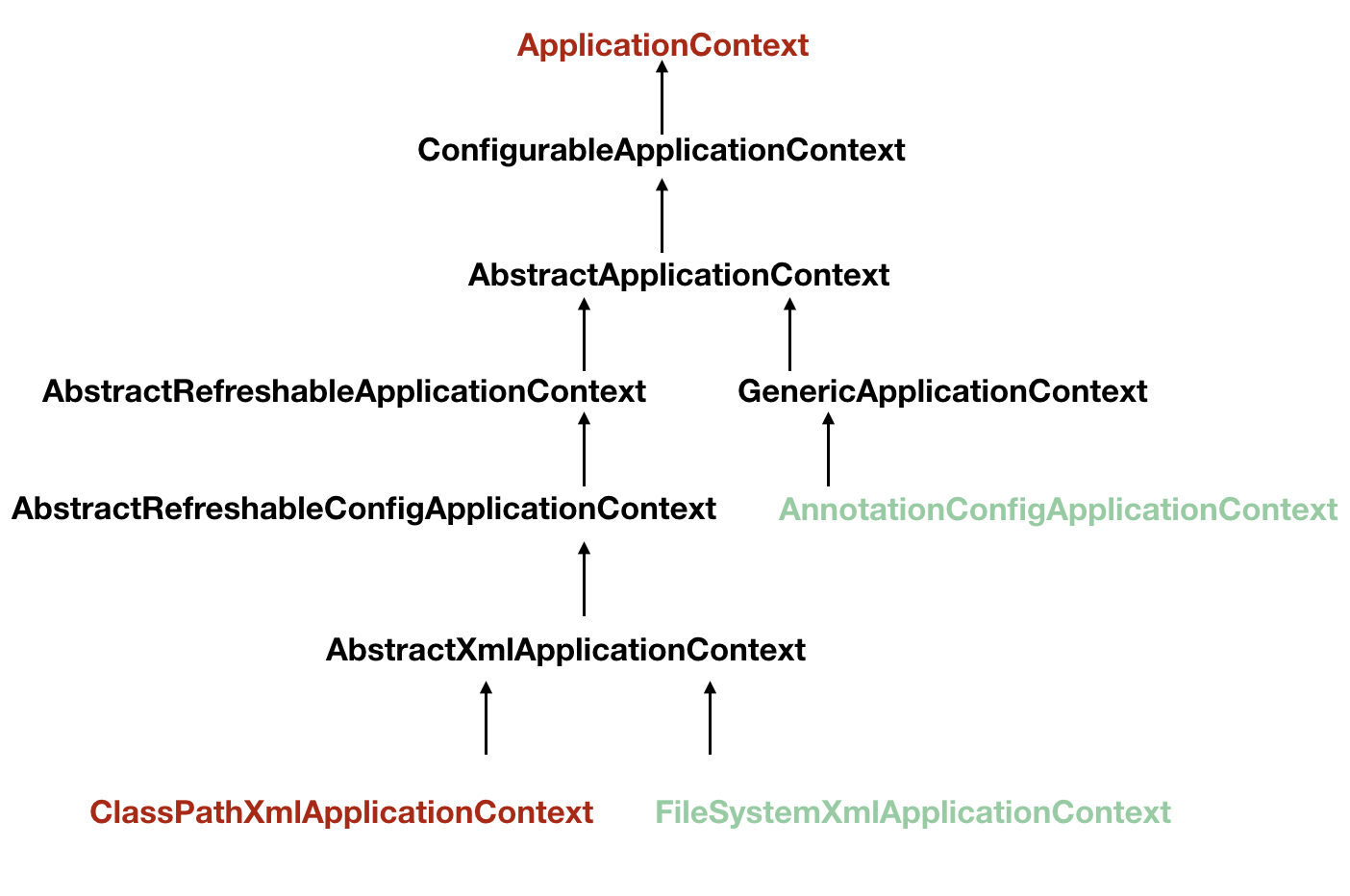
# 前言

为了降低难度，本文所说的所有的内容都是基于 xml 的配置的方式，实际使用已经很少人这么做了，至少不是纯 xml 配置，不过从理解源码的角度来看用这种方式来说无疑是最合适的。

本文要说的 IOC 总体来说有两处地方最重要，一个是创建 Bean 容器，一个是初始化 Bean，如果读者觉得一次性看完本文压力有点大，那么可以按这个思路分两次消化。

## ApplicationContext



ClassPathXmlApplicationContext

FileSystemXmlApplicationContext

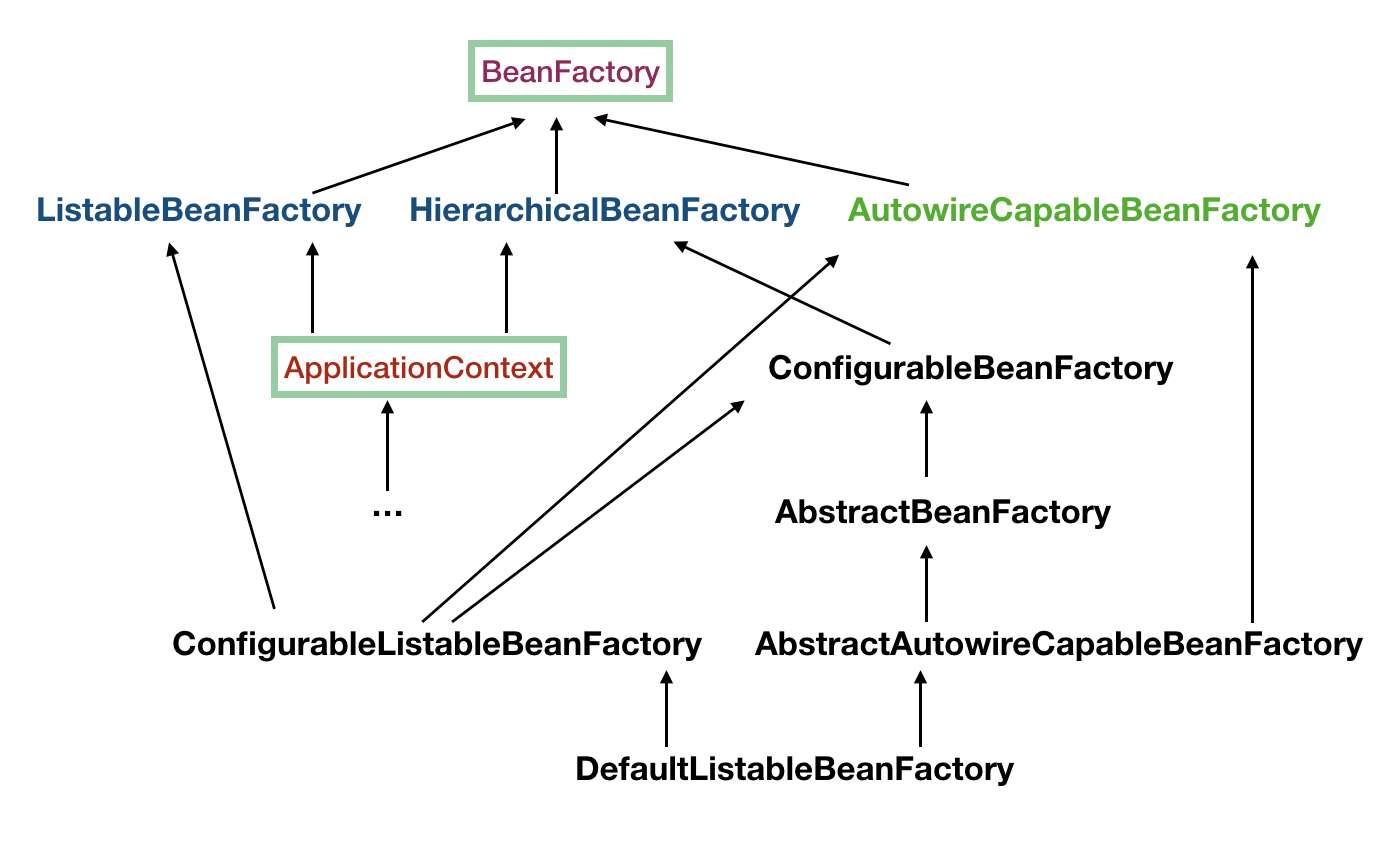
**AnnotationConfigApplicationContext** 是基于注解来使用的，它不需要配置文件，采用 java 配置类和各种注解来配置，是比较简单的方式，也是大势所趋吧。

本文旨在帮助大家理解整个构建流程，所以决定使用 ClassPathXmlApplicationContext 进行分析，重点关注一下ApplicationContext是如何启动的，启动过程中又做了些什么。

## BeanFactory

BeanFactory是生产 bean 的工厂，它负责生产和管理各个 bean 实例。

前面说的 ApplicationContext 是继承自 BeanFactory的，但是它不应该被理解为 BeanFactory 的实现类，而是说其内部持有一个BeanFactory实例（DefaultListableBeanFactory），其所有的 BeanFactory 相关的操作其实是委托给这个实例来处理的。



ApplicationContext 往下的继承结构前面一张图说过了，这里就不重复了。这张图呢，背下来肯定是不需要的，有几个重点和大家说明下就好。

1. ApplicationContext 继承了 **ListableBeanFactory**，这个 Listable 的意思就是，通过这个接口，我们可以获取多个 Bean，大家看源码会发现，最顶层 BeanFactory 接口的方法都是获取单个 Bean 的。
2. ApplicationContext 继承了 **HierarchicalBeanFactory**，Hierarchical 单词本身已经能说明问题了，也就是说我们可以在应用中起多个 BeanFactory，然后可以将各个 BeanFactory 设置为父子关系。
3. **AutowireCapableBeanFactory** 这个名字中的 Autowire 大家都非常熟悉，它就是用来自动装配 Bean 用的，但是仔细看上图，ApplicationContext 并没有继承它，不过不用担心，不使用继承，不代表不可以使用**组合**，如果你看到 ApplicationContext 接口定义中的最后一个方法 getAutowireCapableBeanFactory() 就知道了。
4. ConfigurableListableBeanFactory 也是一个特殊的接口，看图，特殊之处在于它继承了第二层所有的三个接口，而 ApplicationContext 没有。这点之后会用到。
5. 请先不用花时间在其他的接口和类上，先理解我说的这几点就可以了。

请读者打开编辑器，翻一下 BeanFactory、ListableBeanFactory、HierarchicalBeanFactory、AutowireCapableBeanFactory、ApplicationContext 这几个接口的代码，大概看一下各个接口中的方法，做到心里有底。

# ClassPathXmlApplicationContext

## 构造方法

需要关注super(parent)、setConfigLocations(configLocations)、refresh()，特别是refresh()。

|  |
| --- |
| **public** ClassPathXmlApplicationContext() {  }  **public** ClassPathXmlApplicationContext(ApplicationContext parent) {  **super**(parent);  }  // 这两个啥用啊？ |
| **public** ClassPathXmlApplicationContext(String[] configLocations, **boolean** refresh, ApplicationContext parent) **throws** BeansException {  **super**(parent);  // setConfigLocations负责创建环境对象ConfigurableEnvironment ，并用该对象处理传入的路径字符串数组中每个路径中包含的占位符；最后将处理结果的路径数组设置到ApplicationContext相应属性上。  setConfigLocations(configLocations);  **if** (refresh) {  refresh();  }  } |
| **public** ClassPathXmlApplicationContext(String[] paths, Class<?> clazz, ApplicationContext parent)  **throws** BeansException {  **super**(parent);  Assert.*notNull*(paths, "Path array must not be null");  Assert.*notNull*(clazz, "Class argument must not be null");  **this**.configResources = **new** Resource[paths.length];  **for** (**int** i = 0; i < paths.length; i++) {  **this**.configResources[i] = **new** ClassPathResource(paths[i], clazz);  }  refresh();  } |

## refresh()

|  |
| --- |
| AbstractApplicationContext 508 |
| 这里为什么方法名为什么是 refresh，而不是 类似init这种名字呢？因为 ApplicationContext 建立起来以后，其实我们是可以通过调用 refresh() 这个方法重建的，refresh() 会将原来的 ApplicationContext 销毁，然后再重新执行一次初始化操作。 |
| public void refresh() throws BeansException, IllegalStateException {  // 来个锁，不然 refresh() 还没结束，你又来个启动或销毁容器的操作，那不就乱套了嘛  synchronized (this.startupShutdownMonitor) {  // 准备工作，记录下容器的启动时间、标记“已启动”状态、及其它  prepareRefresh();  // 这步比较关键，这步完成后，配置文件中所有的<Bean>配置就会被解析成一个个 BeanDefinition，并注册到 BeanFactory 中了（实际上用Map<beanName,BeanDefinition>保存的，见BeanDefinitionRegistry接口的实现）。 注意在这一步中，bean并没有实例化。  ConfigurableListableBeanFactory beanFactory = obtainFreshBeanFactory();  // 设置 BeanFactory 的类加载器，添加几个 BeanPostProcessor，手动注册几个特殊的 bean  prepareBeanFactory(beanFactory);  try {  // 这里是提供给子类的扩展点，到这里的时候，所有的 Bean 都加载、注册完成了，但是都还没有初始化。 具体的子类可以在这一步对beanFactory做一些后置处理（post-processing）  postProcessBeanFactory(beanFactory);  // 除了上面的扩展点可以对beanFactory进行后置处理，这一步还会调用context中所有实现了BeanFactoryPostProcessor 接口的Bean的 postProcessBeanFactory(factory)方法来对beanFactory进行后置处理。  invokeBeanFactoryPostProcessors(beanFactory);  // 注册 BeanPostProcessor 的实现类， 此接口的两个方法postProcessBeforeInitialization 和 postProcessAfterInitialization分别在 Bean 初始化之前和初始化之后得到执行。这里仅仅是注册，之后会看到回调这两方法的时机  registerBeanPostProcessors(beanFactory);  // ~~Initialize message source for this context. 国际化~~  initMessageSource();  // ~~Initialize event multicaster for this context.事件广播器~~  initApplicationEventMulticaster();  // 钩子方法，具体的子类可以在这里初始化一些特殊的 Bean（在初始化 singleton beans 之前）  onRefresh();  // ~~检查并注册事件监听器（实现了ApplicationListener 接口的Bean）。这也不是我们的重点，过~~  registerListeners();    // 初始化所有的 singleton beans（lazy-init为true的除外）。  finishBeanFactoryInitialization(beanFactory);  //  finishRefresh();  } catch (BeansException ex) {  if (logger.isWarnEnabled()) {  logger.warn("Exception encountered during context initialization - " +  "cancelling refresh attempt: " + ex);  }  // 销毁已经初始化的 singleton 的 Beans，以免有些 bean 会一直占用资源  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();  }  }  } |

## obtainFreshBeanFactory()

|  |
| --- |
| AbstractApplicationContext 613 |
| protected ConfigurableListableBeanFactory obtainFreshBeanFactory() {  refreshBeanFactory();  ConfigurableListableBeanFactory beanFactory = getBeanFactory(); // 获取上一步刚创建的beanFactory  if (logger.isDebugEnabled()) {  logger.debug("Bean factory for " + getDisplayName() + ": " + beanFactory);  }  return beanFactory;  } |

### refreshBeanFactory()

|  |
| --- |
| AbstractRefreshableApplicationContext 120 |
| refreshBeanFactory负责销毁当前ApplicationContext中旧的beanFactory，并为其创建一个新的BeanFactory，并加载BeanDefinition到新的beanFactory中。 |
| protected final void refreshBeanFactory() throws BeansException {  // 首先，如果当前ApplicationContext 中已经加载过 BeanFactory 了，则销毁所有 Bean，并关闭旧 BeanFactory。  if (hasBeanFactory()) {  destroyBeans();  closeBeanFactory();  }  try {  // 这里会创建一个新的DefaultListableBeanFactory，通过BeanFactory类图知道这是功能最强大的BeanFactory了。  DefaultListableBeanFactory beanFactory = createBeanFactory();  beanFactory.setSerializationId(getId()); ~~// 用于 BeanFactory 的序列化，我想不部分人应该都用不到~~  // 设置 BeanFactory 的两个配置属性：是否允许 Bean 覆盖、是否允许循环引用。  默认情况下，如果在同一配置文件中重复了，会抛错，但是如果不是同一配置文件中，会发生覆盖。  默认情况下，允许bean的循环依赖（允许setter造成的循环依赖，不允许构造方法造成的循环依赖）。  customizeBeanFactory(beanFactory);  // 根据配置加载各个BeanDefinition，并注册到注册中心[即beanFactory]中，  XmlBeanDefinitionReader 中责加载配置、解析。  loadBeanDefinitions(beanFactory);  // 将新创建的beanFactory设置到当前ApplicatonContext中  synchronized (this.beanFactoryMonitor) {  this.beanFactory = beanFactory;  }  } catch (IOException ex) {  throw new ApplicationContextException("I/O error parsing bean definition source for " + getDisplayName(), ex);  }  } |

#### loadBeanDefinitions(beanFactory)

|  |
| --- |
| AbstractXmlApplicationContext 80 |
| 为给定的beanFactory创建一个新的XmlBeanDefinitionReader实例，并将当前ApplicationContext的资源加载环境传递给该XmlBeanDefinitionReader实例（如Environment、ResourceLoader、EntityResolver，其中ResourceLoader 是用于加载资源文件的，EntityResolver是用于加载dtd、xsd的）。  最后用该XmlBeanDefinitionReader实例去加载和注册各个BeanDefinition。 |
| protected void loadBeanDefinitions(DefaultListableBeanFactory beanFactory) throws BeansException, IOException {  XmlBeanDefinitionReader beanDefinitionReader = new XmlBeanDefinitionReader(beanFactory);  beanDefinitionReader.setEnvironment(this.getEnvironment());  beanDefinitionReader.setResourceLoader(this);  beanDefinitionReader.setEntityResolver(new ResourceEntityResolver(this));  // ~~【不重要】另外，在用该XmlBeanDefinitionReader实例去加载加载BeanDefinition之前，会先执行一个钩子方法，子类可通过覆写该方法来定制化该XmlBeanDefinitionReader实例。~~  initBeanDefinitionReader(beanDefinitionReader);  loadBeanDefinitions(beanDefinitionReader);  } |

##### loadBeanDefinitions(beanDefinitionReader)

|  |
| --- |
| AbstractXmlApplicationContext 120 |
| 下面代码关注reader.loadBeanDefinitions(Resource... resources)就够了，因为跟进去reader.loadBeanDefinitions(String... locations)里面，你会发现内部会用ResourcePatternResolver将每个location都转换为一个Resource[]，然后再去调用reader.loadBeanDefinitions(Resource... resources),殊途同归。 |
| protected void loadBeanDefinitions(XmlBeanDefinitionReader reader) throws BeansException, IOException {  Resource[] configResources = getConfigResources();  if (configResources != null) {  reader.loadBeanDefinitions(configResources);  }  String[] configLocations = getConfigLocations();  if (configLocations != null) {  reader.loadBeanDefinitions(configLocations);  }  } |

## prepareBeanFactory (beanFactory)

|  |
| --- |
| AbstractApplicationContext 627 |
| 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.  // 添加一个 BeanPostProcessor[ApplicationContextAwareProcessor]，这个 processor 比较简单：  // 实现了 Aware 接口的 beans 在初始化的时候，这个 processor 负责回调。 这个我们很常用，例如我们会为了获取 ApplicationContext 而 implement ApplicationContextAware。 注意：它不仅仅回调 ApplicationContextAware， 还会负责回调 EnvironmentAware、ResourceLoaderAware 等，看下源码就清楚了  beanFactory.addBeanPostProcessor(new ApplicationContextAwareProcessor(this));  // 如果某个 bean 依赖于以下几个接口的实现类，在自动装配的时候忽略它们，Spring 会通过其他方式来处理这些依赖。  beanFactory.ignoreDependencyInterface(EnvironmentAware.class);  beanFactory.ignoreDependencyInterface(EmbeddedValueResolverAware.class);  beanFactory.ignoreDependencyInterface(ResourceLoaderAware.class);  beanFactory.ignoreDependencyInterface(ApplicationEventPublisherAware.class);  beanFactory.ignoreDependencyInterface(MessageSourceAware.class);  beanFactory.ignoreDependencyInterface(ApplicationContextAware.class);  /\*\*  \* 下面几行就是为特殊的几个 bean 赋值，如果有 bean 依赖了以下几个，会注入这边相应的值，  \* 这里第一行解释了"当前 ApplicationContext 持有一个 BeanFactory"， ApplicationContext 还继承了 ResourceLoader、ApplicationEventPublisher、MessageSource 所以对于这几个依赖，可以赋值为 this，注意 this 是一个 ApplicationContext  \* 那这里怎么没看到为 MessageSource 赋值呢？那是因为 MessageSource 被注册成为了一个普通的 bean。  \*/  beanFactory.registerResolvableDependency(BeanFactory.class, beanFactory);  beanFactory.registerResolvableDependency(ResourceLoader.class, this);  beanFactory.registerResolvableDependency(ApplicationEventPublisher.class, this);  beanFactory.registerResolvableDependency(ApplicationContext.class, this);  // ApplicationListenerDetector这个 BeanPostProcessor 也很简单，它负责在 bean 实例化后，如果是 ApplicationListener 的子类， 那么将其添加到 listener 列表中，可以理解成：注册 事件监听器  beanFactory.addBeanPostProcessor(new ApplicationListenerDetector(this));  // ~~这不是我们的重点，忽略它。这里涉及到特殊的 bean名称为loadTimeWeaver， ltw 是 AspectJ 的概念，指的是在运行期进行织入，这个和 Spring AOP 不一样。~~  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()));  }  // 如果我们没有自己定义environment、systemProperties、systemEnvironment 这3个bean，spring会默认帮我们手动注册上。 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());  }  } |

# BeanDefinitionReader【不重要，选择性跳过】

## loadBeanDefinitions(Resource...)

|  |
| --- |
| AbstractBeanDefinitionReader 177 |
| public int loadBeanDefinitions(Resource... resources) throws BeanDefinitionStoreException {  Assert.*notNull*(resources, "Resource array must not be null");  int counter = 0;  // 注意这里是个 for 循环，也就是每个文件是一个 resource  for (Resource resource : resources) {  counter += loadBeanDefinitions(resource);  }  return counter; // 最后返回 counter，表示总共加载了多少的 BeanDefinition  } |

### loadBeanDefinitions(Resource)

|  |
| --- |
| XmlBeanDefinitionReader 303 -> 314 -> 388 -> 505 |
| public int loadBeanDefinitions(Resource resource) throws BeanDefinitionStoreException {  return loadBeanDefinitions(new EncodedResource(resource));  } |
| public int loadBeanDefinitions(EncodedResource encodedResource) throws BeanDefinitionStoreException {  Assert.*notNull*(encodedResource, "EncodedResource must not be null");  if (logger.isInfoEnabled()) {  logger.info("Loading XML bean definitions from " + encodedResource);  }  // 用一个 ThreadLocal 来存放配置文件资源  Set<EncodedResource> currentResources = this.resourcesCurrentlyBeingLoaded.get();  if (currentResources == null) {  currentResources = new HashSet<EncodedResource>(4);  this.resourcesCurrentlyBeingLoaded.set(currentResources);  }  if (!currentResources.add(encodedResource)) {  throw new BeanDefinitionStoreException(  "Detected cyclic loading of " + encodedResource + " - check your import definitions!");  }  try {  InputStream inputStream = encodedResource.getResource().getInputStream();  try {  InputSource inputSource = new InputSource(inputStream);  if (encodedResource.getEncoding() != null) {  inputSource.setEncoding(encodedResource.getEncoding());  }  return doLoadBeanDefinitions(inputSource, encodedResource.getResource());  } finally {  inputStream.close();  }  } catch (IOException ex) {  throw new BeanDefinitionStoreException(  "IOException parsing XML document from " + encodedResource.getResource(), ex);  } finally {  currentResources.remove(encodedResource);  if (currentResources.isEmpty()) {  this.resourcesCurrentlyBeingLoaded.remove();  }  }  } |
| protected int doLoadBeanDefinitions(InputSource inputSource, Resource resource)  throws BeanDefinitionStoreException {  try {  Document doc = doLoadDocument(inputSource, resource); // 将 xml 文件转换为 Document 对象  return registerBeanDefinitions(doc, resource);  } catch...  } |
| public int registerBeanDefinitions(Document doc, Resource resource) throws BeanDefinitionStoreException {  BeanDefinitionDocumentReader documentReader = createBeanDefinitionDocumentReader();  int countBefore = getRegistry().getBeanDefinitionCount();  documentReader.registerBeanDefinitions(doc, createReaderContext(resource));  return getRegistry().getBeanDefinitionCount() - countBefore; // 返回从当前配置文件加载了多少数量的 Bean  } |

# BeanDefinitionDocumentReader

## registerBeanDefinitions(Document, XmlReaderContext)

|  |
| --- |
| DefaultBeanDefinitionDocumentReader 90 -> 116 –> 161 |
| public void registerBeanDefinitions(Document doc, XmlReaderContext readerContext) {  this.readerContext = readerContext;  logger.debug("Loading bean definitions");  Element root = doc.getDocumentElement();  doRegisterBeanDefinitions(root); // 从 xml 根节点开始解析文件  } |
| protected void doRegisterBeanDefinitions(Element root) {  // 此方法负责从xml根节点开始解析，而xml的<beans>是可以嵌套的，每遇到一次嵌套的<beans>都会递归调用一下此方法。所以这个方法的 root 其实不一定就是 xml 的根节点，也可以是嵌套在里面的 <beans /> 节点。  // BeanDefinitionParserDelegate负责解析root[<beans>]。由于子<beans>的各个default-\*属性的配置与父<beans>不同，且子<beans>中未指定的default-\*属性需要从父<beans>继承，所以每次递归都需要依赖parent delegate创建新的delegate来解析子<beans>。子<beans>解析完后要将旧的delegate对象还原回去以便继续解析父<beans>。  BeanDefinitionParserDelegate parent = this.delegate; // 当root是根<beans>时，parent为null  this.delegate = createDelegate(getReaderContext(), root, parent);  // 有一种情况，root对应的<beans>元素是不需要解析的，直接return了。这种情况满足以下两个条件：  // 1、xmlns属性值是默认的beans namespace（即"http://www.springframework.org/schema/beans"）  // 2、profile属性值不能匹配-Dspring.profiles.active  if (this.delegate.isDefaultNamespace(root)) {  String profileSpec = root.getAttribute(*PROFILE\_ATTRIBUTE*);  if (StringUtils.*hasText*(profileSpec)) {  String[] specifiedProfiles = StringUtils.*tokenizeToStringArray*(profileSpec,  BeanDefinitionParserDelegate.*MULTI\_VALUE\_ATTRIBUTE\_DELIMITERS*);  if (!getReaderContext().getEnvironment().acceptsProfiles(specifiedProfiles)) {  if (logger.isInfoEnabled()) {  logger.info("Skipped XML bean definition file due to specified profiles ["  + profileSpec + "] not matching: " + getReaderContext().getResource());  }  return;  }  }  }  preProcessXml(root); // ~~钩子~~  // 核心解析方法（能通过上面的if代码块而未return时才说明<beans>需要被解析）  parseBeanDefinitions(root, this.delegate);  postProcessXml(root); // ~~钩子~~  this.delegate = parent;  } |
| protected void parseBeanDefinitions(Element root, BeanDefinitionParserDelegate delegate) {  if (delegate.isDefaultNamespace(root)) {  NodeList nl = root.getChildNodes();  for (int i = 0; i < nl.getLength(); i++) {  Node node = nl.item(i);  if (node instanceof Element) {  Element ele = (Element) node;  if (delegate.isDefaultNamespace(ele)) {  // 解析默认命名空间[即beans namespace] 下的元素(<import/>、<alias/>、<bean/> 和 <beans />)。  parseDefaultElement(ele, delegate);  } else {  // 解析其他 namespace 的元素，如<mvc />、<context />、<aop />等  delegate.parseCustomElement(ele);  }  }  }  }    else {  // <beans>的xmlns不是beans namespace  delegate.parseCustomElement(root);  }  } |

### parseDefaultElement

|  |
| --- |
| DefaultBeanDefinitionDocumentReader 182 |
| private void parseDefaultElement(Element ele, BeanDefinitionParserDelegate delegate) {  if (delegate.nodeNameEquals(ele, *IMPORT\_ELEMENT*)) { // 处理 <import /> 标签  importBeanDefinitionResource(ele);  } else if (delegate.nodeNameEquals(ele, *ALIAS\_ELEMENT*)) { // 处理 <alias /> 标签  processAliasRegistration(ele);  } else if (delegate.nodeNameEquals(ele, *BEAN\_ELEMENT*)) { // 处理 <bean /> 标签  processBeanDefinition(ele, delegate);  } else if (delegate.nodeNameEquals(ele, *NESTED\_BEANS\_ELEMENT*)) { // 处理嵌套的<beans /> 标签  doRegisterBeanDefinitions(ele); // 递归  }  } |

#### processBeanDefinition

|  |
| --- |
| DefaultBeanDefinitionDocumentReader 298 |
| protected void processBeanDefinition(Element ele, BeanDefinitionParserDelegate delegate) {  // 加载BeanDefinition，将 <bean> 节点转换为 BeanDefinitionHolder  BeanDefinitionHolder bdHolder = delegate.parseBeanDefinitionElement(ele);  if (bdHolder != null) {  ~~// 如果有自定义属性的话，进行相应的解析，先忽略~~  bdHolder = delegate.decorateBeanDefinitionIfRequired(ele, bdHolder);  try {  // 注册Bean  BeanDefinitionReaderUtils.*registerBeanDefinition*(bdHolder, getReaderContext().getRegistry());  } catch (BeanDefinitionStoreException ex) {  getReaderContext().error("Failed to register bean definition with name '" +  bdHolder.getBeanName() + "'", ele, ex);  }  ~~// 注册完成后，发送事件，本文不展开说这个~~  getReaderContext().fireComponentRegistered(new BeanComponentDefinition(bdHolder));  }  } |

# BeanDefinitionParserDelegate

## parseCustomElement

|  |
| --- |
| BeanDefinitionParserDelegate 1403 |
|  |
| public BeanDefinition parseCustomElement(Element ele, BeanDefinition containingBd) {  String namespaceUri = getNamespaceURI(ele);  NamespaceHandler handler = this.readerContext.getNamespaceHandlerResolver().resolve(namespaceUri);  if (handler == null) {  error("Unable to locate Spring NamespaceHandler for XML schema namespace [" + namespaceUri + "]", ele);  return null;  }  return handler.parse(ele, new ParserContext(this.readerContext, this, containingBd));  } |

## BeanDefinitionHolder parseBeanDefinitionElement()

|  |
| --- |
| BeanDefinitionParserDelegate 427 -> 436 |
| public BeanDefinitionHolder parseBeanDefinitionElement(Element ele) {  return parseBeanDefinitionElement(ele, null);  } |
| public BeanDefinitionHolder parseBeanDefinitionElement(Element ele, BeanDefinition containingBean) {  String id = ele.getAttribute(*ID\_ATTRIBUTE*);  String nameAttr = ele.getAttribute(*NAME\_ATTRIBUTE*);  List<String> aliases = new ArrayList<String>();  if (StringUtils.*hasLength*(nameAttr)) {  String[] nameArr = StringUtils.*tokenizeToStringArray*(nameAttr, *MULTI\_VALUE\_ATTRIBUTE\_DELIMITERS*);  aliases.addAll(Arrays.*asList*(nameArr));  }  String beanName = id;  if (!StringUtils.*hasText*(beanName) && !aliases.isEmpty()) {  beanName = aliases.remove(0);  if (logger.isDebugEnabled()) {  logger.debug("No XML 'id' specified - using '" + beanName +  "' as bean name and " + aliases + " as aliases");  }  }  if (containingBean == null) {  // 确保当前bean的beanName和aliases不和当前配置文件中其它已解析的bean的beanName和aliases冲突  checkNameUniqueness(beanName, aliases, ele);  }  // 根据 <bean ...>...</bean> 中的配置创建 BeanDefinition  AbstractBeanDefinition beanDefinition = parseBeanDefinitionElement(ele, beanName, containingBean);  if (beanDefinition != null) {  // <bean>没有设置id和name的处理  if (!StringUtils.*hasText*(beanName)) {  try {  if (containingBean != null) { // 什么时候会走进if？  beanName = BeanDefinitionReaderUtils.*generateBeanName*(  beanDefinition, this.readerContext.getRegistry(), true);  } else { // 正常走这  beanName = this.readerContext.generateBeanName(beanDefinition);  String beanClassName = beanDefinition.getBeanClassName();  if (beanClassName != null && beanName.startsWith(beanClassName) &&  beanName.length() > beanClassName.length() &&  !this.readerContext.getRegistry().isBeanNameInUse(beanClassName)) {  aliases.add(beanClassName);  }  }  if (logger.isDebugEnabled()) {  logger.debug("Neither XML 'id' nor 'name' specified - " +  "using generated bean name [" + beanName + "]");  }  } catch (Exception ex) {  error(ex.getMessage(), ele);  return null;  }  }  String[] aliasesArray = StringUtils.*toStringArray*(aliases);  return new BeanDefinitionHolder(beanDefinition, beanName, aliasesArray);  }  return null;  } |

### AbstractBeanDefinition parseBeanDefinitionElement ()

|  |
| --- |
| BeanDefinitionParserDelegate 521 |
| public AbstractBeanDefinition parseBeanDefinitionElement(  Element ele, String beanName, BeanDefinition containingBean) {  this.parseState.push(new BeanEntry(beanName));  String className = null;  if (ele.hasAttribute(*CLASS\_ATTRIBUTE*)) { // class属性  className = ele.getAttribute(*CLASS\_ATTRIBUTE*).trim();  }  try {  String parent = null;  if (ele.hasAttribute(*PARENT\_ATTRIBUTE*)) { // parent属性  parent = ele.getAttribute(*PARENT\_ATTRIBUTE*);  }  // 创建BeanDefinition，并设置beanClass和parentName属性  AbstractBeanDefinition bd = createBeanDefinition(className, parent);    // 设置 BeanDefinition 的一堆属性，这些属性定义在 AbstractBeanDefinition 中  parseBeanDefinitionAttributes(ele, beanName, containingBean, bd);    // 好吧，<bean>元素下可以配置<description>子元素  bd.setDescription(DomUtils.*getChildElementValueByTagName*(ele, *DESCRIPTION\_ELEMENT*));  /\*\*  \* 下面的一堆是解析 <bean>......</bean> 内部的子元素，  \* 解析出来以后的信息都放到 bd 的属性中  \*/  parseMetaElements(ele, bd); // <meta />  parseLookupOverrideSubElements(ele, bd.getMethodOverrides());// <lookup-method/>  parseReplacedMethodSubElements(ele, bd.getMethodOverrides());// <replaced-method/>  parseConstructorArgElements(ele, bd); // <constructor-arg />  parsePropertyElements(ele, bd); // <property />  parseQualifierElements(ele, bd); // <qualifier />  bd.setResource(this.readerContext.getResource());  bd.setSource(extractSource(ele));  return bd;  } catch (ClassNotFoundException ex) {  error("Bean class [" + className + "] not found", ele, ex);  } catch (NoClassDefFoundError err) {  error("Class that bean class [" + className + "] depends on not found", ele, err);  } catch (Throwable ex) {  error("Unexpected failure during bean definition parsing", ele, ex);  } finally {  this.parseState.pop();  }  return null;  } |

# 注册BeanDefinition

## BeanDefinitionReaderUtils [143]

|  |
| --- |
| BeanDefinitionReaderUtils 143 |
| public static void registerBeanDefinition( BeanDefinitionHolder definitionHolder, BeanDefinitionRegistry registry) throws BeanDefinitionStoreException {  // 根据 Bean的beanName注册其BeanDefinition  String beanName = definitionHolder.getBeanName();  registry.registerBeanDefinition(beanName, definitionHolder.getBeanDefinition());  // 如果Bean还有别名的话，也要根据别名全部注册一遍，不然根据别名就会找不到 Bean 了  String[] aliases = definitionHolder.getAliases();  if (aliases != null) {  for (String alias : aliases) {  // 别名也会注册到beanFactory中，结构是Map<alias,beanName>，见SimpleAliasRegistry  // 获取的时候，会先将 alias 转换为 beanName，然后再查找  registry.registerAlias(beanName, alias);  }  }  } |

## registerBeanDefinition(beanName, beanDefinition)

|  |
| --- |
| DefaultListableBeanFactory 788 |
| public void registerBeanDefinition(String beanName, BeanDefinition beanDefinition)  throws BeanDefinitionStoreException {  Assert.*hasText*(beanName, "Bean name must not be empty");  Assert.*notNull*(beanDefinition, "BeanDefinition must not be null");  if (beanDefinition instanceof AbstractBeanDefinition) {  try {  ((AbstractBeanDefinition) beanDefinition).validate();  } catch (BeanDefinitionValidationException ex) {  throw new BeanDefinitionStoreException(beanDefinition.getResourceDescription(),  beanName, "Validation of bean definition failed", ex);  }  }    // 处理要注册的Bean的beanName与已注册的Bean的BeanName重复的情况（所有的 Bean 注册后都会放入beanDefinitionMap 中）  BeanDefinition existingDefinition = this.beanDefinitionMap.get(beanName);  if (existingDefinition != null) {  // 如果不允许覆盖的话，抛异常  if (!isAllowBeanDefinitionOverriding()) {  throw new BeanDefinitionStoreException(……);  } else if (existingDefinition.getRole() < beanDefinition.getRole()) {  // log...用框架定义的 Bean 覆盖用户自定义的 Bean  } else if (!beanDefinition.equals(existingDefinition)) {  // log...用新的 Bean 覆盖旧的 Bean  } else {  // log...用同等的 Bean 覆盖旧的 Bean，这里指的是 equals 方法返回 true 的 Bean  }  this.beanDefinitionMap.put(beanName, beanDefinition); // 覆盖  }  // 不重复的情况  else {  // 一般不会进入if分支  if (hasBeanCreationStarted()) {  // Cannot modify startup-time collection elements anymore (for stable iteration)  synchronized (this.beanDefinitionMap) {  this.beanDefinitionMap.put(beanName, beanDefinition);  List<String> updatedDefinitions = new ArrayList<String>(this.beanDefinitionNames.size() + 1);  updatedDefinitions.addAll(this.beanDefinitionNames);  updatedDefinitions.add(beanName);  this.beanDefinitionNames = updatedDefinitions;  if (this.manualSingletonNames.contains(beanName)) {  Set<String> updatedSingletons = new LinkedHashSet<String>(this.manualSingletonNames);  updatedSingletons.remove(beanName);  this.manualSingletonNames = updatedSingletons;  }  }  }  // 正常会进入else分支  else {  // 将注册的< beanName, BeanDefinition>放到这个beanDefinitionMap中  this.beanDefinitionMap.put(beanName, beanDefinition);  // beanDefinitionNames[ArrayList]会按照 bean 配置的顺序保存每一个注册的 Bean 的BeanName  this.beanDefinitionNames.add(beanName);  // manualSingletonNames[LinkedHashSet] 代表的是手动注册的 singleton bean，注意这里是 remove 方法，到这里的 Bean 当然不是手动注册的，手动指的是通过调用registerSingleton(String beanName, Object singletonObject)方法注册的 bean 。  //这不是重点，解释只是为了不让大家疑惑。Spring 会在后面"手动"注册一些 Bean，如 "environment"、"systemProperties" 等 bean，我们自己也可以在运行时注册 Bean 到容器中的  this.manualSingletonNames.remove(beanName);  }  ~~// 这个不重要，在预初始化的时候会用到，不必管它。~~  this.frozenBeanDefinitionNames = null;  }  if (existingDefinition != null || containsSingleton(beanName)) {  resetBeanDefinition(beanName);  }  } |

# 初始化所有singleton beans

## finishBeanFactoryInitialization(beanFactory)

|  |
| --- |
| AbstractApplicationContext 834 |
| 到目前为止，应该说 BeanFactory 已经创建完成，并且所有的实现了 BeanFactoryPostProcessor 接口的 Bean 都已经初始化并且其中的 postProcessBeanFactory(factory) 方法已经得到回调执行了。而且 Spring 已经“手动”注册了一些特殊的 Bean，如 ‘environment’、‘systemProperties’ 等。剩下的就是初始化所有非懒加载的单例 bean 了。 |
| protected void finishBeanFactoryInitialization(ConfigurableListableBeanFactory beanFactory) {  // 首先，初始化名字为 conversionService 的 Bean。注意这里的初始化动作包装在 beanFactory.getBean(...) 中了  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(new StringValueResolver() {  @Override  public String resolveStringValue(String strVal) {  return getEnvironment().resolvePlaceholders(strVal);  }  });  }  // ~~先初始化 LoadTimeWeaverAware 类型的 Bean。 之前也说过，这是 AspectJ 相关的内容，放心跳过吧~~  // 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);  // 不希望在初始化 singleton beans时，还出现 bean 定义解析、加载、注册。  beanFactory.freezeConfiguration();  // 开始初始化  beanFactory.preInstantiateSingletons();  } |

## preInstantiateSingletons()

|  |
| --- |
| DefaultListableBeanFactory 723 |
| public void preInstantiateSingletons() throws BeansException {  if (logger.isDebugEnabled()) {  logger.debug("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<String>(this.beanDefinitionNames);  // Trigger initialization of all non-lazy singleton beans...  for (String beanName : beanNames) {  // 合并父 Bean 中的配置，注意 <bean id="" class="" parent="" /> 中的 parent  RootBeanDefinition bd = getMergedLocalBeanDefinition(beanName);  if (!bd.isAbstract() && bd.isSingleton() && !bd.isLazyInit()) {  if (isFactoryBean(beanName)) {  // 对于FactoryBean ，调用 getBean(“&”+beanName)  final FactoryBean<?> factory = (FactoryBean<?>) getBean(*FACTORY\_BEAN\_PREFIX* + beanName);  // ~~判断当前 FactoryBean 是否是 SmartFactoryBean 的实现，此处忽略，直接跳过~~  boolean isEagerInit;  if (System.*getSecurityManager*() != null && factory instanceof SmartFactoryBean) {  isEagerInit = AccessController.*doPrivileged*(new PrivilegedAction<Boolean>() {  @Override public Boolean run() {  return ((SmartFactoryBean<?>) factory).isEagerInit();  }  }, getAccessControlContext());  } else {  isEagerInit = (factory instanceof SmartFactoryBean &&  ((SmartFactoryBean<?>) factory).isEagerInit());  }  if (isEagerInit) {  getBean(beanName);  }  }  // 对于普通的 Bean，调用 getBean(beanName)进行初始化。  else {  getBean(beanName);  }  }  }  // 到这里说明所有的非懒加载的 singleton beans 已经完成了初始化。  // ~~如果我们定义的 bean 是实现了 SmartInitializingSingleton 接口的，那么在这里得到回调，忽略~~  for (String beanName : beanNames) {  Object singletonInstance = getSingleton(beanName);  if (singletonInstance instanceof SmartInitializingSingleton) {  final SmartInitializingSingleton smartSingleton = (SmartInitializingSingleton) singletonInstance;  if (System.*getSecurityManager*() != null) {  AccessController.*doPrivileged*(new PrivilegedAction<Object>() {  @Override public Object run() {  smartSingleton.afterSingletonsInstantiated();  return null;  }  }, getAccessControlContext());  }  else {  smartSingleton.afterSingletonsInstantiated();  }  }  }  } |

# getBean(…)

## doGetBean(name, requiredType, args, typeCheckOnly)

|  |
| --- |
| AbstractBeanFactory 235 |
| 每个getBean()都会转到doGetBean()。 |
| protected <T> T doGetBean( final String name, final Class<T> requiredType,  final Object[] args, boolean typeCheckOnly) throws BeansException {  // 获取一个 “正统的” beanName，处理两种情况，一个是前面说的 FactoryBean(前面带 ‘&’)，一个是别名问题  final String beanName = transformedBeanName(name);  // 返回值  Object bean;  // 检查下是不是已经创建过了  // Eagerly check singleton cache for manually registered singletons.  Object sharedInstance = getSingleton(beanName);  // 这里说下 args ，虽然看上去一点不重要。前面我们一路进来的时候都是 getBean(beanName)， 所以 args 传参其实是 null 的，但是如果 args 不为空的时候，那么意味着调用方不是希望获取 Bean，而是创建 Bean  if (sharedInstance != null && args == null) {  if (logger.isDebugEnabled()) {  if (isSingletonCurrentlyInCreation(beanName)) {  logger.debug("Returning eagerly cached instance of singleton bean '" + beanName +  "' that is not fully initialized yet - a consequence of a circular reference");  } else {  logger.debug("Returning cached instance of singleton bean '" + beanName + "'");  }  }  // 下面这个方法：如果是普通 Bean 的话，直接返回 sharedInstance，如果是 FactoryBean 的话，返回它创建的那个实例对象  bean = getObjectForBeanInstance(sharedInstance, name, beanName, null);  }    else {  // 首先，在创建前，如果发现已经正在创建此 beanName 的 prototype 类型的 bean，那么抛异常， 往往是因为陷入了循环引用  if (isPrototypeCurrentlyInCreation(beanName)) {  throw new BeanCurrentlyInCreationException(beanName);  }  // 如果在当前容器中不存在此beanName对应的BeanDefinition，将尝试委托给父容器来完成getBean()  BeanFactory parentBeanFactory = getParentBeanFactory();  if (parentBeanFactory != null && !containsBeanDefinition(beanName)) {  // Not found -> check parent.  String nameToLookup = originalBeanName(name);  if (args != null) {  // Delegation to parent with explicit args.  return (T) parentBeanFactory.getBean(nameToLookup, args);  } else {  // No args -> delegate to standard getBean method.  return parentBeanFactory.getBean(nameToLookup, requiredType);  }  }  // 从getBean()过来的typeCheckOnly 均为 false，会将当前 beanName 放入一个 alreadyCreated 的 Set 集合中。  if (!typeCheckOnly) {  markBeanAsCreated(beanName);  }  /\*  \* 稍稍总结一下：  \* 到这里的话，要准备创建 Bean 了，对于 singleton 的 Bean 来说，容器中还没创建过此 Bean；  \* 对于 prototype 的 Bean 来说，本来就是要创建一个新的 Bean。  \*/  try {  final RootBeanDefinition mbd = getMergedLocalBeanDefinition(beanName);  checkMergedBeanDefinition(mbd, beanName, args);  // 先初始化依赖的所有 Bean，这里的依赖指的是 depends-on 中定义的依赖  String[] dependsOn = mbd.getDependsOn();  if (dependsOn != null) {  for (String dep : dependsOn) {  // 如果有循环依赖，抛异常  if (isDependent(beanName, dep)) {  throw new BeanCreationException(mbd.getResourceDescription(), beanName,  "Circular depends-on relationship between '" + beanName + "' and '" + dep + "'");  }  // 注册一下依赖关系  registerDependentBean(dep, beanName);  try {  // 先初始化被依赖项  getBean(dep);  } catch (NoSuchBeanDefinitionException ex) {  throw new BeanCreationException(mbd.getResourceDescription(), beanName,  "'" + beanName + "' depends on missing bean '" + dep + "'", ex);  }  }  }  // 1、如果是 singleton scope 的，创建 singleton 的实例  if (mbd.isSingleton()) {  sharedInstance = getSingleton(beanName, new ObjectFactory<Object>() {  @Override  public Object getObject() throws BeansException {  try {  return createBean(beanName, mbd, args);  }  catch (BeansException ex) {  // Explicitly remove instance from singleton cache: It might have been put there  // eagerly by the creation process, to allow for circular reference resolution.  // Also remove any beans that received a temporary reference to the bean.  destroySingleton(beanName);  throw ex;  }  }  });  bean = getObjectForBeanInstance(sharedInstance, name, beanName, mbd);  }  // 2、如果是 prototype scope 的，创建 prototype 的实例  else if (mbd.isPrototype()) {  // It's a prototype -> create a new instance.  Object prototypeInstance = null;  try {  beforePrototypeCreation(beanName);  prototypeInstance = createBean(beanName, mbd, args);  }  finally {  afterPrototypeCreation(beanName);  }  bean = getObjectForBeanInstance(prototypeInstance, name, beanName, mbd);  }  // 3、如果不是 singleton 和 prototype 的话，需要委托给相应的实现类来处理  else {  String scopeName = mbd.getScope();  final Scope scope = this.scopes.get(scopeName);  if (scope == null) {  throw new IllegalStateException("No Scope registered for scope name '" + scopeName + "'");  }  try {  Object scopedInstance = scope.get(beanName, new ObjectFactory<Object>() {  @Override  public Object getObject() throws BeansException {  beforePrototypeCreation(beanName);  try {  return createBean(beanName, mbd, args);  }  finally {  afterPrototypeCreation(beanName);  }  }  });  bean = getObjectForBeanInstance(scopedInstance, name, beanName, mbd);  } catch (IllegalStateException ex) {  throw new BeanCreationException(beanName,  "Scope '" + scopeName + "' is not active for the current thread; consider " + "defining a scoped proxy for this bean if you intend to refer to it from a singleton",  ex);  }  }  }  catch (BeansException ex) {  cleanupAfterBeanCreationFailure(beanName);  throw ex;  }  }  // Check if required type matches the type of the actual bean instance.  if (requiredType != null && bean != null && !requiredType.isInstance(bean)) {  try {  return getTypeConverter().convertIfNecessary(bean, requiredType);  } catch (TypeMismatchException ex) {  if (logger.isDebugEnabled()) {  logger.debug("Failed to convert bean '" + name + "' to required type '" +  ClassUtils.*getQualifiedName*(requiredType) + "'", ex);  }  throw new BeanNotOfRequiredTypeException(name, requiredType, bean.getClass());  }  }  return (T) bean;  } |

## createBean(beanName, mbd, args)

|  |
| --- |
| AbstractAutowireCapableBeanFactory 445 |
|  |
| protected Object createBean(String beanName, RootBeanDefinition mbd, Object[] args)  throws BeanCreationException {  if (logger.isDebugEnabled()) {  logger.debug("Creating instance of bean '" + beanName + "'");  }  RootBeanDefinition mbdToUse = mbd;  // 确保 BeanDefinition 中的 Class 被加载  // 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);  }  // 准备方法覆写(MethodOverrides)，方法覆写概念与 bean 定义中的 <lookup-method /> 和 <replaced-method />相关，感兴趣的话可以回到 bean 解析的地方看看对这两个标签的解析。  try {  mbdToUse.prepareMethodOverrides();  } catch (BeanDefinitionValidationException ex) {  throw new BeanDefinitionStoreException(mbdToUse.getResourceDescription(),  beanName, "Validation of method overrides failed", ex);  }  // ~~这一步给了我们机会可以用InstantiationAwareBeanPostProcessor来返回AOP代理，先跳过~~  try {  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);  }  // 重点，doCreateBean  Object beanInstance = doCreateBean(beanName, mbdToUse, args);  if (logger.isDebugEnabled()) {  logger.debug("Finished creating instance of bean '" + beanName + "'");  }  return beanInstance;  } |

## doCreateBean(beanName, mbdToUse, args)

|  |
| --- |
| AbstractAutowireCapableBeanFactory 502 |
| protected Object doCreateBean(final String beanName, final RootBeanDefinition mbd, final Object[] args)  throws BeanCreationException {  // Instantiate the bean.  BeanWrapper instanceWrapper = null;  if (mbd.isSingleton()) {  instanceWrapper = this.factoryBeanInstanceCache.remove(beanName);  }  // 说明不是 FactoryBean，这里实例化 Bean，这里非常关键，细节之后再说  if (instanceWrapper == null) {  instanceWrapper = createBeanInstance(beanName, mbd, args);  }  // 这个就是 Bean 里面的 我们定义的类 的实例，很多地方我直接描述成 "bean 实例"  final Object bean = (instanceWrapper != null ? instanceWrapper.getWrappedInstance() : null);  Class<?> beanType = (instanceWrapper != null ? instanceWrapper.getWrappedClass() : null);  mbd.resolvedTargetType = beanType;  // ~~涉及MergedBeanDefinitionPostProcessor接口，很少用，建议跳过~~  // 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;  }  }  // 下面这块代码是为了解决循环依赖的问题  // 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.isDebugEnabled()) {  logger.debug("Eagerly caching bean '" + beanName +  "' to allow for resolving potential circular references");  }  addSingletonFactory(beanName, new ObjectFactory<Object>() {  @Override  public Object getObject() throws BeansException {  return getEarlyBeanReference(beanName, mbd, bean);  }  });  }  // Initialize the bean instance.  Object exposedObject = bean;  try {  // 这一步也是非常关键的，这一步负责属性装配，因为前面的实例只是实例化了，并没有设值，这里就是设值  populateBean(beanName, mbd, instanceWrapper);  if (exposedObject != null) {  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<String>(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;  } |