Spring初始化源码分析

接下来我们详细分析下refresh方法的作用。

一、refresh方法

```
    @Override
    public void refresh() throws BeansException, IllegalStateException {
        synchronized (this.startupShutdownMonitor) {
            // 1.context 为刷新做准备
            // Prepare this context for refreshing.
            prepareRefresh();

            // Tell the subclass to refresh the internal bean factory.
            // 2.让子类实现刷新内部持有BeanFactory
            ConfigurableListableBeanFactory beanFactory =
obtainFreshBeanFactory();

            // Prepare the bean factory for use in this context.
            // 3.对beanFactory做一些准备工作：注册一些context回调、bean等
            prepareBeanFactory(beanFactory);

            try {
                // Allows post-processing of the bean factory in context
subclasses.
                // 4.调用留给子类来提供实现逻辑的 对BeanFactory进行处理的钩子方法
                postProcessBeanFactory(beanFactory);

                // Invoke factory processors registered as beans in the context.
                // 5.执行context中注册的 BeanFactoryPostProcessor bean
                invokeBeanFactoryPostProcessors(beanFactory);

                // Register bean processors that intercept bean creation.
                // 6.注册BeanPostProcessor: 获得用户注册的BeanPostProcessor实例，注册
到BeanFactory上
                registerBeanPostProcessors(beanFactory);

                // Initialize message source for this context.
                // 7.初始化国际化资源
                initMessageSource();

                // Initialize event multicaster for this context.
                // 8.初始化Application event 广播器
                initApplicationEventMulticaster();

                // Initialize other special beans in specific context
subclasses.
                // 9.执行 有子类来提供实现逻辑的钩子方法 onRefresh
                onRefresh();

                // Check for listener beans and register them.
                // 10.注册ApplicationListener: 获得用户注册的ApplicationListener
Bean实例，注册到广播器上
```

```
                registerListeners();

                // Instantiate all remaining (non-lazy-init) singletons.
                // 11、完成剩余的单例Bean的实例化
                finishBeanFactoryInitialization(beanFactory);

                // Last step: publish corresponding event.
                // 12 发布对应的事件
                finishRefresh();
            }

            catch (BeansException ex) {
                if (logger.isWarnEnabled()) {
                    logger.warn("Exception encountered during context
initialization - " +
                            "cancelling refresh attempt: " + ex);
                }

                // Destroy already created singletons to avoid dangling
resources.
                destroyBeans();

                // Reset 'active' flag.
                cancelRefresh(ex);

                // Propagate exception to caller.
                throw ex;
            }

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

二、prepareRefresh

完成一些刷新前的准备工作.

```
    protected void prepareRefresh() {
        // Switch to active.
        this.startupDate = System.currentTimeMillis();
// 设置相关的状态
        this.closed.set(false);
        this.active.set(true);

        if (logger.isDebugEnabled()) {
            if (logger.isTraceEnabled()) {
                logger.trace("Refreshing " + this);
            }
            else {
                logger.debug("Refreshing " + getDisplayName());
            }
        }
```

```
        // Initialize any placeholder property sources in the context
environment.
        initPropertySources();

        // Validate that all properties marked as required are resolvable:
        // see ConfigurablePropertyResolver#setRequiredProperties
        getEnvironment().validateRequiredProperties();

        // Store pre-refresh ApplicationListeners...
        if (this.earlyApplicationListeners == null) {
            this.earlyApplicationListeners = new LinkedHashSet<>
(this.applicationListeners);
        }
        else {
            // Reset local application listeners to pre-refresh state.
            this.applicationListeners.clear();
            this.applicationListeners.addAll(this.earlyApplicationListeners);
        }

        // Allow for the collection of early ApplicationEvents,
        // to be published once the multicaster is available...
        this.earlyApplicationEvents = new LinkedHashSet<>();
    }
```

三、obtainFreshBeanFactory

在obtainFreshBeanFactory方法会完成BeanFactory对象的创建。

```
    protected ConfigurableListableBeanFactory obtainFreshBeanFactory() {
        // 刷新容器
        refreshBeanFactory();
        return getBeanFactory();
    }
```

如果是基于XML的方式使用会在refreshBeanFactory中完成配置文件的加载解析操作

```
    @Override
    protected final void refreshBeanFactory() throws BeansException {
        if (hasBeanFactory()) {
            // 销毁前面的 BeanFactory
            destroyBeans();
            closeBeanFactory();
        }
        try {
            // 创建 BeanFactory 对象
            DefaultListableBeanFactory beanFactory = createBeanFactory();
            beanFactory.setSerializationId(getId());
            customizeBeanFactory(beanFactory);
            loadBeanDefinitions(beanFactory); // 加载解析配置文件
            this.beanFactory = beanFactory;
        }
        catch (IOException ex) {
            throw new ApplicationContextException("I/O error parsing bean
definition source for " + getDisplayName(), ex);
        }
```

```
    }
```

四、prepareBeanFactory

上面的obtainFreshBeanFactory中完成了BeanFactory的创建和相关BeanDefinition对象的组装，然后在接下来的prepareBeanFactory中会完成相关的准备工作。

```
    protected void prepareBeanFactory(ConfigurableListableBeanFactory
beanFactory) {
        // Tell the internal bean factory to use the context's class loader etc.
        // 设置beanFactory的classloader为当前context的classloader
        beanFactory.setBeanClassLoader(getClassLoader());
        // 设置beanfactory的表达式语言处理器
        beanFactory.setBeanExpressionResolver(new
StandardBeanExpressionResolver(beanFactory.getBeanClassLoader()));
        // 为beanFactory增加一个默认的propertyEditor，这个主要是对bean的属性等设置管理的
一个工具类
        beanFactory.addPropertyEditorRegistrar(new ResourceEditorRegistrar(this,
getEnvironment()));

        // Configure the bean factory with context callbacks.
        // 添加beanPostProcessor,ApplicationContextAwareProcessor此类用来完成某些
Aware对象的注入
        beanFactory.addBeanPostProcessor(new
ApplicationContextAwareProcessor(this));
        // 设置要忽略自动装配的接口，很多同学理解不了为什么此处要对这些接口进行忽略，原因非常简
单，这些接口的实现是由容器通过set方法进行注入的，
        // 所以在使用autowire进行注入的时候需要将这些接口进行忽略
        beanFactory.ignoreDependencyInterface(EnvironmentAware.class);
        beanFactory.ignoreDependencyInterface(EmbeddedValueResolverAware.class);
        beanFactory.ignoreDependencyInterface(ResourceLoaderAware.class);

beanFactory.ignoreDependencyInterface(ApplicationEventPublisherAware.class);
        beanFactory.ignoreDependencyInterface(MessageSourceAware.class);
        beanFactory.ignoreDependencyInterface(ApplicationContextAware.class);

        // BeanFactory interface not registered as resolvable type in a plain
factory.
        // MessageSource registered (and found for autowiring) as a bean.
        // 设置几个自动装配的特殊规则,当在进行ioc初始化的如果有多个实现，那么就使用指定的对象
进行注入
        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.
        // 注册BPP
        beanFactory.addBeanPostProcessor(new ApplicationListenerDetector(this));

        // Detect a LoadTimeWeaver and prepare for weaving, if found.
```

```
        // 增加对AspectJ的支持，在java中织入分为三种方式，分为编译器织入，类加载器织入，运行期织入，编译器织入是指在java编译器，采用特殊的编译器，将切面织入到java类中，
        // 而类加载期织入则指通过特殊的类加载器，在类字节码加载到JVM时，织入切面，运行期织入则是采用cglib和jdk进行切面的织入
        // aspectj提供了两种织入方式，第一种是通过特殊编译器，在编译器，将aspectj语言编写的切面类织入到java类中，第二种是类加载期织入，就是下面的load time weaving，此处后续讲
        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()));
        }

        // Register default environment beans.
        // 注册默认的系统环境bean到一级缓存中
        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());
        }
    }
```

五、postProcessBeanFactory

该方法是一个空方法，交给子类自己处理的方法

六、invokeBeanFactoryPostProcessors

invokeBeanFactoryPostProcessors是BeanFactory的后置处理方法。核心是会完成注册的BeanFactoryPostProcessor接口和BeanDefinitionRegistryPostProcessor的相关逻辑。invokeBeanFactoryPostProcessors是其核心的方法。

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

        // Invoke BeanDefinitionRegistryPostProcessors first, if any.
        // 无论是什么情况，优先执行BeanDefinitionRegistryPostProcessors
        // 将已经执行过的BFPP存储在processedBeans中，防止重复执行
        Set<String> processedBeans = new HashSet<>();

        // 判断beanfactory是否是BeanDefinitionRegistry类型，此处是DefaultListableBeanFactory,实现了BeanDefinitionRegistry接口，所以为true
        if (beanFactory instanceof BeanDefinitionRegistry) {
            // 类型转换
            BeanDefinitionRegistry registry = (BeanDefinitionRegistry) beanFactory;
```

```
            // 此处希望大家做一个区分，两个接口是不同的，
BeanDefinitionRegistryPostProcessor是BeanFactoryPostProcessor的子集
            // BeanFactoryPostProcessor主要针对的操作对象是BeanFactory，而
BeanDefinitionRegistryPostProcessor主要针对的操作对象是BeanDefinition
            // 存放BeanFactoryPostProcessor的集合
            List<BeanFactoryPostProcessor> regularPostProcessors = new
ArrayList<>();
            // 存放BeanDefinitionRegistryPostProcessor的集合
            List<BeanDefinitionRegistryPostProcessor> registryProcessors = new
ArrayList<>();

            // 首先处理入参中的beanFactoryPostProcessors,遍历所有的
beanFactoryPostProcessors，将BeanDefinitionRegistryPostProcessor
            // 和BeanFactoryPostProcessor区分开
            for (BeanFactoryPostProcessor postProcessor :
beanFactoryPostProcessors) {
                // 如果是BeanDefinitionRegistryPostProcessor
                if (postProcessor instanceof
BeanDefinitionRegistryPostProcessor) {
                    BeanDefinitionRegistryPostProcessor registryProcessor =
                            (BeanDefinitionRegistryPostProcessor) postProcessor;
                    // 直接执行BeanDefinitionRegistryPostProcessor接口中的
postProcessBeanDefinitionRegistry方法

 registryProcessor.postProcessBeanDefinitionRegistry(registry);
                    // 添加到registryProcessors，用于后续执行postProcessBeanFactory
方法
                    registryProcessors.add(registryProcessor);
                } else {
                    // 否则，只是普通的BeanFactoryPostProcessor，添加到
regularPostProcessors，用于后续执行postProcessBeanFactory方法
                    regularPostProcessors.add(postProcessor);
                }
            }

            // Do not initialize FactoryBeans here: We need to leave all regular
beans
            // uninitialized to let the bean factory post-processors apply to
them!
            // Separate between BeanDefinitionRegistryPostProcessors that
implement
            // PriorityOrdered, Ordered, and the rest.
            // 用于保存本次要执行的BeanDefinitionRegistryPostProcessor
            List<BeanDefinitionRegistryPostProcessor> currentRegistryProcessors
= new ArrayList<>();

            // First, invoke the BeanDefinitionRegistryPostProcessors that
implement PriorityOrdered.
            // 调用所有实现PriorityOrdered接口的BeanDefinitionRegistryPostProcessor
实现类
            // 找到所有实现BeanDefinitionRegistryPostProcessor接口bean的beanName
            String[] postProcessorNames =

 beanFactory.getBeanNamesForType(BeanDefinitionRegistryPostProcessor.class,
true, false);
            // 遍历处理所有符合规则的postProcessorNames
            for (String ppName : postProcessorNames) {
                // 检测是否实现了PriorityOrdered接口
```

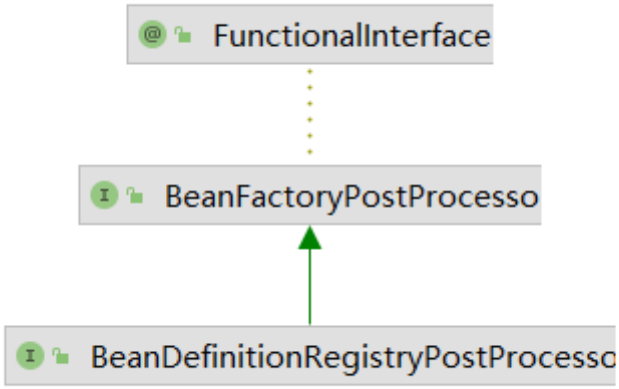
```
                if (beanFactory.isTypeMatch(ppName, PriorityOrdered.class)) {
                    // 获取名字对应的bean实例，添加到currentRegistryProcessors中
                    currentRegistryProcessors.add(beanFactory.getBean(ppName,
BeanDefinitionRegistryPostProcessor.class));
                    // 将要被执行的BFPP名称添加到processedBeans，避免后续重复执行
                    processedBeans.add(ppName);
                }
            }
            // 按照优先级进行排序操作
            sortPostProcessors(currentRegistryProcessors, beanFactory);
            // 添加到registryProcessors中，用于最后执行postProcessBeanFactory方法
            registryProcessors.addAll(currentRegistryProcessors);
            // 遍历currentRegistryProcessors，执行
postProcessBeanDefinitionRegistry方法

 invokeBeanDefinitionRegistryPostProcessors(currentRegistryProcessors,
registry);
            // 执行完毕之后，清空currentRegistryProcessors
            currentRegistryProcessors.clear();

            // Next, invoke the BeanDefinitionRegistryPostProcessors that
implement Ordered.
            // 调用所有实现Ordered接口的BeanDefinitionRegistryPostProcessor实现类
            // 找到所有实现BeanDefinitionRegistryPostProcessor接口bean的beanName，
            // 此处需要重复查找的原因在于上面的执行过程中可能会新增其他的
BeanDefinitionRegistryPostProcessor
            postProcessorNames =
beanFactory.getBeanNamesForType(BeanDefinitionRegistryPostProcessor.class, true,
false);
            for (String ppName : postProcessorNames) {
                // 检测是否实现了Ordered接口，并且还未执行过
                if (!processedBeans.contains(ppName) &&
beanFactory.isTypeMatch(ppName, Ordered.class)) {
                    // 获取名字对应的bean实例，添加到currentRegistryProcessors中
                    currentRegistryProcessors.add(beanFactory.getBean(ppName,
BeanDefinitionRegistryPostProcessor.class));
                    // 将要被执行的BFPP名称添加到processedBeans，避免后续重复执行
                    processedBeans.add(ppName);
                }
            }
            // 按照优先级进行排序操作
            sortPostProcessors(currentRegistryProcessors, beanFactory);
            // 添加到registryProcessors中，用于最后执行postProcessBeanFactory方法
            registryProcessors.addAll(currentRegistryProcessors);
            // 遍历currentRegistryProcessors，执行
postProcessBeanDefinitionRegistry方法

 invokeBeanDefinitionRegistryPostProcessors(currentRegistryProcessors,
registry);
            // 执行完毕之后，清空currentRegistryProcessors
            currentRegistryProcessors.clear();

            // Finally, invoke all other BeanDefinitionRegistryPostProcessors
until no further ones appear.
            // 最后，调用所有剩下的BeanDefinitionRegistryPostProcessors
            boolean reiterate = true;
            while (reiterate) {
                reiterate = false;
```

```
                // 找出所有实现BeanDefinitionRegistryPostProcessor接口的类
                postProcessorNames =
beanFactory.getBeanNamesForType(BeanDefinitionRegistryPostProcessor.class, true,
false);
                // 遍历执行
                for (String ppName : postProcessorNames) {
                    // 跳过已经执行过的BeanDefinitionRegistryPostProcessor
                    if (!processedBeans.contains(ppName)) {
                        // 获取名字对应的bean实例，添加到currentRegistryProcessors中

 currentRegistryProcessors.add(beanFactory.getBean(ppName,
BeanDefinitionRegistryPostProcessor.class));
                        // 将要被执行的BFPP名称添加到processedBeans，避免后续重复执行
                        processedBeans.add(ppName);
                        reiterate = true;
                    }
                }
                // 按照优先级进行排序操作
                sortPostProcessors(currentRegistryProcessors, beanFactory);
                // 添加到registryProcessors中，用于最后执行postProcessBeanFactory方法
                registryProcessors.addAll(currentRegistryProcessors);
                // 遍历currentRegistryProcessors，执行
postProcessBeanDefinitionRegistry方法

 invokeBeanDefinitionRegistryPostProcessors(currentRegistryProcessors,
registry);
                // 执行完毕之后，清空currentRegistryProcessors
                currentRegistryProcessors.clear();
            }

            // Now, invoke the postProcessBeanFactory callback of all processors
handled so far.
            // 调用所有BeanDefinitionRegistryPostProcessor的postProcessBeanFactory
方法
            invokeBeanFactoryPostProcessors(registryProcessors, beanFactory);
            // 最后，调用入参beanFactoryPostProcessors中的普通
BeanFactoryPostProcessor的postProcessBeanFactory方法
            invokeBeanFactoryPostProcessors(regularPostProcessors, beanFactory);
        } else {
            // Invoke factory processors registered with the context instance.
            // 如果beanFactory不归属于BeanDefinitionRegistry类型，那么直接执行
postProcessBeanFactory方法
            invokeBeanFactoryPostProcessors(beanFactoryPostProcessors,
beanFactory);
        }

        // 到这里为止，入参beanFactoryPostProcessors和容器中的所有
BeanDefinitionRegistryPostProcessor已经全部处理完毕，下面开始处理容器中
        // 所有的BeanFactoryPostProcessor
        // 可能会包含一些实现类，只实现了BeanFactoryPostProcessor，并没有实现
BeanDefinitionRegistryPostProcessor接口

        // Do not initialize FactoryBeans here: We need to leave all regular
beans
        // uninitialized to let the bean factory post-processors apply to them!
        // 找到所有实现BeanFactoryPostProcessor接口的类
        String[] postProcessorNames =
```

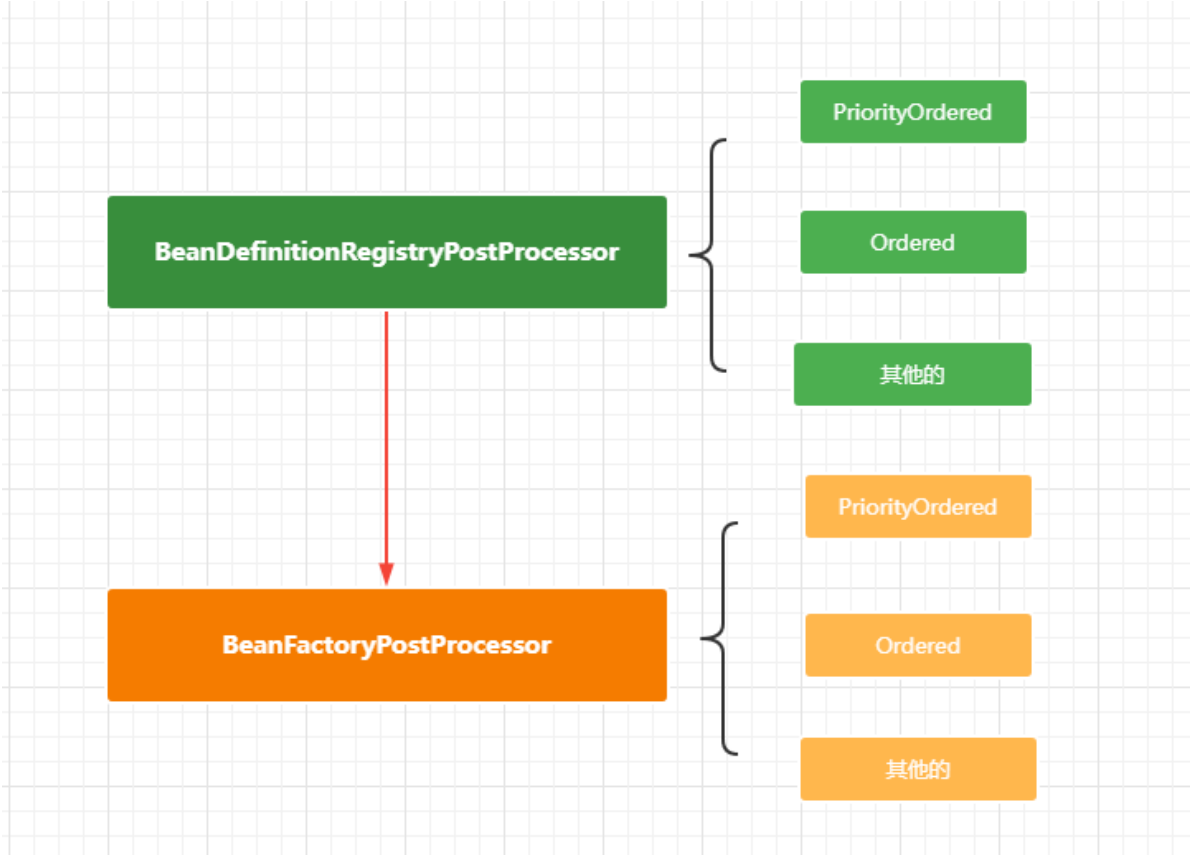
```
                beanFactory.getBeanNamesForType(BeanFactoryPostProcessor.class,
true, false);

        // Separate between BeanFactoryPostProcessors that implement
PriorityOrdered,
        // Ordered, and the rest.
        // 用于存放实现了PriorityOrdered接口的BeanFactoryPostProcessor
        List<BeanFactoryPostProcessor> priorityOrderedPostProcessors = new
ArrayList<>();
        // 用于存放实现了Ordered接口的BeanFactoryPostProcessor的beanName
//      List<String> orderedPostProcessorNames = new ArrayList<>();
        List<BeanFactoryPostProcessor> orderedPostProcessor = new ArrayList<>();
        // 用于存放普通BeanFactoryPostProcessor的beanName
//      List<String> nonOrderedPostProcessorNames = new ArrayList<>();
        List<BeanFactoryPostProcessor> nonOrderedPostProcessorNames = new
ArrayList<>();
        // 遍历postProcessorNames,将BeanFactoryPostProcessor按实现PriorityOrdered、
实现Ordered接口、普通三种区分开
        for (String ppName : postProcessorNames) {
            // 跳过已经执行过的BeanFactoryPostProcessor
            if (processedBeans.contains(ppName)) {
                // skip - already processed in first phase above
            }
            // 添加实现了PriorityOrdered接口的BeanFactoryPostProcessor到
priorityOrderedPostProcessors
            else if (beanFactory.isTypeMatch(ppName, PriorityOrdered.class)) {
                priorityOrderedPostProcessors.add(beanFactory.getBean(ppName,
BeanFactoryPostProcessor.class));
            }
            // 添加实现了Ordered接口的BeanFactoryPostProcessor的beanName到
orderedPostProcessorNames
            else if (beanFactory.isTypeMatch(ppName, Ordered.class)) {
//              orderedPostProcessorNames.add(ppName);
                orderedPostProcessor.add(beanFactory.getBean(ppName,
BeanFactoryPostProcessor.class));
            } else {
                // 添加剩下的普通BeanFactoryPostProcessor的beanName到
nonOrderedPostProcessorNames
//              nonOrderedPostProcessorNames.add(ppName);
                nonOrderedPostProcessorNames.add(beanFactory.getBean(ppName,
BeanFactoryPostProcessor.class));
            }
        }

        // First, invoke the BeanFactoryPostProcessors that implement
PriorityOrdered.
        // 对实现了PriorityOrdered接口的BeanFactoryPostProcessor进行排序
        sortPostProcessors(priorityOrderedPostProcessors, beanFactory);
        // 遍历实现了PriorityOrdered接口的BeanFactoryPostProcessor，执行
postProcessBeanFactory方法
        invokeBeanFactoryPostProcessors(priorityOrderedPostProcessors,
beanFactory);

        // Next, invoke the BeanFactoryPostProcessors that implement Ordered.
        // 创建存放实现了Ordered接口的BeanFactoryPostProcessor集合
//      List<BeanFactoryPostProcessor> orderedPostProcessors = new ArrayList<>
(orderedPostProcessorNames.size());
        // 遍历存放实现了Ordered接口的BeanFactoryPostProcessor名字的集合
```

```
//      for (String postProcessorName : orderedPostProcessorNames) {
        // 将实现了Ordered接口的BeanFactoryPostProcessor添加到集合中
//          orderedPostProcessors.add(beanFactory.getBean(postProcessorName,
BeanFactoryPostProcessor.class));
//      }
        // 对实现了Ordered接口的BeanFactoryPostProcessor进行排序操作
//      sortPostProcessors(orderedPostProcessors, beanFactory);
        sortPostProcessors(orderedPostProcessor, beanFactory);
        // 遍历实现了Ordered接口的BeanFactoryPostProcessor，执行
postProcessBeanFactory方法
//      invokeBeanFactoryPostProcessors(orderedPostProcessors, beanFactory);
        invokeBeanFactoryPostProcessors(orderedPostProcessor, beanFactory);

        // Finally, invoke all other BeanFactoryPostProcessors.
        // 最后，创建存放普通的BeanFactoryPostProcessor的集合
//      List<BeanFactoryPostProcessor> nonOrderedPostProcessors = new
ArrayList<>(nonOrderedPostProcessorNames.size());
        // 遍历存放实现了普通BeanFactoryPostProcessor名字的集合
//      for (String postProcessorName : nonOrderedPostProcessorNames) {
        // 将普通的BeanFactoryPostProcessor添加到集合中
//          nonOrderedPostProcessors.add(beanFactory.getBean(postProcessorName,
BeanFactoryPostProcessor.class));
//      }
        // 遍历普通的BeanFactoryPostProcessor，执行postProcessBeanFactory方法
//      invokeBeanFactoryPostProcessors(nonOrderedPostProcessors, beanFactory);
        invokeBeanFactoryPostProcessors(nonOrderedPostProcessorNames,
beanFactory);

        // Clear cached merged bean definitions since the post-processors might
have
        // modified the original metadata, e.g. replacing placeholders in
values...
        // 清除元数据缓存（mergeBeanDefinitions、allBeanNamesByType、
singletonBeanNameByType）
        // 因为后置处理器可能已经修改了原始元数据，例如，替换值中的占位符
        beanFactory.clearMetadataCache();
    }
```

要搞清楚上面的代码含义首先需要搞清楚出这两者之间的关系



实现的核心流程是



在这个位置核心的代表是 `ConfigurationClassPostProcessor` 用来处理 `@Configuration` 注解表示的Java类，来处理其中的@Bean,@Primary等注解。

七、registerBeanPostProcessors

完成Bean对象的相关后置处理器的注册。具体的代码逻辑和上面是差不多的。

```
    /**
     * 注册beanPostProcessor
     * @param beanFactory
     * @param applicationContext
     */
    public static void registerBeanPostProcessors(
            ConfigurableListableBeanFactory beanFactory,
AbstractApplicationContext applicationContext) {

        // 找到所有实现了BeanPostProcessor接口的类
        String[] postProcessorNames =
beanFactory.getBeanNamesForType(BeanPostProcessor.class, true, false);

        // Register BeanPostProcessorChecker that logs an info message when
        // a bean is created during BeanPostProcessor instantiation, i.e. when
        // a bean is not eligible for getting processed by all
BeanPostProcessors.
        // 记录下BeanPostProcessor的目标计数
        // 此处为什么要+1呢，原因非常简单，在此方法的最后会添加一个
BeanPostProcessorChecker的类
        int beanProcessorTargetCount = beanFactory.getBeanPostProcessorCount() +
1 + postProcessorNames.length;
        // 添加BeanPostProcessorChecker(主要用于记录信息)到beanFactory中
        beanFactory.addBeanPostProcessor(new
BeanPostProcessorChecker(beanFactory, beanProcessorTargetCount));
```

```
        // Separate between BeanPostProcessors that implement PriorityOrdered,
        // Ordered, and the rest.
        // 定义存放实现了PriorityOrdered接口的BeanPostProcessor集合
        List<BeanPostProcessor> priorityOrderedPostProcessors = new ArrayList<>
();
        // 定义存放spring内部的BeanPostProcessor
        List<BeanPostProcessor> internalPostProcessors = new ArrayList<>();
        // 定义存放实现了Ordered接口的BeanPostProcessor的name集合
        List<String> orderedPostProcessorNames = new ArrayList<>();
        // 定义存放普通的BeanPostProcessor的name集合
        List<String> nonOrderedPostProcessorNames = new ArrayList<>();
        // 遍历beanFactory中存在的BeanPostProcessor的集合postProcessorNames，
        for (String ppName : postProcessorNames) {
            // 如果ppName对应的BeanPostProcessor实例实现了PriorityOrdered接口，则获取
到ppName对应的BeanPostProcessor的实例添加到priorityOrderedPostProcessors中
            if (beanFactory.isTypeMatch(ppName, PriorityOrdered.class)) {
                BeanPostProcessor pp = beanFactory.getBean(ppName,
BeanPostProcessor.class);
                priorityOrderedPostProcessors.add(pp);
                // 如果ppName对应的BeanPostProcessor实例也实现了
MergedBeanDefinitionPostProcessor接口，那么则将ppName对应的bean实例添加到
internalPostProcessors中
                if (pp instanceof MergedBeanDefinitionPostProcessor) {
                    internalPostProcessors.add(pp);
                }
            }
            // 如果ppName对应的BeanPostProcessor实例没有实现PriorityOrdered接口，但是
实现了Ordered接口，那么将ppName对应的bean实例添加到orderedPostProcessorNames中
            else if (beanFactory.isTypeMatch(ppName, Ordered.class)) {
                orderedPostProcessorNames.add(ppName);
            } else {
                // 否则将ppName添加到nonOrderedPostProcessorNames中
                nonOrderedPostProcessorNames.add(ppName);
            }
        }

        // First, register the BeanPostProcessors that implement
PriorityOrdered.
        // 首先，对实现了PriorityOrdered接口的BeanPostProcessor实例进行排序操作
        sortPostProcessors(priorityOrderedPostProcessors, beanFactory);
        // 注册实现了PriorityOrdered接口的BeanPostProcessor实例添加到beanFactory中
        registerBeanPostProcessors(beanFactory, priorityOrderedPostProcessors);

        // Next, register the BeanPostProcessors that implement Ordered.
        // 注册所有实现Ordered的beanPostProcessor
        List<BeanPostProcessor> orderedPostProcessors = new ArrayList<>
(orderedPostProcessorNames.size());
        for (String ppName : orderedPostProcessorNames) {
            // 根据ppName找到对应的BeanPostProcessor实例对象
            BeanPostProcessor pp = beanFactory.getBean(ppName,
BeanPostProcessor.class);
            // 将实现了Ordered接口的BeanPostProcessor添加到orderedPostProcessors集合
中
            orderedPostProcessors.add(pp);
            // 如果ppName对应的BeanPostProcessor实例也实现了
MergedBeanDefinitionPostProcessor接口，那么则将ppName对应的bean实例添加到
internalPostProcessors中
```

```
            if (pp instanceof MergedBeanDefinitionPostProcessor) {
                internalPostProcessors.add(pp);
            }
        }
        // 对实现了Ordered接口的BeanPostProcessor进行排序操作
        sortPostProcessors(orderedPostProcessors, beanFactory);
        //  注册实现了Ordered接口的BeanPostProcessor实例添加到beanFactory中
        registerBeanPostProcessors(beanFactory, orderedPostProcessors);

        // Now, register all regular BeanPostProcessors.
        // 创建存放没有实现PriorityOrdered和Ordered接口的BeanPostProcessor的集合
        List<BeanPostProcessor> nonOrderedPostProcessors = new ArrayList<>
(nonOrderedPostProcessorNames.size());
        // 遍历集合
        for (String ppName : nonOrderedPostProcessorNames) {
            // 根据ppName找到对应的BeanPostProcessor实例对象
            BeanPostProcessor pp = beanFactory.getBean(ppName,
BeanPostProcessor.class);
            // 将没有实现PriorityOrdered和Ordered接口的BeanPostProcessor添加到
nonOrderedPostProcessors集合中
            nonOrderedPostProcessors.add(pp);
            // 如果ppName对应的BeanPostProcessor实例也实现了
MergedBeanDefinitionPostProcessor接口，那么则将ppName对应的bean实例添加到
internalPostProcessors中
            if (pp instanceof MergedBeanDefinitionPostProcessor) {
                internalPostProcessors.add(pp);
            }
        }
        //  注册没有实现PriorityOrdered和Ordered的BeanPostProcessor实例添加到
beanFactory中
        registerBeanPostProcessors(beanFactory, nonOrderedPostProcessors);

        // Finally, re-register all internal BeanPostProcessors.
        // 将所有实现了MergedBeanDefinitionPostProcessor类型的BeanPostProcessor进行排
序操作
        sortPostProcessors(internalPostProcessors, beanFactory);
        // 注册所有实现了MergedBeanDefinitionPostProcessor类型的BeanPostProcessor到
beanFactory中
        registerBeanPostProcessors(beanFactory, internalPostProcessors);

        // Re-register post-processor for detecting inner beans as
ApplicationListeners,
        // moving it to the end of the processor chain (for picking up proxies
etc).
        // 注册ApplicationListenerDetector到beanFactory中
        beanFactory.addBeanPostProcessor(new
ApplicationListenerDetector(applicationContext));
    }
```

八、initMessageSource

为上下文初始化message源，即不同语言的消息体，国际化处理.此处不过多介绍、

九、initApplicationEventMulticaster

initApplicationEventMulticaster初始化事件监听多路广播器.

```
    protected void initApplicationEventMulticaster() {
        // 获取当前bean工厂,一般是DefaultListableBeanFactory
        ConfigurableListableBeanFactory beanFactory = getBeanFactory();
        // 判断容器中是否存在bdName为applicationEventMulticaster的bd，也就是说自定义的事
件监听多路广播器，必须实现ApplicationEventMulticaster接口
        if
(beanFactory.containsLocalBean(APPLICATION_EVENT_MULTICASTER_BEAN_NAME)) {
            // 如果有，则从bean工厂得到这个bean对象
            this.applicationEventMulticaster =
                    beanFactory.getBean(APPLICATION_EVENT_MULTICASTER_BEAN_NAME,
ApplicationEventMulticaster.class);
            if (logger.isTraceEnabled()) {
                logger.trace("Using ApplicationEventMulticaster [" +
this.applicationEventMulticaster + "]");
            }
        }
        else {
            // 如果没有，则默认采用SimpleApplicationEventMulticaster
            this.applicationEventMulticaster = new
SimpleApplicationEventMulticaster(beanFactory);

beanFactory.registerSingleton(APPLICATION_EVENT_MULTICASTER_BEAN_NAME,
this.applicationEventMulticaster);
            if (logger.isTraceEnabled()) {
                logger.trace("No '" + APPLICATION_EVENT_MULTICASTER_BEAN_NAME +
"' bean, using " +
                        "[" +
this.applicationEventMulticaster.getClass().getSimpleName() + "]");
            }
        }
    }
```

代码很简单，创建了一个SimpleApplicationEventMulticaster对象，来广播相关的消息事件。

十、onRefresh

留给子类来初始化其他的bean

十一、registerListeners

所有注册的bean中查找listener bean,注册到消息广播器中.

```
    protected void registerListeners() {
        // Register statically specified listeners first.
        // 遍历应用程序中存在的监听器集合，并将对应的监听器添加到监听器的多路广播器中
        for (ApplicationListener<?> listener : getApplicationListeners()) {
            getApplicationEventMulticaster().addApplicationListener(listener);
        }

        // Do not initialize FactoryBeans here: We need to leave all regular
beans
        // uninitialized to let post-processors apply to them!
        // 从容器中获取所有实现了ApplicationListener接口的bd的bdName
        // 放入ApplicationListenerBeans集合中
```

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

getApplicationEventMulticaster().addApplicationListenerBean(listenerBeanName);
//
getApplicationEventMulticaster().addApplicationListener(this.getBean(listenerBea
nName,ApplicationListener.class));
        }

        // Publish early application events now that we finally have a
multicaster...
        // 此处先发布早期的监听器集合
        Set<ApplicationEvent> earlyEventsToProcess =
this.earlyApplicationEvents;
        this.earlyApplicationEvents = null;
        if (!CollectionUtils.isEmpty(earlyEventsToProcess)) {
            for (ApplicationEvent earlyEvent : earlyEventsToProcess) {
                getApplicationEventMulticaster().multicastEvent(earlyEvent);
            }
        }
    }
```

十二、finishBeanFactoryInitialization

finishBeanFactoryInitialization初始化剩下的单实例（非懒加载的）.这个专门单独讲解

十三、finishRefresh

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.

```
    protected void finishRefresh() {
        // Clear context-level resource caches (such as ASM metadata from
scanning).
        // 清除上下文级别的资源缓存(如扫描的ASM元数据)
        // 清空在资源加载器中的所有资源缓存
        clearResourceCaches();

        // Initialize lifecycle processor for this context.
        // 为这个上下文初始化生命周期处理器
        // 初始化LifecycleProcessor.如果上下文中找到'lifecycleProcessor'的
LifecycleProcessor Bean对象，
        // 则使用DefaultLifecycleProcessor
        initLifecycleProcessor();

        // Propagate refresh to lifecycle processor first.
        // 首先将刷新传播到生命周期处理器
        // 上下文刷新的通知，例如自动启动的组件
        getLifecycleProcessor().onRefresh();

        // Publish the final event.
        // 发布最终事件
        // 新建ContextRefreshedEvent事件对象，将其发布到所有监听器。
        publishEvent(new ContextRefreshedEvent(this));
```

```
        // Participate in LiveBeansView MBean, if active.
        // 参与LiveBeansView MBean，如果是活动的
        // LiveBeansView:Sping用于支持JMX 服务的类
        // 注册当前上下文到LiveBeansView，以支持JMX服务
        LiveBeansView.registerApplicationContext(this);
    }
```