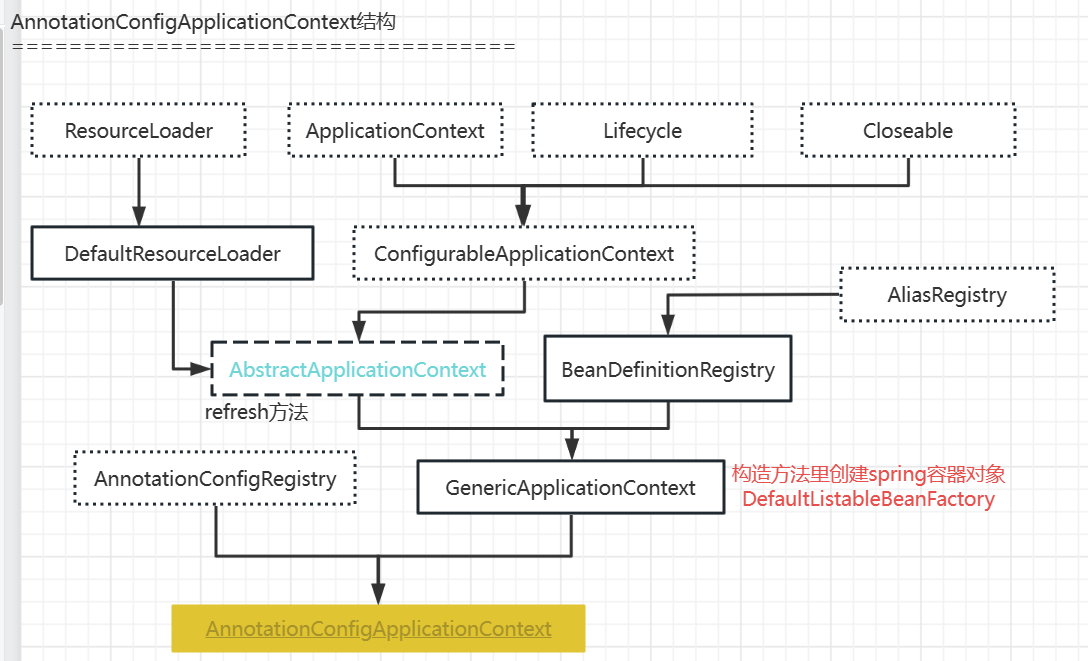
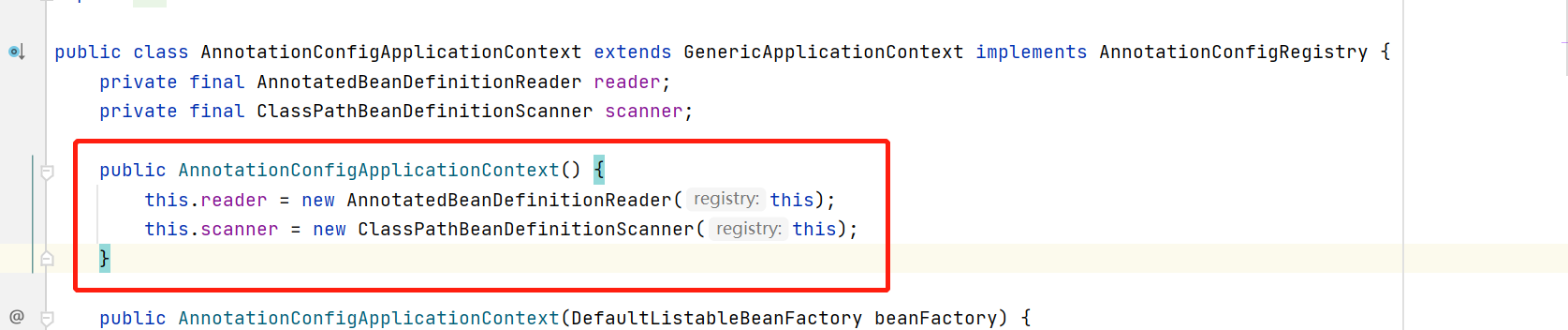
# 一、创建AnnotationConfigApplicationContext对象



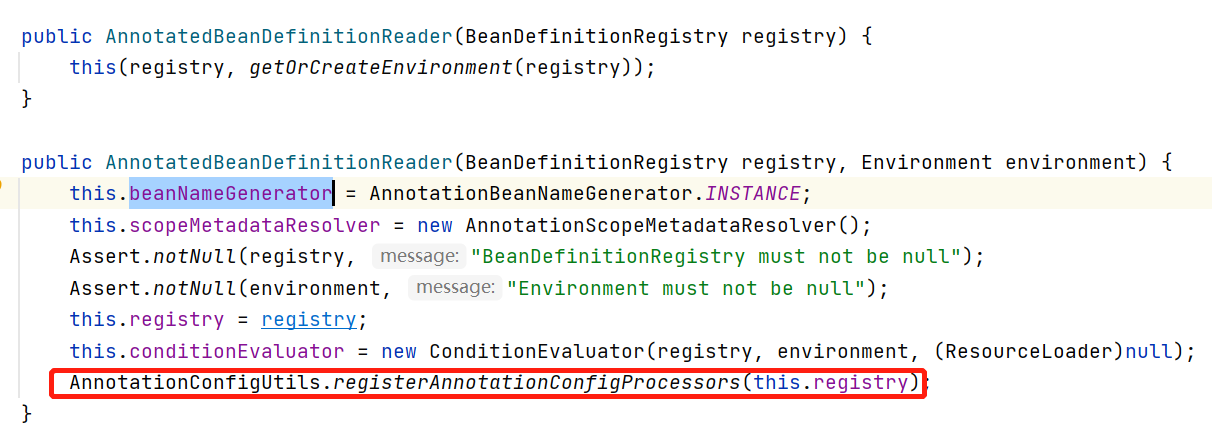
AnnotationConfigApplicationContext的构造方法：



这里主要创建了两个类AnnotatedBeanDefinitionReader和ClassPathBeanDefinitionScanner，

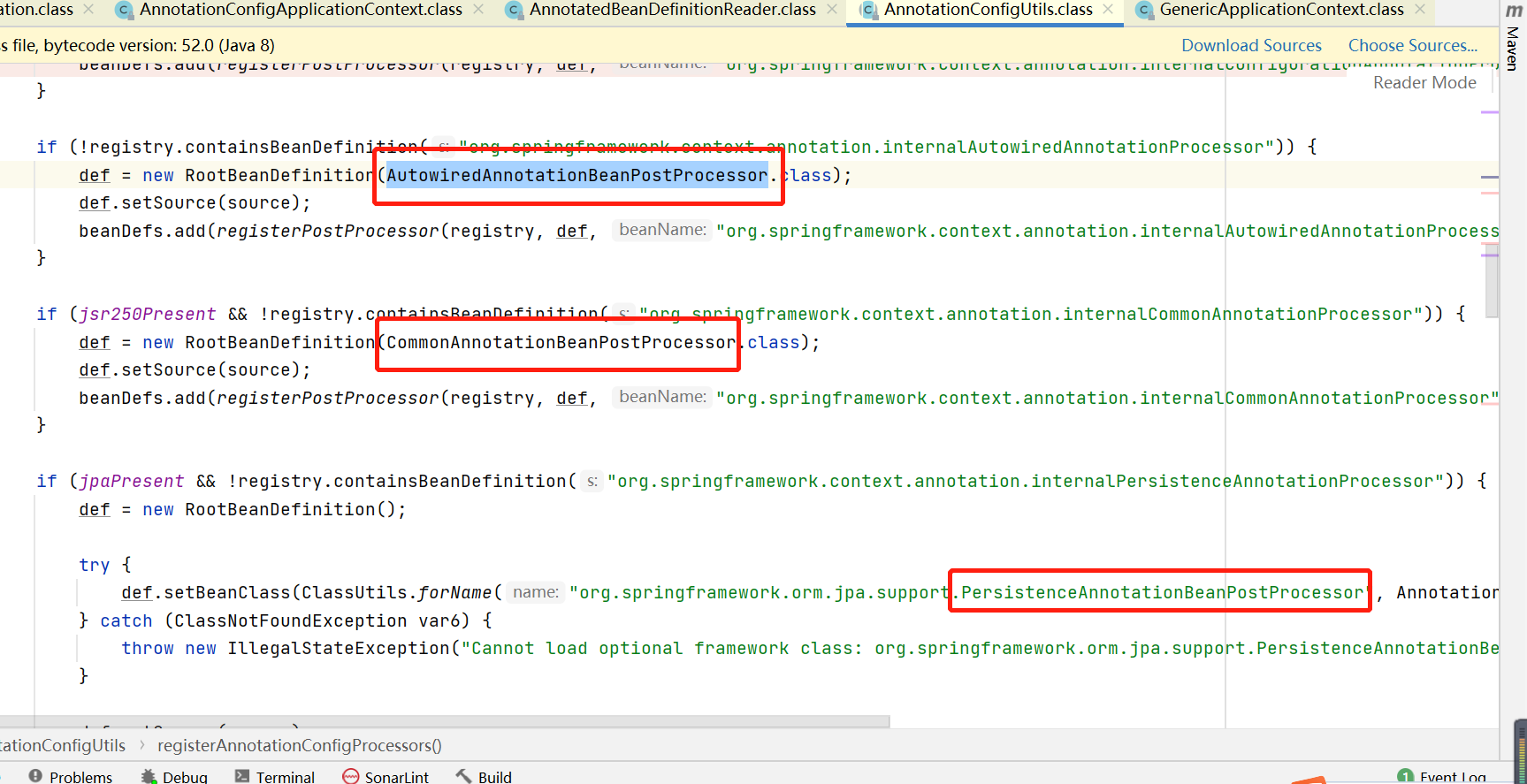
### AnnotatedBeanDefinitionReader

下面看下AnnotatedBeanDefinitionReader的构造方法：



我们重点看下AnnotationConfigUtils#registerAnnotationConfigProcessors方法：







这里重点说明的是：

**ConfigurationClassPostProcessor类**，它是BeanFactoryPostProcessor接口的实现类；

**AutowiredAnnotationBeanPostProcessor类**，它是BeanPostProcessor类的实现类；

### ClassPathBeanDefinitionScanner

扫描解析使用@Component(包含@Controller、@Repository、@Service)、javax.annotation.ManagedBean、javax.inject.Named的类，注入到spring容器中。

# 二、调用AbstractApplicationContext#refresh方法

到这一步，spring容器的初始化的前期准备工作已经就绪，DefaultListableBeanFactory对象（创建AnnotationConfigApplicationContext对象时候的父类GenericApplicationContext构造方法里创建的DefaultListableBeanFactory对象）的Map<String, BeanDefinition> beanDefinitionMap 属性已经有了所有待实例化的对象。

下面我们具体看下spring是如何一步步的去初始化对象的。

## 2.1 AbstractApplicationContext#prepareRefresh

刷新前的准备工作：



核心点：

给AbstractApplicationContext 类的Set<ApplicationListener<?>> applicationListeners集合赋值

## 2.2AbstractApplicationContext#obtainFreshBeanFactory

* AbstractApplicationContext#refreshBeanFactory

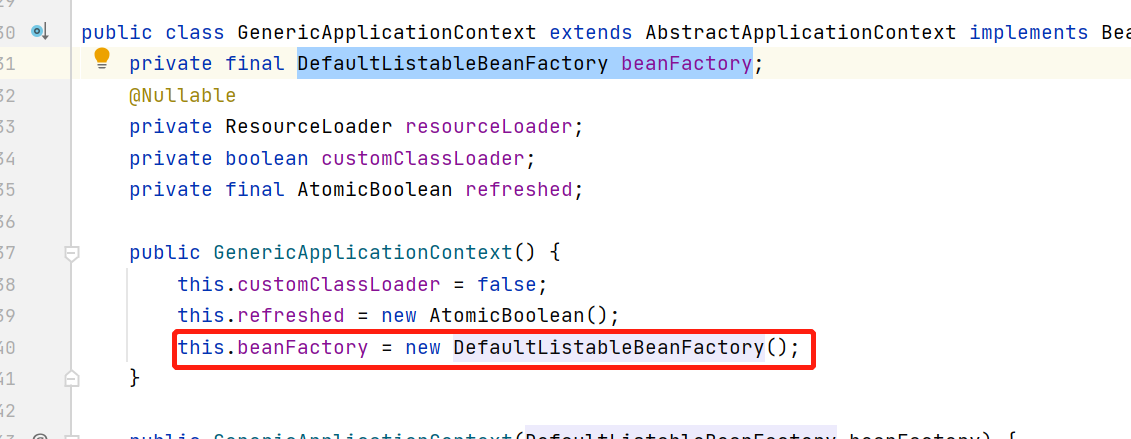
设置DefaultListableBeanFactory beanFactory的id：



* AbstractApplicationContext#getBeanFactory

获取DefaultListableBeanFactory beanFactory对象；

转调的是GenericApplicationContext#getBeanFactory方法，在GenericApplicationContext的构造方法里已经对beanFactory赋值，而GenericApplicationContext是AnnotationConfigApplicationContext类的父类，所以我们在创建AnnotationConfigApplicationContext对象的时候，其父类GenericApplicationContext的构造方法也被调用了：



## 2.3. AbstractApplicationContext#prepareBeanFactory

对2.2获取的beanFactory进行赋值，做一些前期的准备工作；





1、AbstractBeanFactory类的属性BeanExpressionResolver beanExpressionResolver添加值：

**new StandardBeanExpressionResolver(beanFactory.getBeanClassLoader()；**

后续在创建完实例后，对属性进行赋值的时候，会解析属性上的el表达式，进而进行属性值的替换。

2、AbstractBeanFactory类添加Set<PropertyEditorRegistrar> propertyEditorRegistrars ：

**new ResourceEditorRegistrar(this, this.getEnvironment())；**

3、AbstractAutowireCapableBeanFactory类的 Set<Class<?>> ignoredDependencyInterfaces:

**EnvironmentAware\EmbeddedValueResolverAware\ResourceLoaderAware\ApplicationEventPublisherAware\MessageSourceAware\ApplicationContextAware;**

这个就是把这几个类给存入集合中标记下，后续在调用AbstractAutowireCapableBeanFactory#isExcludedFromDependencyCheck方法时，不会校验这些被标记的接口项的依赖。

4、DefaultListableBeanFactory类的 Map<Class<?>, Object> resolvableDependencies：

**BeanFactory、ResourceLoader、ApplicationEventPublisher、ApplicationContext**

5、AbstractBeanFactory类添加 List<BeanPostProcessor> beanPostProcessors

**ApplicationContextAwareProcessor\LoadTimeWeaverAwareProcessor\ApplicationListenerDetector**

6、DefaultListableBeanFactory#registerSingleton 注册单例bean：

**"environment", this.getEnvironment()；**

**"systemProperties", this.getEnvironment().getSystemProperties()**

**"systemEnvironment", this.getEnvironment().getSystemEnvironment()**

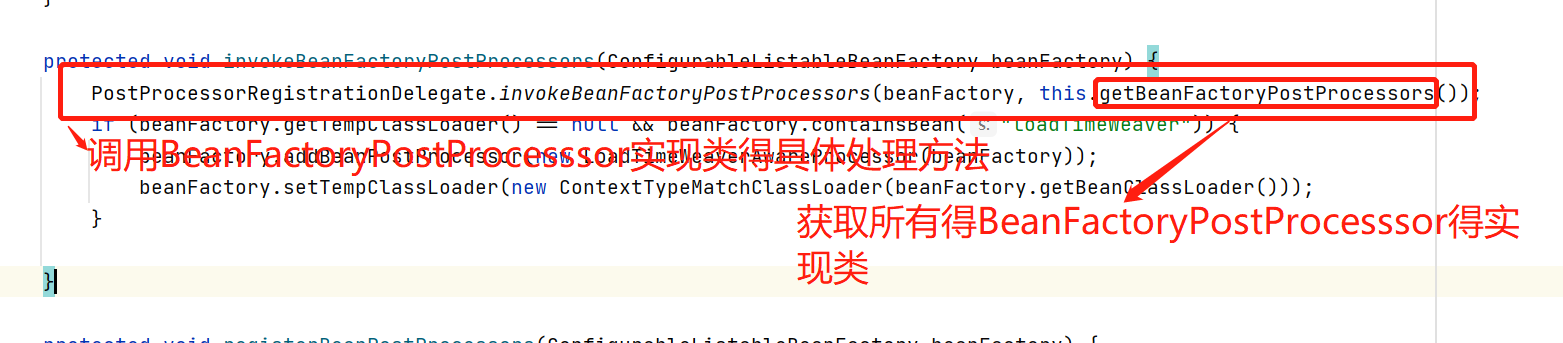
## 2.4 AbstractApplicationContext#postProcessBeanFactory

对beanFactory的后置处理；

2.2 获取beanFactory对象，2.3对beanFactory进行赋值；2.4是对beanFactory的后置处理；2.2-2.3是公共的代码，2.4是子类需要实现的(按需实现)。

## 2.5 AbstractApplicationContext#invokeBeanFactoryPostProcessors

调用**BeanFactoryPostProcessor**实现类的postProcessBeanFactory方法：



在AbstractApplicationContext 类List<BeanFactoryPostProcessor> beanFactoryPostProcessors中目前有三个值：

* **CachingMetadataReaderFactoryPostProcessor**

SharedMetadataReaderFactoryContextInitializer#initialize方法通过调用AbstractApplicationContext#addBeanFactoryPostProcessor方法给赋值；

* **ConfigurationWarningsPostProcessor**

ConfigurationWarningsApplicationContextInitializer#initialize方法通过调用AbstractApplicationContext#addBeanFactoryPostProcessor方法给赋值；

* **PropertySourceOrderingPostProcessor**

ConfigFileApplicationListener#onApplicationPreparedEvent -->#addPostProcessors方法通过调用AbstractApplicationContext#addBeanFactoryPostProcessor方法给赋值；

下面我们看它对BeanFactoryPostProcessor得实现类进行了什么处理。

### PostProcessorRegistrationDelegate#invokeBeanFactoryPostProcessors

|  |
| --- |
| /\*\*  \* 入参beanFactory，就是在AbstractApplicationContext#refresh初始化的beanFactory；  \* 入参beanFactoryPostProcessors 是传入的BeanFactoryPostProcessor得实现类，前面已经介绍  \*/  public static void invokeBeanFactoryPostProcessors(ConfigurableListableBeanFactory beanFactory, List<BeanFactoryPostProcessor> beanFactoryPostProcessors) {  Set<String> processedBeans = new HashSet(); ArrayList regularPostProcessors; //存入BeanFactoryPostProcessor的其他实现类(非BeanDefinitionRegistryPostProcessor的实现类)  ArrayList registryProcessors; //存入BeanDefinitionRegistryPostProcessor的实现类  int var9;  ArrayList currentRegistryProcessors;  String[] postProcessorNames;  if (beanFactory instanceof BeanDefinitionRegistry) {  BeanDefinitionRegistry registry = (BeanDefinitionRegistry)beanFactory;  regularPostProcessors = new ArrayList();  registryProcessors = new ArrayList();  Iterator var6 = beanFactoryPostProcessors.iterator();    while(var6.hasNext()) { //遍历入参的三个BeanFactoryPostProcessor得实现类  BeanFactoryPostProcessor postProcessor = (BeanFactoryPostProcessor)var6.next();  // ConfigurationWarningsPostProcessor和CachingMetadataReaderFactoryPostProcessor是、、//BeanDefinitionRegistryPostProcessor的子类  if (postProcessor instanceof BeanDefinitionRegistryPostProcessor) {  BeanDefinitionRegistryPostProcessor registryProcessor = (BeanDefinitionRegistryPostProcessor)postProcessor; //调用他们的 postProcessBeanDefinitionRegistry 方法：  // CachingMetadataReaderFactoryPostProcessor#postProcessBeanDefinitionRegistry方法干了两件事情：   1. 注入一个bean 2. 给ConfigurationClassPostProcessor对象的 metadataReaderFactory属性赋值   // ConfigurationWarningsPostProcessor#postProcessBeanDefinitionRegistry  registryProcessor.postProcessBeanDefinitionRegistry(registry);  registryProcessors.add(registryProcessor);  } else {  regularPostProcessors.add(postProcessor);  }  }   currentRegistryProcessors = new ArrayList();  //获取BeanDefinitionRegistryPostProcessor在spring容器中定义的实现类：ConfigurationClassPostProcessor(AnnotationConfigApplicationContext的构造方法-->AnnotatedBeanDefinitionReader的构造方法里把它塞入spring容器的)  postProcessorNames = beanFactory.getBeanNamesForType(BeanDefinitionRegistryPostProcessor.class, true, false);  String[] var16 = postProcessorNames;  var9 = postProcessorNames.length;   int var10;  String ppName;  for(var10 = 0; var10 < var9; ++var10) {  ppName = var16[var10];  // BeanDefinitionRegistryPostProcessor 且是PriorityOrdered接口的是实现类放入currentRegistryProcessors集合中  if (beanFactory.isTypeMatch(ppName, PriorityOrdered.class)) {  currentRegistryProcessors.add(beanFactory.getBean(ppName, BeanDefinitionRegistryPostProcessor.class));  processedBeans.add(ppName);//加入已处理的ban集合  }  }  //BeanDefinitionRegistryPostProcessor 且是PriorityOrdered接口的是实现类排序  *sortPostProcessors*(currentRegistryProcessors, beanFactory);  registryProcessors.addAll(currentRegistryProcessors);//加入集合中  //调用postProcessBeanDefinitionRegistry方法，这里核心是调用ConfigurationClassPostProcessor类的postProcessBeanDefinitionRegistry方法  *invokeBeanDefinitionRegistryPostProcessors*(currentRegistryProcessors, registry);  currentRegistryProcessors.clear();  } //调用postProcessBeanFactory方法，特别是调用ConfigurationClassPostProcessor类的postProcessBeanFactory方法  *invokeBeanFactoryPostProcessors*((Collection)registryProcessors, (ConfigurableListableBeanFactory)beanFactory); //调用*BeanFactoryPostProcessor且不是*BeanDefinitionRegistryPostProcessor接口的实现类的具体处理方法  *invokeBeanFactoryPostProcessors*((Collection)regularPostProcessors, (ConfigurableListableBeanFactory)beanFactory); } else {  *invokeBeanFactoryPostProcessors*((Collection)beanFactoryPostProcessors, (ConfigurableListableBeanFactory)beanFactory); }  //获取spring容器中BeanFactoryPostProcessor的实现beanName  String[] postProcessorNames = beanFactory.getBeanNamesForType(BeanFactoryPostProcessor.class, true, false);  regularPostProcessors = new ArrayList();  registryProcessors = new ArrayList();  currentRegistryProcessors = new ArrayList();  postProcessorNames = postProcessorNames;  int var20 = postProcessorNames.length;   String ppName;  for(var9 = 0; var9 < var20; ++var9) {  ppName = postProcessorNames[var9];  if (!processedBeans.contains(ppName)) {  //*BeanFactoryPostProcessor且是*PriorityOrdered接口的实现类放入集合regularPostProcessors中  if (beanFactory.isTypeMatch(ppName, PriorityOrdered.class)) {  regularPostProcessors.add(beanFactory.getBean(ppName, BeanFactoryPostProcessor.class));  //*BeanFactoryPostProcessor且是Ordered接口的实现类放入集合registryProcessors中*  } else if (beanFactory.isTypeMatch(ppName, Ordered.class)) {  registryProcessors.add(ppName);  } else {  //*BeanFactoryPostProcessor实现类没有实现Ordered的放入集合currentRegistryProcessors中*  currentRegistryProcessors.add(ppName);  }  }  } //*BeanFactoryPostProcessor且是*PriorityOrdered接口的实现类放入集合regularPostProcessors中：排序  *sortPostProcessors*(regularPostProcessors, beanFactory);  //调用集合regularPostProcessors集合中对象的postProcessBeanFactory  *invokeBeanFactoryPostProcessors*((Collection)regularPostProcessors, (ConfigurableListableBeanFactory)beanFactory);  List<BeanFactoryPostProcessor> orderedPostProcessors = new ArrayList(registryProcessors.size());  Iterator var21 = registryProcessors.iterator();   while(var21.hasNext()) {  String postProcessorName = (String)var21.next();  orderedPostProcessors.add(beanFactory.getBean(postProcessorName, BeanFactoryPostProcessor.class));  }  //对orderedPostProcessors集合元素排序  *sortPostProcessors*(orderedPostProcessors, beanFactory);  //调用集合orderedPostProcessors集合中对象的postProcessBeanFactory方法  *invokeBeanFactoryPostProcessors*((Collection)orderedPostProcessors, (ConfigurableListableBeanFactory)beanFactory);  List<BeanFactoryPostProcessor> nonOrderedPostProcessors = new ArrayList(currentRegistryProcessors.size());  Iterator var24 = currentRegistryProcessors.iterator();   while(var24.hasNext()) {  ppName = (String)var24.next();  nonOrderedPostProcessors.add(beanFactory.getBean(ppName, BeanFactoryPostProcessor.class));  }  //剩余无序的*BeanFactoryPostProcessor实现类调用*postProcessBeanFactory方法  *invokeBeanFactoryPostProcessors*((Collection)nonOrderedPostProcessors, (ConfigurableListableBeanFactory)beanFactory);  beanFactory.clearMetadataCache(); } |

上面讲了那么多代码，逻辑很简单，就是调用BeanFactoryPostProcessor实现类的处理逻辑，但是有些实现类又是BeanDefinitionRegistryPostProcessor类的实现类。所以大体分为两类：

1. 实现了BeanDefinitionRegistryPostProcessor接口(它继承了BeanFactoryPostProcessor接口)的实现类集合

* 1.1 执行传入的参数中BeanDefinitionRegistryPostProcessor接口的实现类postProcessBeanDefinitionRegistry方法；
* 1.2 获取所有类型是BeanDefinitionRegistryPostProcessor的spring容器中定义的bean；

\*\*实现PriorityOrdered接口的，进行排序，然后执行postProcessBeanDefinitionRegistry方法；

\*\*直接实现Ordered接口且没有实现PriorityOrdered接口的，进行排序，然后执行postProcessBeanDefinitionRegistry方法；

\*\* 执spring容器中的BeanDefinitionRegistryPostProcessor的spring容器中定义的bean，并且没有实现PriorityOrdered接口和Ordered接口的bean的postProcessBeanDefinitionRegistry方法；

\*\* 执行1.1和1.2获取BeanDefinitionRegistryPostProcessor实现类的postProcessBeanFactory方法；

1. 实现了BeanFactoryPostProcessor接口且没有实现BeanDefinitionRegistryPostProcessor的是实现类

* 执行传入参数的BeanFactoryPostProcessor实现类的postProcessBeanFactory方法；
* 获取spring容器中BeanFactoryPostProcessor类型的所有容器

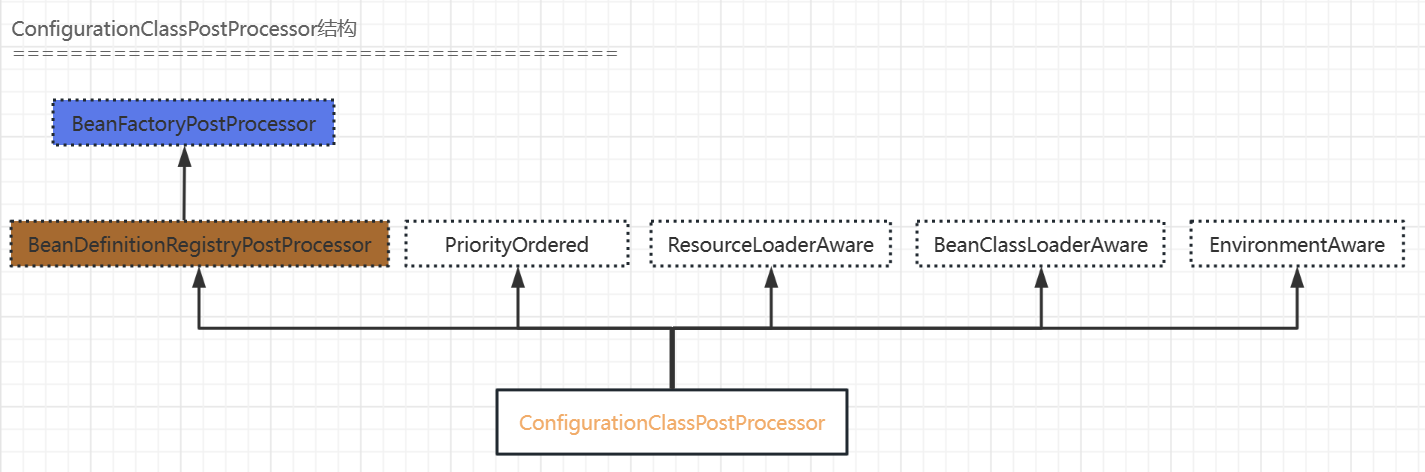
\*\*实现PriorityOrdered接口的，进行排序，然后执postProcessBeanFactory方法；

\*\*直接实现Ordered接口且没有实现PriorityOrdered接口的，进行排序，然后执行postProcessBeanFactory方法；

\*\* 执spring容器中的BeanFactoryPostProcessor的spring容器中定义的bean，并且没有实现PriorityOrdered接口和Ordered接口的bean的postProcessBeanFactory方法；

下面看下这里的一个特别的例子ConfigurationClassPostProcessor

### ConfigurationClassPostProcessor

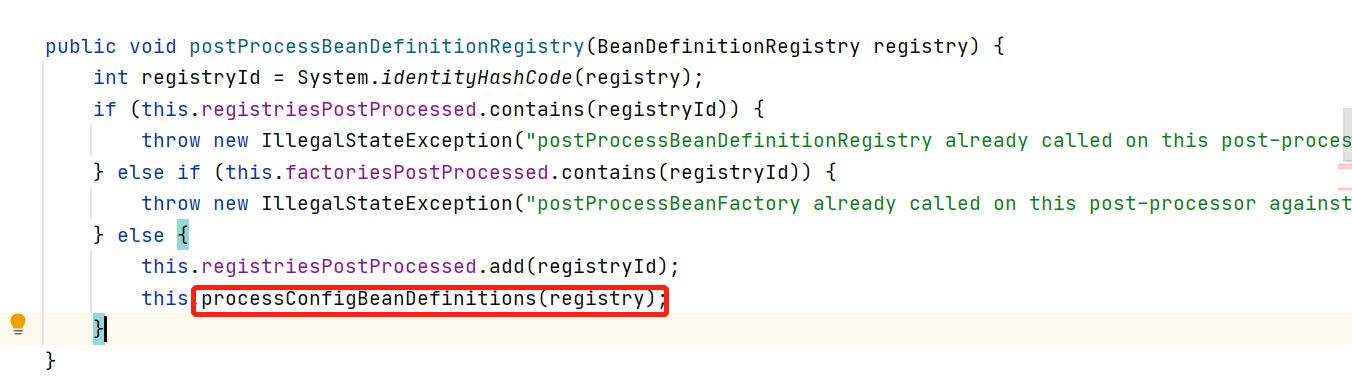


这个类在AnnotationConfigApplicationContext类的构造方法里，通过创建AnnotatedBeanDefinitionReader对象里的构造方法里，又调用AnnotationConfigUtils#registerAnnotationConfigProcessors方法，在这里把ConfigurationClassPostProcessor塞入bean容器中去。

通过对PostProcessorRegistrationDelegate#invokeBeanFactoryPostProcessors方法的分析，我们可以看出ConfigurationClassPostProcessor是BeanDefinitionRegistryPostProcessor接口的实现类，它先执行ConfigurationClassPostProcessor# postProcessBeanDefinitionRegistry方法，然后再执行ConfigurationClassPostProcessor#postProcessBeanFactory方法，下面我们看下ConfigurationClassPostProcessor类的这两个方法：

##### ConfigurationClassPostProcessor#processConfigBeanDefinitions

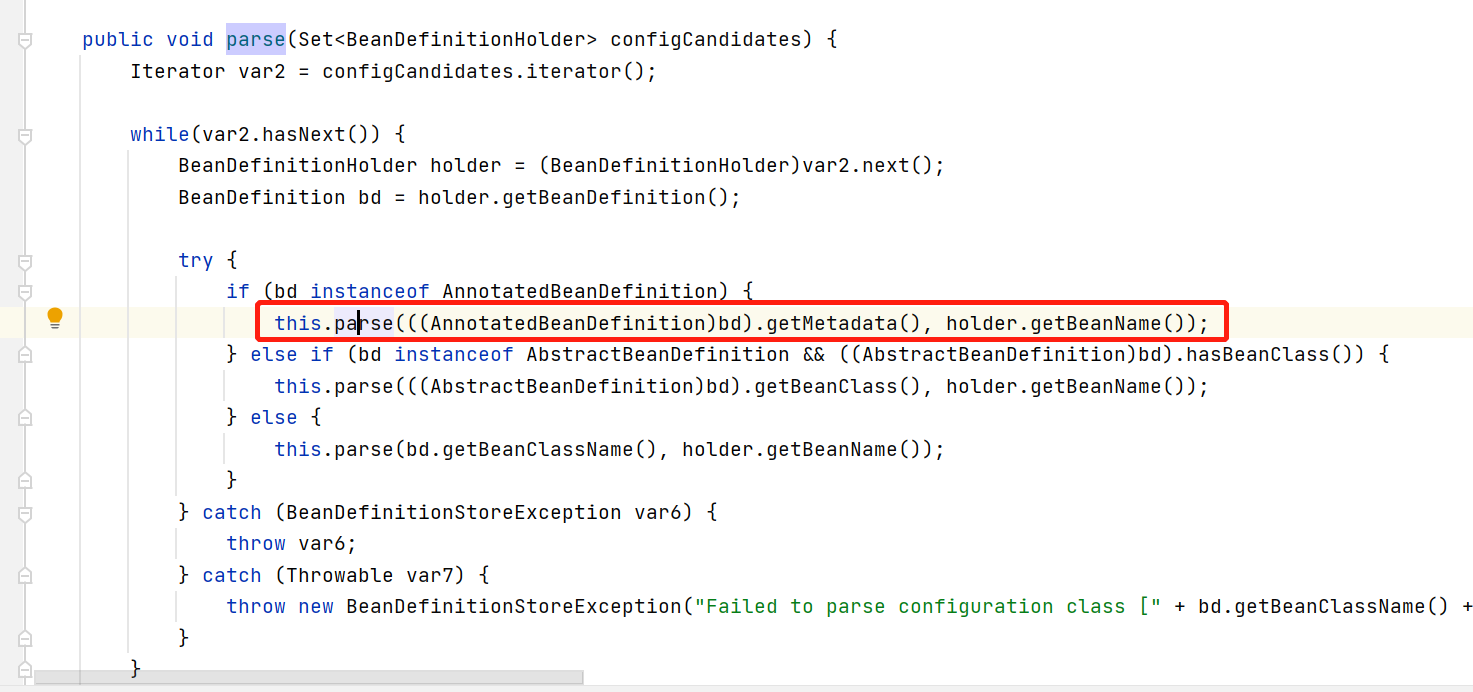
在ConfigurationClassPostProcessor#postProcessBeanDefinitionRegistry方法中，主要调用了processConfigBeanDefinitions方法：



下面看ConfigurationClassPostProcessor#processConfigBeanDefinitions方法：

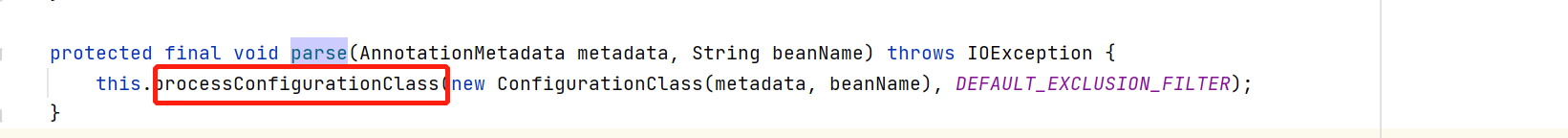
|  |
| --- |
| public void processConfigBeanDefinitions(BeanDefinitionRegistry registry) {  List<BeanDefinitionHolder> configCandidates = new ArrayList();  //1、 获取所有注册到spring容器的bean 名字  String[] candidateNames = registry.getBeanDefinitionNames();  String[] var4 = candidateNames;  int var5 = candidateNames.length;   for(int var6 = 0; var6 < var5; ++var6) {  String beanName = var4[var6];  BeanDefinition beanDef = registry.getBeanDefinition(beanName);  if (beanDef.getAttribute(ConfigurationClassUtils.*CONFIGURATION\_CLASS\_ATTRIBUTE*) != null) {  if (this.logger.isDebugEnabled()) {  this.logger.debug("Bean definition has already been processed as a configuration class: " + beanDef);  }  //2、检查是否是配置类，是的话，放入configCandidates集合中  } else if (ConfigurationClassUtils.*checkConfigurationClassCandidate*(beanDef, this.metadataReaderFactory)) {  configCandidates.add(new BeanDefinitionHolder(beanDef, beanName));  }  }  //3、若候选的配置类集合不为空，则继续进行处理  if (!configCandidates.isEmpty()) {  //3.1 排序  configCandidates.sort((bd1, bd2) -> {  int i1 = ConfigurationClassUtils.*getOrder*(bd1.getBeanDefinition());  int i2 = ConfigurationClassUtils.*getOrder*(bd2.getBeanDefinition());  return Integer.*compare*(i1, i2);  });  SingletonBeanRegistry sbr = null;  if (registry instanceof SingletonBeanRegistry) {  sbr = (SingletonBeanRegistry)registry;  if (!this.localBeanNameGeneratorSet) {  BeanNameGenerator generator = (BeanNameGenerator)sbr.getSingleton("org.springframework.context.annotation.internalConfigurationBeanNameGenerator");  if (generator != null) {  this.componentScanBeanNameGenerator = generator;  this.importBeanNameGenerator = generator;  }  }  }   if (this.environment == null) {  this.environment = new StandardEnvironment();  }  //3.2 创建候选配置类的解析类对象  ConfigurationClassParser parser = new ConfigurationClassParser(this.metadataReaderFactory, this.problemReporter, this.environment, this.resourceLoader, this.componentScanBeanNameGenerator, registry);  Set<BeanDefinitionHolder> candidates = new LinkedHashSet(configCandidates);  HashSet alreadyParsed = new HashSet(configCandidates.size());  //3.3 循环解析  do {  parser.parse(candidates); //解析配置类  parser.validate();  // 获取解析了哪些配置类  Set<ConfigurationClass> configClasses = new LinkedHashSet(parser.getConfigurationClasses());  configClasses.removeAll(alreadyParsed);  if (this.reader == null) {  this.reader = new ConfigurationClassBeanDefinitionReader(registry, this.sourceExtractor, this.resourceLoader, this.environment, this.importBeanNameGenerator, parser.getImportRegistry());  }  //加载到spring容器中  this.reader.loadBeanDefinitions(configClasses);  alreadyParsed.addAll(configClasses); //记录下哪些配置类已经被解析并且加载到spring容器中  candidates.clear();  //说明spring容器中有新的bean加入了，下面判断是否有新的配置类还没有被解析，如有的话继续走一遍do【】循环的操作  if (registry.getBeanDefinitionCount() > candidateNames.length) {  String[] newCandidateNames = registry.getBeanDefinitionNames();  Set<String> oldCandidateNames = new HashSet(Arrays.*asList*(candidateNames));  Set<String> alreadyParsedClasses = new HashSet();  Iterator var12 = alreadyParsed.iterator();   while(var12.hasNext()) {  ConfigurationClass configurationClass = (ConfigurationClass)var12.next();  alreadyParsedClasses.add(configurationClass.getMetadata().getClassName());  }   String[] var23 = newCandidateNames;  int var24 = newCandidateNames.length;   for(int var14 = 0; var14 < var24; ++var14) {  String candidateName = var23[var14];  if (!oldCandidateNames.contains(candidateName)) {  BeanDefinition bd = registry.getBeanDefinition(candidateName);  if (ConfigurationClassUtils.*checkConfigurationClassCandidate*(bd, this.metadataReaderFactory) && !alreadyParsedClasses.contains(bd.getBeanClassName())) {  candidates.add(new BeanDefinitionHolder(bd, candidateName));  }  }  }   candidateNames = newCandidateNames;  }  } while(!candidates.isEmpty());   if (sbr != null && !sbr.containsSingleton(*IMPORT\_REGISTRY\_BEAN\_NAME*)) {  sbr.registerSingleton(*IMPORT\_REGISTRY\_BEAN\_NAME*, parser.getImportRegistry());  }   if (this.metadataReaderFactory instanceof CachingMetadataReaderFactory) {  ((CachingMetadataReaderFactory)this.metadataReaderFactory).clearCache();  }   } } |

###### ConfigurationClassParser#parse

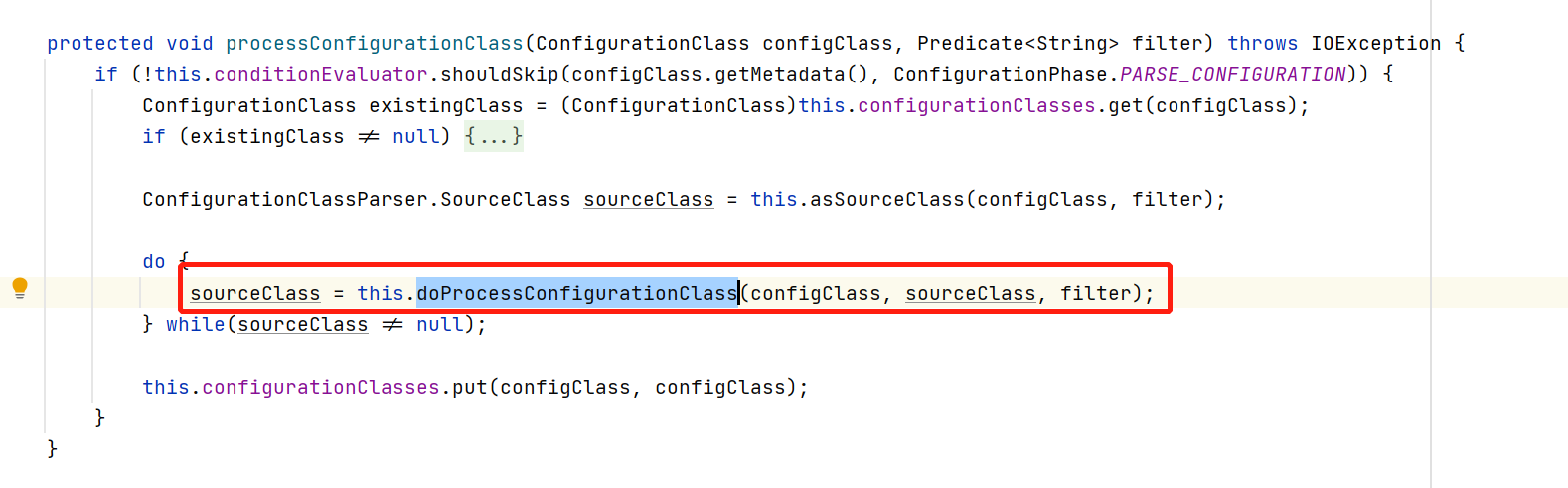


上面对候选配置类进行循环遍历，然后逐一进行解析。

下面我们看ConfigurationClassParser#parse(AnnotationMetadata, String)方法：



转调用processConfigurationClass方法：



下面看下doProcessConfigurationClass方法

|  |
| --- |
| protected final ConfigurationClassParser.SourceClass doProcessConfigurationClass(ConfigurationClass configClass, ConfigurationClassParser.SourceClass sourceClass, Predicate<String> filter) throws IOException { //1、解析@Component注解  if (configClass.getMetadata().isAnnotated(Component.class.getName())) {  this.processMemberClasses(configClass, sourceClass, filter);  }  //2、解析@PropertySources注解  Iterator var4 = AnnotationConfigUtils.*attributesForRepeatable*(sourceClass.getMetadata(), PropertySources.class, PropertySource.class).iterator();   AnnotationAttributes importResource;  while(var4.hasNext()) {  importResource = (AnnotationAttributes)var4.next();  if (this.environment instanceof ConfigurableEnvironment) {  this.processPropertySource(importResource);  } else {  this.logger.info("Ignoring @PropertySource annotation on [" + sourceClass.getMetadata().getClassName() + "]. Reason: Environment must implement ConfigurableEnvironment");  }  } //3、解析@ComponentScans注解  Set<AnnotationAttributes> componentScans = AnnotationConfigUtils.*attributesForRepeatable*(sourceClass.getMetadata(), ComponentScans.class, ComponentScan.class);  if (!componentScans.isEmpty() && !this.conditionEvaluator.shouldSkip(sourceClass.getMetadata(), ConfigurationPhase.*REGISTER\_BEAN*)) {  Iterator var14 = componentScans.iterator();   while(var14.hasNext()) {  AnnotationAttributes componentScan = (AnnotationAttributes)var14.next();  Set<BeanDefinitionHolder> scannedBeanDefinitions = this.componentScanParser.parse(componentScan, sourceClass.getMetadata().getClassName());  Iterator var8 = scannedBeanDefinitions.iterator();   while(var8.hasNext()) {  BeanDefinitionHolder holder = (BeanDefinitionHolder)var8.next();  BeanDefinition bdCand = holder.getBeanDefinition().getOriginatingBeanDefinition();  if (bdCand == null) {  bdCand = holder.getBeanDefinition();  }   if (ConfigurationClassUtils.*checkConfigurationClassCandidate*(bdCand, this.metadataReaderFactory)) {  this.parse(bdCand.getBeanClassName(), holder.getBeanName());  }  }  }  }  //4、处理@Import注解，获取该注解的代码是getImports方法  this.processImports(configClass, sourceClass, this.getImports(sourceClass), filter, true);  //5、处理@ImportResource注解  importResource = AnnotationConfigUtils.*attributesFor*(sourceClass.getMetadata(), ImportResource.class);  if (importResource != null) {  String[] resources = importResource.getStringArray("locations");  Class<? extends BeanDefinitionReader> readerClass = importResource.getClass("reader");  String[] var20 = resources;  int var22 = resources.length;   for(int var23 = 0; var23 < var22; ++var23) {  String resource = var20[var23];  String resolvedResource = this.environment.resolveRequiredPlaceholders(resource);  configClass.addImportedResource(resolvedResource, readerClass);  }  }  //6、处理@Bean注解  Set<MethodMetadata> beanMethods = this.retrieveBeanMethodMetadata(sourceClass);  Iterator var18 = beanMethods.iterator();   while(var18.hasNext()) {  MethodMetadata methodMetadata = (MethodMetadata)var18.next();  configClass.addBeanMethod(new BeanMethod(methodMetadata, configClass));  }    this.processInterfaces(configClass, sourceClass);  if (sourceClass.getMetadata().hasSuperClass()) {  String superclass = sourceClass.getMetadata().getSuperClassName();  if (superclass != null && !superclass.startsWith("java") && !this.knownSuperclasses.containsKey(superclass)) {  this.knownSuperclasses.put(superclass, configClass);  return sourceClass.getSuperClass();  }  }   return null; } |

以上方法主要处理：

1、解析@Component注解；

2、解析@PropertySources注解；

3、解析@ComponentScans注解；

4、处理@Import注解；

5、处理@ImportResource注解；

6、处理@Bean注解；

下面我们详细看下处理@ Import注解的方法：调用的是ConfigurationClassParser#processImports方法

###### @Import注解

提供了三种用法

1、@Import一个类 该类实现了**ImportSelector** 重写selectImports方法该方法返回了String[]数组的对象，数组里面的类都会注入到spring容器当中;

2、@Import一个类，该类实现了**ImportBeanDefinitionRegistrar**接口，在重写的registerBeanDefinitions方法里面，能拿到BeanDefinitionRegistry bd的注册器，能手工往beanDefinitionMap中注册 beanDefinition;

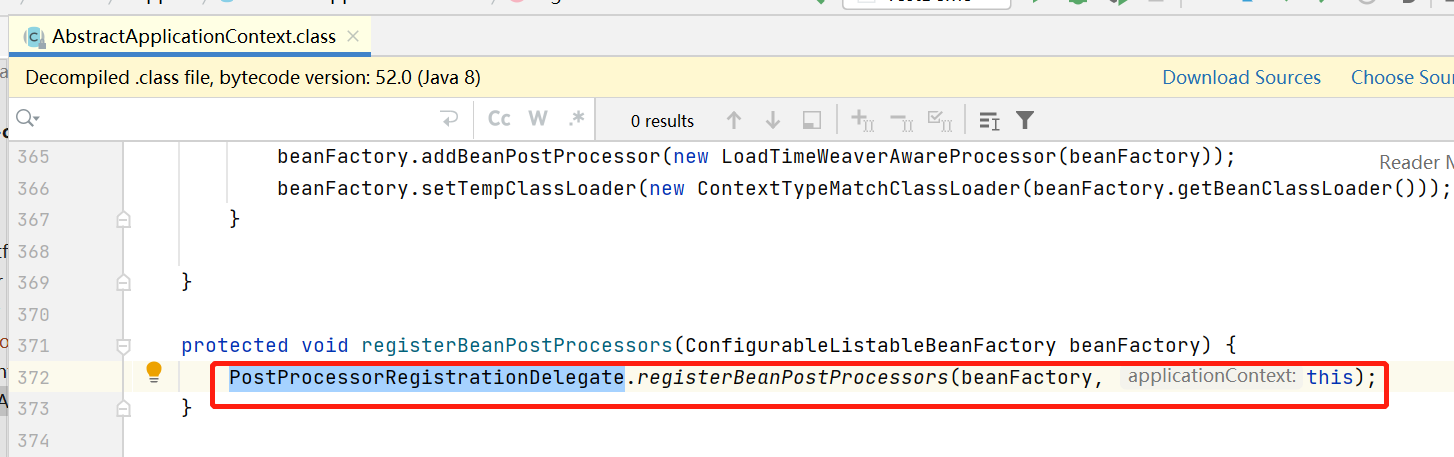
3、@Import一个普通类 spring会将该类加载到spring容器中;

|  |
| --- |
| private void processImports(ConfigurationClass configClass, ConfigurationClassParser.SourceClass currentSourceClass, Collection<ConfigurationClassParser.SourceClass> importCandidates, Predicate<String> exclusionFilter, boolean checkForCircularImports) {  if (!importCandidates.isEmpty()) {  if (checkForCircularImports && this.isChainedImportOnStack(configClass)) {  this.problemReporter.error(new ConfigurationClassParser.CircularImportProblem(configClass, this.importStack));  } else {  this.importStack.push(configClass);   try {  Iterator var6 = importCandidates.iterator();   while(var6.hasNext()) {  ConfigurationClassParser.SourceClass candidate = (ConfigurationClassParser.SourceClass)var6.next();  Class candidateClass;  //1、若是ImportSelector的实现类  if (candidate.isAssignable(ImportSelector.class)) {  candidateClass = candidate.loadClass();  ImportSelector selector = (ImportSelector)ParserStrategyUtils.*instantiateClass*(candidateClass, ImportSelector.class, this.environment, this.resourceLoader, this.registry);  Predicate<String> selectorFilter = selector.getExclusionFilter();  if (selectorFilter != null) {  exclusionFilter = exclusionFilter.or(selectorFilter);  }   if (selector instanceof DeferredImportSelector) {  this.deferredImportSelectorHandler.handle(configClass, (DeferredImportSelector)selector);  } else {  String[] importClassNames = selector.selectImports(currentSourceClass.getMetadata());  Collection<ConfigurationClassParser.SourceClass> importSourceClasses = this.asSourceClasses(importClassNames, exclusionFilter);  this.processImports(configClass, currentSourceClass, importSourceClasses, exclusionFilter, false);  }  } else  //2、若是ImportBeanDefinitionRegistrar的实现类  if (candidate.isAssignable(ImportBeanDefinitionRegistrar.class)) {  candidateClass = candidate.loadClass();  ImportBeanDefinitionRegistrar registrar = (ImportBeanDefinitionRegistrar)ParserStrategyUtils.*instantiateClass*(candidateClass, ImportBeanDefinitionRegistrar.class, this.environment, this.resourceLoader, this.registry);  configClass.addImportBeanDefinitionRegistrar(registrar, currentSourceClass.getMetadata());  } else {  //3、其他情况  this.importStack.registerImport(currentSourceClass.getMetadata(), candidate.getMetadata().getClassName());  this.processConfigurationClass(candidate.asConfigClass(configClass), exclusionFilter);  }  }  } catch (BeanDefinitionStoreException var17) {  throw var17;  } catch (Throwable var18) {  throw new BeanDefinitionStoreException("Failed to process import candidates for configuration class [" + configClass.getMetadata().getClassName() + "]", var18);  } finally {  this.importStack.pop();  }  }   } } |

##### ConfigurationClassPostProcessor#postProcessBeanFactory

## 2.6 AbstractApplicationContext#registerBeanPostProcessors

~~实现已经在spring容器中存在~~**~~BeanPostProcessor~~**~~实现类的逻辑：~~



调用的是PostProcessorRegistrationDelegate#registerBeanPostProcessors方法：

|  |
| --- |
| public static void registerBeanPostProcessors(ConfigurableListableBeanFactory beanFactory, AbstractApplicationContext applicationContext) {  //1、获取目前spring容器中所有BeanPostProcessor实现类  String[] postProcessorNames = beanFactory.getBeanNamesForType(BeanPostProcessor.class, true, false);  int beanProcessorTargetCount = beanFactory.getBeanPostProcessorCount() + 1 + postProcessorNames.length;  beanFactory.addBeanPostProcessor(new PostProcessorRegistrationDelegate.BeanPostProcessorChecker(beanFactory, beanProcessorTargetCount));  //保存实现了 PriorityOrdered、BeanPostProcessor的实现类  List<BeanPostProcessor> priorityOrderedPostProcessors = new ArrayList();  //记录又实现了 MergedBeanDefinitionPostProcessor、BeanPostProcessor的实现类  List<BeanPostProcessor> internalPostProcessors = new ArrayList();  //保存实现了 Ordered、BeanPostProcessor的实现类  List<String> orderedPostProcessorNames = new ArrayList();  //记录无序的BeanPostProcessor的实现类  List<String> nonOrderedPostProcessorNames = new ArrayList();  String[] var8 = postProcessorNames;  int var9 = postProcessorNames.length;   String ppName;  BeanPostProcessor pp;  for(int var10 = 0; var10 < var9; ++var10) {  ppName = var8[var10];  if (beanFactory.isTypeMatch(ppName, PriorityOrdered.class)) {  pp = (BeanPostProcessor)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);  }  }  //对PriorityOrdered、BeanPostProcessor的实现类集合排序  *sortPostProcessors*(priorityOrderedPostProcessors, beanFactory);  //放入beanFactory的List<BeanPostProcessor> beanPostProcessors集合中去  *registerBeanPostProcessors*(beanFactory, (List)priorityOrderedPostProcessors);  List<BeanPostProcessor> orderedPostProcessors = new ArrayList(orderedPostProcessorNames.size());  Iterator var14 = orderedPostProcessorNames.iterator();   while(var14.hasNext()) {  String ppName = (String)var14.next();  BeanPostProcessor pp = (BeanPostProcessor)beanFactory.getBean(ppName, BeanPostProcessor.class);  orderedPostProcessors.add(pp);  if (pp instanceof MergedBeanDefinitionPostProcessor) {  internalPostProcessors.add(pp);  }  }   *sortPostProcessors*(orderedPostProcessors, beanFactory);  *registerBeanPostProcessors*(beanFactory, (List)orderedPostProcessors);  List<BeanPostProcessor> nonOrderedPostProcessors = new ArrayList(nonOrderedPostProcessorNames.size());  Iterator var17 = nonOrderedPostProcessorNames.iterator();   while(var17.hasNext()) {  ppName = (String)var17.next();  pp = (BeanPostProcessor)beanFactory.getBean(ppName, BeanPostProcessor.class);  nonOrderedPostProcessors.add(pp);  if (pp instanceof MergedBeanDefinitionPostProcessor) {  internalPostProcessors.add(pp);  }  }   *registerBeanPostProcessors*(beanFactory, (List)nonOrderedPostProcessors);  *sortPostProcessors*(internalPostProcessors, beanFactory);  *registerBeanPostProcessors*(beanFactory, (List)internalPostProcessors);  beanFactory.addBeanPostProcessor(new ApplicationListenerDetector(applicationContext)); } |

以上代码：

1、获取目前容器所有的BeanPostProcessor实现类

2、把又实现了PriorityOrdered接口的放入一个集合中；

3、把又实现了Ordered接口的放入一个集合中；

4、无序的放入一个集合中；

5、把2中的集合排序，然后放入beanFactory的List<BeanPostProcessor> beanPostProcessors集合中去；

6、把3的集合排序，然后放入beanFactory的List<BeanPostProcessor> beanPostProcessors集合中去；

7、把4的集合放入beanFactory的List<BeanPostProcessor> beanPostProcessors集合中去；

### AutowiredAnnotationBeanPostProcessor

在AnnotatedBeanDefinitionReader的构造方法里通过AnnotationConfigUtils#registerAnnotationConfigProcessors(BeanDefinitionRegistry)赋值；

**主要解析@Autowired、@Value注解**

****

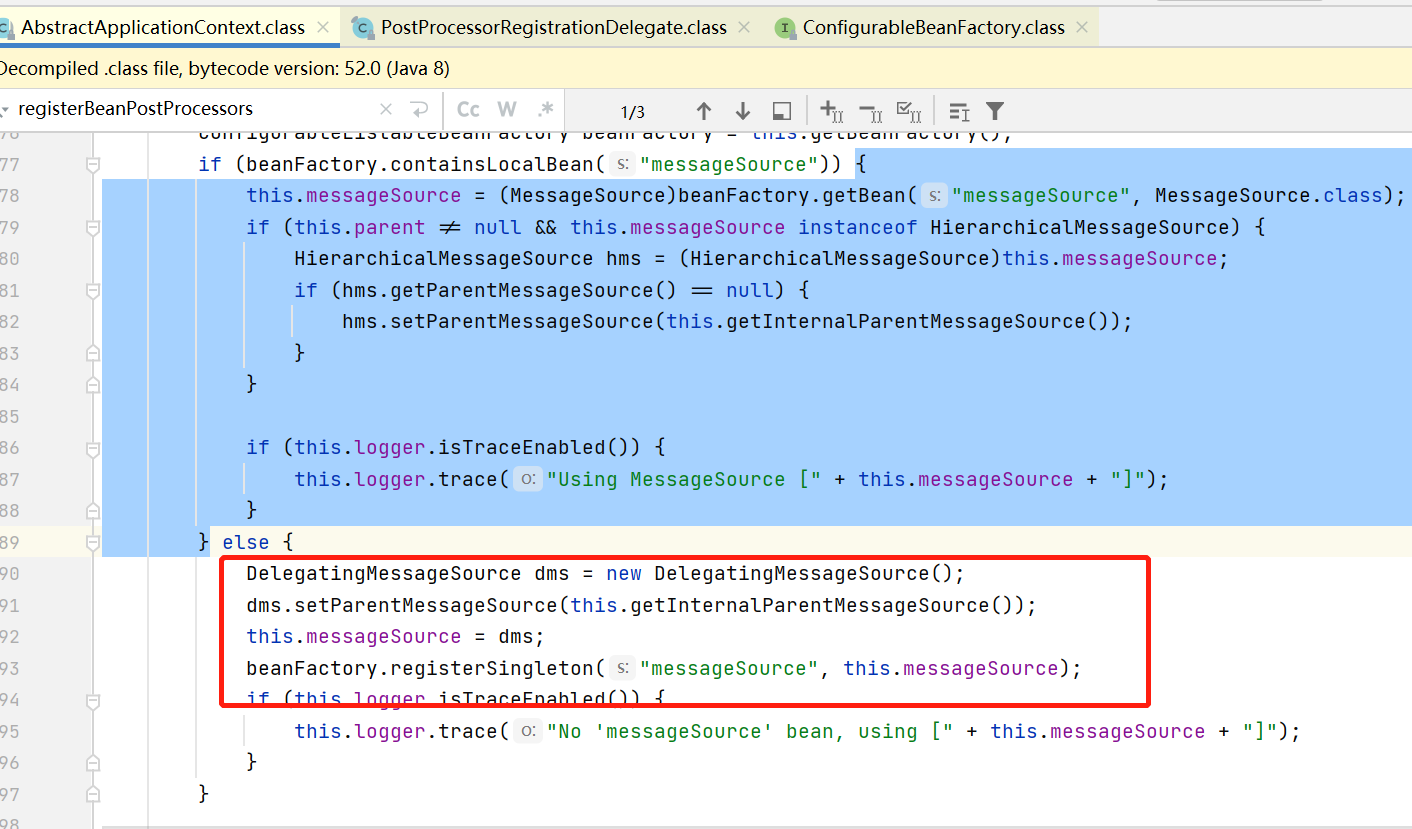
### CommonAnnotationBeanPostProcessor

在AnnotatedBeanDefinitionReader的构造方法里通过AnnotationConfigUtils#registerAnnotationConfigProcessors(BeanDefinitionRegistry)赋值；

主要解析@ PostConstruct和@ PreDestroy注解。

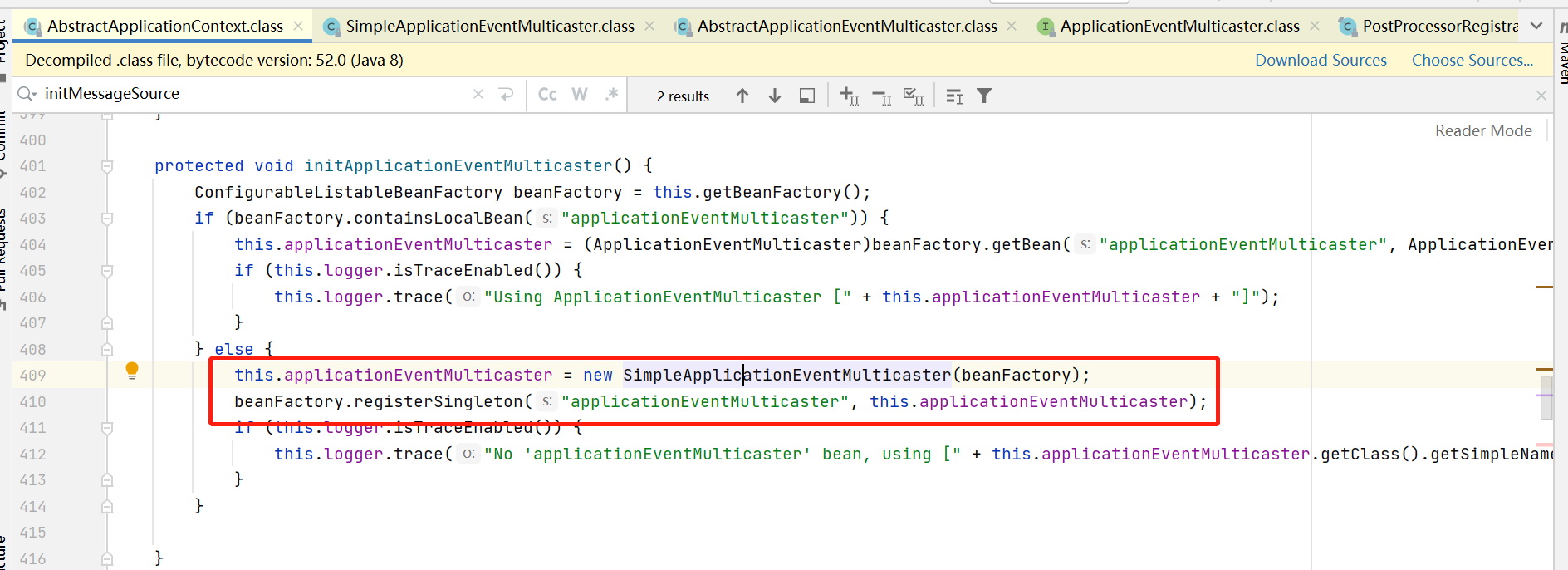
## 2.7 AbstractApplicationContext#initMessageSource

国际化；



## 2.8 AbstractApplicationContext#initApplicationEventMulticaster

创建事件监听器的启动类，也就是ApplicationEventMulticaster的实现类，把它加入spring容器中去。



## 2.9 AbstractApplicationContext#onRefresh

子类可自行实现，像在spring-boot里内嵌的有tomcat，tomcat服务的启动就是在这个方法里实现的。

## 2.10AbstractApplicationContext#registerListeners

调用各个事件监听器

|  |
| --- |
| protected void registerListeners() {  //获取本类中 Set<ApplicationListener<?>> applicationListeners属性值  Iterator var1 = this.getApplicationListeners().iterator();   while(var1.hasNext()) {  ApplicationListener<?> listener = (ApplicationListener)var1.next();  // 放入八中创建的SimpleApplicationEventMulticaster的属性defaultRetriever.applicationListenerBeans 中  this.getApplicationEventMulticaster().addApplicationListener(listener);  }  //获取容器中所有ApplicationListener实现  String[] listenerBeanNames = this.getBeanNamesForType(ApplicationListener.class, true, false);  String[] var7 = listenerBeanNames;  int var3 = listenerBeanNames.length;   for(int var4 = 0; var4 < var3; ++var4) {  String listenerBeanName = var7[var4];  //放入八中创建的SimpleApplicationEventMulticaster的属性defaultRetriever.applicationListenerBeans 中  this.getApplicationEventMulticaster().addApplicationListenerBean(listenerBeanName);  }   Set<ApplicationEvent> earlyEventsToProcess = this.earlyApplicationEvents;  this.earlyApplicationEvents = null;  if (!CollectionUtils.*isEmpty*(earlyEventsToProcess)) {  Iterator var9 = earlyEventsToProcess.iterator();   while(var9.hasNext()) {  ApplicationEvent earlyEvent = (ApplicationEvent)var9.next();  //如果earlyApplicationEvents有值，执行对应事件监听器的逻辑。  this.getApplicationEventMulticaster().multicastEvent(earlyEvent);  }  }  } |

## 2.11 AbstractApplicationContext#finishBeanFactoryInitialization

这里主要完成bean的实例化操作：

|  |
| --- |
| protected void finishBeanFactoryInitialization(ConfigurableListableBeanFactory beanFactory) {  if (beanFactory.containsBean("conversionService") && beanFactory.isTypeMatch("conversionService", ConversionService.class)) {  beanFactory.setConversionService((ConversionService)beanFactory.getBean("conversionService", ConversionService.class));  }   if (!beanFactory.hasEmbeddedValueResolver()) {  beanFactory.addEmbeddedValueResolver((strVal) -> {  return this.getEnvironment().resolvePlaceholders(strVal);  });  }   String[] weaverAwareNames = beanFactory.getBeanNamesForType(LoadTimeWeaverAware.class, false, false);  String[] var3 = weaverAwareNames;  int var4 = weaverAwareNames.length;   for(int var5 = 0; var5 < var4; ++var5) {  String weaverAwareName = var3[var5];  this.getBean(weaverAwareName);  }   beanFactory.setTempClassLoader((ClassLoader)null);  //把spring容器中需要实例化的放入一个临时变量保存  beanFactory.freezeConfiguration();  //实例化spring容器中所有待实例化的类———DefaultListableBeanFactory 中List<String> beanDefinitionNames属性值  beanFactory.preInstantiateSingletons(); } |

下面我们继续看spring实例化对象的过程是如何做的：

### DefaultListableBeanFactory#preInstantiateSingletons

1. 实例化对象；
2. 对于实现了SmartInitializingSingleton接口的实例，调用它的afterSingletonsInstantiated方法：



下面我们看下核心的点：创建实例化过程。



要求：

1. 不能是抽象类；
2. 不能是非单例的；
3. 不能是懒加载的；

满足以上条件的再次分类处理：

1. 实现了FactoryBean的类；
2. 未实现FactoryBean的类；

下面我们看实例化对象的过程，调用的AbstractBeanFactory类的getBean(string beanName)方法：



继续看AbstractBeanFactory#doGetBean方法：

## 2.12 AbstractApplicationContext#finishRefresh

