Spring 版本 **spring-context-5.0.6.RELEASE.jar**

解读BeanPostPrecessor 源码

加载我们配置类@Configurable类文件是AnnotationConfigApplicationContext开始

找到我们的AnnotationConfigApplicationContext类

|  |
| --- |
| AnnotationConfigApplicationContext anno =  new AnnotationConfigApplicationContext(Config.class) |

|  |
| --- |
| **public** AnnotationConfigApplicationContext(Class... annotatedClasses) {  **this**();  **this**.register(annotatedClasses);  **this**.refresh(); } |

找到this.refresh()方法，进行刷新操作

|  |
| --- |
| **public void** refresh() **throws** BeansException, IllegalStateException {  Object var1 = **this**.startupShutdownMonitor;  **synchronized**(**this**.startupShutdownMonitor) {  **this**.prepareRefresh();  ConfigurableListableBeanFactory beanFactory = **this**.obtainFreshBeanFactory();  **this**.prepareBeanFactory(beanFactory);   **try** {  **this**.postProcessBeanFactory(beanFactory);  **this**.invokeBeanFactoryPostProcessors(beanFactory);  **this**.registerBeanPostProcessors(beanFactory);  **this**.initMessageSource();  **this**.initApplicationEventMulticaster();  **this**.onRefresh();  **this**.registerListeners();  **this**.finishBeanFactoryInitialization(beanFactory);  **this**.finishRefresh();  } **catch** (BeansException var9) {  **if** (**this**.logger.isWarnEnabled()) {  **this**.logger.warn(**"Exception encountered during context initialization - cancelling refresh attempt: "** + var9);  }   **this**.destroyBeans();  **this**.cancelRefresh(var9);  **throw** var9;  } **finally** {  **this**.resetCommonCaches();  }   } } |

找到其中的 this.finishBeanFactoryInitialization(beanFactory) 完成Bean工厂初始化

|  |
| --- |
| **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**);  beanFactory.freezeConfiguration();  beanFactory.preInstantiateSingletons(); } |

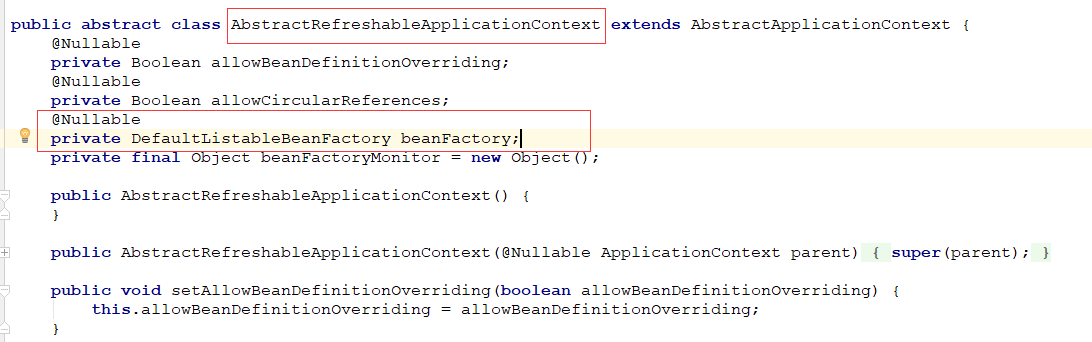
找到beanFactory.preInstantiateSingletons() 实例化之前的单例

这个方法是implements ConfigurableListableBeanFactory 的

preInstantiateSingletons

被

这里beanFactory 实际上是



DefaultListableBeanFactory.java

|  |
| --- |
| **public class** DefaultListableBeanFactory **extends** AbstractAutowireCapableBeanFactory **implements** ConfigurableListableBeanFactory, BeanDefinitionRegistry, Serializable {…} |

ConfigurableListableBeanFactory.java

|  |
| --- |
| **public interface** ConfigurableListableBeanFactory **extends** ListableBeanFactory, AutowireCapableBeanFactory, ConfigurableBeanFactory {  **void** ignoreDependencyType(Class<?> var1);   **void** ignoreDependencyInterface(Class<?> var1);   **void** registerResolvableDependency(Class<?> var1, @Nullable Object var2);   **boolean** isAutowireCandidate(String var1, DependencyDescriptor var2) **throws** NoSuchBeanDefinitionException;   BeanDefinition getBeanDefinition(String var1) **throws** NoSuchBeanDefinitionException;   Iterator<String> getBeanNamesIterator();   **void** clearMetadataCache();   **void** freezeConfiguration();   **boolean** isConfigurationFrozen();   **void** preInstantiateSingletons() **throws** BeansException; } |

所以这里的 beanFactory.preInstantiateSingletons()

最终是由 DefaultListableBeanFactory.java实现

|  |
| --- |
| **public void** preInstantiateSingletons() **throws** BeansException {  **if** (**this**.logger.isDebugEnabled()) {  **this**.logger.debug(**"Pre-instantiating singletons in "** + **this**);  }   List<String> beanNames = **new** ArrayList(**this**.beanDefinitionNames);  Iterator var2 = beanNames.iterator();   **while**(**true**) {  String beanName;  Object bean;  **do** {  **while**(**true**) {  RootBeanDefinition bd;  **do** {  **do** {  **do** {  **if** (!var2.hasNext()) {  var2 = beanNames.iterator();   **while**(var2.hasNext()) {  beanName = (String)var2.next();  Object singletonInstance = **this**.getSingleton(beanName);  **if** (singletonInstance **instanceof** SmartInitializingSingleton) {  SmartInitializingSingleton smartSingleton = (SmartInitializingSingleton)singletonInstance;  **if** (System.getSecurityManager() != **null**) {  AccessController.doPrivileged(() -> {  smartSingleton.afterSingletonsInstantiated();  **return null**;  }, **this**.getAccessControlContext());  } **else** {  smartSingleton.afterSingletonsInstantiated();  }  }  }   **return**;  }   beanName = (String)var2.next();  bd = **this**.getMergedLocalBeanDefinition(beanName);  } **while**(bd.isAbstract());  } **while**(!bd.isSingleton());  } **while**(bd.isLazyInit());   **if** (**this**.isFactoryBean(beanName)) {  bean = **this**.getBean(**"&"** + beanName);  **break**;  }   **this**.getBean(beanName);  }  } **while**(!(bean **instanceof** FactoryBean));   FactoryBean<?> factory = (FactoryBean)bean;  **boolean** isEagerInit;  **if** (System.getSecurityManager() != **null** && factory **instanceof** SmartFactoryBean) {  SmartFactoryBean var10000 = (SmartFactoryBean)factory;  ((SmartFactoryBean)factory).getClass();  isEagerInit = ((Boolean)AccessController.doPrivileged(var10000::isEagerInit, **this**.getAccessControlContext())).booleanValue();  } **else** {  isEagerInit = factory **instanceof** SmartFactoryBean && ((SmartFactoryBean)factory).isEagerInit();  }   **if** (isEagerInit) {  **this**.getBean(beanName);  }  } } |

|  |
| --- |
| **public class** DefaultListableBeanFactory **extends** AbstractAutowireCapableBeanFactory **implements** ConfigurableListableBeanFactory, BeanDefinitionRegistry, Serializable {…} |

DefaultListableBeanFactory.java extends

AbstractAutowireCapableBeanFactory.java

AbstractAutowireCapableBeanFactory.java

|  |
| --- |
| **public abstract class** AbstractAutowireCapableBeanFactory **extends** AbstractBeanFactory **implements** AutowireCapableBeanFactory {…} |

所有这个 this.getbean(beanName) 会执行AbstractBeanFactory父类的 getBean方法

|  |
| --- |
| **public abstract class** AbstractBeanFactory **extends** FactoryBeanRegistrySupport **implements** ConfigurableBeanFactory {  **public** Object getBean(String name) **throws** BeansException {  **return this**.doGetBean(name, (Class)**null**, (Object[])**null**, **false**);  }  **protected** <T> T doGetBean(String name, @Nullable Class<T> requiredType, @Nullable Object[] args, **boolean** typeCheckOnly) **throws** BeansException {  String beanName = **this**.transformedBeanName(name);  Object sharedInstance = **this**.getSingleton(beanName);  Object bean;  **if** (sharedInstance != **null** && args == **null**) {  **if** (**this**.logger.isDebugEnabled()) {  **if** (**this**.isSingletonCurrentlyInCreation(beanName)) {  **this**.logger.debug(**"Returning eagerly cached instance of singleton bean '"** + beanName + **"' that is not fully initialized yet - a consequence of a circular reference"**);  } **else** {  **this**.logger.debug(**"Returning cached instance of singleton bean '"** + beanName + **"'"**);  }  }   bean = **this**.getObjectForBeanInstance(sharedInstance, name, beanName, (RootBeanDefinition)**null**);  } **else** {  **if** (**this**.isPrototypeCurrentlyInCreation(beanName)) {  **throw new** BeanCurrentlyInCreationException(beanName);  }   BeanFactory parentBeanFactory = **this**.getParentBeanFactory();  **if** (parentBeanFactory != **null** && !**this**.containsBeanDefinition(beanName)) {  String nameToLookup = **this**.originalBeanName(name);  **if** (parentBeanFactory **instanceof** AbstractBeanFactory) {  **return** ((AbstractBeanFactory)parentBeanFactory).doGetBean(nameToLookup, requiredType, args, typeCheckOnly);  }   **if** (args != **null**) {  **return** parentBeanFactory.getBean(nameToLookup, args);  }   **return** parentBeanFactory.getBean(nameToLookup, requiredType);  }   **if** (!typeCheckOnly) {  **this**.markBeanAsCreated(beanName);  }   **try** {  RootBeanDefinition mbd = **this**.getMergedLocalBeanDefinition(beanName);  **this**.checkMergedBeanDefinition(mbd, beanName, args);  String[] dependsOn = mbd.getDependsOn();  String[] var11;  **if** (dependsOn != **null**) {  var11 = dependsOn;  **int** var12 = dependsOn.length;   **for**(**int** var13 = 0; var13 < var12; ++var13) {  String dep = var11[var13];  **if** (**this**.isDependent(beanName, dep)) {  **throw new** BeanCreationException(mbd.getResourceDescription(), beanName, **"Circular depends-on relationship between '"** + beanName + **"' and '"** + dep + **"'"**);  }   **this**.registerDependentBean(dep, beanName);   **try** {  **this**.getBean(dep);  } **catch** (NoSuchBeanDefinitionException var24) {  **throw new** BeanCreationException(mbd.getResourceDescription(), beanName, **"'"** + beanName + **"' depends on missing bean '"** + dep + **"'"**, var24);  }  }  }   **if** (mbd.isSingleton()) {  sharedInstance = **this**.getSingleton(beanName, () -> {  **try** {  **return this**.createBean(beanName, mbd, args);  } **catch** (BeansException var5) {  **this**.destroySingleton(beanName);  **throw** var5;  }  });  bean = **this**.getObjectForBeanInstance(sharedInstance, name, beanName, mbd);  } **else if** (mbd.isPrototype()) {  var11 = **null**;   Object prototypeInstance;  **try** {  **this**.beforePrototypeCreation(beanName);  prototypeInstance = **this**.createBean(beanName, mbd, args);  } **finally** {  **this**.afterPrototypeCreation(beanName);  }   bean = **this**.getObjectForBeanInstance(prototypeInstance, name, beanName, mbd);  } **else** {  String scopeName = mbd.getScope();  Scope 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, () -> {  **this**.beforePrototypeCreation(beanName);   Object var4;  **try** {  var4 = **this**.createBean(beanName, mbd, args);  } **finally** {  **this**.afterPrototypeCreation(beanName);  }   **return** var4;  });  bean = **this**.getObjectForBeanInstance(scopedInstance, name, beanName, mbd);  } **catch** (IllegalStateException var23) {  **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"**, var23);  }  }  } **catch** (BeansException var26) {  **this**.cleanupAfterBeanCreationFailure(beanName);  **throw** var26;  }  }   **if** (requiredType != **null** && !requiredType.isInstance(bean)) {  **try** {  T convertedBean = **this**.getTypeConverter().convertIfNecessary(bean, requiredType);  **if** (convertedBean == **null**) {  **throw new** BeanNotOfRequiredTypeException(name, requiredType, bean.getClass());  } **else** {  **return** convertedBean;  }  } **catch** (TypeMismatchException var25) {  **if** (**this**.logger.isDebugEnabled()) {  **this**.logger.debug(**"Failed to convert bean '"** + name + **"' to required type '"** + ClassUtils.getQualifiedName(requiredType) + **"'"**, var25);  }   **throw new** BeanNotOfRequiredTypeException(name, requiredType, bean.getClass());  }  } **else** {  **return** bean;  } }  **protected abstract** Object createBean(String var1, RootBeanDefinition var2, @Nullable Object[] var3) **throws** BeanCreationException;  } |

然后执行doGetBean(name, **null**, **null**, **false**) -- >

createBean(beanName, mbd, args) -->

我们发现createBean 在这里是abstract方法没有实现我们看是否子类去实现了 也就是我们的AbstractAutowireCapableBeanFactory.java

|  |
| --- |
| **public abstract class** AbstractAutowireCapableBeanFactory **extends** AbstractBeanFactory **implements** AutowireCapableBeanFactory {  **public** <T> T createBean(Class<T> beanClass) **throws** BeansException {  RootBeanDefinition bd = **new** RootBeanDefinition(beanClass);  bd.setScope(**"prototype"**);  bd.allowCaching = ClassUtils.isCacheSafe(beanClass, **this**.getBeanClassLoader());  **return this**.createBean(beanClass.getName(), bd, (Object[])**null**); }  **protected** Object createBean(String beanName, RootBeanDefinition mbd, @Nullable Object[] args) **throws** BeanCreationException {  **if** (**this**.logger.isDebugEnabled()) {  **this**.logger.debug(**"Creating instance of bean '"** + beanName + **"'"**);  }   RootBeanDefinition mbdToUse = mbd;  Class<?> resolvedClass = **this**.resolveBeanClass(mbd, beanName, **new** Class[0]);  **if** (resolvedClass != **null** && !mbd.hasBeanClass() && mbd.getBeanClassName() != **null**) {  mbdToUse = **new** RootBeanDefinition(mbd);  mbdToUse.setBeanClass(resolvedClass);  }   **try** {  mbdToUse.prepareMethodOverrides();  } **catch** (BeanDefinitionValidationException var9) {  **throw new** BeanDefinitionStoreException(mbdToUse.getResourceDescription(), beanName, **"Validation of method overrides failed"**, var9);  }   Object beanInstance;  **try** {  beanInstance = **this**.resolveBeforeInstantiation(beanName, mbdToUse);  **if** (beanInstance != **null**) {  **return** beanInstance;  }  } **catch** (Throwable var10) {  **throw new** BeanCreationException(mbdToUse.getResourceDescription(), beanName, **"BeanPostProcessor before instantiation of bean failed"**, var10);  }   **try** {  beanInstance = **this**.doCreateBean(beanName, mbdToUse, args);  **if** (**this**.logger.isDebugEnabled()) {  **this**.logger.debug(**"Finished creating instance of bean '"** + beanName + **"'"**);  }   **return** beanInstance;  } **catch** (ImplicitlyAppearedSingletonException | BeanCreationException var7) {  **throw** var7;  } **catch** (Throwable var8) {  **throw new** BeanCreationException(mbdToUse.getResourceDescription(), beanName, **"Unexpected exception during bean creation"**, var8);  } }  **protected** Object doCreateBean(String beanName, RootBeanDefinition mbd, @Nullable Object[] args) **throws** BeanCreationException {  BeanWrapper instanceWrapper = **null**;  **if** (mbd.isSingleton()) {  instanceWrapper = (BeanWrapper)**this**.factoryBeanInstanceCache.remove(beanName);  }   **if** (instanceWrapper == **null**) {  instanceWrapper = **this**.createBeanInstance(beanName, mbd, args);  }   Object bean = instanceWrapper.getWrappedInstance();  Class<?> beanType = instanceWrapper.getWrappedClass();  **if** (beanType != NullBean.**class**) {  mbd.resolvedTargetType = beanType;  }   Object var7 = mbd.postProcessingLock;  **synchronized**(mbd.postProcessingLock) {  **if** (!mbd.postProcessed) {  **try** {  **this**.applyMergedBeanDefinitionPostProcessors(mbd, beanType, beanName);  } **catch** (Throwable var17) {  **throw new** BeanCreationException(mbd.getResourceDescription(), beanName, **"Post-processing of merged bean definition failed"**, var17);  }   mbd.postProcessed = **true**;  }  }   **boolean** earlySingletonExposure = mbd.isSingleton() && **this**.allowCircularReferences && **this**.isSingletonCurrentlyInCreation(beanName);  **if** (earlySingletonExposure) {  **if** (**this**.logger.isDebugEnabled()) {  **this**.logger.debug(**"Eagerly caching bean '"** + beanName + **"' to allow for resolving potential circular references"**);  }   **this**.addSingletonFactory(beanName, () -> {  **return this**.getEarlyBeanReference(beanName, mbd, bean);  });  }   Object exposedObject = bean;   **try** {  **this**.populateBean(beanName, mbd, instanceWrapper);  exposedObject = **this**.initializeBean(beanName, exposedObject, mbd);  } **catch** (Throwable var18) {  **if** (var18 **instanceof** BeanCreationException && beanName.equals(((BeanCreationException)var18).getBeanName())) {  **throw** (BeanCreationException)var18;  }   **throw new** BeanCreationException(mbd.getResourceDescription(), beanName, **"Initialization of bean failed"**, var18);  }   **if** (earlySingletonExposure) {  Object earlySingletonReference = **this**.getSingleton(beanName, **false**);  **if** (earlySingletonReference != **null**) {  **if** (exposedObject == bean) {  exposedObject = earlySingletonReference;  } **else if** (!**this**.allowRawInjectionDespiteWrapping && **this**.hasDependentBean(beanName)) {  String[] dependentBeans = **this**.getDependentBeans(beanName);  Set<String> actualDependentBeans = **new** LinkedHashSet(dependentBeans.length);  String[] var12 = dependentBeans;  **int** var13 = dependentBeans.length;   **for**(**int** var14 = 0; var14 < var13; ++var14) {  String dependentBean = var12[var14];  **if** (!**this**.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."**);  }  }  }  }   **try** {  **this**.registerDisposableBeanIfNecessary(beanName, bean, mbd);  **return** exposedObject;  } **catch** (BeanDefinitionValidationException var16) {  **throw new** BeanCreationException(mbd.getResourceDescription(), beanName, **"Invalid destruction signature"**, var16);  } }  } |

在 AbstractAutowireCapableBeanFactory.java

createBean(beanName, mbd, args)-->

doCreateBean(beanName, mbdToUse, args)-->

createBeanInstance(beanName, mbd, args)(完成bean创建)-->

populateBean(beanName, mbd, instanceWrapper)(属性赋值)-->

initializeBean(beanName, exposedObject, mbd)(Bean初始化)->

再看initializeBean 方法里面

|  |
| --- |
| **protected** Object initializeBean(String beanName, Object bean, @Nullable RootBeanDefinition mbd) {  **if** (System.getSecurityManager() != **null**) {  AccessController.doPrivileged(() -> {  **this**.invokeAwareMethods(beanName, bean);  **return null**;  }, **this**.getAccessControlContext());  } **else** {  **this**.invokeAwareMethods(beanName, bean);  }   Object wrappedBean = bean;  **if** (mbd == **null** || !mbd.isSynthetic()) {  wrappedBean = **this**.applyBeanPostProcessorsBeforeInitialization(bean, beanName);  }   **try** {  **this**.invokeInitMethods(beanName, wrappedBean, mbd);  } **catch** (Throwable var6) {  **throw new** BeanCreationException(mbd != **null** ? mbd.getResourceDescription() : **null**, beanName, **"Invocation of init method failed"**, var6);  }   **if** (mbd == **null** || !mbd.isSynthetic()) {  wrappedBean = **this**.applyBeanPostProcessorsAfterInitialization(wrappedBean, beanName);  }   **return** wrappedBean; } |

**这三个方法 执行顺序** 注意 与我们 实现 BeanPostPrecessor 继承的方法

|  |
| --- |
| @Component **public class** CustomBeanPostProcessor **implements** BeanPostProcessor {   *// 这个方法在初始化之前调用* @Nullable  @Override  **public** Object postProcessBeforeInitialization(Object bean, String beanName) **throws** BeansException {  System.***out***.println(beanName + **"CustomBeanPostProcessor 1"**);  **return** bean;  }   *//这个方法在初始化之后调用* @Nullable  @Override  **public** Object postProcessAfterInitialization(Object bean, String beanName) **throws** BeansException {  System.***out***.println(beanName + **"CustomBeanPostProcessor 2"**);  **return** bean;  } } |

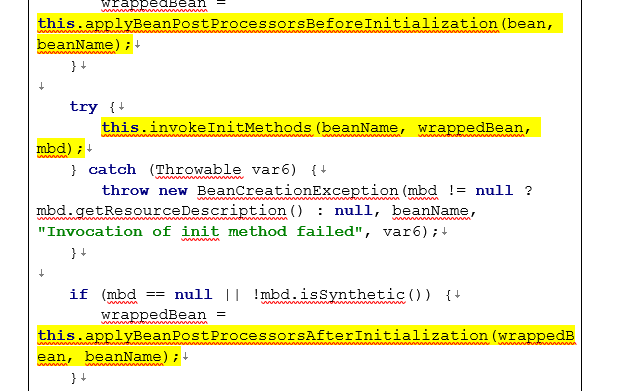
所以你就会明白 这两个方法为什么是 spring ioc 初始化Construct 🡪

postProcessBeforeInitialization 方法 🡪

init 方法 🡪

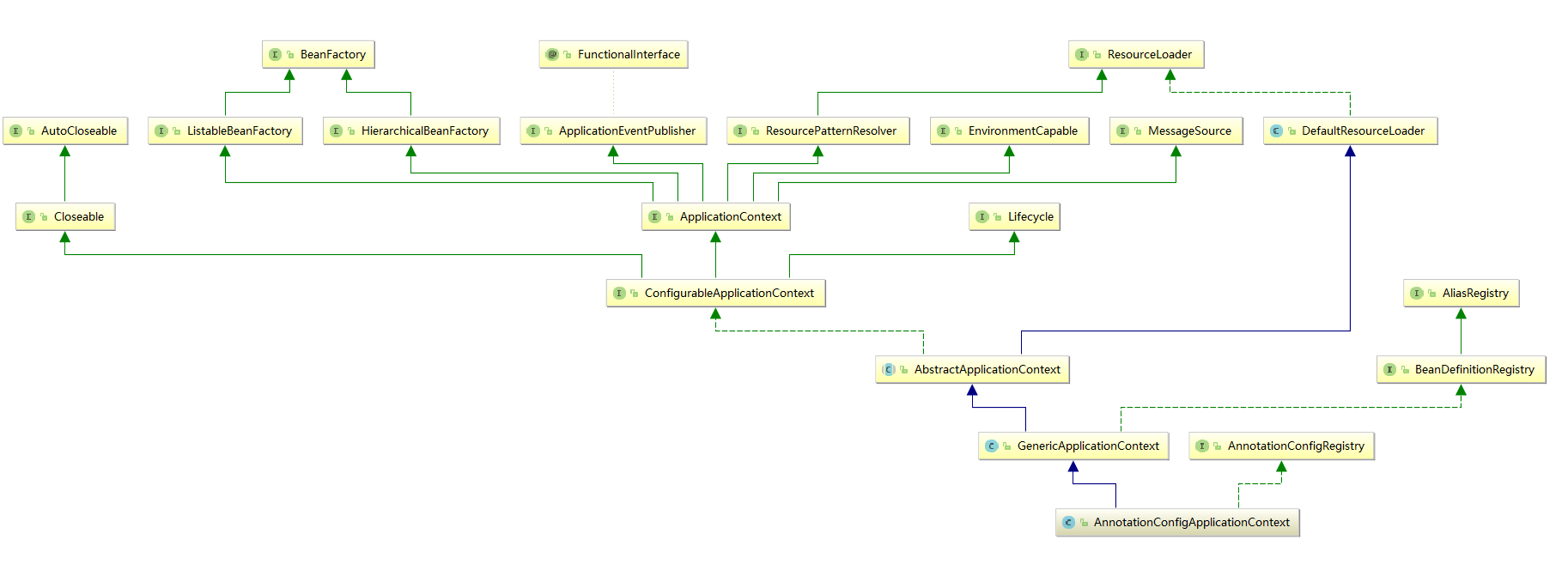
postProcessAfterInitialization 方法

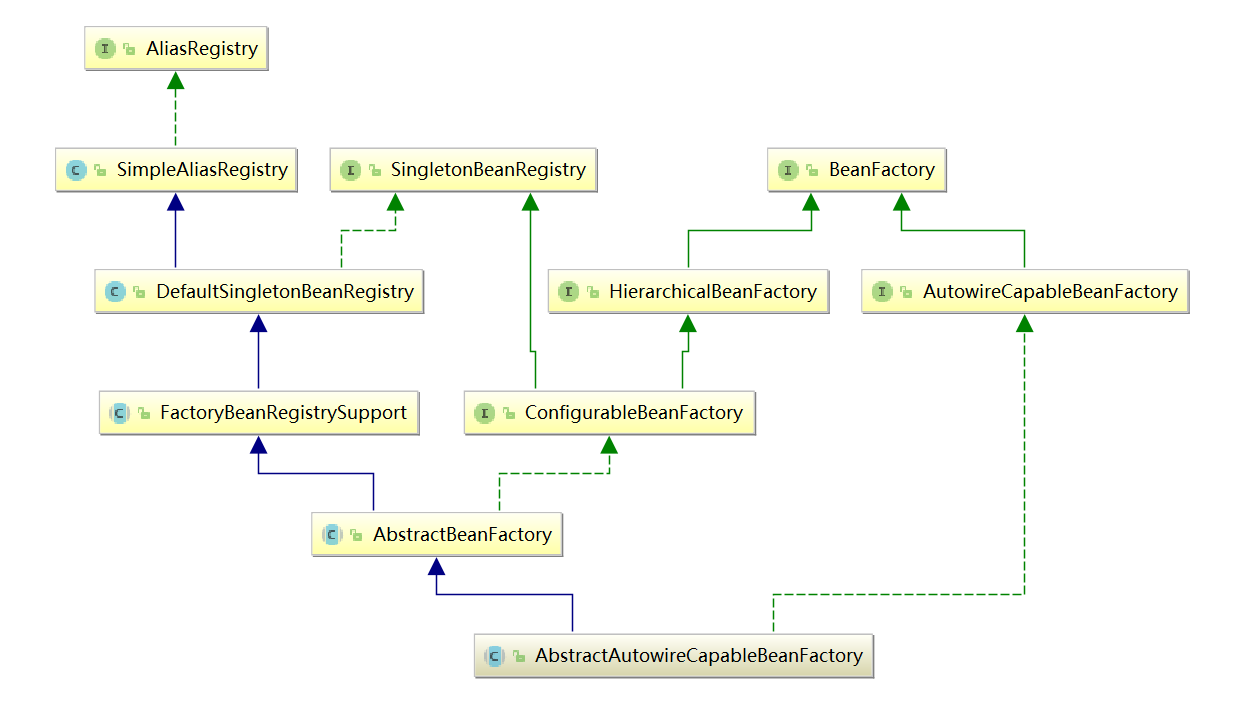
对应这里的 代码顺序



This.invokeIniMethods()利用java的反射执行init方法

了解下 AnnotationConfigApplicationContext.java





BeanPostProcessor原理:

可从容器类跟进顺序为:

AnnotationConfigApplicationContext-->refresh()-->

finishBeanFactoryInitialization(beanFactory)--->

beanFactory.preInstantiateSingletons()-->

760行getBean(beanName)--->

199行doGetBean(name, **null**, **null**, **false**)-->

317行createBean(beanName, mbd, args)-->

501行doCreateBean(beanName, mbdToUse, args)-->

541行createBeanInstance(beanName, mbd, args)(完成bean创建)-->

578行populateBean(beanName, mbd, instanceWrapper)(属性赋值)-->

579行initializeBean(beanName, exposedObject, mbd)(Bean初始化)->

1069行到1710行,后置处理器完成对init方法的前后处理.

最终得到如下如下

createBeanInstance(beanName, mbd, args)(完成bean创建)

populateBean(beanName, mbd, instanceWrapper); 给bean进行属性赋值

initializeBean() //初始化Bean方法内容如下,后置处理器对init方法的前后处理

{

applyBeanPostProcessorsBeforeInitialization(wrappedBean, beanName);

**invokeInitMethods(beanName, wrappedBean, mbd)** //执行自定义初始化

applyBeanPostProcessorsAfterInitialization(wrappedBean, beanName)

}

从以上分析不难发现,bean的生命周期为bean的创建, 初始化, 当容器关闭时对单实例的bean进行销毁.