Spring入口源码分析

方法入口

Spring容器创建之后，会调用它的refresh方法刷新Spring应用的上下文。

具体查看AbstractApplicationContext#refresh源码

```
public void refresh() throws BeansException, IllegalStateException {
    synchronized (this.startupShutdownMonitor) {
        //刷新前的预处理;
        prepareRefresh();

        //获取BeanFactory；默认实现是DefaultListableBeanFactory，在创建容器的时候创建的
        ConfigurableListableBeanFactory beanFactory = obtainFreshBeanFactory();

        //BeanFactory的预准备工作（BeanFactory进行一些设置，比如context的类加载器，BeanPostPr
        prepareBeanFactory(beanFactory);

        try {
            //BeanFactory准备工作完成后进行的后置处理工作
            postProcessBeanFactory(beanFactory);

            //执行BeanFactoryPostProcessor的方法；
            invokeBeanFactoryPostProcessors(beanFactory);

            //注册BeanPostProcessor（Bean的后置处理器），在创建bean的前后等执行
            registerBeanPostProcessors(beanFactory);

            //初始化MessageSource组件（做国际化功能；消息绑定，消息解析）；
            initMessageSource();

            //初始化事件派发器
            initApplicationEventMulticaster();

            //子类重写这个方法，在容器刷新的时候可以自定义逻辑；如创建Tomcat，Jetty等WEB服务器
            onRefresh();

            //注册应用的监听器。就是注册实现了ApplicationListener接口的监听器bean，这些监听器是
            registerListeners();

            //初始化所有剩下的非懒加载的单例bean
            finishBeanFactoryInitialization(beanFactory);

```

```
37                //完成context的刷新。主要是调用LifecycleProcessor的onRefresh()方法，并且发布事件
38                finishRefresh();
39            }
40
41            ......
42    }
```

prepareRefresh方法

表示在真正做refresh操作之前需要准备做的事情：

- 设置Spring容器的启动时间，
- 开启活跃状态，撤销关闭状态，。
- 初始化context environment（上下文环境）中的占位符属性来源。
- 验证环境信息里一些必须存在的属性

ConfigurableListableBeanFactory beanFactory = obtainFreshBeanFactory()

让这个类（AbstractApplicationContext）的子类刷新内部bean工厂。

- AbstractRefreshableApplicationContext容器：实际上就是重新创建一个bean工厂，并设置工厂的一些属性。
- GenericApplicationContext容器：获取创建容器的就创建的bean工厂，并且设置工厂的ID.

prepareBeanFactory方法

上一步已经把工厂建好了，但是还不能投入使用，因为工厂里什么都没有，还需要配置一些东西。看看这个方法的注释

```
1     /**
2      * Configure the factory's standard context characteristics,
3      * such as the context's ClassLoader and post-processors.
4      * @param beanFactory the BeanFactory to configure
5      */
```

他说配置这个工厂的标准环境，比如context的类加载器和post-processors后处理器。

```
1     protected void prepareBeanFactory(ConfigurableListableBeanFactory beanFactory) {
```

```
        //设置BeanFactory的类加载器
        beanFactory.setBeanClassLoader(getClassLoader());
        //设置支持表达式解析器
        beanFactory.setBeanExpressionResolver(new StandardBeanExpressionResolver(beanFactory.g
        beanFactory.addPropertyEditorRegistrar(new ResourceEditorRegistrar(this, getEnvironmen

        //添加部分BeanPostProcessor【ApplicationContextAwareProcessor】
        beanFactory.addBeanPostProcessor(new ApplicationContextAwareProcessor(this));
        //设置忽略的自动装配的接口EnvironmentAware、EmbeddedValueResolverAware、xx,因为Applicati
        beanFactory.ignoreDependencyInterface(EnvironmentAware.class);
        beanFactory.ignoreDependencyInterface(EmbeddedValueResolverAware.class);
        beanFactory.ignoreDependencyInterface(ResourceLoaderAware.class);
        beanFactory.ignoreDependencyInterface(ApplicationEventPublisherAware.class);
        beanFactory.ignoreDependencyInterface(MessageSourceAware.class);
        beanFactory.ignoreDependencyInterface(ApplicationContextAware.class);

        //注册可以解析的自动装配；我们能直接在任何组件中自动注入：BeanFactory、ResourceLoader、App
        //其他组件中可以通过 @autowired 直接注册使用
        beanFactory.registerResolvableDependency(BeanFactory.class, beanFactory);
        beanFactory.registerResolvableDependency(ResourceLoader.class, this);
        beanFactory.registerResolvableDependency(ApplicationEventPublisher.class, this);
        beanFactory.registerResolvableDependency(ApplicationContext.class, this);

        //添加BeanPostProcessor【ApplicationListenerDetector】后置处理器，在bean初始化前后的一些
        beanFactory.addBeanPostProcessor(new ApplicationListenerDetector(this));

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

        //给BeanFactory中注册一些能用的组件；
        if (!beanFactory.containsLocalBean(ENVIRONMENT_BEAN_NAME)) {
            //环境信息ConfigurableEnvironment
            beanFactory.registerSingleton(ENVIRONMENT_BEAN_NAME, getEnvironment());
        }
        if (!beanFactory.containsLocalBean(SYSTEM_PROPERTIES_BEAN_NAME)) {
          //系统属性，systemProperties【Map<String, Object>】
            beanFactory.registerSingleton(SYSTEM_PROPERTIES_BEAN_NAME, getEnvironment().getSys
        }
        if (!beanFactory.containsLocalBean(SYSTEM_ENVIRONMENT_BEAN_NAME)) {
          //系统环境变量systemEnvironment【Map<String, Object>】
            beanFactory.registerSingleton(SYSTEM_ENVIRONMENT_BEAN_NAME, getEnvironment().getSy
        }
    }
```

postProcessBeanFactory方法

上面对bean工厂进行了许多配置，现在需要对bean工厂进行一些处理。不同的Spring容器做不同的操作。比如GenericWebApplicationContext容器的操作会在BeanFactory中添加ServletContextAwareProcessor用于处理ServletContextAware类型的bean初始化的时候调用setServletContext或者setServletConfig方法(跟ApplicationContextAwareProcessor原理一样)。

> GenericWebApplicationContext#postProcessBeanFactory源码：

```
1    protected void postProcessBeanFactory(ConfigurableListableBeanFactory beanFactory) {
2        if (this.servletContext != null) {
3            beanFactory.addBeanPostProcessor(new ServletContextAwareProcessor(this.servletConte
4            beanFactory.ignoreDependencyInterface(ServletContextAware.class);
5        }
6        WebApplicationContextUtils.registerWebApplicationScopes(beanFactory, this.servletContex
7        WebApplicationContextUtils.registerEnvironmentBeans(beanFactory, this.servletContext);
8    }
```

> AnnotationConfigServletWebServerApplicationContext#postProcessBeanFactory方法

```
1    @Override
2    protected void postProcessBeanFactory(ConfigurableListableBeanFactory beanFactory) {
3        super.postProcessBeanFactory(beanFactory);
4        // 查看basePackages属性，如果设置了会使用ClassPathBeanDefinitionScanner去扫描basePackage
5        if (this.basePackages != null && this.basePackages.length > 0) {
6            this.scanner.scan(this.basePackages);
7        }
8        // 查看annotatedClasses属性，如果设置了会使用AnnotatedBeanDefinitionReader去注册这些bean
9        if (!this.annotatedClasses.isEmpty()) {
10           this.reader.register(ClassUtils.toClassArray(this.annotatedClasses));
11       }
12   }
```

invokeBeanFactoryPostProcessors方法

先介绍两个接口：

- BeanFactoryPostProcessor：用来修改Spring容器中已经存在的bean的定义，使用ConfigurableListableBeanFactory对bean进行处理
- BeanDefinitionRegistryPostProcessor：继承BeanFactoryPostProcessor，作用跟BeanFactoryPostProcessor一样，只不过是使用BeanDefinitionRegistry对bean进行处理

在Spring容器中找出实现了BeanFactoryPostProcessor接口的processor并执行。Spring容器会委托给PostProcessorRegistrationDelegate的invokeBeanFactoryPostProcessors方法执行。

注

1. 在springboot的web程序初始化AnnotationConfigServletWebServerApplicationContext容器时，会初始化内部属性AnnotatedBeanDefinitionReader reader，这个reader构造的时候会在BeanFactory中注册一些post processor，包括BeanPostProcessor和BeanFactoryPostProcessor(比如ConfigurationClassPostProcessor、AutowiredAnnotationBeanPostProcessor)：

```
1 AnnotationConfigUtils.registerAnnotationConfigProcessors(this.registry);
```

2. 在使用mybatis时，一般配置了MapperScannerConfigurer的bean，这个bean就是继承的BeanDefinitionRegistryPostProcessor，所以也是这个地方把扫描的mybatis的接口注册到容器中的。

invokeBeanFactoryPostProcessors方法处理BeanFactoryPostProcessor的逻辑如下：

从Spring容器中找出BeanDefinitionRegistryPostProcessor类型的bean(这些processor是在容器刚创建的时候通过构造AnnotatedBeanDefinitionReader的时候注册到容器中的)，然后按照优先级分别执行，优先级的逻辑如下：

1. 实现PriorityOrdered接口的BeanDefinitionRegistryPostProcessor先全部找出来，然后排序后依次执行
2. 实现Ordered接口的BeanDefinitionRegistryPostProcessor找出来，然后排序后依次执行
3. 没有实现PriorityOrdered和Ordered接口的BeanDefinitionRegistryPostProcessor找出来执行并依次执行

接下来从Spring容器内查找BeanFactoryPostProcessor接口的实现类，然后执行(如果processor已经执行过，则忽略)，这里的查找规则跟上面查找BeanDefinitionRegistryPostProcessor一样，先找PriorityOrdered，然后是Ordered，最后是两者都没。

这里需要说明的是ConfigurationClassPostProcessor这个processor是优先级最高的被执行的processor(实现了PriorityOrdered接口)。这个ConfigurationClassPostProcessor会去BeanFactory中找出所有有@Configuration注解的bean，然后使用ConfigurationClassParser去解析这个类。ConfigurationClassParser内部有个Map<ConfigurationClass, ConfigurationClass>类型的configurationClasses属性用于保存解析的类，ConfigurationClass是一个对要解析的配置类的封装，内部存储了配置类的注解信息、被@Bean注解修饰的方法、@ImportResource注解修饰的信息、ImportBeanDefinitionRegistrar等都存储在这个封装类中。

这里ConfigurationClassPostProcessor最先被处理还有另外一个原因是如果程序中有自定义的BeanFactoryPostProcessor，那么这个PostProcessor首先得通过ConfigurationClassPostProcessor被解析出来，然后才

能被Spring容器找到并执行。(ConfigurationClassPostProcessor不先执行的话，这个Processor是不会被解析的，不会被解析的话也就不会执行了)。

1. 处理@PropertySources注解：进行一些配置信息的解析
2. 处理@ComponentScan注解：使用ComponentScanAnnotationParser扫描basePackage下的需要解析的类(@SpringBootApplication注解也包括了@ComponentScan注解，只不过basePackages是空的，空的话会去获取当前@Configuration修饰的类所在的包)，并注册到BeanFactory中(这个时候bean并没有进行实例化，而是进行了注册。具体的实例化在finishBeanFactoryInitialization方法中执行)。对于扫描出来的类，递归解析
3. 处理@Import注解：先递归找出所有的注解，然后再过滤出只有@Import注解的类，得到@Import注解的值。比如查找@SpringBootApplication注解的@Import注解数据的话，首先发现@SpringBootApplication不是一个@Import注解，然后递归调用修饰了@SpringBootApplication的注解，发现有个@EnableAutoConfiguration注解，再次递归发现被@Import(EnableAutoConfigurationImportSelector.class)修饰，还有@AutoConfigurationPackage注解修饰，再次递归@AutoConfigurationPackage注解，发现被@Import(AutoConfigurationPackages.Registrar.class)注解修饰，所以@SpringBootApplication注解对应的@Import注解有2个，分别是@Import(AutoConfigurationPackages.Registrar.class)和@Import(EnableAutoConfigurationImportSelector.class)。找出所有的@Import注解之后，开始处理逻辑：
   a. 遍历这些@Import注解内部的属性类集合
   b. 如果这个类是个ImportSelector接口的实现类，实例化这个ImportSelector，如果这个类也是DeferredImportSelector接口的实现类，那么加入ConfigurationClassParser的deferredImportSelectors属性中让第6步处理。否则调用ImportSelector的selectImports方法得到需要Import的类，然后对这些类递归做@Import注解的处理
   c. 如果这个类是ImportBeanDefinitionRegistrar接口的实现类，设置到配置类的importBeanDefinitionRegistrars属性中
   d. 其它情况下把这个类入队到ConfigurationClassParser的importStack(队列)属性中，然后把这个类当成是@Configuration注解修饰的类递归重头开始解析这个类
4. 处理@ImportResource注解：获取@ImportResource注解的locations属性，得到资源文件的地址信息。然后遍历这些资源文件并把它们添加到配置类的importedResources属性中
5. 处理@Bean注解：获取被@Bean注解修饰的方法，然后添加到配置类的beanMethods属性中
6. 处理DeferredImportSelector：处理第3步@Import注解产生的DeferredImportSelector，进行selectImports方法的调用找出需要import的类，然后再调用第3步相同的处理逻辑处理

这里@SpringBootApplication注解被@EnableAutoConfiguration修饰，@EnableAutoConfiguration注解被@Import(EnableAutoConfigurationImportSelector.class)修饰，所以在第3步会找出这个@Import修饰的类EnableAutoConfigurationImportSelector，这个类刚好实现了DeferredImportSelector接口，接着就会在第6步被执行。第6步selectImport得到的类就是自动化配置类。

EnableAutoConfigurationImportSelector的selectImport方法会在spring-boot-autoconfigure包的META-INF里面的spring.factories文件中找出key为org.springframework.boot.autoconfigure.EnableAutoConfiguration对应的值，有109个，这109个就是所谓的自动化配置类(XXXAutoConfiguration)。（如果引入了mybatis和pagehelper，也会在对应的XXXautoconfigure包的META-INF里面的spring.factories找到EnableAutoConfiguration，这样可能最后得到的自动配置类会大于109个。）然后在过滤排除一下不需要的配置，最后返回实际用到的。

ConfigurationClassParser解析完成之后，被解析出来的类会放到configurationClasses属性中。然后使用ConfigurationClassBeanDefinitionReader去解析这些类。

这个时候这些bean只是被加载到了Spring容器中。下面这段代码是ConfigurationClassBeanDefinitionReader的解析bean过程：这个时候这些bean只是被加载到了Spring容器中。下面这段代码是ConfigurationClassBeanDefinitionReader#loadBeanDefinitions的解析bean过程：

```
public void loadBeanDefinitions(Set<ConfigurationClass> configurationModel) {
    TrackedConditionEvaluator trackedConditionEvaluator = new TrackedConditionEvaluator();
    for (ConfigurationClass configClass : configurationModel) {
        //对每一个配置类，调用loadBeanDefinitionsForConfigurationClass方法
        loadBeanDefinitionsForConfigurationClass(configClass, trackedConditionEvaluator);
    }
}
```

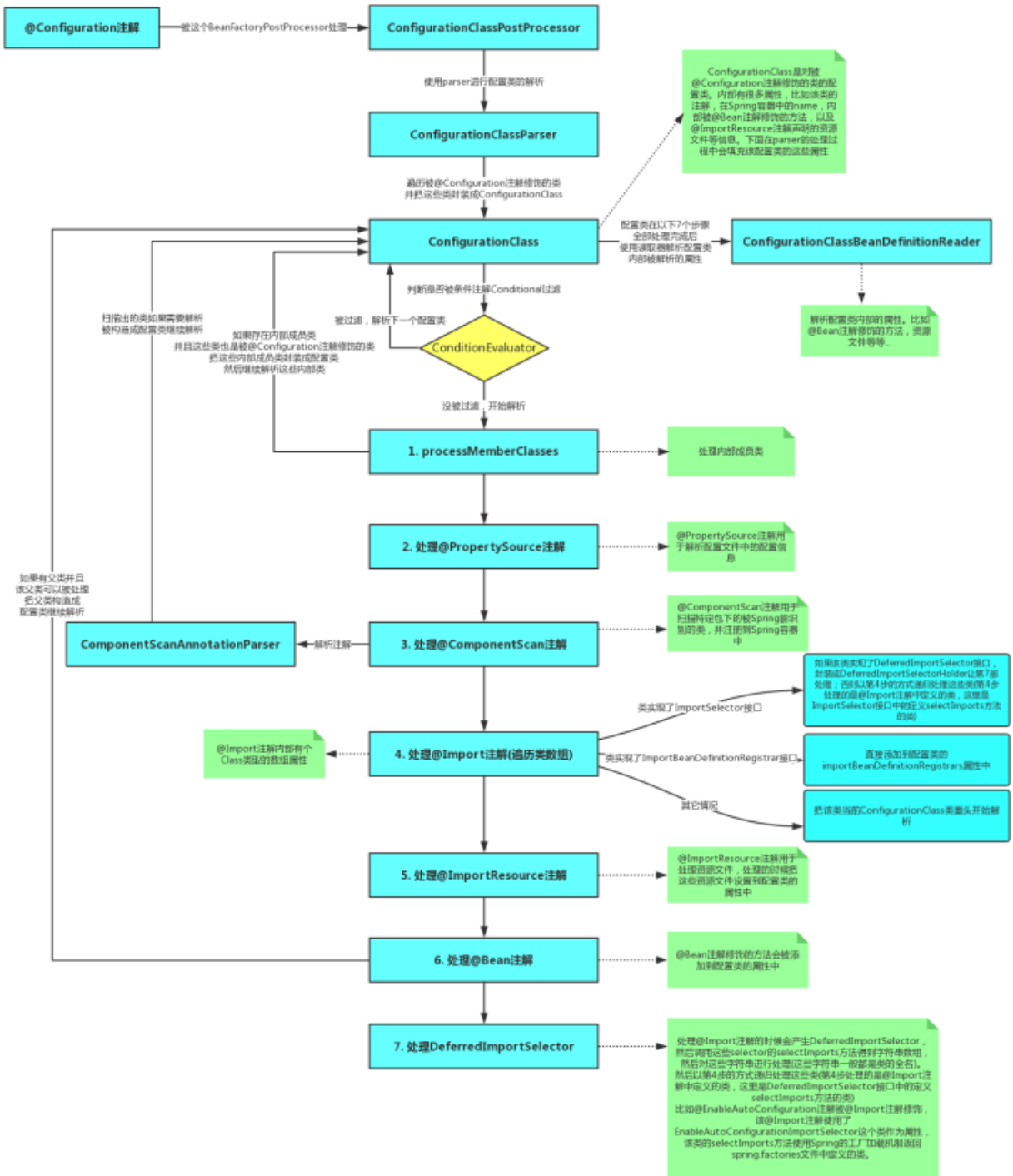
```
 private void loadBeanDefinitionsForConfigurationClass(ConfigurationClass configClass,
         TrackedConditionEvaluator trackedConditionEvaluator) {
     //使用条件注解判断是否需要跳过这个配置类
     if (trackedConditionEvaluator.shouldSkip(configClass)) {
         //跳过配置类的话在Spring容器中移除bean的注册
         String beanName = configClass.getBeanName();
         if (StringUtils.hasLength(beanName) && this.registry.containsBeanDefinition(beanNa
             this.registry.removeBeanDefinition(beanName);
         }
         this.importRegistry.removeImportingClass(configClass.getMetadata().getClassName())
         return;
     }

     if (configClass.isImported()) {
         //如果自身是被@Import注释所import的，注册自己
         registerBeanDefinitionForImportedConfigurationClass(configClass);
     }
     //注册方法中被@Bean注解修饰的bean
     for (BeanMethod beanMethod : configClass.getBeanMethods()) {
         loadBeanDefinitionsForBeanMethod(beanMethod);
     }
     //注册@ImportResource注解注释的资源文件中的bean
     loadBeanDefinitionsFromImportedResources(configClass.getImportedResources());
     //注册@Import注解中的ImportBeanDefinitionRegistrar接口的registerBeanDefinitions
     loadBeanDefinitionsFromRegistrars(configClass.getImportBeanDefinitionRegistrars());
 }
```

invokeBeanFactoryPostProcessors方法总结来说就是从Spring容器中找出BeanDefinitionRegistryPostProcessor和BeanFactoryPostProcessor接口的实现类并按照一定的规则顺序进行执行。 其中ConfigurationClassPostProcessor这个BeanDefinitionRegistryPostProcessor优先级最高，它会对项目中的@Configuration注解修饰的类(@Component、

@ComponentScan、@Import、@ImportResource修饰的类也会被处理)进行解析，解析完成之后把这些bean注册到BeanFactory中。需要注意的是这个时候注册进来的bean还没有实例化。

下面这图就是对ConfigurationClassPostProcessor后置器的总结：



registerBeanPostProcessors方法

从Spring容器中找出的BeanPostProcessor接口的bean，并设置到BeanFactory的属性中。之后bean被实例化的时候会调用这个BeanPostProcessor。

该方法委托给了PostProcessorRegistrationDelegate类的registerBeanPostProcessors方法执行。这里的过程跟invokeBeanFactoryPostProcessors类似：

1. 先找出实现了PriorityOrdered接口的BeanPostProcessor并排序后加到BeanFactory的BeanPostProcessor集合中
2. 找出实现了Ordered接口的BeanPostProcessor并排序后加到BeanFactory的BeanPostProcessor集合中
3. 没有实现PriorityOrdered和Ordered接口的BeanPostProcessor加到BeanFactory的BeanPostProcessor集合中

这些已经存在的BeanPostProcessor在postProcessBeanFactory方法中已经说明，都是由AnnotationConfigUtils的registerAnnotationConfigProcessors方法注册的。这些BeanPostProcessor包括有AutowiredAnnotationBeanPostProcessor(处理被@Autowired注解修饰的bean并注入)、RequiredAnnotationBeanPostProcessor(处理被@Required注解修饰的方法)、CommonAnnotationBeanPostProcessor(处理@PreDestroy、@PostConstruct、@Resource等多个注解的作用)等。

如果是自定义的BeanPostProcessor，已经被ConfigurationClassPostProcessor注册到容器内。

这些BeanPostProcessor会在这个方法内被实例化(通过调用BeanFactory的getBean方法，如果没有找到实例化的类，就会去实例化)。

initMessageSource方法

初始化MessageSource组件（做国际化功能；消息绑定，消息解析）,这个接口提供了消息处理功能。主要用于国际化/i18n。

initApplicationEventMulticaster方法

在Spring容器中初始化事件广播器，事件广播器用于事件的发布。

程序首先会检查bean工厂中是否有bean的名字和这个常量(applicationEventMulticaster)相同的，如果没有则说明没有那么就使用默认的ApplicationEventMulticaster 的实现：SimpleApplicationEventMulticaster

onRefresh方法

一个模板方法，不同的Spring容器做不同的事情。

比如web程序的容器ServletWebServerApplicationContext中会调用createWebServer方法去创建内置的Servlet容器。

目前SpringBoot只支持3种内置的Servlet容器：

- Tomcat
- Jetty
- Undertow

registerListeners方法

注册应用的监听器。就是注册实现了ApplicationListener接口的监听器bean，这些监听器是注册到ApplicationEventMulticaster中的。这不会影响到其它监听器bean。在注册完以后，还会将其前期的事件发布给相匹配的监听器。

```
protected void registerListeners() {
    //1、从容器中拿到所有已经创建的ApplicationListener
    for (ApplicationListener<?> listener : getApplicationListeners()) {
        //2、将每个监听器添加到事件派发器中；
        getApplicationEventMulticaster().addApplicationListener(listener);
    }

    // Do not initialize FactoryBeans here: We need to leave all regular beans
    // uninitialized to let post-processors apply to them!
    // 1.获取所有还没有创建的ApplicationListener
    String[] listenerBeanNames = getBeanNamesForType(ApplicationListener.class, true, fals
    for (String listenerBeanName : listenerBeanNames) {
        //2、将每个监听器添加到事件派发器中；
        getApplicationEventMulticaster().addApplicationListenerBean(listenerBeanName);
    }

    // earlyApplicationEvents 中保存之前的事件，
    Set<ApplicationEvent> earlyEventsToProcess = this.earlyApplicationEvents;
    this.earlyApplicationEvents = null;
    if (earlyEventsToProcess != null) {
        for (ApplicationEvent earlyEvent : earlyEventsToProcess) {
            //3、派发之前步骤产生的事件；
            getApplicationEventMulticaster().multicastEvent(earlyEvent);
        }
    }
}
```

finishBeanFactoryInitialization方法

实例化BeanFactory中已经被注册但是未实例化的所有实例(懒加载的不需要实例化)。

比如invokeBeanFactoryPostProcessors方法中根据各种注解解析出来的类，在这个时候都会被初始化。

实例化的过程各种BeanPostProcessor开始起作用。

后面在详细分析此步骤

finishRefresh方法

refresh做完之后需要做的其他事情。

- 初始化生命周期处理器，并设置到Spring容器中(LifecycleProcessor)
- 调用生命周期处理器的onRefresh方法，这个方法会找出Spring容器中实现了SmartLifecycle接口的类并进行start方法的调用
- 发布ContextRefreshedEvent事件告知对应的ApplicationListener进行响应的操作

如果是web容器ServletWebServerApplicationContext还会启动web服务和发布消息

```
1     protected void finishRefresh() {
2         super.finishRefresh();
3         WebServer webServer = startWebServer();
4         if (webServer != null) {
5             publishEvent(new ServletWebServerInitializedEvent(webServer, this));
6         }
7     }
```