芋道源码 —— 知识星球

2019-09-29
Spring

【死磕 Spring】—— 深入分析 ApplicationContext 的 refresh()

本文主要基于 Spring 5.0.6.RELEASE

摘要: 原创出处 http://cmsblogs.com/?p=todo 「小明哥」，谢谢！

作为「小明哥」的忠实读者，「老艿艿」略作修改，记录在理解过程中，参考的资料。

上篇博客只是对 ApplicationContext 相关的接口做了一个简单的介绍，作为一个高富帅级别的 Spring 容器，它涉及的方法实在是太多了，全部介绍是不可能的，而且大部分功能都已经在前面系列博客中做了详细的介绍，所以这篇博问介绍 ApplicationContext 最重要的方法（小编认为的）：#refresh() 方法。

> 艿艿：我也这么认为，#refresh() 方法是关键的关键！

#refresh() 方法，是定义在 ConfigurableApplicationContext 类中的，如下：

```
// ConfigurableApplicationContext.java

/**
 * Load or refresh the persistent representation of the configuration,
 * which might an XML file, properties file, or relational database schema.
 * As this is a startup method, it should destroy already created singletons
 * if it fails, to avoid dangling resources. In other words, after invocation
 * of that method, either all or no singletons at all should be instantiated.
 * @throws BeansException if the bean factory could not be initialized
 * @throws IllegalStateException if already initialized and multiple refresh
 * attempts are not supported
 */
void refresh() throws BeansException, IllegalStateException;
```

> 作用就是：刷新 Spring 的应用上下文。

其实现是在 AbstractApplicationContext 中实现。如下：

```
// AbstractApplicationContext.java
```

```java
@Override
public void refresh() throws BeansException, IllegalStateException {
 synchronized (this.startupShutdownMonitor) {
     // 准备刷新上下文环境
        prepareRefresh();

     // 创建并初始化 BeanFactory
        ConfigurableListableBeanFactory beanFactory = obtainFreshBeanFactory();

     // 填充BeanFactory功能
        prepareBeanFactory(beanFactory);

     try {
         // 提供子类覆盖的额外处理，即子类处理自定义的BeanFactoryPostProcess
            postProcessBeanFactory(beanFactory);

         // 激活各种BeanFactory处理器
            invokeBeanFactoryPostProcessors(beanFactory);

         // 注册拦截Bean创建的Bean处理器，即注册 BeanPostProcessor
            registerBeanPostProcessors(beanFactory);

         // 初始化上下文中的资源文件，如国际化文件的处理等
            initMessageSource();

         // 初始化上下文事件广播器
            initApplicationEventMulticaster();

         // 给子类扩展初始化其他Bean
            onRefresh();

         // 在所有bean中查找listener bean，然后注册到广播器中
            registerListeners();

         // 初始化剩下的单例Bean(非延迟加载的)
            finishBeanFactoryInitialization(beanFactory);

         // 完成刷新过程,通知生命周期处理器lifecycleProcessor刷新过程,同时发出ContextRefreshEvent通知别人
            finishRefresh();
        } catch (BeansException ex) {
         if (logger.isWarnEnabled()) {
                logger.warn("Exception encountered during context initialization - " +
                     "cancelling refresh attempt: " + ex);
            }

         //  销毁已经创建的Bean
            destroyBeans();

         // 重置容器激活标签
            cancelRefresh(ex);

         // 抛出异常
         throw ex;
        } finally {
         // Reset common introspection caches in Spring's core, since we
         // might not ever need metadata for singleton beans anymore...
            resetCommonCaches();
        }
    }
}
```

这里每一个方法都非常重要，需要一个一个地解释说明。

1. prepareRefresh()

初始化上下文环境，对系统的环境变量或者系统属性进行准备和校验,如环境变量中必须设置某个值才能运行，否则不能运行，这个时候可以在这里加这个校验，重写 initPropertySources 方法就好了

```
// AbstractApplicationContext.java

protected void prepareRefresh() {
   // 设置启动日期
 this.startupDate = System.currentTimeMillis();
 // 设置 context 当前状态
 this.closed.set(false);
 this.active.set(true);

 if (logger.isInfoEnabled()) {
        logger.info("Refreshing " + this);
    }

 // 初始化context environment（上下文环境）中的占位符属性来源
    initPropertySources();

 // 对属性进行必要的验证
    getEnvironment().validateRequiredProperties();

 this.earlyApplicationEvents = new LinkedHashSet<>();
}
```

该方法主要是做一些准备工作，如：

1. 设置 context 启动时间
2. 设置 context 的当前状态
3. 初始化 context environment 中占位符
4. 对属性进行必要的验证

2. obtainFreshBeanFactory()

创建并初始化 BeanFactory

```
// AbstractApplicationContext.java

protected ConfigurableListableBeanFactory obtainFreshBeanFactory() {
   // 刷新 BeanFactory
    refreshBeanFactory();
 // 获取 BeanFactory
    ConfigurableListableBeanFactory beanFactory = getBeanFactory();
 if (logger.isDebugEnabled()) {
```

```
            logger.debug("Bean factory for " + getDisplayName() + ": " + beanFactory);
        }
     return beanFactory;
    }
```

核心方法就在 #refreshBeanFactory() 方法，该方法的核心任务就是创建 BeanFactory 并对其就行一番初始化。如下：

```
// AbstractRefreshableApplicationContext.java

@Override
protected final void refreshBeanFactory() throws BeansException {
    // 若已有 BeanFactory ，销毁它的 Bean 们，并销毁 BeanFactory
    if (hasBeanFactory()) {
        destroyBeans();
        closeBeanFactory();
    }
    try {
        // 创建 BeanFactory 对象
        DefaultListableBeanFactory beanFactory = createBeanFactory();
        // 指定序列化编号
        beanFactory.setSerializationId(getId());
        // 定制 BeanFactory 设置相关属性
        customizeBeanFactory(beanFactory);
        // 加载 BeanDefinition 们
        loadBeanDefinitions(beanFactory);
        // 设置 Context 的 BeanFactory
        synchronized (this.beanFactoryMonitor) {
            this.beanFactory = beanFactory;
        }
    } catch (IOException ex) {
        throw new ApplicationContextException("I/O error parsing bean definition source for " + getDisplayName(), ex)
    }
}
```

1. 判断当前容器是否存在一个 BeanFactory，如果存在则对其进行销毁和关闭
2. 调用 #createBeanFactory() 方法，创建一个 BeanFactory 实例，其实就是 DefaultListableBeanFactory 。
3. 自定义 BeanFactory
4. 加载 BeanDefinition 。
5. 将创建好的 bean 工厂的引用交给的 context 来管理

上面 5 个步骤，都是比较简单的，但是有必要讲解下第 4 步：加载 BeanDefinition。如果各位看过 【死磕 Spring】系列的话，在刚刚开始分析源码的时候，小编就是以 BeanDefinitionReader#loadBeanDefinitions(Resource resource) 方法，作为入口来分析的，示例如下：

```
// 示例代码

ClassPathResource resource = new ClassPathResource("bean.xml");
DefaultListableBeanFactory factory = new DefaultListableBeanFactory();
XmlBeanDefinitionReader reader = new XmlBeanDefinitionReader(factory);
reader.loadBeanDefinitions(resource);
```

只不过这段代码的 BeanDefinitionReader#loadBeanDefinitions(Resource) 方法，是定义在

BeanDefinitionReader 中，而此处的 #loadBeanDefinitions(DefaultListableBeanFactory beanFactory) 则是定义在 AbstractRefreshableApplicationContext 中，如下：

```
// AbstractRefreshableApplicationContext.java

protected abstract void loadBeanDefinitions(DefaultListableBeanFactory beanFactory) throws BeansException, IOExceptio
```

由具体的子类实现，我们以 AbstractXmlApplicationContext 为例，实现如下：

```
// AbstractXmlApplicationContext.java

@Override
protected void loadBeanDefinitions(DefaultListableBeanFactory beanFactory) throws BeansException, IOException {
    // Create a new XmlBeanDefinitionReader for the given BeanFactory.
    // 创建 XmlBeanDefinitionReader 对象
    XmlBeanDefinitionReader beanDefinitionReader = new XmlBeanDefinitionReader(beanFactory);

    // Configure the bean definition reader with this context's
    // resource loading environment.
    // 对 XmlBeanDefinitionReader 进行环境变量的设置
    beanDefinitionReader.setEnvironment(this.getEnvironment());
    beanDefinitionReader.setResourceLoader(this);
    beanDefinitionReader.setEntityResolver(new ResourceEntityResolver(this));

    // Allow a subclass to provide custom initialization of the reader,
    // then proceed with actually loading the bean definitions.
    // 对 XmlBeanDefinitionReader 进行设置，可以进行覆盖
    initBeanDefinitionReader(beanDefinitionReader);

    // 从 Resource 们中，加载 BeanDefinition 们
    loadBeanDefinitions(beanDefinitionReader);
}
```

新建 XmlBeanDefinitionReader 实例对象 beanDefinitionReader，调用 initBeanDefinitionReader() 对其进行初始化，然后调用 loadBeanDefinitions() 加载 BeanDefinition。代码如下：

```
// AbstractXmlApplicationContext.java

protected void loadBeanDefinitions(XmlBeanDefinitionReader reader) throws BeansException, IOException {
    // 从配置文件 Resource 中，加载 BeanDefinition 们
    Resource[] configResources = getConfigResources();
    if (configResources != null) {
        reader.loadBeanDefinitions(configResources);
    }
    // 从配置文件地址中，加载 BeanDefinition 们
    String[] configLocations = getConfigLocations();
    if (configLocations != null) {
        reader.loadBeanDefinitions(configLocations);
    }
}
```

- 到这里我们发现，其实内部依然是调用 BeanDefinitionReader#loadBeanDefinitionn() 进行 BeanDefinition 的加载进程。

3. prepareBeanFactory(beanFactory)

填充 BeanFactory 功能

上面获取获取的 BeanFactory 除了加载了一些 BeanDefinition 就没有其他任何东西了，这个时候其实还不能投入生产，因为还少配置了一些东西，比如 context的 ClassLoader 和 后置处理器等等。

```
// AbstractApplicationContext.java

protected void prepareBeanFactory(ConfigurableListableBeanFactory beanFactory) {
 // 设置beanFactory的classLoader
    beanFactory.setBeanClassLoader(getClassLoader());

 // 设置beanFactory的表达式语言处理器,Spring3开始增加了对语言表达式的支持,默认可以使用#{bean.xxx}的形式来调用相关属性
    beanFactory.setBeanExpressionResolver(new StandardBeanExpressionResolver(beanFactory.getBeanClassLoader()));
 // 为beanFactory增加一个默认的propertyEditor
    beanFactory.addPropertyEditorRegistrar(new ResourceEditorRegistrar(this, getEnvironment()));

 // 添加ApplicationContextAwareProcessor
    beanFactory.addBeanPostProcessor(new ApplicationContextAwareProcessor(this));
 // 设置忽略自动装配的接口
    beanFactory.ignoreDependencyInterface(EnvironmentAware.class);
    beanFactory.ignoreDependencyInterface(EmbeddedValueResolverAware.class);
    beanFactory.ignoreDependencyInterface(ResourceLoaderAware.class);
    beanFactory.ignoreDependencyInterface(ApplicationEventPublisherAware.class);
    beanFactory.ignoreDependencyInterface(MessageSourceAware.class);
    beanFactory.ignoreDependencyInterface(ApplicationContextAware.class);

 // 设置几个自动装配的特殊规则
    beanFactory.registerResolvableDependency(BeanFactory.class, beanFactory);
    beanFactory.registerResolvableDependency(ResourceLoader.class, this);
    beanFactory.registerResolvableDependency(ApplicationEventPublisher.class, this);
    beanFactory.registerResolvableDependency(ApplicationContext.class, this);

 // Register early post-processor for detecting inner beans as ApplicationListeners.
    beanFactory.addBeanPostProcessor(new ApplicationListenerDetector(this));

 // 增加对AspectJ的支持
 if (beanFactory.containsBean(LOAD_TIME_WEAVER_BEAN_NAME)) {
        beanFactory.addBeanPostProcessor(new LoadTimeWeaverAwareProcessor(beanFactory));
     // Set a temporary ClassLoader for type matching.
        beanFactory.setTempClassLoader(new ContextTypeMatchClassLoader(beanFactory.getBeanClassLoader()));
    }

 // 注册默认的系统环境bean
 if (!beanFactory.containsLocalBean(ENVIRONMENT_BEAN_NAME)) {
        beanFactory.registerSingleton(ENVIRONMENT_BEAN_NAME, getEnvironment());
    }
 if (!beanFactory.containsLocalBean(SYSTEM_PROPERTIES_BEAN_NAME)) {
        beanFactory.registerSingleton(SYSTEM_PROPERTIES_BEAN_NAME, getEnvironment().getSystemProperties());
    }
 if (!beanFactory.containsLocalBean(SYSTEM_ENVIRONMENT_BEAN_NAME)) {
        beanFactory.registerSingleton(SYSTEM_ENVIRONMENT_BEAN_NAME, getEnvironment().getSystemEnvironment());
    }
}
```

看上面的源码知道这个就是对 BeanFactory 设置各种各种的功能。

4. postProcessBeanFactory()

提供子类覆盖的额外处理，即子类处理自定义的BeanFactoryPostProcess

```
// AbstractApplicationContext.java

protected void postProcessBeanFactory(ConfigurableListableBeanFactory beanFactory) {
    beanFactory.addBeanPostProcessor(new ServletContextAwareProcessor(this.servletContext, this.servletConfig));
    beanFactory.ignoreDependencyInterface(ServletContextAware.class);
    beanFactory.ignoreDependencyInterface(ServletConfigAware.class);

    WebApplicationContextUtils.registerWebApplicationScopes(beanFactory, this.servletContext);
    WebApplicationContextUtils.registerEnvironmentBeans(beanFactory, this.servletContext, this.servletConfig);
}
```

1. 添加 ServletContextAwareProcessor 到 BeanFactory 容器中，该 processor 实现 BeanPostProcessor 接口，主要用于将ServletContext 传递给实现了 ServletContextAware 接口的 bean
2. 忽略 ServletContextAware、ServletConfigAware
3. 注册 WEB 应用特定的域（scope）到 beanFactory 中，以便 WebApplicationContext 可以使用它们。比如 “request” ， “session” ， “globalSession” ， “application”
4. 注册 WEB 应用特定的 Environment bean 到 beanFactory 中，以便WebApplicationContext 可以使用它们。如：”contextParameters”， “contextAttributes”

5. invokeBeanFactoryPostProcessors()

激活各种BeanFactory处理器

```
// AbstractApplicationContext.java

public static void invokeBeanFactoryPostProcessors(
        ConfigurableListableBeanFactory beanFactory, List<BeanFactoryPostProcessor> beanFactoryPostProcessors) {

 // 定义一个 set 保存所有的 BeanFactoryPostProcessors
    Set<String> processedBeans = new HashSet<>();

 // 如果当前 BeanFactory 为 BeanDefinitionRegistry
 if (beanFactory instanceof BeanDefinitionRegistry) {
        BeanDefinitionRegistry registry = (BeanDefinitionRegistry) beanFactory;
     // BeanFactoryPostProcessor 集合
        List<BeanFactoryPostProcessor> regularPostProcessors = new ArrayList<>();
     // BeanDefinitionRegistryPostProcessor 集合
        List<BeanDefinitionRegistryPostProcessor> registryProcessors = new ArrayList<>();

     // 迭代注册的 beanFactoryPostProcessors
     for (BeanFactoryPostProcessor postProcessor : beanFactoryPostProcessors) {
         // 如果是 BeanDefinitionRegistryPostProcessor，则调用 postProcessBeanDefinitionRegistry 进行注册，
         // 同时加入到 registryProcessors 集合中
         if (postProcessor instanceof BeanDefinitionRegistryPostProcessor) {
                BeanDefinitionRegistryPostProcessor registryProcessor =
                        (BeanDefinitionRegistryPostProcessor) postProcessor;
```

```
           registryProcessor.postProcessBeanDefinitionRegistry(registry);
           registryProcessors.add(registryProcessor);
       }
    else {
        // 否则当做普通的 BeanFactoryPostProcessor 处理
        // 添加到 regularPostProcessors 集合中即可，便于后面做后续处理
           regularPostProcessors.add(postProcessor);
       }
   }

// 用于保存当前处理的 BeanDefinitionRegistryPostProcessor
   List<BeanDefinitionRegistryPostProcessor> currentRegistryProcessors = new ArrayList<>();

// 首先处理实现了 PriorityOrdered (有限排序接口)的 BeanDefinitionRegistryPostProcessor
   String[] postProcessorNames =
           beanFactory.getBeanNamesForType(BeanDefinitionRegistryPostProcessor.class, true, false);
for (String ppName : postProcessorNames) {
    if (beanFactory.isTypeMatch(ppName, PriorityOrdered.class)) {
           currentRegistryProcessors.add(beanFactory.getBean(ppName, BeanDefinitionRegistryPostProcessor.class))
           processedBeans.add(ppName);
       }
   }

// 排序
   sortPostProcessors(currentRegistryProcessors, beanFactory);

// 加入registryProcessors集合
   registryProcessors.addAll(currentRegistryProcessors);

// 调用所有实现了 PriorityOrdered 的 BeanDefinitionRegistryPostProcessors 的 postProcessBeanDefinitionRegistry()
   invokeBeanDefinitionRegistryPostProcessors(currentRegistryProcessors, registry);

// 清空，以备下次使用
   currentRegistryProcessors.clear();

// 其次，调用是实现了 Ordered（普通排序接口）的 BeanDefinitionRegistryPostProcessors
// 逻辑和 上面一样
   postProcessorNames = beanFactory.getBeanNamesForType(BeanDefinitionRegistryPostProcessor.class, true, false);
for (String ppName : postProcessorNames) {
    if (!processedBeans.contains(ppName) && beanFactory.isTypeMatch(ppName, Ordered.class)) {
           currentRegistryProcessors.add(beanFactory.getBean(ppName, BeanDefinitionRegistryPostProcessor.class))
           processedBeans.add(ppName);
       }
   }
   sortPostProcessors(currentRegistryProcessors, beanFactory);
   registryProcessors.addAll(currentRegistryProcessors);
   invokeBeanDefinitionRegistryPostProcessors(currentRegistryProcessors, registry);
   currentRegistryProcessors.clear();

// 最后调用其他的 BeanDefinitionRegistryPostProcessors
boolean reiterate = true;
while (reiterate) {
       reiterate = false;
    // 获取 BeanDefinitionRegistryPostProcessor
       postProcessorNames = beanFactory.getBeanNamesForType(BeanDefinitionRegistryPostProcessor.class, true, fal
    for (String ppName : postProcessorNames) {

        // 没有包含在 processedBeans 中的（因为包含了的都已经处理了）
        if (!processedBeans.contains(ppName)) {
               currentRegistryProcessors.add(beanFactory.getBean(ppName, BeanDefinitionRegistryPostProcessor.cla
                   processedBeans.add(ppName);
                   reiterate = true;
               }
           }

        // 与上面处理逻辑一致
           sortPostProcessors(currentRegistryProcessors, beanFactory);
           registryProcessors.addAll(currentRegistryProcessors);
           invokeBeanDefinitionRegistryPostProcessors(currentRegistryProcessors, registry);
           currentRegistryProcessors.clear();
       }

    // 调用所有 BeanDefinitionRegistryPostProcessor (包括手动注册和通过配置文件注册)
    // 和 BeanFactoryPostProcessor(只有手动注册)的回调函数(postProcessBeanFactory())
       invokeBeanFactoryPostProcessors(registryProcessors, beanFactory);
       invokeBeanFactoryPostProcessors(regularPostProcessors, beanFactory);
   }

else {
    // 如果不是 BeanDefinitionRegistry 只需要调用其回调函数（postProcessBeanFactory()）即可
       invokeBeanFactoryPostProcessors(beanFactoryPostProcessors, beanFactory);
   }

//
   String[] postProcessorNames =
           beanFactory.getBeanNamesForType(BeanFactoryPostProcessor.class, true, false);

// 这里同样需要区分 PriorityOrdered 、Ordered 和 no Ordered
   List<BeanFactoryPostProcessor> priorityOrderedPostProcessors = new ArrayList<>();
   List<String> orderedPostProcessorNames = new ArrayList<>();
   List<String> nonOrderedPostProcessorNames = new ArrayList<>();
for (String ppName : postProcessorNames) {
    // 已经处理过了的，跳过
    if (processedBeans.contains(ppName)) {
        // skip - already processed in first phase above
       }
    // PriorityOrdered
    else if (beanFactory.isTypeMatch(ppName, PriorityOrdered.class)) {
           priorityOrderedPostProcessors.add(beanFactory.getBean(ppName, BeanFactoryPostProcessor.class));
       }
    // Ordered
    else if (beanFactory.isTypeMatch(ppName, Ordered.class)) {
           orderedPostProcessorNames.add(ppName);
       }
    // no Ordered
    else {
           nonOrderedPostProcessorNames.add(ppName);
       }
   }

// First, PriorityOrdered 接口
   sortPostProcessors(priorityOrderedPostProcessors, beanFactory);
   invokeBeanFactoryPostProcessors(priorityOrderedPostProcessors, beanFactory);

// Next, Ordered 接口
   List<BeanFactoryPostProcessor> orderedPostProcessors = new ArrayList<>();
for (String postProcessorName : orderedPostProcessorNames) {
       orderedPostProcessors.add(beanFactory.getBean(postProcessorName, BeanFactoryPostProcessor.class));
   }
   sortPostProcessors(orderedPostProcessors, beanFactory);
```

```
    invokeBeanFactoryPostProcessors(orderedPostProcessors, beanFactory);

 // Finally, no ordered
    List<BeanFactoryPostProcessor> nonOrderedPostProcessors = new ArrayList<>();
 for (String postProcessorName : nonOrderedPostProcessorNames) {
        nonOrderedPostProcessors.add(beanFactory.getBean(postProcessorName, BeanFactoryPostProcessor.class));
    }
    invokeBeanFactoryPostProcessors(nonOrderedPostProcessors, beanFactory);

 // Clear cached merged bean definitions since the post-processors might have
 // modified the original metadata, e.g. replacing placeholders in values...
    beanFactory.clearMetadataCache();
}
```

上述代码较长，但是处理逻辑较为单一，就是对所有的 BeanDefinitionRegistryPostProcessors 、手动注册的 BeanFactoryPostProcessor 以及通过配置文件方式的 BeanFactoryPostProcessor 按照 PriorityOrdered 、 Ordered、no ordered 三种方式分开处理、调用。

6. registerBeanPostProcessors

注册拦截Bean创建的Bean处理器，即注册 BeanPostProcessor

与 BeanFactoryPostProcessor 一样，也是委托给 PostProcessorRegistrationDelegate 来实现的。

```
// AbstractApplicationContext.java

public static void registerBeanPostProcessors(
        ConfigurableListableBeanFactory beanFactory, AbstractApplicationContext applicationContext) {

 // 所有的 BeanPostProcessors
    String[] postProcessorNames = beanFactory.getBeanNamesForType(BeanPostProcessor.class, true, false);

 // 注册 BeanPostProcessorChecker
 // 主要用于记录一些 bean 的信息，这些 bean 不符合所有 BeanPostProcessors 处理的资格时
 int beanProcessorTargetCount = beanFactory.getBeanPostProcessorCount() + 1 + postProcessorNames.length;
    beanFactory.addBeanPostProcessor(new BeanPostProcessorChecker(beanFactory, beanProcessorTargetCount));

 // 区分 PriorityOrdered、Ordered 、 no ordered
    List<BeanPostProcessor> priorityOrderedPostProcessors = new ArrayList<>();
    List<String> orderedPostProcessorNames = new ArrayList<>();
    List<String> nonOrderedPostProcessorNames = new ArrayList<>();
 // MergedBeanDefinition
    List<BeanPostProcessor> internalPostProcessors = new ArrayList<>();
 for (String ppName : postProcessorNames) {
     if (beanFactory.isTypeMatch(ppName, PriorityOrdered.class)) {
            BeanPostProcessor pp = beanFactory.getBean(ppName, BeanPostProcessor.class);
            priorityOrderedPostProcessors.add(pp);
         if (pp instanceof MergedBeanDefinitionPostProcessor) {
                internalPostProcessors.add(pp);
            }
        }
     else if (beanFactory.isTypeMatch(ppName, Ordered.class)) {
            orderedPostProcessorNames.add(ppName);
        }
     else {
            nonOrderedPostProcessorNames.add(ppName);
        }
    }

 // First, PriorityOrdered
    sortPostProcessors(priorityOrderedPostProcessors, beanFactory);
    registerBeanPostProcessors(beanFactory, priorityOrderedPostProcessors);

 // Next, Ordered
    List<BeanPostProcessor> orderedPostProcessors = new ArrayList<>();
 for (String ppName : orderedPostProcessorNames) {
        BeanPostProcessor pp = beanFactory.getBean(ppName, BeanPostProcessor.class);
        orderedPostProcessors.add(pp);
     if (pp instanceof MergedBeanDefinitionPostProcessor) {
            internalPostProcessors.add(pp);
        }
    }
    sortPostProcessors(orderedPostProcessors, beanFactory);
    registerBeanPostProcessors(beanFactory, orderedPostProcessors);

 // onOrdered
    List<BeanPostProcessor> nonOrderedPostProcessors = new ArrayList<>();
 for (String ppName : nonOrderedPostProcessorNames) {
        BeanPostProcessor pp = beanFactory.getBean(ppName, BeanPostProcessor.class);
        nonOrderedPostProcessors.add(pp);
     if (pp instanceof MergedBeanDefinitionPostProcessor) {
            internalPostProcessors.add(pp);
        }
    }
    registerBeanPostProcessors(beanFactory, nonOrderedPostProcessors);

 // Finally, all internal BeanPostProcessors.
    sortPostProcessors(internalPostProcessors, beanFactory);
    registerBeanPostProcessors(beanFactory, internalPostProcessors);

 // 重新注册用来自动探测内部ApplicationListener的post-processor，这样可以将他们移到处理器链条的末尾
    beanFactory.addBeanPostProcessor(new ApplicationListenerDetector(applicationContext));
}
```

7. initMessageSource

初始化上下文中的资源文件，如国际化文件的处理等

其实该方法就是初始化一个 MessageSource 接口的实现类，主要用于国际化/i18n。

```
// AbstractApplicationContext.java

protected void initMessageSource() {
    ConfigurableListableBeanFactory beanFactory = getBeanFactory();
 // 包含 “messageSource” bean
 if (beanFactory.containsLocalBean(MESSAGE_SOURCE_BEAN_NAME)) {
     this.messageSource = beanFactory.getBean(MESSAGE_SOURCE_BEAN_NAME, MessageSource.class);
     // 如果有父类
     // HierarchicalMessageSource 分级处理的 MessageSource
     if (this.parent != null && this.messageSource instanceof HierarchicalMessageSource) {
            HierarchicalMessageSource hms = (HierarchicalMessageSource) this.messageSource;
         if (hms.getParentMessageSource() == null) {
             // 如果没有注册父 MessageSource，则设置为父类上下文的的 MessageSource
                hms.setParentMessageSource(getInternalParentMessageSource());
            }
        }
     if (logger.isDebugEnabled()) {
            logger.debug("Using MessageSource [" + this.messageSource + "]");
        }
    }
 else {
     // 使用 空 MessageSource
        DelegatingMessageSource dms = new DelegatingMessageSource();
        dms.setParentMessageSource(getInternalParentMessageSource());
     this.messageSource = dms;
        beanFactory.registerSingleton(MESSAGE_SOURCE_BEAN_NAME, this.messageSource);
     if (logger.isDebugEnabled()) {
            logger.debug("Unable to locate MessageSource with name '" + MESSAGE_SOURCE_BEAN_NAME +
                 "': using default [" + this.messageSource + "]");
        }
    }
}
```

8. initApplicationEventMulticaster

初始化上下文事件广播器

```
// AbstractApplicationContext.java

protected void initApplicationEventMulticaster() {
    ConfigurableListableBeanFactory beanFactory = getBeanFactory();

 // 如果存在 applicationEventMulticaster bean，则获取赋值
 if (beanFactory.containsLocalBean(APPLICATION_EVENT_MULTICASTER_BEAN_NAME)) {
     this.applicationEventMulticaster =
                beanFactory.getBean(APPLICATION_EVENT_MULTICASTER_BEAN_NAME, ApplicationEventMulticaster.class);
     if (logger.isDebugEnabled()) {
            logger.debug("Using ApplicationEventMulticaster [" + this.applicationEventMulticaster + "]");
        }
    }
 else {
     // 没有则新建 SimpleApplicationEventMulticaster，并完成 bean 的注册
     this.applicationEventMulticaster = new SimpleApplicationEventMulticaster(beanFactory);
        beanFactory.registerSingleton(APPLICATION_EVENT_MULTICASTER_BEAN_NAME, this.applicationEventMulticaster);
     if (logger.isDebugEnabled()) {
            logger.debug("Unable to locate ApplicationEventMulticaster with name '" +
                    APPLICATION_EVENT_MULTICASTER_BEAN_NAME +
                 "': using default [" + this.applicationEventMulticaster + "]");
        }
    }
}
```

如果当前容器中存在 applicationEventMulticaster 的 bean，则对 applicationEventMulticaster 赋值，否则新建一个 SimpleApplicationEventMulticaster 的对象（默认的），并完成注册。

9. onRefresh

给子类扩展初始化其他Bean

预留给 AbstractApplicationContext 的子类用于初始化其他特殊的 bean，该方法需要在所有单例 bean 初始化之前调用。

10. registerListeners

在所有 bean 中查找 listener bean，然后注册到广播器中

```
// AbstractApplicationContext.java

protected void registerListeners() {
 // 注册静态 监听器
 for (ApplicationListener<?> listener : getApplicationListeners()) {
        getApplicationEventMulticaster().addApplicationListener(listener);
    }

    String[] listenerBeanNames = getBeanNamesForType(ApplicationListener.class, true, false);
 for (String listenerBeanName : listenerBeanNames) {
        getApplicationEventMulticaster().addApplicationListenerBean(listenerBeanName);
    }

 // 至此，已经完成将监听器注册到ApplicationEventMulticaster中，下面将发布前期的事件给监听器。
    Set<ApplicationEvent> earlyEventsToProcess = this.earlyApplicationEvents;
 this.earlyApplicationEvents = null;
 if (earlyEventsToProcess != null) {
     for (ApplicationEvent earlyEvent : earlyEventsToProcess) {
            getApplicationEventMulticaster().multicastEvent(earlyEvent);
        }
    }
}
```

10. finishBeanFactoryInitialization

初始化剩下的单例Bean(非延迟加载的)

```
// AbstractApplicationContext.java

protected void finishBeanFactoryInitialization(ConfigurableListableBeanFactory beanFactory) {
 // 初始化转换器
 if (beanFactory.containsBean(CONVERSION_SERVICE_BEAN_NAME) &&
            beanFactory.isTypeMatch(CONVERSION_SERVICE_BEAN_NAME, ConversionService.class)) {
        beanFactory.setConversionService(
                beanFactory.getBean(CONVERSION_SERVICE_BEAN_NAME, ConversionService.class));
    }

 // 如果之前没有注册 bean 后置处理器（例如PropertyPlaceholderConfigurer），则注册默认的解析器
 if (!beanFactory.hasEmbeddedValueResolver()) {
        beanFactory.addEmbeddedValueResolver(strVal -> getEnvironment().resolvePlaceholders(strVal));
    }

  // 初始化 Initialize LoadTimeWeaverAware beans early to allow for registering their transformers early.
    String[] weaverAwareNames = beanFactory.getBeanNamesForType(LoadTimeWeaverAware.class, false, false);
  for (String weaverAwareName : weaverAwareNames) {
        getBean(weaverAwareName);
    }

  // 停止使用临时的 ClassLoader
    beanFactory.setTempClassLoader(null);

  //
    beanFactory.freezeConfiguration();

  // 初始化所有剩余的单例（非延迟初始化）
    beanFactory.preInstantiateSingletons();
}
```

11. finishRefresh

> 完成刷新过程,通知生命周期处理器 lifecycleProcessor 刷新过程,同时发出 ContextRefreshEvent 通知别人

主要是调用 `LifecycleProcessor#onRefresh()` ，并且发布事件（ContextRefreshedEvent）。

```
// AbstractApplicationContext.java

protected void finishRefresh() {
  // Clear context-level resource caches (such as ASM metadata from scanning).
    clearResourceCaches();

  // Initialize lifecycle processor for this context.
    initLifecycleProcessor();

  // Propagate refresh to lifecycle processor first.
    getLifecycleProcessor().onRefresh();

  // Publish the final event.
    publishEvent(new ContextRefreshedEvent(this));

  // Participate in LiveBeansView MBean, if active.
    LiveBeansView.registerApplicationContext(this);
}
```

文章目录