到prepareRefresh方法处时，按下 **F5** 快捷键进入该方法里面，如下图所示，可以看到会先清理一些缓存，我们的关注点不在这儿，所以略过。



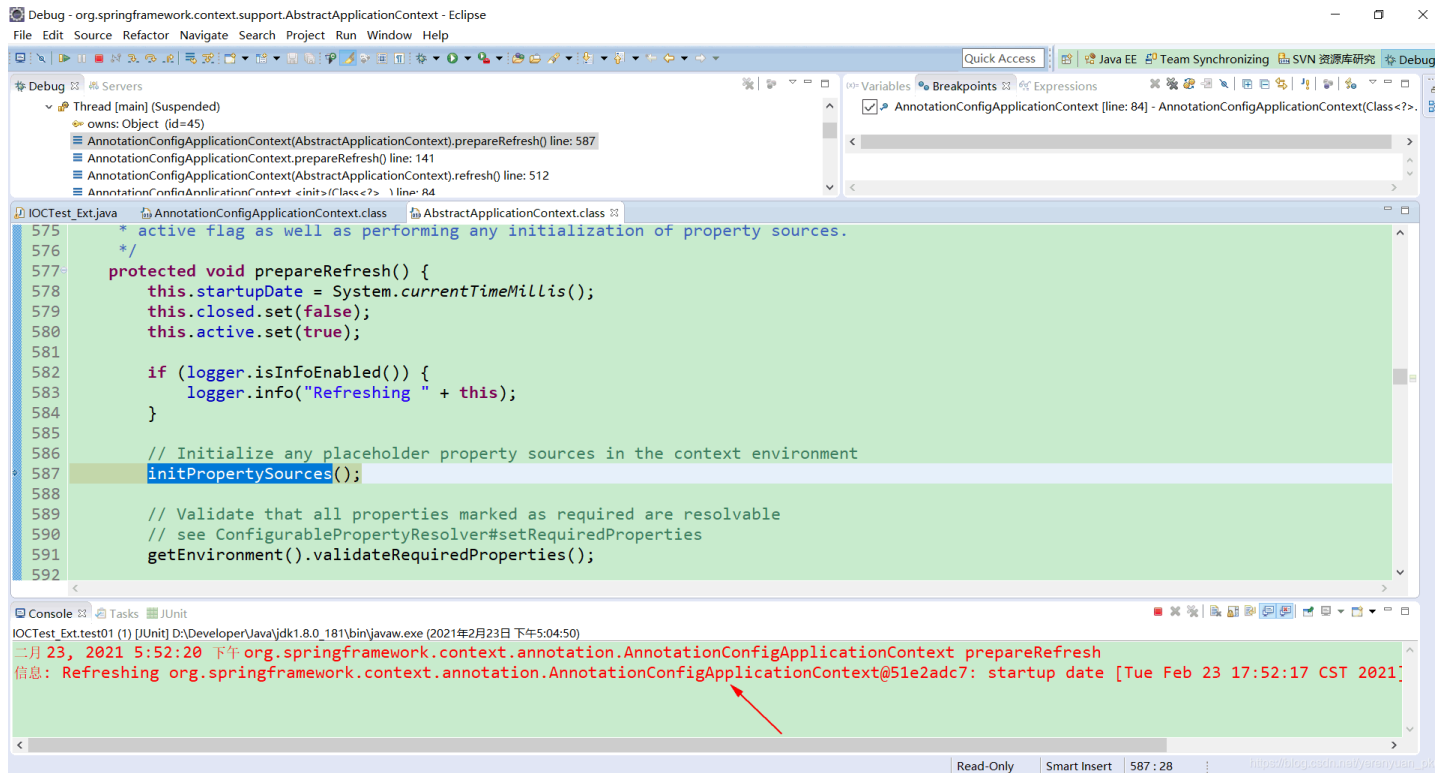
```
130: * <p>Any call to this method must occur prior to calls to {@link #register(Class...)}
131: * and/or {@link #scan(String...)}
132: */
133: public void setScopeMetadataResolver(ScopeMetadataResolver scopeMetadataResolver) {
134:     this.reader.setScopeMetadataResolver(scopeMetadataResolver);
135:     this.scanner.setScopeMetadataResolver(scopeMetadataResolver);
136: }
137:
138: @Override
139: protected void prepareRefresh() {
140:     this.scanner.clearCache();
141:     super.prepareRefresh();
142: }
143:
144: //-----
145: // Implementation of AnnotationConfigRegistry
146: //-----
147:
```

继续按下 **F6** 快捷键让程序往下运行，运行到 `super.prepareRefresh()` 这行代码处，这儿也是来执行刷新容器前的预处理工作的。按下 **F5** 快捷键进入该方法里面，如下图所示，我们可以看到它里面都做了些什么预处理工作。



```
575: * active flag as well as performing any initialization of property sources.
576: */
577: protected void prepareRefresh() {
578:     this.startupDate = System.currentTimeMillis();
579:     this.closed.set(false);
580:     this.active.set(true);
581:
582:     if (logger.isInfoEnabled()) {
583:         logger.info("Refreshing " + this);
584:     }
585:
586:     // Initialize any placeholder property sources in the context environment
587:     initPropertySources();
588:
589:     // Validate that all properties marked as required are resolvable
590:     // see ConfigurablePropertyResolver#setRequiredProperties
591:     getEnvironment().validateRequiredProperties();
592:
593:     // Allow for the collection of early ApplicationEvents,
594:     // to be published once the multicaster is available...
595:     this.earlyApplicationEvents = new LinkedHashSet<ApplicationEvent>();
596: }
597:
598: /**
```

发现就是先记录下当前时间，然后设置下当前容器是否是关闭、是否是活跃等状态，除此之外，还会打印当前容器的刷新日志。如果你要是不信的话，那么可以按下 **F6** 快捷键让程序往下运行，直至运行到initPropertySources方法处，你便能看到Eclipse 控制台打印出了一些当前容器的刷新日志，如下图所示。



这时，我们看到了第一个方法，即initPropertySources方法。那么，它里面做了些啥事呢？

initPropertySources(): 子类自定义个性化的属性设置的方法

顾名思义，该方法是来初始化一些属性设置的。那么，该方法里面究竟做了些啥事呢？我们不妨进去一探究竟，按下 **F5** 快捷键进入该方法中，如下图所示，发现它是空的，没有做任何事情。



但是，我们要注意该方法是protected类型的，这意味着它是留给子类自定义个性化的属性设置的。例如，我们可以自己来写一个AnnotationConfigApplicationContext的子类，在容器刷新的时候，重写这个方法，这样，我们就可以在子类（也叫子容器）的方法中自定义一些个性化的属性设置了。

这个方法只有在子类自定义的时候有用，只不过现在它还是空的，里面啥也没做。

getEnvironment().validateRequiredProperties(): 获取其环境变量，然后校验属性的合法性

继续按下 **F6** 快捷键让程序往下运行，直至运行到以下这行代码处。



```
576
577     protected void prepareRefresh() {
578         this.startupDate = System.currentTimeMillis();
579         this.closed.set(false);
580         this.active.set(true);
581
582         if (logger.isInfoEnabled()) {
583             logger.info("Refreshing " + this);
584         }
585
586         // Initialize any placeholder property sources in the context environment
587         initPropertySources();
588
589         // Validate that all properties marked as required are resolvable
590         // see ConfigurablePropertyResolver#setRequiredProperties
591         getEnvironment().validateRequiredProperties();
592
593         // Allow for the collection of early ApplicationEvents,
594         // to be published once the multicaster is available...
595         this.earlyApplicationEvents = new LinkedHashSet<ApplicationEvent>();
596     }
```

首先是来获取其环境变量，然后校验属性的合法性

这段代码的意思很容易知道，前面不是自定义了一些个性化的属性吗？这儿就是来校验这些属性的合法性的。

那么是怎么来进行属性校验的呢？首先是要来获取其环境变量，你可以按下 **F5** 快捷键进入 `getEnvironment` 方法中去看看，如下图所示，可以看到该方法就是用来获取其环境变量的。



```
294     }
295
296     /**
297      * Return the {@code Environment} for this application context in configurable
298      * form, allowing for further customization.
299      * <p>If none specified, a default environment will be initialized via
300      * {@link #createEnvironment()}.
301      */
302     @Override
303     public ConfigurableEnvironment getEnvironment() {
304         if (this.environment == null) {
305             this.environment = createEnvironment();
306         }
307         return this.environment;
308     }
309
310     /**
311      * Create and return a new {@link StandardEnvironment}.
312      * <p>Subclasses may override this method in order to supply
313      * a custom {@link ConfigurableEnvironment} implementation.
314     */
```

继续按下 **F6** 快捷键让程序往下运行，让程序再次运行到 `getEnvironment().validateRequiredProperties()` 这行代码处。然后，再次按下 **F5** 快捷键进入 `validateRequiredProperties` 方法中去看看，如下图所示，可以看到就是使用属性解析器来进行属性校验的。



```
505     @Override
506     public void setRequiredProperties(String... requiredProperties) {
507         this.propertyResolver.setRequiredProperties(requiredProperties);
508     }
509
510     @Override
511     public void validateRequiredProperties() throws MissingRequiredPropertiesException {
512         this.propertyResolver.validateRequiredProperties();
513     }
514
515     //-----
516     // Implementation of PropertyResolver interface
517     //-----
518
519     @Override
520     public boolean containsProperty(String key) {
521         return this.propertyResolver.containsProperty(key);
522     }
523
524     @Override
```

使用属性解析器来进行属性校验

只不过，我们现在没有自定义什么属性，所以，此时并没有做任何属性校验工作。

保存容器中早期的事件

继续按下 **F6** 快捷键让程序往下运行，直至运行到以下这行代码处。


```
577     protected void prepareRefresh() {
578         this.startupDate = System.currentTimeMillis();
579         this.closed.set(false);
580         this.active.set(true);
581
582         if (logger.isInfoEnabled()) {
583             logger.info("Refreshing " + this);
584         }
585
586         // Initialize any placeholder property sources in the context environment
587         initPropertySources();
588
589         // Validate that all properties marked as required are resolvable
590         // see ConfigurablePropertyResolver#setRequiredProperties
591         getEnvironment().validateRequiredProperties();
592
593         // Allow for the collection of early ApplicationEvents,
594         // to be published once the multicaster is available...
595         this.earlyApplicationEvents = new LinkedHashSet<ApplicationEvent>();
596     }
```

这儿是new了一个LinkedHashSet，它主要是来临时保存一些容器中早期的事件的。如果有事件发生，那么就存放在这个LinkedHashSet里面，这样，当事件派发器好了以后，直接用事件派发器把这些事件都派出去。

总结一下就是一句话，即允许收集早期的容器事件，等待事件派发器可用之后，即可进行发布。

至此，我们就分析完了prepareRefresh方法，以上就是该方法所做的事情。我们发现这个方法和BeanFactory并没有太大关系，因此，接下来我们还得来看下一个方法，即obtainFreshBeanFactory方法。

obtainFreshBeanFactory(): 获取BeanFactory对象

继续按下 F6 快捷键让程序往下运行，直至运行至以下这行代码处。

```
508     @Override
509     public void refresh() throws BeansException, IllegalStateException {
510         synchronized (this.startupShutdownMonitor) {
511             // Prepare this context for refreshing.
512             prepareRefresh();
513
514             // Tell the subclass to refresh the internal bean factory.
515             ConfigurableListableBeanFactory beanFactory = obtainFreshBeanFactory();
516
517             // Prepare the bean factory for use in this context.
518             prepareBeanFactory(beanFactory);
519
520             try {
521                 // Allows post-processing of the bean factory in context subclasses.
522                 postProcessBeanFactory(beanFactory);
523
524                 // Invoke factory processors registered as beans in the context.
525                 invokeBeanFactoryPostProcessors(beanFactory);
526
527                 // Register bean processors that intercept bean creation.
528                 registerBeanPostProcessors(beanFactory);
```

可以看到一个叫obtainFreshBeanFactory的方法，顾名思义，它是来获取BeanFactory的实例的。接下来，我们就来看看该方法里面究竟做了哪些事。

refreshBeanFactory(): 创建BeanFactory对象，并为其设置一个序列化id

按下 F5 快捷键进入该方法中，如下图所示，可以看到其获取BeanFactory实例的过程是下面这样子的。

```
605     }
606
607     /**
608      * Tell the subclass to refresh the internal bean factory.
609      * @return the fresh BeanFactory instance
610      * @see #refreshBeanFactory()
611      * @see #getBeanFactory()
612      */
613     protected ConfigurableListableBeanFactory obtainFreshBeanFactory() {
614         refreshBeanFactory();
615         ConfigurableListableBeanFactory beanFactory = getBeanFactory();
616         if (logger.isDebugEnabled()) {
617             logger.debug("Bean factory for " + getDisplayName() + ": " + beanFactory);
618         }
619         return beanFactory;
620     }
621
622     /**
623      * Configure the factory's standard context characteristics,
624      * such as the context's ClassLoader and post-processors.
625      * @param beanFactory the BeanFactory to configure
```

发现首先调用了个叫refreshBeanFactory的方法，该方法见名思义，应该是来刷新BeanFactory的。那么，该方法里面又做了哪些事呢？

我们可以按下 F5 快捷键进入该方法中去看看，如下图所示，发现程序来到了GenericApplicationContext类里面。

```
250
251 //-----
252 // Implementations of AbstractApplicationContext's template methods
253 //-----
254
255 /**
256  * Do nothing: We hold a single internal BeanFactory and rely on callers
257  * to register beans through our public methods (or the BeanFactory's).
258  * @see #registerBeanDefinition
259  */
260 @Override
261 protected final void refreshBeanFactory() throws IllegalStateException {
262     if (!this.refreshed.compareAndSet(false, true)) {
263         throw new IllegalStateException(
264             "GenericApplicationContext does not support multiple refresh attempts: just call 'refresh' once");
265     }
266     this.beanFactory.setSerializationId(getId());
267 }
268
269 @Override
270 protected void cancelRefresh(BeansException ex) {
```

而且，我们还可以看到在以上refreshBeanFactory方法中，会先判断是不是重复刷新了。于是，我们继续按下 **F6** 快捷键让程序往下运行，发现程序并没有进入到if判断语句中，而是来到了下面这行代码处。

```
250
251 //-----
252 // Implementations of AbstractApplicationContext's template methods
253 //-----
254
255 /**
256  * Do nothing: We hold a single internal BeanFactory and rely on callers
257  * to register beans through our public methods (or the BeanFactory's).
258  * @see #registerBeanDefinition
259  */
260 @Override
261 protected final void refreshBeanFactory() throws IllegalStateException {
262     if (!this.refreshed.compareAndSet(false, true)) {
263         throw new IllegalStateException(
264             "GenericApplicationContext does not support multiple refresh attempts: just call 'refresh' once");
265     }
266     this.beanFactory.setSerializationId(getId());
267 }
268
269 @Override
270 protected void cancelRefresh(BeansException ex) {
```

程序运行到这里，你会不会有一个大大的疑问，那就是我们的beanFactory不是还没创建么，怎么在这儿又开始调用方法了呢，难道是已经创建了吗？

我们向上翻阅GenericApplicationContext类的代码，发现原来是在这个类的无参构造方法里面，就已经实例化了beanFactory这个对象。也就是说，在创建GenericApplicationContext对象时，无参构造器里面就new出来了beanFactory这个对象。

```
87
88 public class GenericApplicationContext extends AbstractApplicationContext implements BeanDefinitionRegistry {
89
90     private final DefaultListableBeanFactory beanFactory;
91
92     private ResourceLoader resourceLoader;
93
94     private boolean customClassLoader = false;
95
96     private final AtomicBoolean refreshed = new AtomicBoolean();
97
98
99     /**
100      * Create a new GenericApplicationContext.
101      * @see #registerBeanDefinition
102      * @see #refresh
103      */
104     public GenericApplicationContext() {
105         this.beanFactory = new DefaultListableBeanFactory();
106     }
107
108     /**
```

相当于我们做了非常核心的一步，即创建了一个beanFactory对象，而且该对象还是DefaultListableBeanFactory类型的。

现在，我们已经知道了在GenericApplicationContext这个类的无参构造方法里面，就已经实例化了beanFactory这个对象。那么，你可能会有疑问，究竟是在什么地方调用GenericApplicationContext类的无参构造方法的呢？

这时，我们可以去看一下我们的单元测试类（例如IOCTest_Ext），如下图所示。

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

只要点进去AnnotationConfigApplicationContext类里面去看一看，你就知道大概了，如下图所示，原来AnnotationConfigApplicationContext类继承了GenericApplicationContext这个类，所以，当我们实例化AnnotationConfigApplicationContext时就会调用其父类的构造方法，相应地这时就会对我们的BeanFactory进行实例化了。

```
43 * @see #register
44 * @see #scan
45 * @see AnnotatedBeanDefinitionReader
46 * @see ClassPathBeanDefinitionScanner
47 * @see org.springframework.context.support.GenericXmlApplicationContext
48 */
49 public class AnnotationConfigApplicationContext extends GenericApplicationContext implements AnnotationConfigRegistry {
50
51     private final AnnotatedBeanDefinitionReader reader;
52
53     private final ClassPathBeanDefinitionScanner scanner;
54
55     /**
56      * Create a new AnnotationConfigApplicationContext that needs to be populated
57      * through {@link #register} calls and then manually {@link #refresh} refreshed}.
58      */
59     public AnnotationConfigApplicationContext() {
60         this.reader = new AnnotatedBeanDefinitionReader(this);
61         this.scanner = new ClassPathBeanDefinitionScanner(this);
62     }
63
64 }
```

好了，我们不扯远了，还是回到主题。BeanFactory对象创建好了之后，接下来就是要给其设置一个序列化id，相当于打了一个id标识。我们不妨Inspect一下getId方法，发现它是org.springframework.context.annotation.AnnotationConfigApplicationContext@51e2adc7这么一长串的字符串，原来这个序列化id就是它啊！

```
Debug - org.springframework.context.support.GenericApplicationContext - Eclipse
File Edit Source Refactor Navigate Search Project Run Window Help

Thread [main] (Suspended)
  owns: Object (id=45)
  AnnotationConfigApplicationContext(GenericApplicationContext).refreshBeanFactory() line: 266
  AnnotationConfigApplicationContext(AbtractApplicationContext).obtainFreshBeanFactory() line: 614
  AnnotationConfigApplicationContext(AbtractApplicationContext).refresh() line: 515
  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
  DelegationMethodAccessorImpl.invoke(Object, Object[]) line: 43

Variables Breakpoints Expressions
AnnotationConfigApplicationContext [line: 84] - AnnotationConfigApplicationContext(Class<?>)

// Implementations of AbstractApplicationContext's template methods
// -----
/**
 * Do nothing: We hold a single internal BeanFactory
 * to register beans through our public methods.
 * @see #registerBeanDefinition
 */
@Override
protected final void refreshBeanFactory() throws BeansException {
    if (!this.refreshed.compareAndSet(false, true))
        throw new IllegalStateException(
            "GenericApplicationContext already initialized."
        );
    this.beanFactory.setSerializationId(getId());
}

@Override
protected void cancelRefresh(BeansException ex) {
    this.beanFactory.setSerializationId(null);
    super.cancelRefresh(ex);
}
```

按下 F6 快捷键让程序往下运行，直至程序运行到下面这行代码处，refreshBeanFactory方法就执行完了。



```
605 }
606
607 /**
608  * Tell the subclass to refresh the internal bean factory.
609  * @return the fresh BeanFactory instance
610  * @see #refreshBeanFactory()
611  * @see #getBeanFactory()
612  */
613 protected ConfigurableListableBeanFactory obtainFreshBeanFactory() {
614     refreshBeanFactory();
615     ConfigurableListableBeanFactory beanFactory = getBeanFactory();
616     if (logger.isDebugEnabled()) {
617         logger.debug("Bean factory for " + getDisplayName() + ": " + beanFactory);
618     }
619     return beanFactory;
620 }
621
622 /**
623  * Configure the factory's standard context characteristics,
624  * such as the context's ClassLoader and post-processors.
625  * @param beanFactory the BeanFactory to configure
626  */
```

该方法所做的事情很简单，无非就是创建了一个BeanFactory对象（DefaultListableBeanFactory类型的），并为其设置好了一个序列化id。

getBeanFactory(): 返回设置了序列化id后的BeanFactory对象

接下来，我们就要看看getBeanFactory方法了。按下 **F5** 快捷键进入该方法里面，如下图所示，发现它里面就只是做了一件事，即返回设置了序列化id后的BeanFactory对象。



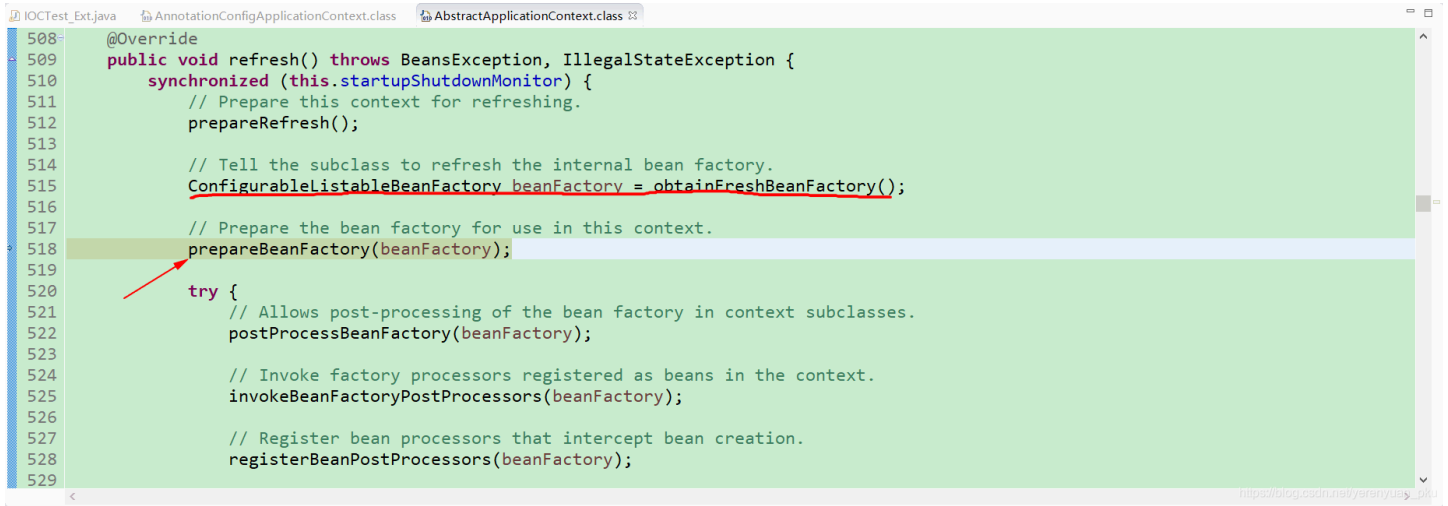
```
280 protected final void closeBeanFactory() {
281     this.beanFactory.setSerializationId(null);
282 }
283
284 /**
285  * Return the single internal BeanFactory held by this context
286  * (as ConfigurableListableBeanFactory).
287  */
288 @Override
289 public final ConfigurableListableBeanFactory getBeanFactory() {
290     return this.beanFactory;
291 }
292
293 /**
294  * Return the underlying bean factory of this context,
295  * available for registering bean definitions.
296  * <p><b>NOTE:</b> You need to call {@link #refresh()} to initialize the
297  * bean factory and its contained beans with application context semantics
298  * (autodetecting BeanFactoryPostProcessors, etc).
299  * @return the internal bean factory (as DefaultListableBeanFactory)
300  */
301 public final DefaultListableBeanFactory getDefaultListableBeanFactory() {
```

按下 **F6** 快捷键让程序往下运行，还是运行到下面这行代码处，可以看到这儿是用ConfigurationListableBeanFactory接口去接受我们刚刚实例化的BeanFactory对象（DefaultListableBeanFactory类型的）。



```
605 }
606
607 /**
608  * Tell the subclass to refresh the internal bean factory.
609  * @return the fresh BeanFactory instance
610  * @see #refreshBeanFactory()
611  * @see #getBeanFactory()
612  */
613 protected ConfigurableListableBeanFactory obtainFreshBeanFactory() {
614     refreshBeanFactory();
615     ConfigurableListableBeanFactory beanFactory = getBeanFactory();
616     if (logger.isDebugEnabled()) {
617         logger.debug("Bean factory for " + getDisplayName() + ": " + beanFactory);
618     }
619     return beanFactory;
620 }
621
622 /**
623  * Configure the factory's standard context characteristics,
624  * such as the context's ClassLoader and post-processors.
625  * @param beanFactory the BeanFactory to configure
626  */
```

继续按下 **F6** 快捷键让程序往下运行，一直让程序运行到下面这行代码处。程序运行至此，就返回了我们刚刚创建好的那个BeanFactory对象，只不过这个BeanFactory对象，由于我们刚创建，所以它里面的什么东西都是默认的一些设置。



```
508- @Override
509- public void refresh() throws BeansException, IllegalStateException {
510-     synchronized (this.startupShutdownMonitor) {
511-         // Prepare this context for refreshing.
512-         prepareRefresh();
513-
514-         // Tell the subclass to refresh the internal bean factory.
515-         ConfigurableListableBeanFactory beanFactory = obtainFreshBeanFactory();
516-
517-         // Prepare the bean factory for use in this context.
518-         prepareBeanFactory(beanFactory);
519-
520-         try {
521-             // Allows post-processing of the bean factory in context subclasses.
522-             postProcessBeanFactory(beanFactory);
523-
524-             // Invoke factory processors registered as beans in the context.
525-             invokeBeanFactoryPostProcessors(beanFactory);
526-
527-             // Register bean processors that intercept bean creation.
528-             registerBeanPostProcessors(beanFactory);
529-         }
```

至此，我们就分析完了obtainFreshBeanFactory方法，以上就是该方法所做的事情，即获取BeanFactory对象。

prepareBeanFactory(beanFactory): BeanFactory的预准备工作，即对BeanFactory进行一些预处理

接下来，我们就得来说道说道prepareBeanFactory方法了。顾名思义，该方法就是对BeanFactory做一些预处理，即BeanFactory的预准备工作。


为什么要在这儿对BeanFactory做一些预处理啊？因为我们前面刚刚创建好的BeanFactory还没有做任何设置呢，所以就在这儿对BeanFactory做一些设置了。那到底做了哪些设置呢？下面我们就得进入prepareBeanFactory方法里面一探究竟了。

按下 **F5** 快捷键进入该方法中，如下图所示，我们发现会对BeanFactory进行一系列的赋值（即设置一些属性）。比方说，设置BeanFactory的类加载器，就得像下面这样。



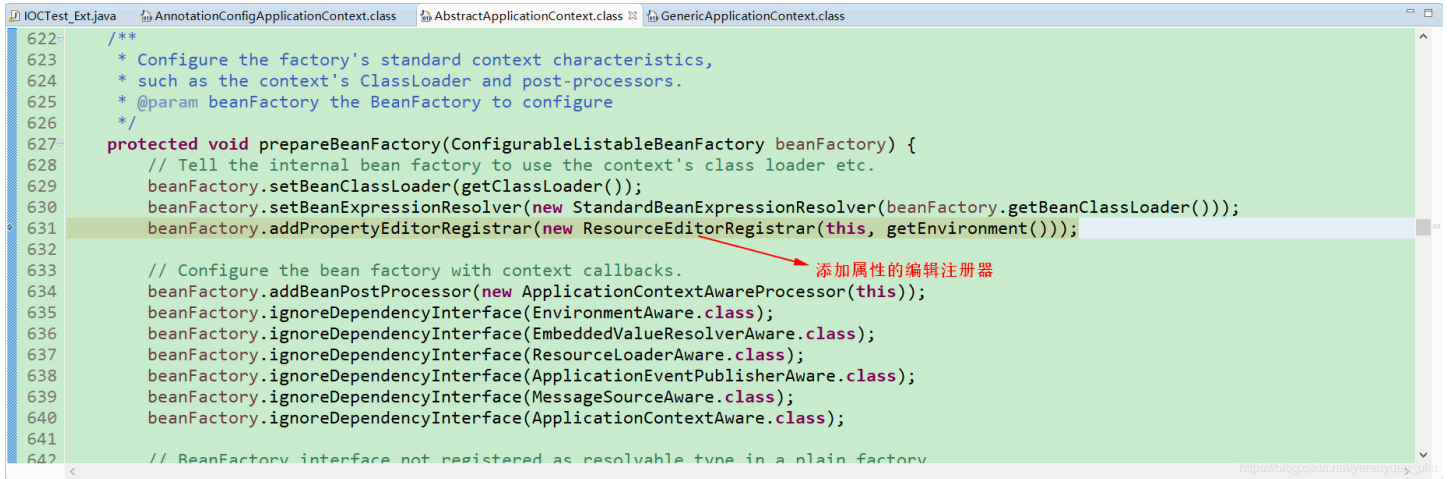
```
622- /**
623-  * Configure the factory's standard context characteristics,
624-  * such as the context's ClassLoader and post-processors.
625-  * @param beanFactory the BeanFactory to configure
626-  */
627- protected void prepareBeanFactory(ConfigurableListableBeanFactory beanFactory) {
628-     // Tell the internal bean factory to use the context's class loader etc.
629-     beanFactory.setBeanClassLoader(getClassLoader());
630-     beanFactory.setBeanExpressionResolver(new StandardBeanExpressionResolver(beanFactory.getBeanClassLoader());
631-     beanFactory.addPropertyEditorRegistrar(new ResourceEditorRegistrar(this, getEnvironment()));
632-
633-     // Configure the bean factory with context callbacks.
634-     beanFactory.addBeanPostProcessor(new ApplicationContextAwareProcessor(this));
635-     beanFactory.ignoreDependencyInterface(EnvironmentAware.class);
636-     beanFactory.ignoreDependencyInterface(EmbeddedValueResolverAware.class);
637-     beanFactory.ignoreDependencyInterface(ResourceLoaderAware.class);
638-     beanFactory.ignoreDependencyInterface(ApplicationEventPublisherAware.class);
639-     beanFactory.ignoreDependencyInterface(MessageSourceAware.class);
640-     beanFactory.ignoreDependencyInterface(ApplicationContextAware.class);
641-
642-     // BeanFactory interface not registered as resolvable type in a plain factory
```

以及，设置支持相关表达式语言的解析器。



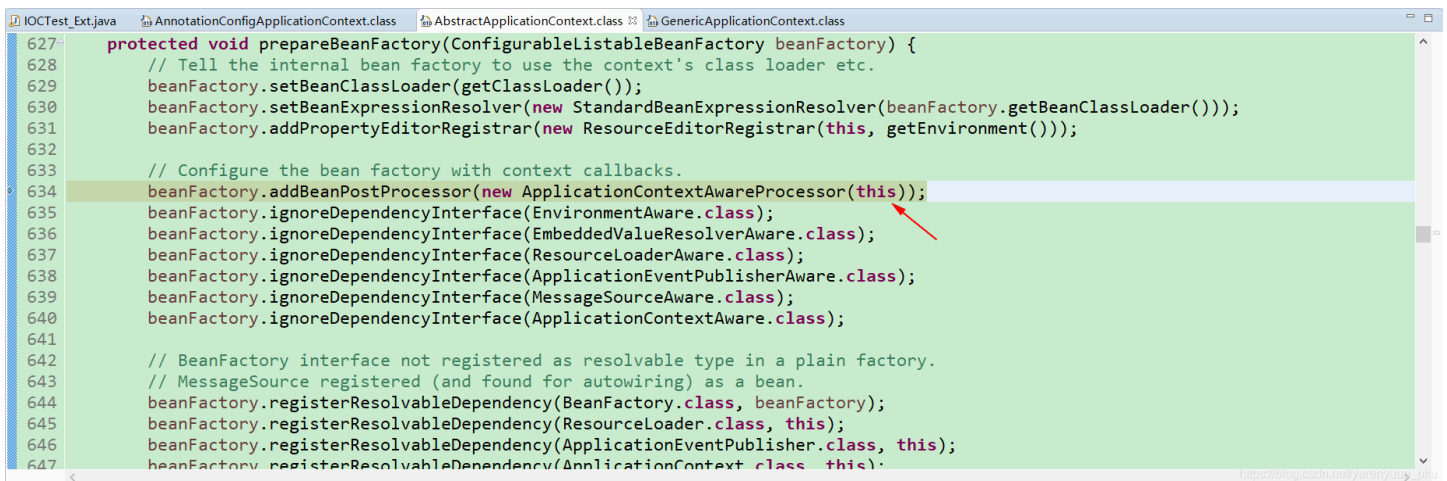
```
622- /**
623-  * Configure the factory's standard context characteristics,
624-  * such as the context's ClassLoader and post-processors.
625-  * @param beanFactory the BeanFactory to configure
626-  */
627- protected void prepareBeanFactory(ConfigurableListableBeanFactory beanFactory) {
628-     // Tell the internal bean factory to use the context's class loader etc.
629-     beanFactory.setBeanClassLoader(getClassLoader());
630-     beanFactory.setBeanExpressionResolver(new StandardBeanExpressionResolver(beanFactory.getBeanClassLoader());
631-     beanFactory.addPropertyEditorRegistrar(new ResourceEditorRegistrar(this, getEnvironment()));
632-
633-     // Configure the bean factory with context callbacks.
634-     beanFactory.addBeanPostProcessor(new ApplicationContextAwareProcessor(this));
635-     beanFactory.ignoreDependencyInterface(EnvironmentAware.class);
636-     beanFactory.ignoreDependencyInterface(EmbeddedValueResolverAware.class);
637-     beanFactory.ignoreDependencyInterface(ResourceLoaderAware.class);
638-     beanFactory.ignoreDependencyInterface(ApplicationEventPublisherAware.class);
639-     beanFactory.ignoreDependencyInterface(MessageSourceAware.class);
640-     beanFactory.ignoreDependencyInterface(ApplicationContextAware.class);
641-
642-     // BeanFactory interface not registered as resolvable type in a plain factory
```

还有，添加属性的编辑注册器等等，总之，会对BeanFactory设置非常多的东西。



```
622- /**
623-  * Configure the factory's standard context characteristics,
624-  * such as the context's ClassLoader and post-processors.
625-  * @param beanFactory the BeanFactory to configure
626-  */
627- protected void prepareBeanFactory(ConfigurableListableBeanFactory beanFactory) {
628-     // Tell the internal bean factory to use the context's class loader etc.
629-     beanFactory.setBeanClassLoader(getClassLoader());
630-     beanFactory.setBeanExpressionResolver(new StandardBeanExpressionResolver(beanFactory.getBeanClassLoader()));
631-     beanFactory.addPropertyEditorRegistrar(new ResourceEditorRegistrar(this, getEnvironment()));
632-
633-     // Configure the bean factory with context callbacks.
634-     beanFactory.addBeanPostProcessor(new ApplicationContextAwareProcessor(this));
635-     beanFactory.ignoreDependencyInterface(EnvironmentAware.class);
636-     beanFactory.ignoreDependencyInterface(EmbeddedValueResolverAware.class);
637-     beanFactory.ignoreDependencyInterface(ResourceLoaderAware.class);
638-     beanFactory.ignoreDependencyInterface(ApplicationEventPublisherAware.class);
639-     beanFactory.ignoreDependencyInterface(MessageSourceAware.class);
640-     beanFactory.ignoreDependencyInterface(ApplicationContextAware.class);
641-
642-     // BeanFactory interface not registered as resolvable type in a plain factory
```

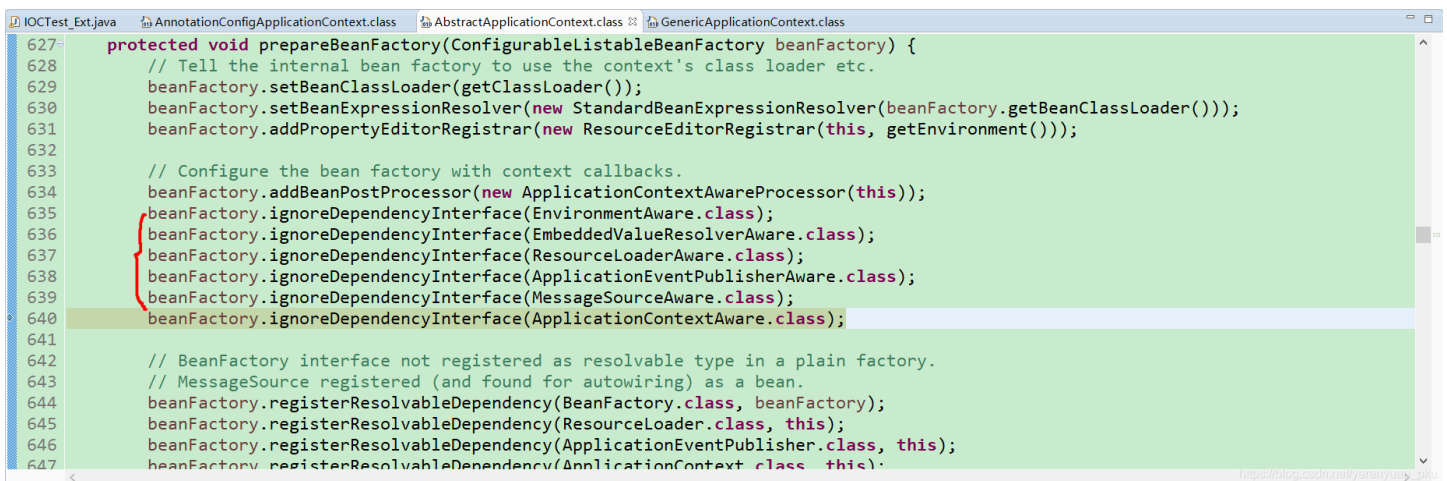
继续向下看prepareBeanFactory方法，你会发现还向BeanFactory中添加了一个BeanPostProcessor，即ApplicationContextAwareProcessor。



```
627- protected void prepareBeanFactory(ConfigurableListableBeanFactory beanFactory) {
628-     // Tell the internal bean factory to use the context's class loader etc.
629-     beanFactory.setBeanClassLoader(getClassLoader());
630-     beanFactory.setBeanExpressionResolver(new StandardBeanExpressionResolver(beanFactory.getBeanClassLoader()));
631-     beanFactory.addPropertyEditorRegistrar(new ResourceEditorRegistrar(this, getEnvironment()));
632-
633-     // Configure the bean factory with context callbacks.
634-     beanFactory.addBeanPostProcessor(new ApplicationContextAwareProcessor(this));
635-     beanFactory.ignoreDependencyInterface(EnvironmentAware.class);
636-     beanFactory.ignoreDependencyInterface(EmbeddedValueResolverAware.class);
637-     beanFactory.ignoreDependencyInterface(ResourceLoaderAware.class);
638-     beanFactory.ignoreDependencyInterface(ApplicationEventPublisherAware.class);
639-     beanFactory.ignoreDependencyInterface(MessageSourceAware.class);
640-     beanFactory.ignoreDependencyInterface(ApplicationContextAware.class);
641-
642-     // BeanFactory interface not registered as resolvable type in a plain factory.
643-     // MessageSource registered (and found for autowiring) as a bean.
644-     beanFactory.registerResolvableDependency(BeanFactory.class, beanFactory);
645-     beanFactory.registerResolvableDependency(ResourceLoader.class, this);
646-     beanFactory.registerResolvableDependency(ApplicationEventPublisher.class, this);
647-     beanFactory.registerResolvableDependency(ApplicationContext.class, this);
```

温馨提示：这儿只是向BeanFactory中添加了部分的BeanPostProcessor，而不是添加所有的，比如我们现在只是向BeanFactory中添加了一个叫ApplicationContextAwareProcessor的BeanPostProcessor。它的作用，我们之前也看过了，就是在bean初始化以后来判断这个bean是不是实现了ApplicationContextAware接口。

继续向下看prepareBeanFactory方法，可以看到现在是来为BeanFactory设置忽略的自动装配的接口，比如说像EnvironmentAware、EmbeddedValueResolverAware等等这些接口。



```
627- protected void prepareBeanFactory(ConfigurableListableBeanFactory beanFactory) {
628-     // Tell the internal bean factory to use the context's class loader etc.
629-     beanFactory.setBeanClassLoader(getClassLoader());
630-     beanFactory.setBeanExpressionResolver(new StandardBeanExpressionResolver(beanFactory.getBeanClassLoader()));
631-     beanFactory.addPropertyEditorRegistrar(new ResourceEditorRegistrar(this, getEnvironment()));
632-
633-     // Configure the bean factory with context callbacks.
634-     beanFactory.addBeanPostProcessor(new ApplicationContextAwareProcessor(this));
635-     beanFactory.ignoreDependencyInterface(EnvironmentAware.class);
636-     beanFactory.ignoreDependencyInterface(EmbeddedValueResolverAware.class);
637-     beanFactory.ignoreDependencyInterface(ResourceLoaderAware.class);
638-     beanFactory.ignoreDependencyInterface(ApplicationEventPublisherAware.class);
639-     beanFactory.ignoreDependencyInterface(MessageSourceAware.class);
640-     beanFactory.ignoreDependencyInterface(ApplicationContextAware.class);
641-
642-     // BeanFactory interface not registered as resolvable type in a plain factory.
643-     // MessageSource registered (and found for autowiring) as a bean.
644-     beanFactory.registerResolvableDependency(BeanFactory.class, beanFactory);
645-     beanFactory.registerResolvableDependency(ResourceLoader.class, this);
646-     beanFactory.registerResolvableDependency(ApplicationEventPublisher.class, this);
647-     beanFactory.registerResolvableDependency(ApplicationContext.class, this);
```

那么，设置忽略的自动装配的这些接口有什么作用呢？作用就是，这些接口的实现类不能通过接口类型来自动注入。

继续向下看prepareBeanFactory方法，可以看到现在是来为BeanFactory注册可以解析的自动装配。

```
IOCTest_Ext.java AnnotationConfigApplicationContext.class AbstractApplicationContext.class GenericApplicationContext.class
633 // Configure the bean factory with context callbacks.
634 beanFactory.addBeanPostProcessor(new ApplicationContextAwareProcessor(this));
635 beanFactory.ignoreDependencyInterface(EnvironmentAware.class);
636 beanFactory.ignoreDependencyInterface(EmbeddedValueResolverAware.class);
637 beanFactory.ignoreDependencyInterface(ResourceLoaderAware.class);
638 beanFactory.ignoreDependencyInterface(ApplicationEventPublisherAware.class);
639 beanFactory.ignoreDependencyInterface(MessageSourceAware.class);
640 beanFactory.ignoreDependencyInterface(ApplicationContextAware.class);
641
642 // BeanFactory interface not registered as resolvable type in a plain factory.
643 // MessageSource registered (and found for autowiring) as a bean.
644 beanFactory.registerResolvableDependency(BeansFactory.class, beanFactory);
645 beanFactory.registerResolvableDependency(ResourceLoader.class, this);
646 beanFactory.registerResolvableDependency(ApplicationEventPublisher.class, this);
647 beanFactory.registerResolvableDependency(ApplicationContext.class, this);
648
649 // Register early post-processor for detecting inner beans as ApplicationListeners.
650 beanFactory.addBeanPostProcessor(new ApplicationListenerDetector(this));
651
652 // Detect a LoadTimeWeaver and prepare for weaving, if found.
653 if (beanFactory.containsBean(LOAD_TIME_WEAVER_BEAN_NAME)) {
```

所谓的可以解析的自动装配，就是说，我们可以直接在任何组件里面自动注入像BeanFactory、ResourceLoader、ApplicationEventPublisher（它就是上一讲我们讲述的事件派发器）以及ApplicationContext（也就是我们的IOC容器）这些东西。

继续向下看prepareBeanFactory方法，可以看到现在又向BeanFactory中添加了一个BeanPostProcessor，只不过现在添加的是一个叫ApplicationListenerDetector的BeanPostProcessor。

```
IOCTest_Ext.java AnnotationConfigApplicationContext.class AbstractApplicationContext.class GenericApplicationContext.class
642 // BeanFactory interface not registered as resolvable type in a plain factory.
643 // MessageSource registered (and found for autowiring) as a bean.
644 beanFactory.registerResolvableDependency(BeansFactory.class, beanFactory);
645 beanFactory.registerResolvableDependency(ResourceLoader.class, this);
646 beanFactory.registerResolvableDependency(ApplicationEventPublisher.class, this);
647 beanFactory.registerResolvableDependency(ApplicationContext.class, this);
648
649 // Register early post-processor for detecting inner beans as ApplicationListeners.
650 beanFactory.addBeanPostProcessor(new ApplicationListenerDetector(this));
651
652 // Detect a LoadTimeWeaver and prepare for weaving, if found.
653 if (beanFactory.containsBean(LOAD_TIME_WEAVER_BEAN_NAME)) {
654     beanFactory.addBeanPostProcessor(new LoadTimeWeaverAwareProcessor(beanFactory));
655     // Set a temporary ClassLoader for type matching.
656     beanFactory.setTempClassLoader(new ContextTypeMatchClassLoader(beanFactory.getBeanClassLoader()));
657 }
658
659 // Register default environment beans.
660 if (!beanFactory.containsLocalBean(ENVIRONMENT_BEAN_NAME)) {
661     beanFactory.registerSingleton(ENVIRONMENT_BEAN_NAME, getEnvironment());
662 }
```

添加对事件进行监听的处理器

也就是说，会向BeanFactory中添加很多的后置处理器，后置处理器的作用就是在bean初始化前后做一些工作。

继续向下看prepareBeanFactory方法，可以看到有一个if判断语句，它这是向BeanFactory中添加编译时与AspectJ支持相关的东西。

```
IOCTest_Ext.java AnnotationConfigApplicationContext.class AbstractApplicationContext.class GenericApplicationContext.class
651
652 // Detect a LoadTimeWeaver and prepare for weaving, if found.
653 if (beanFactory.containsBean(LOAD_TIME_WEAVER_BEAN_NAME)) {
654     beanFactory.addBeanPostProcessor(new LoadTimeWeaverAwareProcessor(beanFactory));
655     // Set a temporary ClassLoader for type matching.
656     beanFactory.setTempClassLoader(new ContextTypeMatchClassLoader(beanFactory.getBeanClassLoader()));
657 }
658
659 // Register default environment beans.
660 if (!beanFactory.containsLocalBean(ENVIRONMENT_BEAN_NAME)) {
661     beanFactory.registerSingleton(ENVIRONMENT_BEAN_NAME, getEnvironment());
662 }
663 if (!beanFactory.containsLocalBean(SYSTEM_PROPERTIES_BEAN_NAME)) {
664     beanFactory.registerSingleton(SYSTEM_PROPERTIES_BEAN_NAME, getEnvironment().getSystemProperties());
665 }
666 if (!beanFactory.containsLocalBean(SYSTEM_ENVIRONMENT_BEAN_NAME)) {
667     beanFactory.registerSingleton(SYSTEM_ENVIRONMENT_BEAN_NAME, getEnvironment().getSystemEnvironment());
668 }
669 }
670
671 /**
```

添加编译时
与AspectJ
支持相关的
东西

而我们现在默认的都是运行时的动态代理，所以你会看到这样一个现象，按下F6快捷键让程序往下运行，程序并不会进入到以上if判断语句中，而是来到了下面这个if判断语句句处。


```
IOCTest_Ext.java AnnotationConfigApplicationContext.class AbstractApplicationContext.class GenericApplicationContext.class
651
652 // Detect a LoadTimeWeaver and prepare for weaving, if found.
653 if (beanFactory.containsBean(LOAD_TIME_WEAVER_BEAN_NAME)) {
654     beanFactory.addBeanPostProcessor(new LoadTimeWeaverAwareProcessor(beanFactory));
655     // Set a temporary ClassLoader for type matching.
656     beanFactory.setTempClassLoader(new ContextTypeMatchClassLoader(beanFactory.getBeanClassLoader()));
657 }
658
659 // Register default environment beans.
660 if (!beanFactory.containsLocalBean(ENVIRONMENT_BEAN_NAME)) {
661     beanFactory.registerSingleton(ENVIRONMENT_BEAN_NAME, getEnvironment());
662 }
663 if (!beanFactory.containsLocalBean(SYSTEM_PROPERTIES_BEAN_NAME)) {
664     beanFactory.registerSingleton(SYSTEM_PROPERTIES_BEAN_NAME, getEnvironment().getSystemProperties());
665 }
666 if (!beanFactory.containsLocalBean(SYSTEM_ENVIRONMENT_BEAN_NAME)) {
667     beanFactory.registerSingleton(SYSTEM_ENVIRONMENT_BEAN_NAME, getEnvironment().getSystemEnvironment());
668 }
669 }
670
671 /**
672
```

这儿是在向BeanFactory中注册一些与环境变量相关的bean，比如注册了一个名字是environment，值是当前环境对象（其类型是ConfigurableEnvironment）的bean。

```
IOCTest_Ext.java AnnotationConfigApplicationContext.class AbstractApplicationContext.class GenericApplicationContext.class
652 // Detect a LoadTimeWeaver and prepare for weaving, if found.
653 if (beanFactory.containsBean(LOAD_TIME_WEAVER_BEAN_NAME)) {
654     beanFactory.addBeanPostProcessor(new LoadTimeWeaverAwareProcessor(beanFactory));
655     // Set a temporary ClassLoader for type matching.
656     beanFactory.setTempClassLoader(new ContextTypeMatchClassLoader(beanFactory.getBeanClassLoader()));
657 }
658
659 // Register default environment beans.
660 if (!beanFactory.containsLocalBean(ENVIRONMENT_BEAN_NAME)) {
661     beanFactory.registerSingleton(ENVIRONMENT_BEAN_NAME, getEnvironment());
662 }
663 if (!beanFactory.containsLocalBean(SYSTEM_PROPERTIES_BEAN_NAME)) {
664     beanFactory.registerSingleton(SYSTEM_PROPERTIES_BEAN_NAME, getEnvironment().getSystemProperties());
665 }
666 if (!beanFactory.containsLocalBean(SYSTEM_ENVIRONMENT_BEAN_NAME)) {
667     beanFactory.registerSingleton(SYSTEM_ENVIRONMENT_BEAN_NAME, getEnvironment().getSystemEnvironment());
668 }
669 }
670
671 /**
672 * Modify the application context's internal bean factory after its standard

```

注册当前的环境对象

除此之外，还注册了一个名字为systemProperties的bean，也即系统属性，它是通过当前环境对象的getSystemProperties方法获得的。我们来看一下系统属性是个什么东西，进入getSystemProperties方法里面，如下图所示，可以看到系统属性就是一个Map<String, Object>，该Map里面的key/value就是环境变量里面的key/value。

```
IOCTest_Ext.java AnnotationConfigApplicationContext.class AbstractApplicationContext.class AbstractEnvironment.class
419 @Override
420 @SuppressWarnings({"unchecked", "rawtypes"})
421 public Map<String, Object> getSystemProperties() {
422     try {
423         return (Map) System.getProperties();
424     }
425     catch (AccessControlException ex) {
426         return (Map) new ReadOnlySystemAttributesMap() {
427             @Override
428             protected String getSystemAttribute(String attributeName) {
429                 try {
430                     return System.getProperty(attributeName);
431                 }
432                 catch (AccessControlException ex) {
433                     if (logger.isInfoEnabled()) {
434                         logger.info("Caught AccessControlException when accessing system property '" +
435                             attributeName + "': its value will be returned [null]. Reason: " + ex.getMessage());
436                     }
437                     return null;
438                 }
439             }
440         };
441     }
442 }
```

最后，还会注册一个名字为systemEnvironment的bean，即系统的整个环境信息。我们也不妨点进去getSystemEnvironment方法里面去看一下，如下图所示，发现系统的整个环境信息也是一个Map<String, Object>。


```
376- @Override
377- @SuppressWarnings({"unchecked", "rawtypes"})
378- public Map<String, Object> getSystemEnvironment() {
379-     if (suppressGetenvAccess()) {
380-         return Collections.emptyMap();
381-     }
382-     try {
383-         return (Map) System.getenv();
384-     }
385-     catch (AccessControlException ex) {
386-         return (Map) new ReadOnlySystemAttributesMap() {
387-             @Override
388-             protected String getSystemAttribute(String attributeName) {
389-                 try {
390-                     return System.getenv(attributeName);
391-                 }
392-                 catch (AccessControlException ex) {
393-                     if (logger.isInfoEnabled()) {
394-                         logger.info("Caught AccessControlException when accessing system environment variable '" +
395-                             attributeName + "'; its value will be returned [null]. Reason: " + ex.getMessage());
396-                     }
397-                 }
398-             }
399-         };
400-     }
401- }
```

也就是说，我们向BeanFactory中注册了以上三个与环境变量相关的bean。以后，如果我们想用的话，只须将它们自动注入即可。

继续按下 **F6** 快捷键让程序往下运行，一直让程序运行到下面这行代码处。程序运行至此，说明prepareBeanFactory方法就执行完了，相应地，BeanFactory就已经创建好了，里面该设置的属性也都设置了。

```
508- @Override
509- public void refresh() throws BeansException, IllegalStateException {
510-     synchronized (this.startupShutdownMonitor) {
511-         // Prepare this context for refreshing.
512-         prepareRefresh();
513-
514-         // Tell the subclass to refresh the internal bean factory.
515-         ConfigurableListableBeanFactory beanFactory = obtainFreshBeanFactory();
516-
517-         // Prepare the bean factory for use in this context.
518-         prepareBeanFactory(beanFactory);
519-
520-         try {
521-             // Allows post-processing of the bean factory in context subclasses.
522-             postProcessBeanFactory(beanFactory);
523-
524-             // Invoke factory processors registered as beans in the context.
525-             invokeBeanFactoryPostProcessors(beanFactory);
526-
527-             // Register bean processors that intercept bean creation.
528-             registerBeanPostProcessors(beanFactory);
529-         }
530-         catch (BeansException ex) {
531-             // ...
532-         }
533-     }
534- }
```

postProcessBeanFactory(beanFactory): BeanFactory准备工作完成后进行的后置处理工作

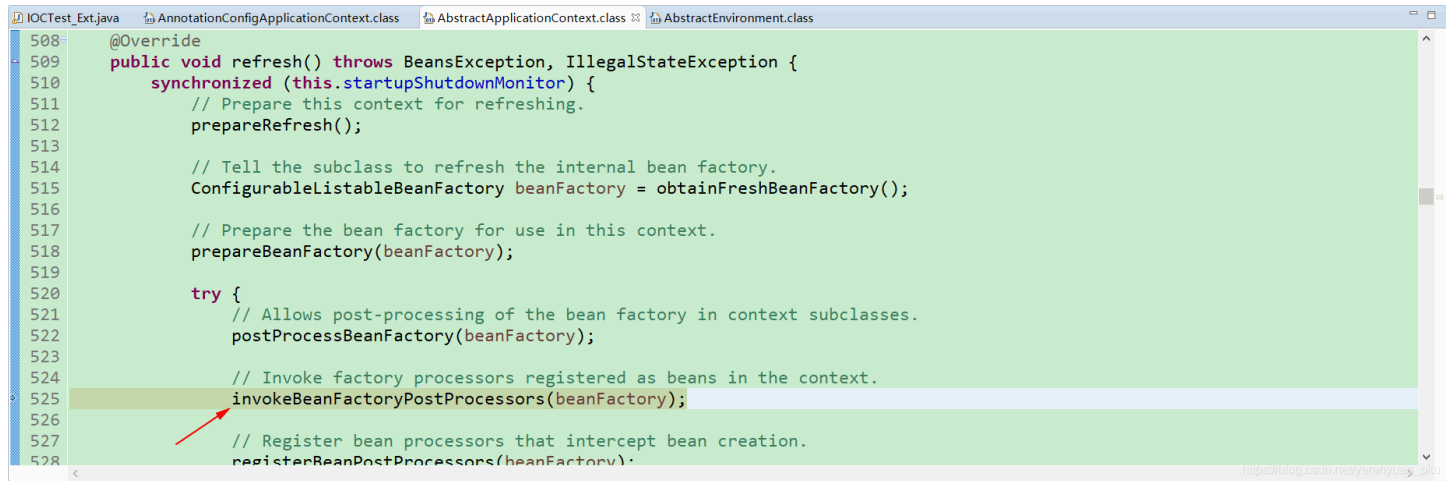
接下来，我们就得来说说道postProcessBeanFactory方法了。它说的就是在BeanFactory准备工作完成之后进行的后置处理工作。我们不妨点进去该方法里面看看，它究竟做了哪些事，如下图所示，发现它里面是空的。

```
668- }
669- }
670-
671- /**
672-  * Modify the application context's internal bean factory after its standard
673-  * initialization. All bean definitions will have been loaded, but no beans
674-  * will have been instantiated yet. This allows for registering special
675-  * BeanPostProcessors etc in certain ApplicationContext implementations.
676-  * @param beanFactory the bean factory used by the application context
677-  */
678- protected void postProcessBeanFactory(ConfigurableListableBeanFactory beanFactory) {
679- }
680-
681- /**
682-  * Instantiate and invoke all registered BeanFactoryPostProcessor beans,
683-  * respecting explicit order if given.
684-  * <p>Must be called before singleton instantiation.
685-  */
686- protected void invokeBeanFactoryPostProcessors(ConfigurableListableBeanFactory beanFactory) {
687-     PostProcessorRegistrationDelegate.invokeBeanFactoryPostProcessors(beanFactory, getBeanFactoryPostProcessors());
688- }
```

这不是和我们刷新容器前的预处理工作中的initPropertySources方法一样吗？方法里面都是空的，默认都是不进行任何处理的，但是方法都是protected类型的，这也就是说子类可以通过重写这个方法，在BeanFactory创建并预处理完成以后做进一步的设置。

这个方法只有在子类重写的时候有用，只不过现在它还是空的，里面啥也没做。

继续按下 **F6** 快捷键让程序往下运行，一直让程序运行到下面这行代码处。程序运行到这里之后，我们先让它停一停。



```
508- @Override
509- public void refresh() throws BeansException, IllegalStateException {
510-     synchronized (this.startupShutdownMonitor) {
511-         // Prepare this context for refreshing.
512-         prepareRefresh();
513-
514-         // Tell the subclass to refresh the internal bean factory.
515-         ConfigurableListableBeanFactory beanFactory = obtainFreshBeanFactory();
516-
517-         // Prepare the bean factory for use in this context.
518-         prepareBeanFactory(beanFactory);
519-
520-         try {
521-             // Allows post-processing of the bean factory in context subclasses.
522-             postProcessBeanFactory(beanFactory);
523-
524-             // Invoke factory processors registered as beans in the context.
525-             invokeBeanFactoryPostProcessors(beanFactory);
526-
527-             // Register bean processors that intercept bean creation.
528-             registerBeanPostProcessors(beanFactory);
```

至此，BeanFactory的创建以及预准备工作就已经完成啦😄

既然有了BeanFactory对象，那么接下来我们就要利用BeanFactory来创建各种组件了。在下一讲，我们就来看看后续的流程。