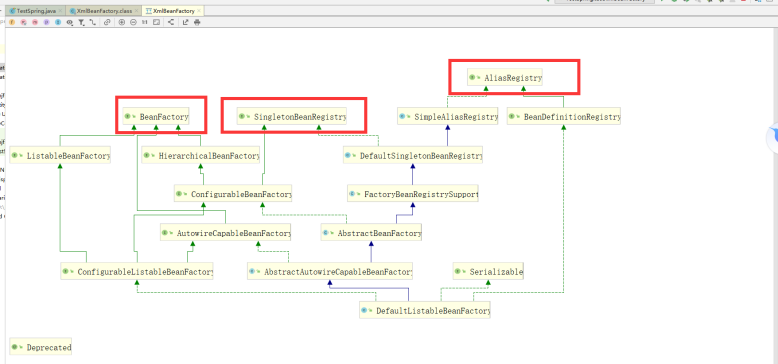
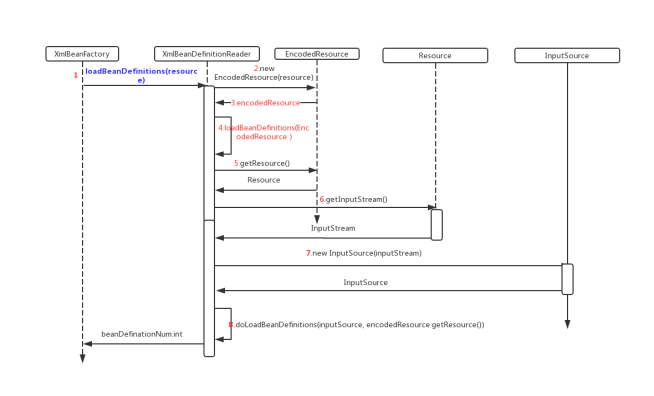
spring源码剖析

# BeanFactory

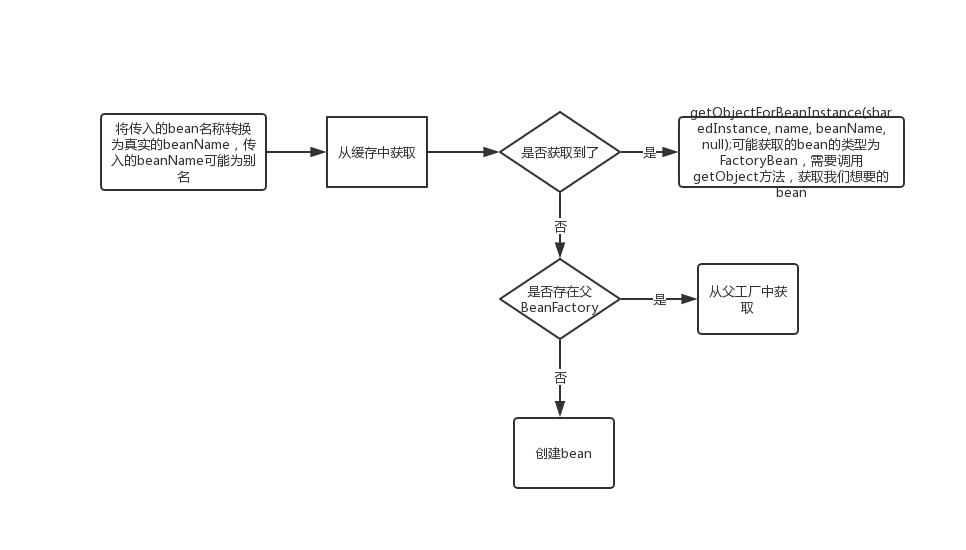
## 类图



## 时序图

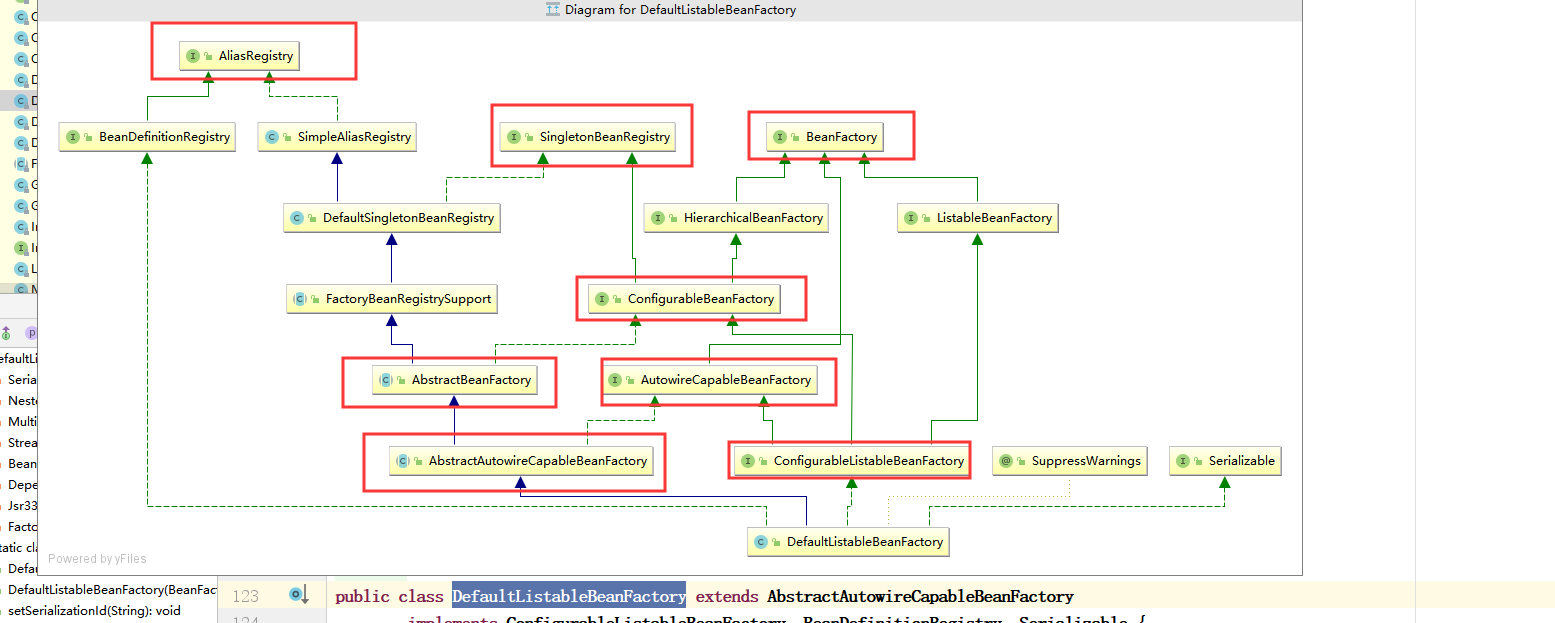


## getBean的流程图



## DefaultListableBeanFactory

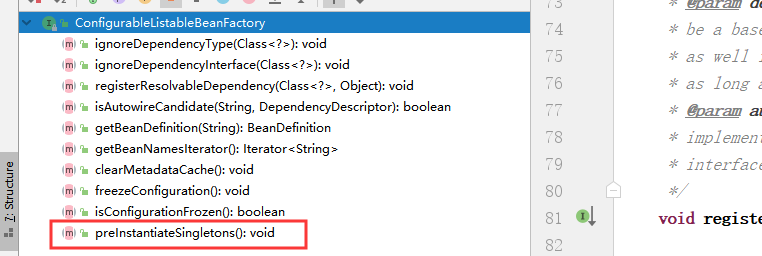
### 类图



### preInstantiateSingletons

|  |
| --- |
| @Override **public void** preInstantiateSingletons() **throws** BeansException {  **if** (**logger**.isTraceEnabled()) {  **logger**.trace(**"Pre-instantiating singletons in "** + **this**);  }   *// Iterate over a copy to allow for init methods which in turn register new bean definitions.  // While this may not be part of the regular factory bootstrap, it does otherwise work fine.* List<String> beanNames = **new** ArrayList<>(**this**.**beanDefinitionNames**);   *// Trigger initialization of all non-lazy singleton beans...* **for** (String beanName : beanNames) {  RootBeanDefinition bd = getMergedLocalBeanDefinition(beanName);  **if** (!bd.isAbstract() && bd.isSingleton() && !bd.isLazyInit()) {  **if** (isFactoryBean(beanName)) {  Object bean = getBean(***FACTORY\_BEAN\_PREFIX*** + beanName);  **if** (bean **instanceof** FactoryBean) {  **final** FactoryBean<?> factory = (FactoryBean<?>) bean;  **boolean** isEagerInit;  **if** (System.*getSecurityManager*() != **null** && factory **instanceof** SmartFactoryBean) {  isEagerInit = AccessController.*doPrivileged*((PrivilegedAction<Boolean>)  ((SmartFactoryBean<?>) factory)::isEagerInit,  getAccessControlContext());  }  **else** {  isEagerInit = (factory **instanceof** SmartFactoryBean &&  ((SmartFactoryBean<?>) factory).isEagerInit());  }  **if** (isEagerInit) {  getBean(beanName);  }  }  }  **else** {  getBean(beanName);  }  }  }   *// Trigger post-initialization callback for all applicable beans...* **for** (String beanName : beanNames) {  Object singletonInstance = getSingleton(beanName);  **if** (singletonInstance **instanceof** SmartInitializingSingleton) {  **final** SmartInitializingSingleton smartSingleton = (SmartInitializingSingleton) singletonInstance;  **if** (System.*getSecurityManager*() != **null**) {  AccessController.*doPrivileged*((PrivilegedAction<Object>) () -> {  smartSingleton.afterSingletonsInstantiated();  **return null**;  }, getAccessControlContext());  }  **else** {  smartSingleton.afterSingletonsInstantiated();  }  }  } } |

## ConfigurableListableBeanFactory



# 创建单例bean

## DefaultSingletonBeanRegistry

|  |
| --- |
| public Object getSingleton(String beanName, ObjectFactory<?> singletonFactory) {  Assert.notNull(beanName, "Bean name must not be null");  synchronized (this.singletonObjects) {  Object singletonObject = this.singletonObjects.get(beanName);  if (singletonObject == null) {  if (this.singletonsCurrentlyInDestruction) {  throw new BeanCreationNotAllowedException(beanName,  "Singleton bean creation not allowed while singletons of this factory are in destruction " +  "(Do not request a bean from a BeanFactory in a destroy method implementation!)");  }  if (logger.isDebugEnabled()) {  logger.debug("Creating shared instance of singleton bean '" + beanName + "'");  }  beforeSingletonCreation(beanName);  boolean newSingleton = false;  boolean recordSuppressedExceptions = (this.suppressedExceptions == null);  if (recordSuppressedExceptions) {  this.suppressedExceptions = new LinkedHashSet<>();  }  try {  singletonObject = singletonFactory.getObject();  newSingleton = true;  }  catch (IllegalStateException ex) {  // Has the singleton object implicitly appeared in the meantime ->  // if yes, proceed with it since the exception indicates that state.  singletonObject = this.singletonObjects.get(beanName);  if (singletonObject == null) {  throw ex;  }  }  catch (BeanCreationException ex) {  if (recordSuppressedExceptions) {  for (Exception suppressedException : this.suppressedExceptions) {  ex.addRelatedCause(suppressedException);  }  }  throw ex;  }  finally {  if (recordSuppressedExceptions) {  this.suppressedExceptions = null;  }  afterSingletonCreation(beanName);  }  if (newSingleton) {  addSingleton(beanName, singletonObject);  }  }  return singletonObject;  }  } |



## 存储bean

|  |
| --- |
| ***/\*\*  \* Add the given singleton object to the singleton cache of this factory.  \* <p>To be called for eager registration of singletons.  \* @param beanName the name of the bean  \* @param singletonObject the singleton object  \*/* protected void** addSingleton(**String beanName**, **Object singletonObject**) {  **synchronized** (**this**.**singletonObjects**) {  **this**.**singletonObjects**.**put**(**beanName**, **singletonObject**);  **this**.**singletonFactories**.**remove**(**beanName**);  **this**.**earlySingletonObjects**.**remove**(**beanName**);  **this**.**registeredSingletons**.**add**(**beanName**);  } } |

## 存储单例的map



# AbstractAutowireCapableBeanFactory

## createBean

|  |
| --- |
| ***/\*\*  \* Central method of this class: creates a bean instance,  \* populates the bean instance, applies post-processors, etc.  \* @see #*doCreateBean  *\*/* @Override protected Object** createBean(**String beanName**, **RootBeanDefinition mbd**, **@Nullable Object**[] **args**)  **throws BeanCreationException** {   **if** (**logger**.**isTraceEnabled**()) {  **logger**.**trace**(**"Creating instance of bean '" + beanName + "'"**);  }  **RootBeanDefinition mbdToUse = mbd**;   *// Make sure bean class is actually resolved at this point, and  // clone the bean definition in case of a dynamically resolved Class  // which cannot be stored in the shared merged bean definition.* **Class<?> resolvedClass = resolveBeanClass**(**mbd**, **beanName**);  **if** (**resolvedClass != null && !mbd**.**hasBeanClass**() **&& mbd**.**getBeanClassName**() **!= null**) {  **mbdToUse = new RootBeanDefinition**(**mbd**);  **mbdToUse**.**setBeanClass**(**resolvedClass**);  }   *// Prepare method overrides.* **try** {  **mbdToUse**.**prepareMethodOverrides**();  }  **catch** (**BeanDefinitionValidationException ex**) {  **throw new BeanDefinitionStoreException**(**mbdToUse**.**getResourceDescription**(),  **beanName**, **"Validation of method overrides failed"**, **ex**);  }   **try** {  *// Give BeanPostProcessors a chance to return a proxy instead of the target bean instance.* **Object bean = resolveBeforeInstantiation**(**beanName**, **mbdToUse**);  **if** (**bean != null**) {  **return bean**;  }  }  **catch** (**Throwable ex**) {  **throw new BeanCreationException**(**mbdToUse**.**getResourceDescription**(), **beanName**,  **"BeanPostProcessor before instantiation of bean failed"**, **ex**);  }   **try** {  // 真正的创建bean  **Object beanInstance = doCreateBean(beanName, mbdToUse, args);**  **if** (**logger**.**isTraceEnabled**()) {  **logger**.**trace**(**"Finished creating instance of bean '" + beanName + "'"**);  }  **return beanInstance**;  }  **catch** (**BeanCreationException | ImplicitlyAppearedSingletonException ex**) {  *// A previously detected exception with proper bean creation context already,  // or illegal singleton state to be communicated up to DefaultSingletonBeanRegistry.* **throw ex**;  }  **catch** (**Throwable ex**) {  **throw new BeanCreationException**(  **mbdToUse**.**getResourceDescription**(), **beanName**, **"Unexpected exception during bean creation"**, **ex**);  } } |

## doCreateBean

|  |
| --- |
| ***/\*\*  \* Actually create the specified bean. Pre-creation processing has already happened  \* at this point, e.g. checking {@code postProcessBeforeInstantiation} callbacks.  \* <p>Differentiates between default bean instantiation, use of a  \* factory method, and autowiring a constructor.  \* @param beanName the name of the bean  \* @param mbd the merged bean definition for the bean  \* @param args explicit arguments to use for constructor or factory method invocation  \* @return a new instance of the bean  \* @throws* BeanCreationException *if the bean could not be created  \* @see #*instantiateBean  *\* @see #*instantiateUsingFactoryMethod  *\* @see #*autowireConstructor  *\*/* protected Object** doCreateBean(**final String beanName**, **final RootBeanDefinition mbd**, **final @Nullable Object**[] **args**)  **throws BeanCreationException** {   *// Instantiate the bean.* **BeanWrapper instanceWrapper = null**;  **if** (**mbd**.**isSingleton**()) {  **instanceWrapper = this**.**factoryBeanInstanceCache**.**remove**(**beanName**);  }  **if** (**instanceWrapper == null**) {  **instanceWrapper = createBeanInstance**(**beanName**, **mbd**, **args**);  }  **final Object bean = instanceWrapper**.**getWrappedInstance**();  **Class<?> beanType = instanceWrapper**.**getWrappedClass**();  **if** (**beanType != NullBean**.**class**) {  **mbd**.**resolvedTargetType = beanType**;  }   *// Allow post-processors to modify the merged bean definition.* **synchronized** (**mbd**.**postProcessingLock**) {  **if** (**!mbd**.**postProcessed**) {  **try** {  **applyMergedBeanDefinitionPostProcessors**(**mbd**, **beanType**, **beanName**);  }  **catch** (**Throwable ex**) {  **throw new BeanCreationException**(**mbd**.**getResourceDescription**(), **beanName**,  **"Post-processing of merged bean definition failed"**, **ex**);  }  **mbd**.**postProcessed = true**;  }  }   *// Eagerly cache singletons to be able to resolve circular references  // even when triggered by lifecycle interfaces like BeanFactoryAware.* **boolean earlySingletonExposure =** (**mbd**.**isSingleton**() **&& this**.**allowCircularReferences &&  isSingletonCurrentlyInCreation**(**beanName**));  **if** (**earlySingletonExposure**) {  **if** (**logger**.**isTraceEnabled**()) {  **logger**.**trace**(**"Eagerly caching bean '" + beanName +  "' to allow for resolving potential circular references"**);  }  **// 解决循环依赖问题**  **addSingletonFactory**(**beanName**, () -> **getEarlyBeanReference**(**beanName**, **mbd**, **bean**));  }   *// Initialize the bean instance.* **Object exposedObject = bean**;  **try** {  **// 注入属性**  **populateBean(beanName, mbd, instanceWrapper);**  **// 调用init-method方法（先调用初始化前的后置处理器方法，再调用init-method，再调用初始化后的后置处理器方法）**  **exposedObject = initializeBean**(**beanName**, **exposedObject**, **mbd**);  }  **catch** (**Throwable ex**) {  **if** (**ex instanceof BeanCreationException && beanName**.**equals**(((**BeanCreationException**) **ex**).**getBeanName**())) {  **throw** (**BeanCreationException**) **ex**;  }  **else** {  **throw new BeanCreationException**(  **mbd**.**getResourceDescription**(), **beanName**, **"Initialization of bean failed"**, **ex**);  }  }   **if** (**earlySingletonExposure**) {  **Object earlySingletonReference = getSingleton**(**beanName**, **false**);  **if** (**earlySingletonReference != null**) {  **if** (**exposedObject == bean**) {  **exposedObject = earlySingletonReference**;  }  **else if** (**!this**.**allowRawInjectionDespiteWrapping && hasDependentBean**(**beanName**)) {  **String**[] **dependentBeans = getDependentBeans**(**beanName**);  **Set<String> actualDependentBeans = new LinkedHashSet<>**(**dependentBeans**.**length**);  **for** (**String dependentBean : dependentBeans**) {  **if** (**!removeSingletonIfCreatedForTypeCheckOnly**(**dependentBean**)) {  **actualDependentBeans**.**add**(**dependentBean**);  }  }  **if** (**!actualDependentBeans**.**isEmpty**()) {  **throw new BeanCurrentlyInCreationException**(**beanName**,  **"Bean with name '" + beanName + "' has been injected into other beans [" +  StringUtils**.*collectionToCommaDelimitedString*(**actualDependentBeans**) **+  "] in its raw version as part of a circular reference, but has eventually been " +  "wrapped. This means that said other beans do not use the final version of the " +  "bean. This is often the result of over-eager type matching - consider using " +  "'getBeanNamesOfType' with the 'allowEagerInit' flag turned off, for example."**);  }  }  }  }   *// Register bean as disposable.* **try** {  **registerDisposableBeanIfNecessary**(**beanName**, **bean**, **mbd**);  }  **catch** (**BeanDefinitionValidationException ex**) {  **throw new BeanCreationException**(  **mbd**.**getResourceDescription**(), **beanName**, **"Invalid destruction signature"**, **ex**);  }   **return exposedObject**; } |

## 初始化bean

|  |
| --- |
| **protected Object initializeBean(final String beanName, final Object bean, @Nullable RootBeanDefinition mbd) {  if (System.*getSecurityManager*() != null) {  AccessController.*doPrivileged*((PrivilegedAction<Object>) () -> {  invokeAwareMethods(beanName, bean);  return null;  }, getAccessControlContext());  }  else {  invokeAwareMethods(beanName, bean);  }   Object wrappedBean = bean;  if (mbd == null || !mbd.isSynthetic()) {  wrappedBean = applyBeanPostProcessorsBeforeInitialization(wrappedBean, beanName);  }   try {  invokeInitMethods(beanName, wrappedBean, mbd);  }  catch (Throwable ex) {  throw new BeanCreationException(  (mbd != null ? mbd.getResourceDescription() : null),  beanName, "Invocation of init method failed", ex);  }  if (mbd == null || !mbd.isSynthetic()) {  wrappedBean = applyBeanPostProcessorsAfterInitialization(wrappedBean, beanName);  }   return wrappedBean; }** |

# BeanPostProcessor体系

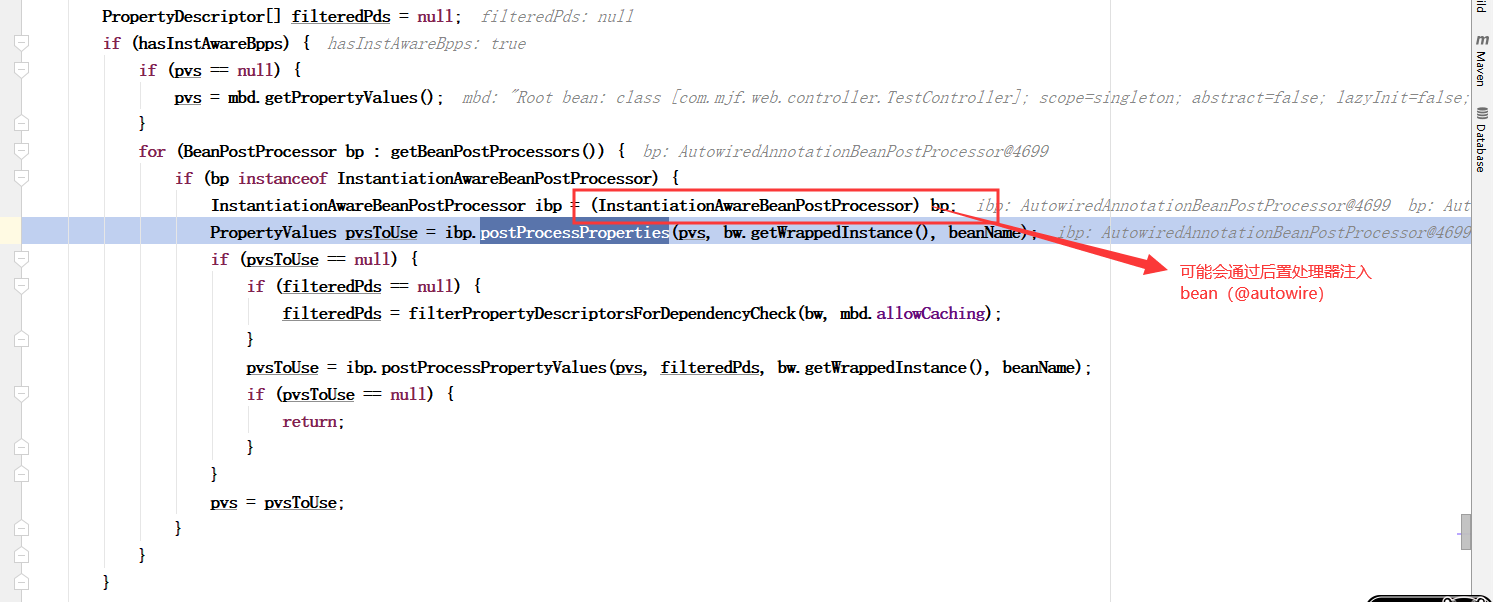
## BeanPostProcessor

### 适用场景

* Springmvc的注解

## InstantiationAwareBeanPostProcessor

### 使用场景



* Aop
* Autowire注解

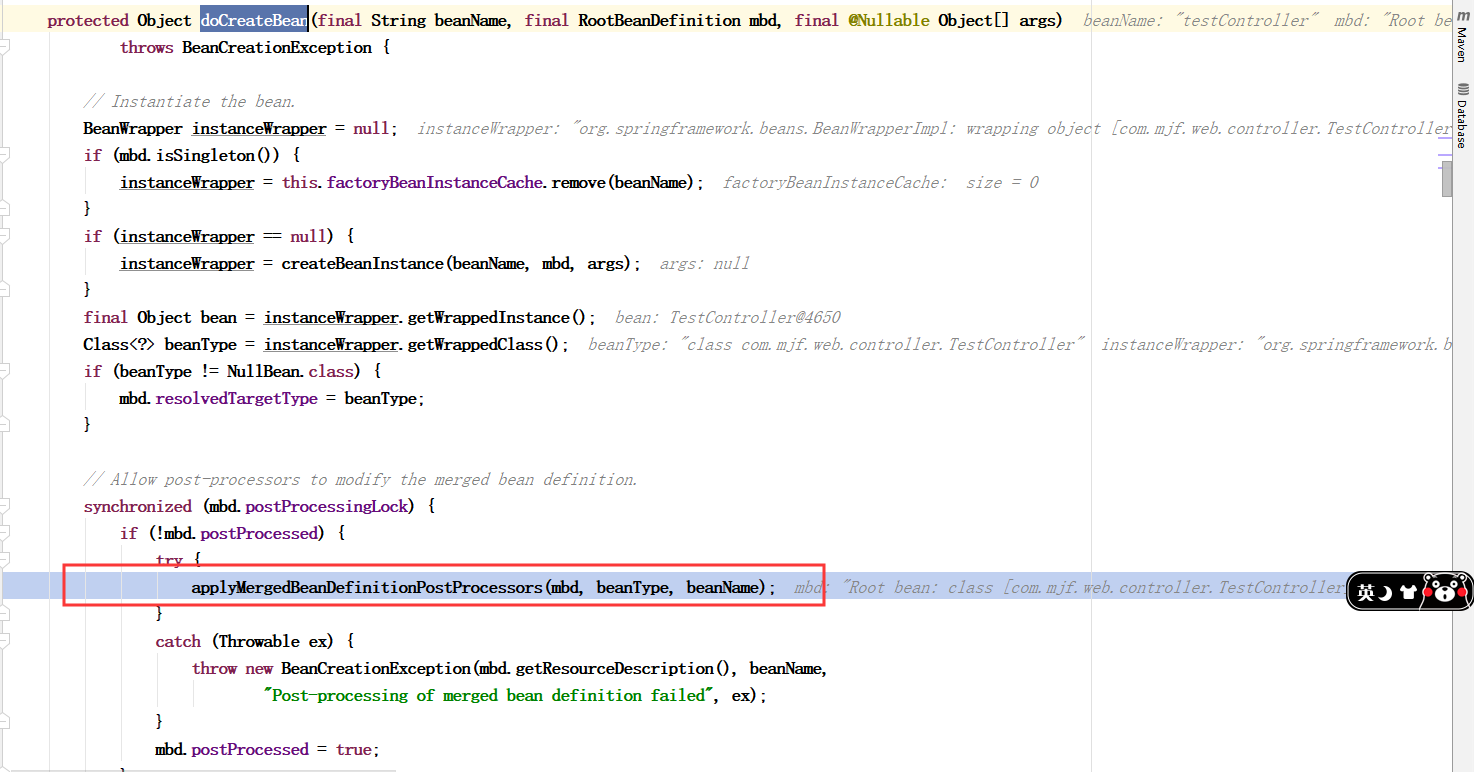
### AutowiredAnnotationBeanPostProcessor

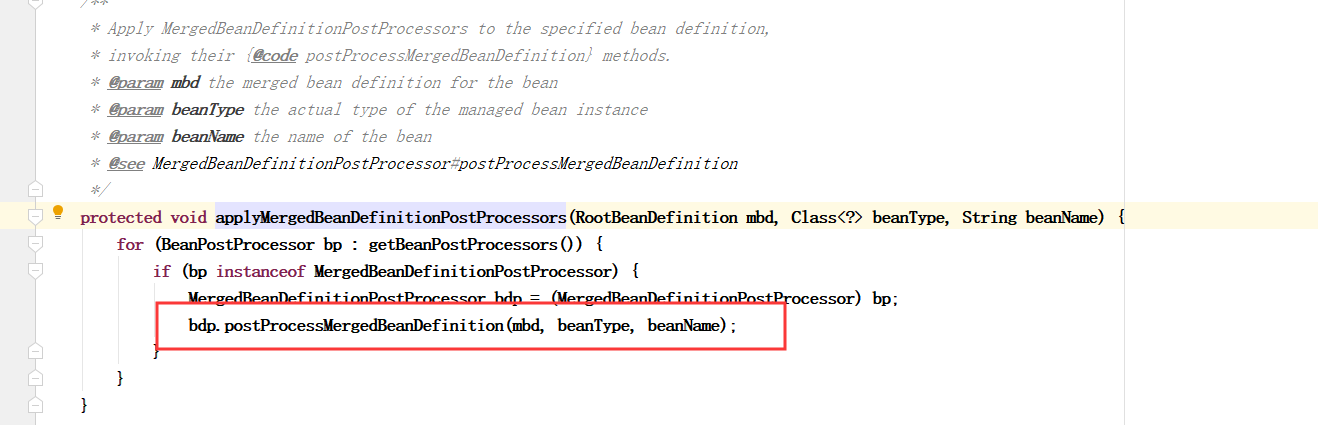
#### 作用

用来处理Autowired注解的

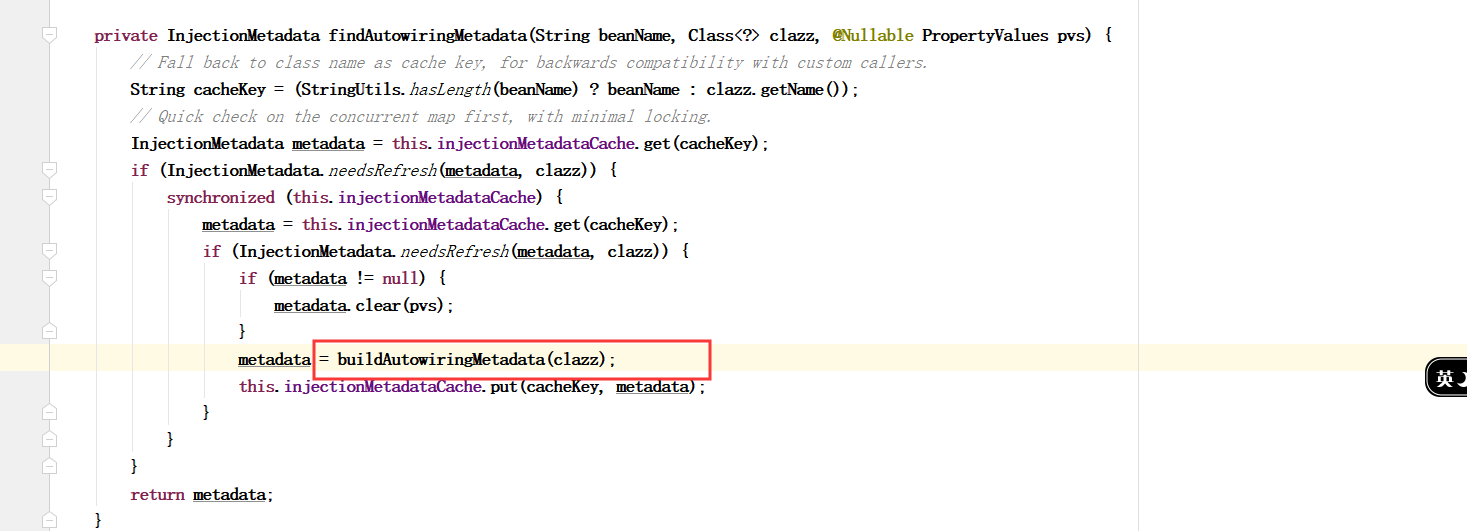
#### 处理流程

在进行创建bean实例后，会调用后置处理器查找带@auwored注解，在进行依赖注入的时候，再通过后置处理器进行注入









**private InjectionMetadata buildAutowiringMetadata(final Class<?> clazz) {  
 if (!AnnotationUtils.*isCandidateClass*(clazz, this.autowiredAnnotationTypes)) {  
 return InjectionMetadata.*EMPTY*;  
 }  
  
 List<InjectionMetadata.InjectedElement> elements = new ArrayList<>();  
 Class<?> targetClass = clazz;  
  
 do {  
 final List<InjectionMetadata.InjectedElement> currElements = new ArrayList<>();  
  
 ReflectionUtils.*doWithLocalFields*(targetClass, field -> {**

**// 查找字段上的注解  
 MergedAnnotation<?> ann = findAutowiredAnnotation(field);  
 if (ann != null) {**

**// Autowired注解不支持静态字段  
 if (Modifier.*isStatic*(field.getModifiers())) {  
 if (logger.isInfoEnabled()) {  
 logger.info("Autowired annotation is not supported on static fields: " + field);  
 }  
 return;  
 }  
 boolean required = determineRequiredStatus(ann);  
 currElements.add(new AutowiredFieldElement(field, required));  
 }  
 });  
  
 ReflectionUtils.*doWithLocalMethods*(targetClass, method -> {  
 Method bridgedMethod = BridgeMethodResolver.*findBridgedMethod*(method);  
 if (!BridgeMethodResolver.*isVisibilityBridgeMethodPair*(method, bridgedMethod)) {  
 return;  
 }**

**// 在方法上查找注解(@Autowired和@Value)  
 MergedAnnotation<?> ann = findAutowiredAnnotation(bridgedMethod);  
 if (ann != null && method.equals(ClassUtils.*getMostSpecificMethod*(method, clazz))) {  
 if (Modifier.*isStatic*(method.getModifiers())) {  
 if (logger.isInfoEnabled()) {  
 logger.info("Autowired annotation is not supported on static methods: " + method);  
 }  
 return;  
 }  
 if (method.getParameterCount() == 0) {  
 if (logger.isInfoEnabled()) {  
 logger.info("Autowired annotation should only be used on methods with parameters: " +  
 method);  
 }  
 }  
 boolean required = determineRequiredStatus(ann);  
 PropertyDescriptor pd = BeanUtils.*findPropertyForMethod*(bridgedMethod, clazz);  
 currElements.add(new AutowiredMethodElement(method, required, pd));  
 }  
 });  
  
 elements.addAll(0, currElements);  
 targetClass = targetClass.getSuperclass();  
 }  
 while (targetClass != null && targetClass != Object.class);  
  
 return InjectionMetadata.*forElements*(elements, clazz);  
}**

#### AutowiredFieldElement

用来封装在字段上添加@autowired注解的信息

## SmartInstantiationAwareBeanPostProcessor

### 作用

循环依赖的扩展

## MergedBeanDefinitionPostProcessor

# Aware体系

## BeanFactoryAware

## ApplicationContextAware

BeanNameAware

# Spring集成到web中

## 配置web.xml

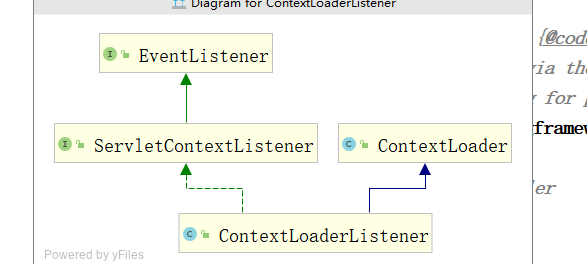
|  |
| --- |
| <**context-param**>  <**param-name**>contextConfigLocation</**param-name**>  <**param-value**>classpath:application.xml</**param-value**> </**context-param**> <**listener**>  <**listener-class**>org.springframework.web.context.ContextLoaderListener</**listener-class**> </**listener**> |

## ContextLoaderListener

### 作用

用来初始化spring容器

### 类结构

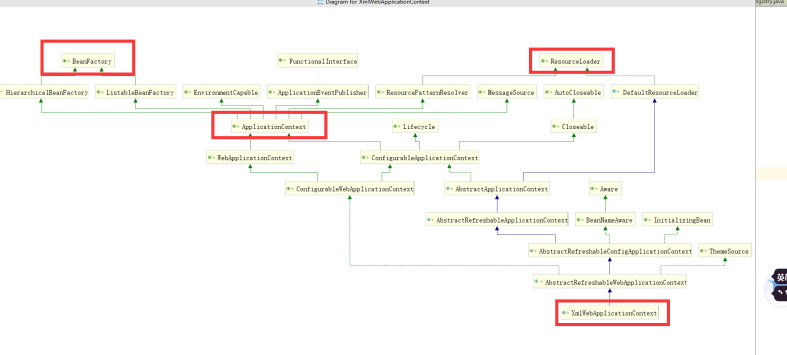


### 核心代码

|  |
| --- |
| ***/\*\*  \* Initialize Spring's web application context for the given servlet context,  \* using the application context provided at construction time, or creating a new one  \* according to the "{@link #CONTEXT\_CLASS\_PARAM contextClass}" and  \* "{@link #CONFIG\_LOCATION\_PARAM contextConfigLocation}" context-params.  \* @param servletContext current servlet context  \* @return the new WebApplicationContext  \* @see #*ContextLoader*(*WebApplicationContext*)  \* @see #CONTEXT\_CLASS\_PARAM  \* @see #CONFIG\_LOCATION\_PARAM  \*/* public WebApplicationContext** initWebApplicationContext(**ServletContext servletContext**) {  **if** (**servletContext**.**getAttribute**(**WebApplicationContext**.***ROOT\_WEB\_APPLICATION\_CONTEXT\_ATTRIBUTE***) **!= null**) {  **throw new IllegalStateException**(  **"Cannot initialize context because there is already a root application context present - " +  "check whether you have multiple ContextLoader\* definitions in your web.xml!"**);  }   **servletContext**.**log**(**"Initializing Spring root WebApplicationContext"**);  **Log logger = LogFactory**.*getLog*(**ContextLoader**.**class**);  **if** (**logger**.**isInfoEnabled**()) {  **logger**.**info**(**"Root WebApplicationContext: initialization started"**);  }  **long startTime = System**.*currentTimeMillis*();   **try** {  *// Store context in local instance variable, to guarantee that  // it is available on ServletContext shutdown.* **if** (**this**.**context == null**) {  // 创建上下文(XmlWebApplicationContext)  **this**.**context = createWebApplicationContext**(**servletContext**);  }  **if** (**this**.**context instanceof ConfigurableWebApplicationContext**) {  **ConfigurableWebApplicationContext cwac =** (**ConfigurableWebApplicationContext**) **this**.**context**;  **if** (**!cwac**.**isActive**()) {  *// The context has not yet been refreshed -> provide services such as  // setting the parent context, setting the application context id, etc* **if** (**cwac**.**getParent**() **== null**) {  *// The context instance was injected without an explicit parent ->  // determine parent for root web application context, if any.* **ApplicationContext parent = loadParentContext**(**servletContext**);  **cwac**.**setParent**(**parent**);  }  // 刷新容器（主要创建bean，还有依赖注入）  **configureAndRefreshWebApplicationContext**(**cwac**, **servletContext**);  }  }  // 将spring容器设置为父容器  **servletContext**.**setAttribute**(**WebApplicationContext**.***ROOT\_WEB\_APPLICATION\_CONTEXT\_ATTRIBUTE***, **this**.**context**);   **ClassLoader ccl = Thread**.*currentThread*().**getContextClassLoader**();  **if** (**ccl == ContextLoader**.**class**.**getClassLoader**()) {  ***currentContext* = this**.**context**;  }  **else if** (**ccl != null**) {  ***currentContextPerThread***.**put**(**ccl**, **this**.**context**);  }   **if** (**logger**.**isInfoEnabled**()) {  **long elapsedTime = System**.*currentTimeMillis*() **- startTime**;  **logger**.**info**(**"Root WebApplicationContext initialized in " + elapsedTime + " ms"**);  }   **return this**.**context**;  }  **catch** (**RuntimeException | Error ex**) {  **logger**.**error**(**"Context initialization failed"**, **ex**);  **servletContext**.**setAttribute**(**WebApplicationContext**.***ROOT\_WEB\_APPLICATION\_CONTEXT\_ATTRIBUTE***, **ex**);  **throw ex**;  } } |

## XmlWebApplicationContext

### 类结构

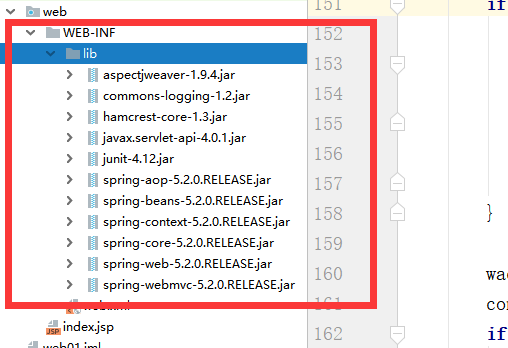


### 核心代码

|  |
| --- |
| ***/\*\*  \* Loads the bean definitions via an XmlBeanDefinitionReader.  \* @see* org.springframework.beans.factory.xml.XmlBeanDefinitionReader  *\* @see #*initBeanDefinitionReader  *\* @see #loadBeanDefinitions  \*/* @Override protected void** loadBeanDefinitions(**DefaultListableBeanFactory beanFactory**) **throws BeansException**, **IOException** {  *// Create a new XmlBeanDefinitionReader for the given BeanFactory.* **XmlBeanDefinitionReader beanDefinitionReader = new XmlBeanDefinitionReader**(**beanFactory**);  *// Configure the bean definition reader with this context's  // resource loading environment.* **beanDefinitionReader**.**setEnvironment**(**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.* **initBeanDefinitionReader**(**beanDefinitionReader**);  // 解析bean（将bean从xml中解析出来，封装到BeanDefination中）  **loadBeanDefinitions**(**beanDefinitionReader**); } |

## 注意

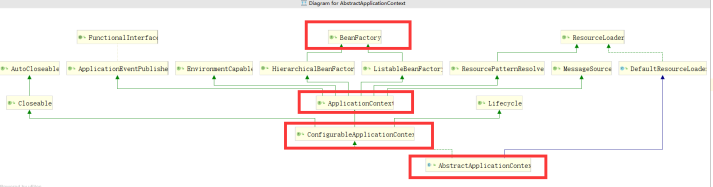
如果不是maven项目，一定要将jar包放到WEB-INF下面



# ApplicationContext家族

## AbstractApplicationContext

### 结构



### 核心代码

|  |
| --- |
| ***/\*\*  \* Return the list of statically specified ApplicationListeners.  \*/* public Collection<ApplicationListener<?>>** getApplicationListeners() {  **return this**.**applicationListeners**; }  **@Override public void** refresh() **throws BeansException**, **IllegalStateException** {  **synchronized** (**this**.**startupShutdownMonitor**) {  *// Prepare this context for refreshing.* **prepareRefresh**();   *// Tell the subclass to refresh the internal bean factory.*  ***// 初始化beanfactory，并且将bean加载到BeanDefinition中*****ConfigurableListableBeanFactory beanFactory = obtainFreshBeanFactory**();   *// Prepare the bean factory for use in this context.* **prepareBeanFactory**(**beanFactory**);   **try** {  *// Allows post-processing of the bean factory in context subclasses.* **postProcessBeanFactory**(**beanFactory**);   *// Invoke factory processors registered as beans in the context.* **invokeBeanFactoryPostProcessors**(**beanFactory**);   *// Register bean processors that intercept bean creation.* **registerBeanPostProcessors**(**beanFactory**);   *// Initialize message source for this context.* **initMessageSource**();   *// Initialize event multicaster for this context.* **initApplicationEventMulticaster**();   *// Initialize other special beans in specific context subclasses.* **onRefresh**();   *// Check for listener beans and register them.* **registerListeners**();   *// Instantiate all remaining (non-lazy-init) singletons.*  ***// 初始化bean，并且进行依赖注入*****finishBeanFactoryInitialization**(**beanFactory**);   *// Last step: publish corresponding event.* **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**();  }  } } |

### prepareBeanFactory

|  |
| --- |
| */\*\*  \* Configure the factory's standard context characteristics,  \* such as the context's ClassLoader and post-processors.  \** ***@param beanFactory*** *the BeanFactory to configure  \*/* **protected void** prepareBeanFactory(ConfigurableListableBeanFactory beanFactory) {  *// Tell the internal bean factory to use the context's class loader etc.* beanFactory.setBeanClassLoader(getClassLoader());  beanFactory.setBeanExpressionResolver(**new** StandardBeanExpressionResolver(beanFactory.getBeanClassLoader()));  beanFactory.addPropertyEditorRegistrar(**new** ResourceEditorRegistrar(**this**, getEnvironment()));   *// Configure the bean factory with context callbacks.  // 添加bean后置处理器*  beanFactory.addBeanPostProcessor(**new** ApplicationContextAwareProcessor(**this**));  **// 忽略某些接口**  beanFactory.ignoreDependencyInterface(EnvironmentAware.**class**);  beanFactory.ignoreDependencyInterface(EmbeddedValueResolverAware.**class**);  beanFactory.ignoreDependencyInterface(ResourceLoaderAware.**class**);  beanFactory.ignoreDependencyInterface(ApplicationEventPublisherAware.**class**);  beanFactory.ignoreDependencyInterface(MessageSourceAware.**class**);  beanFactory.ignoreDependencyInterface(ApplicationContextAware.**class**);   *// BeanFactory interface not registered as resolvable type in a plain factory.  // MessageSource registered (and found for autowiring) as a bean.* 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**));   *// 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.getBeanClassLoader()));  }   ***// Register default environment beans.*****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());  } } |

### finishBeanFactoryInitialization

|  |
| --- |
| */\*\*  \* Finish the initialization of this context's bean factory,  \* initializing all remaining singleton beans.  \*/* **protected void** finishBeanFactoryInitialization(ConfigurableListableBeanFactory beanFactory) {  *// Initialize conversion service for this context.* **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**));  }   *// Register a default embedded value resolver if no bean post-processor  // (such as a PropertyPlaceholderConfigurer bean) registered any before:  // at this point, primarily for resolution in annotation attribute values.* **if** (!beanFactory.hasEmbeddedValueResolver()) {  beanFactory.addEmbeddedValueResolver(strVal -> getEnvironment().resolvePlaceholders(strVal));  }   *// 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);  }   *// Stop using the temporary ClassLoader for type matching.* beanFactory.setTempClassLoader(**null**);   *// Allow for caching all bean definition metadata, not expecting further changes.* beanFactory.freezeConfiguration();   *//* ***Instantiate all remaining (non-lazy-init) singletons.*** beanFactory.preInstantiateSingletons(); // DefaultListableBeanFactory实现该方法 } |