# Spring

## Ioc 流程

* ApplicationContext context = new AnnotationConfigApplicationContext(MyConfiguration.class);
* public AnnotationConfigApplicationContext(Class<?>... annotatedClasses) {

this();

register(annotatedClasses);

refresh();

}

* AbstractApplicationContext. refresh()

invokeBeanFactoryPostProcessors(beanFactory);

// 调用 BeanFactoryPostProcessor 各个实现类的 postProcessBeanFactory(factory) 方法

* PostProcessorRegistrationDelegate

private static void invokeBeanDefinitionRegistryPostProcessors(

Collection<? extends BeanDefinitionRegistryPostProcessor> postProcessors, BeanDefinitionRegistry registry) {

for (BeanDefinitionRegistryPostProcessor postProcessor : postProcessors) {

postProcessor.postProcessBeanDefinitionRegistry(registry);

}

}

* ConfigurationClassPostProcessor

public void processConfigBeanDefinitions(BeanDefinitionRegistry registry)

if (sbr != null && !sbr.containsSingleton(IMPORT\_REGISTRY\_BEAN\_NAME)) {

sbr.registerSingleton(IMPORT\_REGISTRY\_BEAN\_NAME, parser.getImportRegistry());

}

* DefaultListableBeanFactory

@Override

public void registerSingleton(String beanName, Object singletonObject) throws IllegalStateException {

super.registerSingleton(beanName, singletonObject);

updateManualSingletonNames(set -> set.add(beanName), set -> !this.beanDefinitionMap.containsKey(beanName));

clearByTypeCache();

}

## IOC主要方法

### registerBeanDefinition

this.beanDefinitionMap.put(beanName, beanDefinition);

//将BeanDefinition放到这个map中，key:bean名称 value:beanDefinition

this.beanDefinitionNames.add(beanName);

//会按照bean配置的顺序保存每一个注册的Bean的名字

类: DefaultListableBeanFactory

refresh()

AbstractApplicationContext

### registerSingleton

public void registerSingleton(String beanName, Object singletonObject) throws IllegalStateException {

synchronized (this.singletonObjects) {

Object oldObject = this.singletonObjects.get(beanName);

if (oldObject != null) {

throw new IllegalStateException("Could not register object [" + singletonObject +

"] under bean name '" + beanName + "': there is already object [" + oldObject + "] bound");

}

addSingleton(beanName, singletonObject);

}

}

protected void addSingleton(String beanName, Object singletonObject) {

synchronized (this.singletonObjects) {

this.singletonObjects.put(beanName, singletonObject); //加入到单例缓存

this.singletonFactories.remove(beanName); // 三级缓存删除

this.earlySingletonObjects.remove(beanName); //二级缓存删除

this.registeredSingletons.add(beanName); //用来记录保存已经处理的bean

}

}

### 标记Bean创建

alreadyCreated容器

protected void markBeanAsCreated(String beanName) {

if (!this.alreadyCreated.contains(beanName)) {

synchronized (this.mergedBeanDefinitions) {

if (!this.alreadyCreated.contains(beanName)) {

// Let the bean definition get re-merged now that we're actually creating

// the bean... just in case some of its metadata changed in the meantime.

clearMergedBeanDefinition(beanName);

this.alreadyCreated.add(beanName);

}

}

}

}

### dependsOn

String[] dependsOn = mbd.getDependsOn();

if (dependsOn != null) {

for (String dep : dependsOn) {

if (isDependent(beanName, dep)) {

throw new BeanCreationException(mbd.getResourceDescription(), beanName,

"Circular depends-on relationship between '" + beanName + "' and '" + dep + "'");

}

registerDependentBean(dep, beanName);

try {

getBean(dep);

}

catch (NoSuchBeanDefinitionException ex) {

throw new BeanCreationException(mbd.getResourceDescription(), beanName,

"'" + beanName + "' depends on missing bean '" + dep + "'", ex);

}

}

}

### addSingleton

protected void addSingleton(String beanName, Object singletonObject) {  
 synchronized (this.singletonObjects) {  
 this.singletonObjects.put(beanName, singletonObject); //加入到单例缓存  
 this.singletonFactories.remove(beanName); // 三级缓存删除  
 this.earlySingletonObjects.remove(beanName); //二级缓存删除  
 this.registeredSingletons.add(beanName); //用来记录保存已经处理的bean  
 }  
}

### addSingletonFactory

protected void addSingletonFactory(String beanName, ObjectFactory<?> singletonFactory) {  
 Assert.*notNull*(singletonFactory, "Singleton factory must not be null");  
 synchronized (this.singletonObjects) {  
 if (!this.singletonObjects.containsKey(beanName)) {  
 this.singletonFactories.put(beanName, singletonFactory);  
 this.earlySingletonObjects.remove(beanName);  
 this.registeredSingletons.add(beanName);  
 }  
 }  
}

### getSingleton 三级缓存 重要

DefaultSingletonBeanRegistry

protected Object getSingleton(String beanName, boolean allowEarlyReference) {  
 Object singletonObject = this.singletonObjects.get(beanName); // 先从singletonObjects寻找 一级缓存  
 if (singletonObject == null && isSingletonCurrentlyInCreation(beanName)) {  
 synchronized (this.singletonObjects) {  
 singletonObject = this.earlySingletonObjects.get(beanName); // 如果找不到，再从earlySingletonObjects寻找 二级缓存  
 if (singletonObject == null && allowEarlyReference) {  
 ObjectFactory<?> singletonFactory = this.singletonFactories.get(beanName); // 从singletonFactories寻找对应的singleton的工厂 三级缓存  
 if (singletonFactory != null) {  
 singletonObject = singletonFactory.getObject(); // 调用工厂的getObject方法，得到对应的SingletonBean  
 this.earlySingletonObjects.put(beanName, singletonObject); // 并放入earlySingletonObjects中  
 this.singletonFactories.remove(beanName);  
 }  
 }  
 }  
 }  
 return singletonObject;  
}

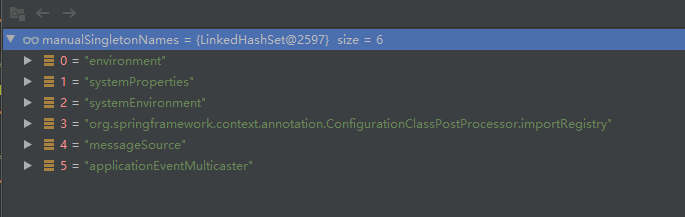
### alias

public class SimpleAliasRegistry implements AliasRegistry {  
  
 */\*\* Map from alias to canonical name. \*/* private final Map<String, String> aliasMap = new ConcurrentHashMap<>(16);  
  
  
 @Override  
 public void registerAlias(String name, String alias) {

### manualSingletonNames

手工注册 系统自动注册的

DefaultListableBeanFactory



### 依赖注入

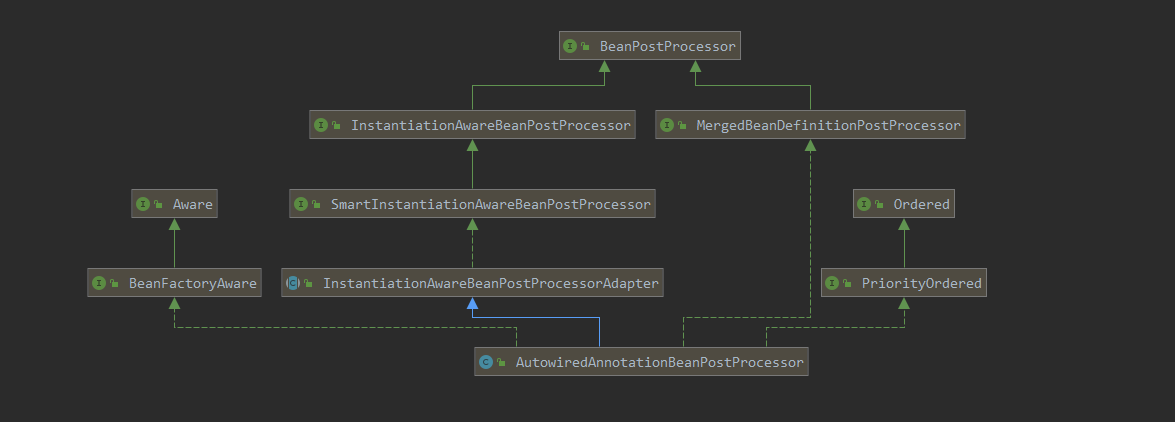
DefaultListableBeanFactory

protected Map<String, Object> findAutowireCandidates(  
 @Nullable String beanName, Class<?> requiredType, DependencyDescriptor descriptor)

*/\*\*  
 \* Determine whether the given beanName/candidateName pair indicates a self reference,  
 \* i.e. whether the candidate points back to the original bean or to a factory method  
 \* on the original bean.  
 \*/*private boolean isSelfReference(@Nullable String beanName, @Nullable String candidateName) {  
 return (beanName != null && candidateName != null &&  
 (beanName.equals(candidateName) || (containsBeanDefinition(candidateName) &&  
 beanName.equals(getMergedLocalBeanDefinition(candidateName).getFactoryBeanName()))));  
}

### InjectionMetadata

### AutowiredAnnotationBeanPostProcessor Autowired注入



#### @Autowired注入

private class AutowiredFieldElement extends InjectionMetadata.InjectedElement {

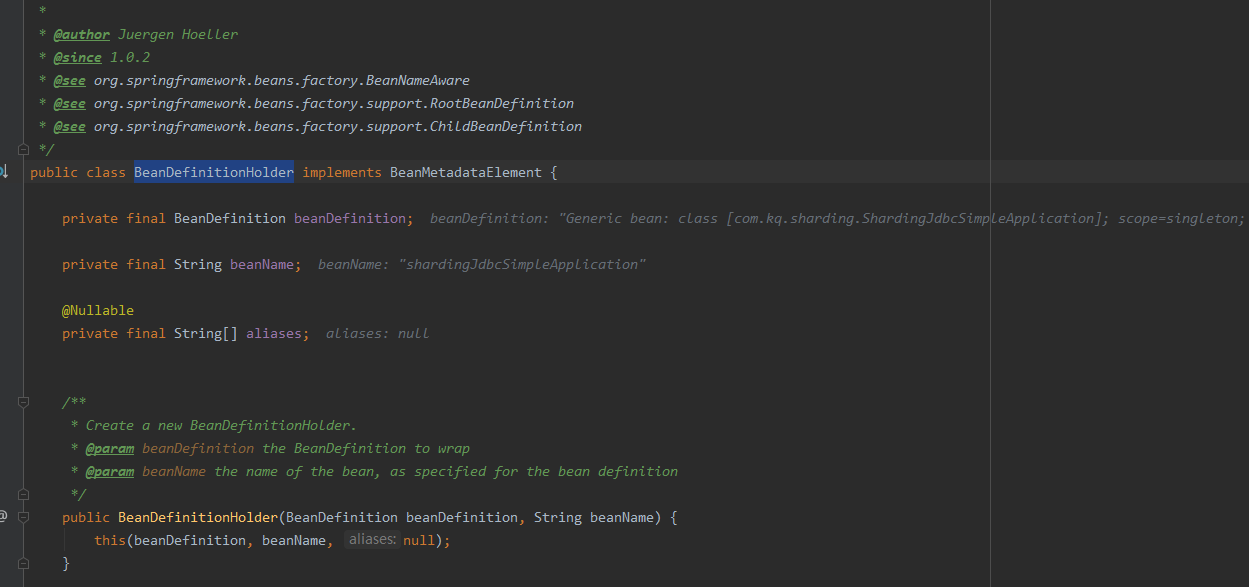
if (value != null) {  
 ReflectionUtils.makeAccessible(field);  
 field.set(bean, value); // Autowired注入  
}

}

## Bean

### BeanDefinitionHolder

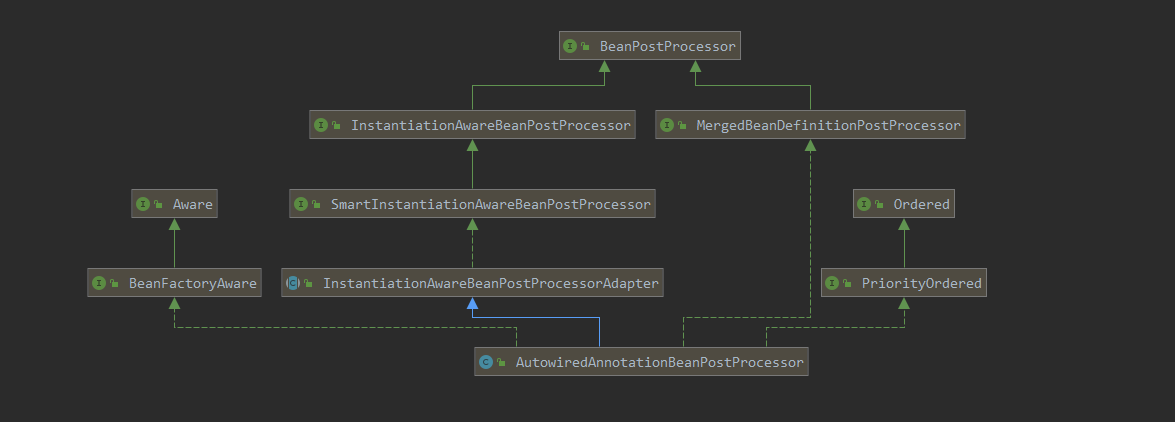
Bean持有者

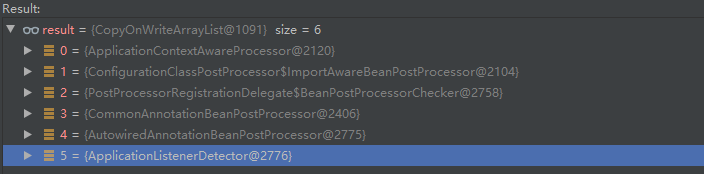


### DefaultListableBeanFactory(重要)

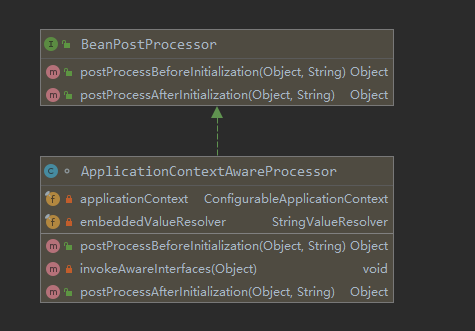
## BeanPostProcessor

refresh() -> prepareBeanFactory(beanFactory); //这里添加BeanPostProcessor





### ApplicationContextAwareProcessor



#### postProcessBeforeInitialization

注入ApplicationContext

##### 注入ApplicationContext

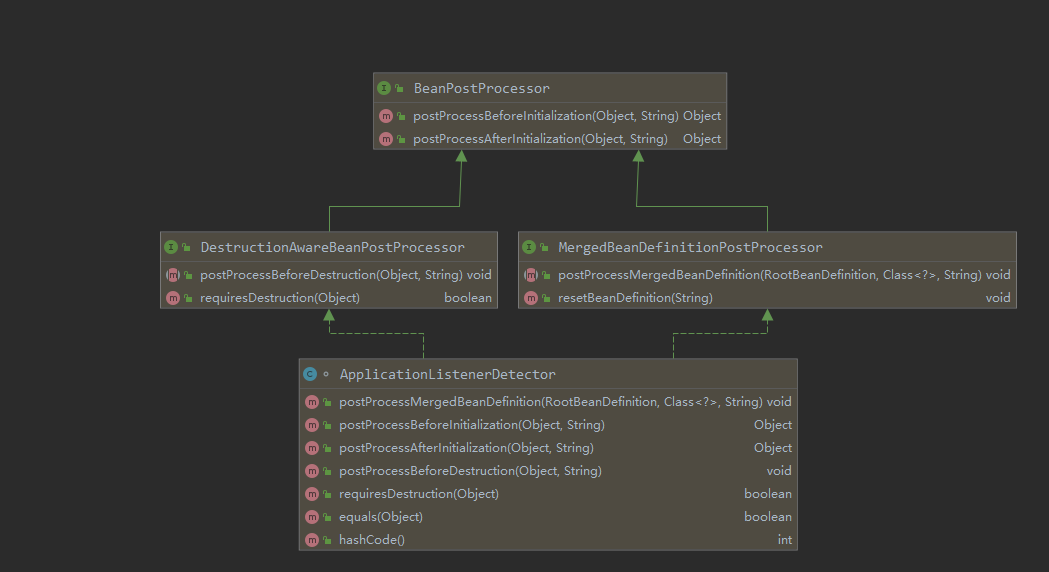
private void invokeAwareInterfaces(Object bean) {  
 if (bean instanceof Aware) {  
 if (bean instanceof EnvironmentAware) {  
 ((EnvironmentAware) bean).setEnvironment(this.applicationContext.getEnvironment());  
 }  
 if (bean instanceof EmbeddedValueResolverAware) {  
 ((EmbeddedValueResolverAware) bean).setEmbeddedValueResolver(this.embeddedValueResolver);  
 }  
 if (bean instanceof ResourceLoaderAware) {  
 ((ResourceLoaderAware) bean).setResourceLoader(this.applicationContext);  
 }  
 if (bean instanceof ApplicationEventPublisherAware) {  
 ((ApplicationEventPublisherAware) bean).setApplicationEventPublisher(this.applicationContext);  
 }  
 if (bean instanceof MessageSourceAware) {  
 ((MessageSourceAware) bean).setMessageSource(this.applicationContext);  
 }  
 if (bean instanceof ApplicationContextAware) {  
 ((ApplicationContextAware) bean).setApplicationContext(this.applicationContext);  
 }  
 }  
}

#### postProcessAfterInitialization

直接返回bean

@Override  
public Object postProcessAfterInitialization(Object bean, String beanName) {  
 return bean;  
}

### ApplicationListenerDetector



#### postProcessBeforeInitialization

直接返回bean

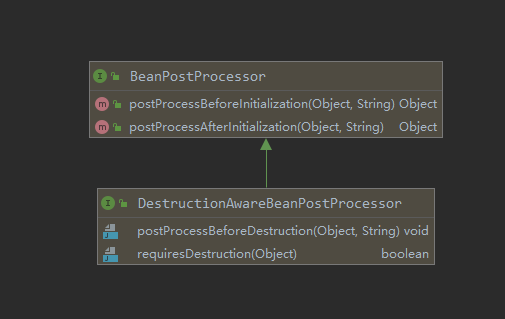
public Object postProcessBeforeInitialization(Object bean, String beanName) {  
 return bean;  
}

#### postProcessAfterInitialization

public Object postProcessAfterInitialization(Object bean, String beanName) {  
 if (bean instanceof ApplicationListener) {  
 // potentially not detected as a listener by getBeanNamesForType retrieval  
 Boolean flag = this.singletonNames.get(beanName);  
 if (Boolean.*TRUE*.equals(flag)) {  
 // singleton bean (top-level or inner): register on the fly  
 this.applicationContext.addApplicationListener((ApplicationListener<?>) bean);  
 }  
 else if (Boolean.*FALSE*.equals(flag)) {  
 if (*logger*.isWarnEnabled() && !this.applicationContext.containsBean(beanName)) {  
 // inner bean with other scope - can't reliably process events  
 *logger*.warn("Inner bean '" + beanName + "' implements ApplicationListener interface " +  
 "but is not reachable for event multicasting by its containing ApplicationContext " +  
 "because it does not have singleton scope. Only top-level listener beans are allowed " +  
 "to be of non-singleton scope.");  
 }  
 this.singletonNames.remove(beanName);  
 }  
 }  
 return bean;  
}

### DestructionAwareBeanPostProcessor 销毁对象

#### 类图



#### postProcessAfterInitialization

判断是否是ApplicationListener类型，如果是，则注册事件监听器

@Override  
public Object postProcessAfterInitialization(Object bean, String beanName) {  
 if (bean instanceof ApplicationListener) { //bean是ApplicationListener类型  
 // potentially not detected as a listener by getBeanNamesForType retrieval  
 Boolean flag = this.singletonNames.get(beanName); //是否单例标志  
 if (Boolean.*TRUE*.equals(flag)) { //是否单例 true:单例  
 // singleton bean (top-level or inner): register on the fly  
 this.applicationContext.addApplicationListener((ApplicationListener<?>) bean); //注册事件监听器  
 }  
 else if (Boolean.*FALSE*.equals(flag)) {  
 this.singletonNames.remove(beanName);  
 }  
 }  
 return bean;  
}

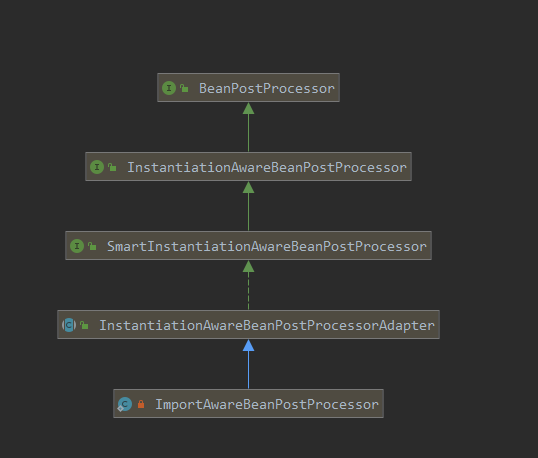
#### postProcessBeforeDestruction

销毁之前调用，是否是aplicationListener类型，如果是，则删除该注册事件监听器

*/\*\* 销毁对象之前 执行 \*/*@Override  
public void postProcessBeforeDestruction(Object bean, String beanName) {  
 if (bean instanceof ApplicationListener) { // 如果bean ApplicationListener类型的  
 try {  
 ApplicationEventMulticaster multicaster = this.applicationContext.getApplicationEventMulticaster();  
 multicaster.removeApplicationListener((ApplicationListener<?>) bean); // 删除该注册事件监听器  
 multicaster.removeApplicationListenerBean(beanName);  
 }  
 catch (IllegalStateException ex) {  
 // ApplicationEventMulticaster not initialized yet - no need to remove a listener  
 }  
 }  
}

### ImportAwareBeanPostProcessor

ConfigurationClassPostProcessor内部类

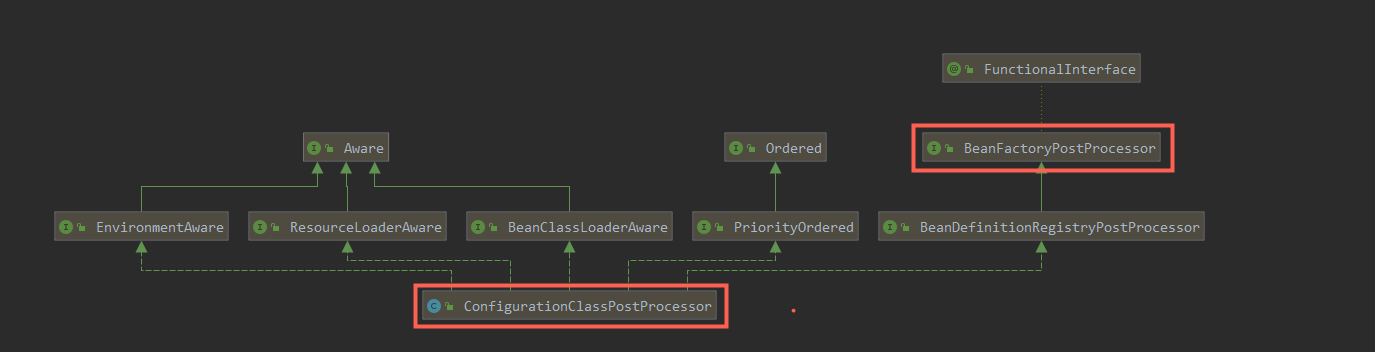


private static class ImportAwareBeanPostProcessor extends InstantiationAwareBeanPostProcessorAdapter {  
  
 private final BeanFactory beanFactory;  
  
 public ImportAwareBeanPostProcessor(BeanFactory beanFactory) {  
 this.beanFactory = beanFactory;  
 }  
  
 @Override  
 public PropertyValues postProcessProperties(@Nullable PropertyValues pvs, Object bean, String beanName) {  
 // Inject the BeanFactory before AutowiredAnnotationBeanPostProcessor's  
 // postProcessProperties method attempts to autowire other configuration beans.  
 if (bean instanceof EnhancedConfiguration) {  
 ((EnhancedConfiguration) bean).setBeanFactory(this.beanFactory);  
 }  
 return pvs;  
 }  
  
 @Override  
 public Object postProcessBeforeInitialization(Object bean, String beanName) {  
 if (bean instanceof ImportAware) {  
 ImportRegistry ir = this.beanFactory.getBean(*IMPORT\_REGISTRY\_BEAN\_NAME*, ImportRegistry.class);  
 AnnotationMetadata importingClass = ir.getImportingClassFor(bean.getClass().getSuperclass().getName());  
 if (importingClass != null) {  
 ((ImportAware) bean).setImportMetadata(importingClass);  
 }  
 }  
 return bean;  
 }  
}

## BeanFactoryPostProcessor

void postProcessBeanFactory(ConfigurableListableBeanFactory beanFactory) throws BeansException;

### ConfigurationClassPostProcessor



#### Configuration类增强

@Override  
public void postProcessBeanFactory(ConfigurableListableBeanFactory beanFactory) {  
 if (!this.registriesPostProcessed.contains(factoryId)) {  
 // BeanDefinitionRegistryPostProcessor hook apparently not supported...  
 // Simply call processConfigurationClasses lazily at this point then.  
 processConfigBeanDefinitions((BeanDefinitionRegistry) beanFactory);  
 }  
  
 enhanceConfigurationClasses(beanFactory); //Configuration类增强 cglib  
 beanFactory.addBeanPostProcessor(new ImportAwareBeanPostProcessor(beanFactory));  
}

## PostProcessorRegistrationDelegate

### 处理 BeanDefinitionRegistry

public static void invokeBeanFactoryPostProcessors(  
 ConfigurableListableBeanFactory beanFactory**,** List<BeanFactoryPostProcessor> beanFactoryPostProcessors) {  
  
 // Invoke BeanDefinitionRegistryPostProcessors first, if any. 处理过的Bean  
 Set<String> processedBeans = new HashSet<>()**;** if (beanFactory instanceof BeanDefinitionRegistry) {  
 BeanDefinitionRegistry registry = (BeanDefinitionRegistry) beanFactory**;**

**// 正常的**BeanFactoryPostProcessorList<BeanFactoryPostProcessor> regularPostProcessors = new ArrayList<>()**;** // 非BeanDefinitionRegistryPostProcessor类型的BeanFactoryPostProcessor  
 List<BeanDefinitionRegistryPostProcessor> registryProcessors = new ArrayList<>()**;** // 注册的BeanDefinitionRegistryPostProcessor类型的BeanFactoryPostProcessor  
 // 这里的beanFactoryPostProcessors 是前面1. AnnotationConfigUtils.registerAnnotationConfigProcessors 2. AbstractApplicationContext.prepareBeanFactory  
 for (BeanFactoryPostProcessor postProcessor : beanFactoryPostProcessors) {  
 if (postProcessor instanceof BeanDefinitionRegistryPostProcessor) {  
 BeanDefinitionRegistryPostProcessor registryProcessor =  
 (BeanDefinitionRegistryPostProcessor) postProcessor**;** registryProcessor.postProcessBeanDefinitionRegistry(registry)**;** registryProcessors.add(registryProcessor)**;** }  
 else {  
 regularPostProcessors.add(postProcessor)**;** }  
 }

先判断是否是BeanDefinitionRegistry，如果是，则得到BeanFactory内部得到beanFactoryPostProcessors然后遍历，判断是否是BeanDefinitionRegistryPostProcessor，如果是，则执行postProcessBeanDefinitionRegistry，并把实例加到注册的Processor列表里，如果不是，则把实例放到正常的Processor列表里

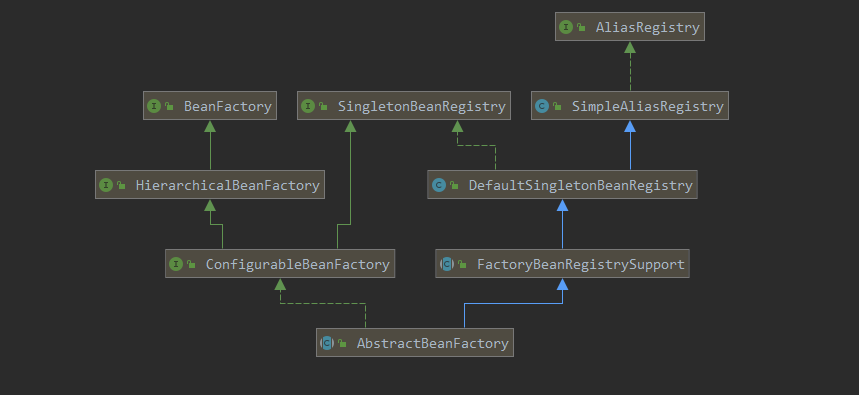
### 从Spring内部得到实现了PriorityOrdered接口的BeanDefinitionRegistryPostProcessor

// PriorityOrdered, Ordered, and the rest.  
List<BeanDefinitionRegistryPostProcessor> currentRegistryProcessors = new ArrayList<>()**;**// First, invoke the BeanDefinitionRegistryPostProcessors that implement PriorityOrdered.  
String[] postProcessorNames = // 从Spring容器中得到所有的BeanDefinitionRegistryPostProcessor  
 beanFactory.getBeanNamesForType(BeanDefinitionRegistryPostProcessor.class**,** true**,** false)**;**for (String ppName : postProcessorNames) { // 第1个Processors:这里找到key:org.springframework.context.annotation.internalConfigurationAnnotationProcessor value: ConfigurationClassPostProcessor  
 if (beanFactory.isTypeMatch(ppName**,** PriorityOrdered.class)) { // 优先取PriorityOrdered  
 currentRegistryProcessors.add(beanFactory.getBean(ppName**,** BeanDefinitionRegistryPostProcessor.class))**;** //spring找到ConfigurationClassPostProcessor并实例化  
 processedBeans.add(ppName)**;** }  
}  
*sortPostProcessors*(currentRegistryProcessors**,** beanFactory)**;** // 排序  
registryProcessors.addAll(currentRegistryProcessors)**;** // 把currentRegistryProcessors添加到registryProcessors  
*invokeBeanDefinitionRegistryPostProcessors*(currentRegistryProcessors**,** registry)**;**currentRegistryProcessors.clear()**;**

## BeanFactory

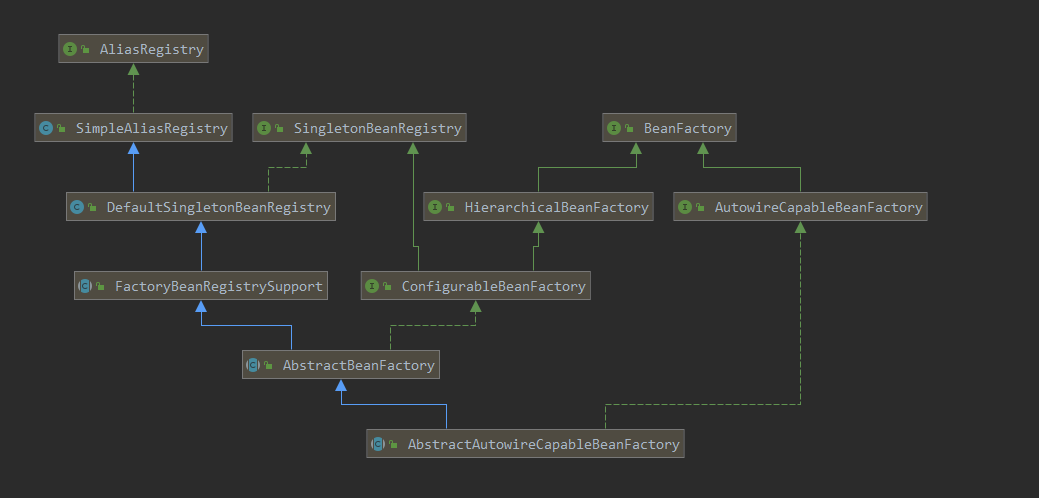
#### AbstractBeanFactory

##### 类图

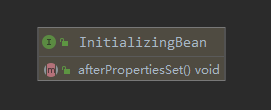


#### AbstractAutowireCapableBeanFactory

##### 类图

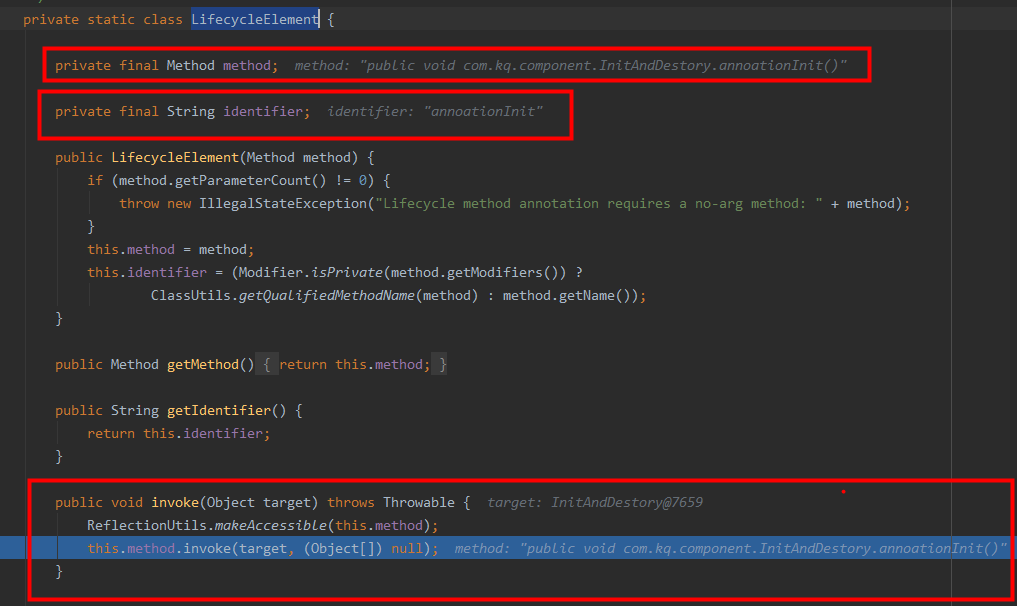


## InitializingBean实现



### 注解PostConstruct

javax.annotation.PostConstruct

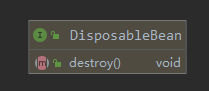


### 先后顺序

构造函数 -> @PostConstruct -> InitializingBean接口

## 销毁详解

### DisposableBean

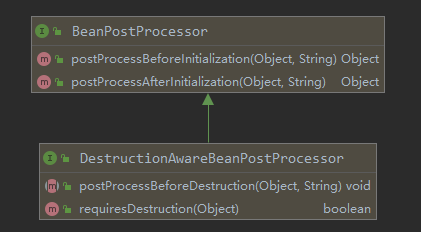


### 注解@PreDestroy

javax.annotation包下

### DestructionAwareBeanPostProcessor

#### 类图



##### 主要方法

###### postProcessBeforeDestruction

void postProcessBeforeDestruction(Object bean, String beanName) throws BeansException;

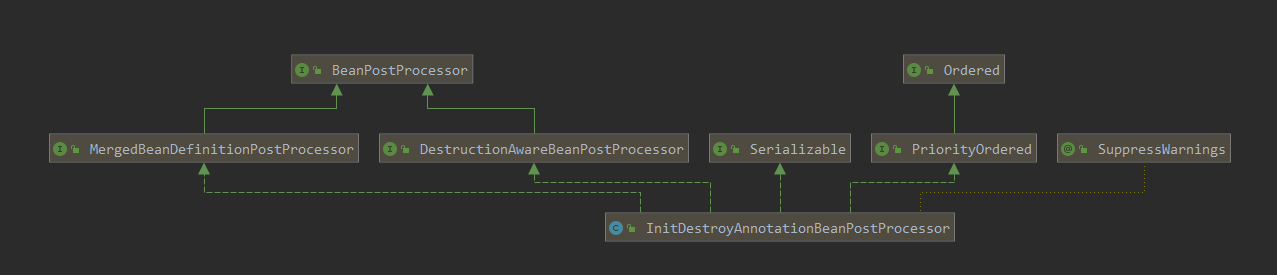
###### requiresDestruction

default boolean requiresDestruction(Object bean) {  
 return true;  
}

### DestructionAwareBeanPostProcessor实现类

#### ApplicationListenerDetector

#### InitDestroyAnnotationBeanPostProcessor



## 初始化

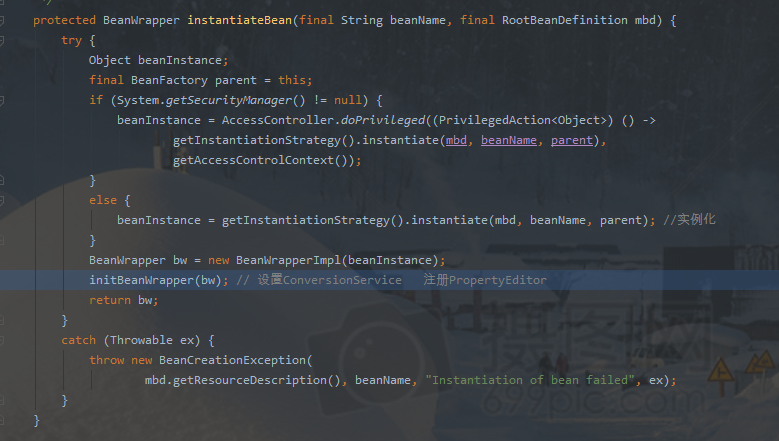
BeanUtils.instantiateClass(constructorToUse);

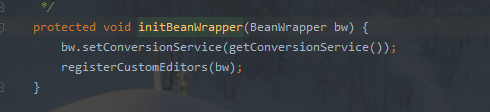
beanInstance = getInstantiationStrategy().instantiate(mbd, beanName, parent);

BeanWrapper bw = new BeanWrapperImpl(beanInstance);

initBeanWrapper(bw); //类型转化注册

### 类型转化





## BeanMethod

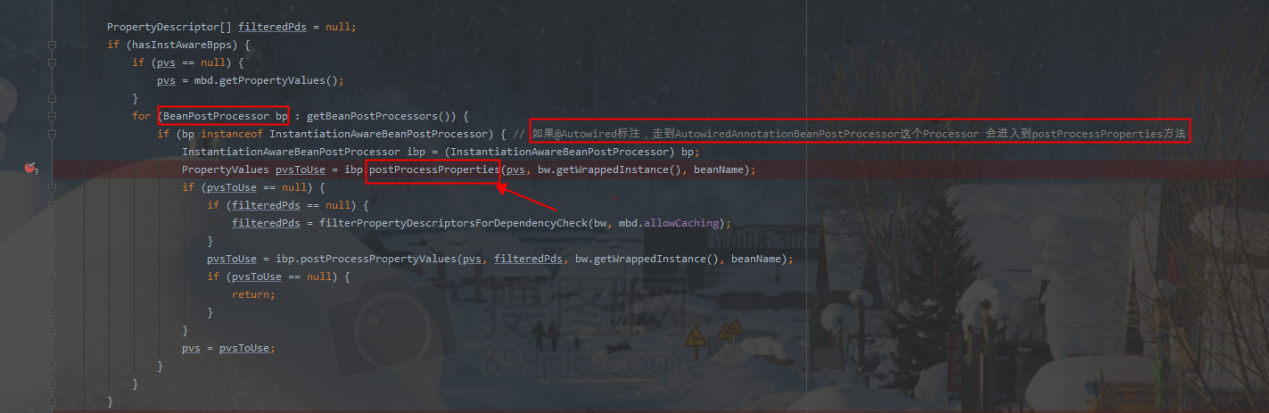
ConfigurationClassBeanDefinitionReader.loadBeanDefinitionsForBeanMethod

## @Autowired 注入

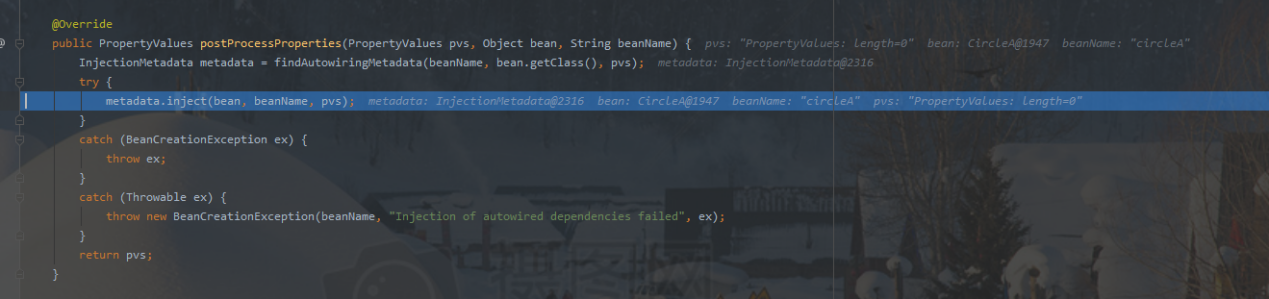
### populateBean (AbstractAutowireCapableBeanFactory )

protected void populateBean(String beanName, RootBeanDefinition mbd, @Nullable BeanWrapper bw) {

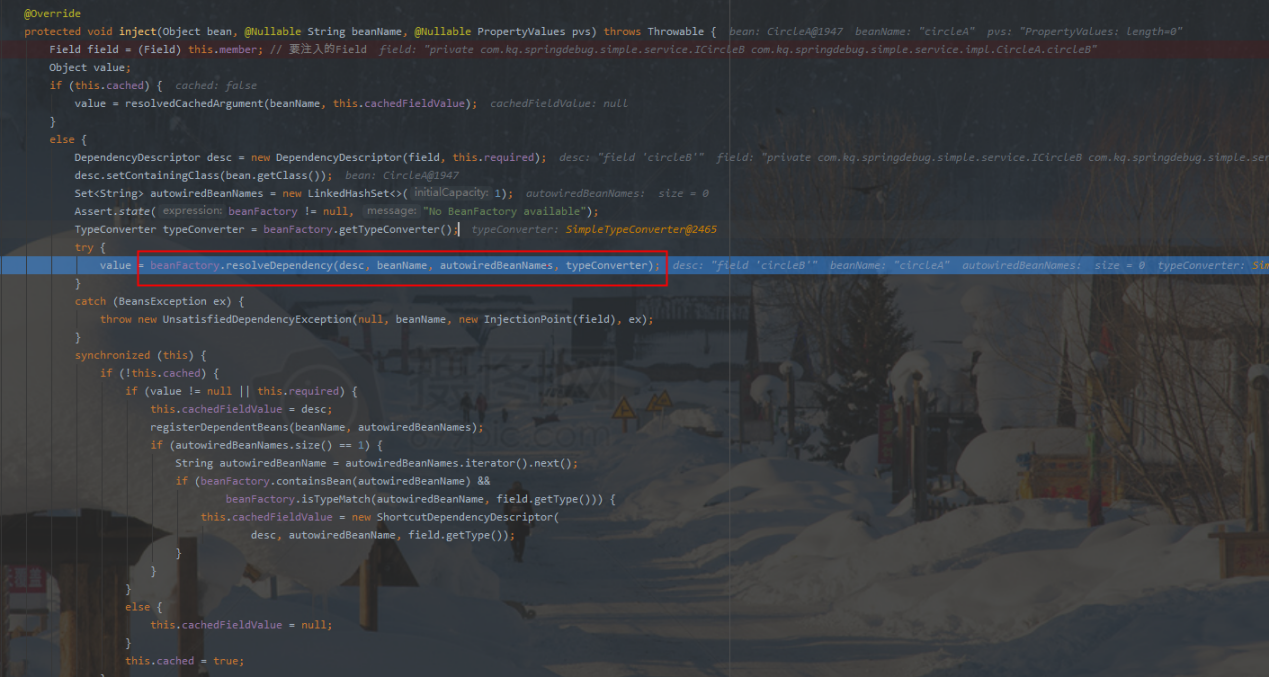
}



### AutowiredAnnotationBeanPostProcessor

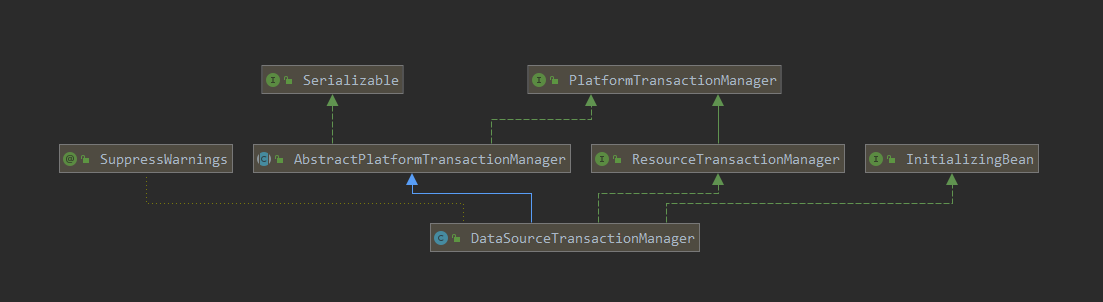


### beanFactory.resolveDependency(重要)

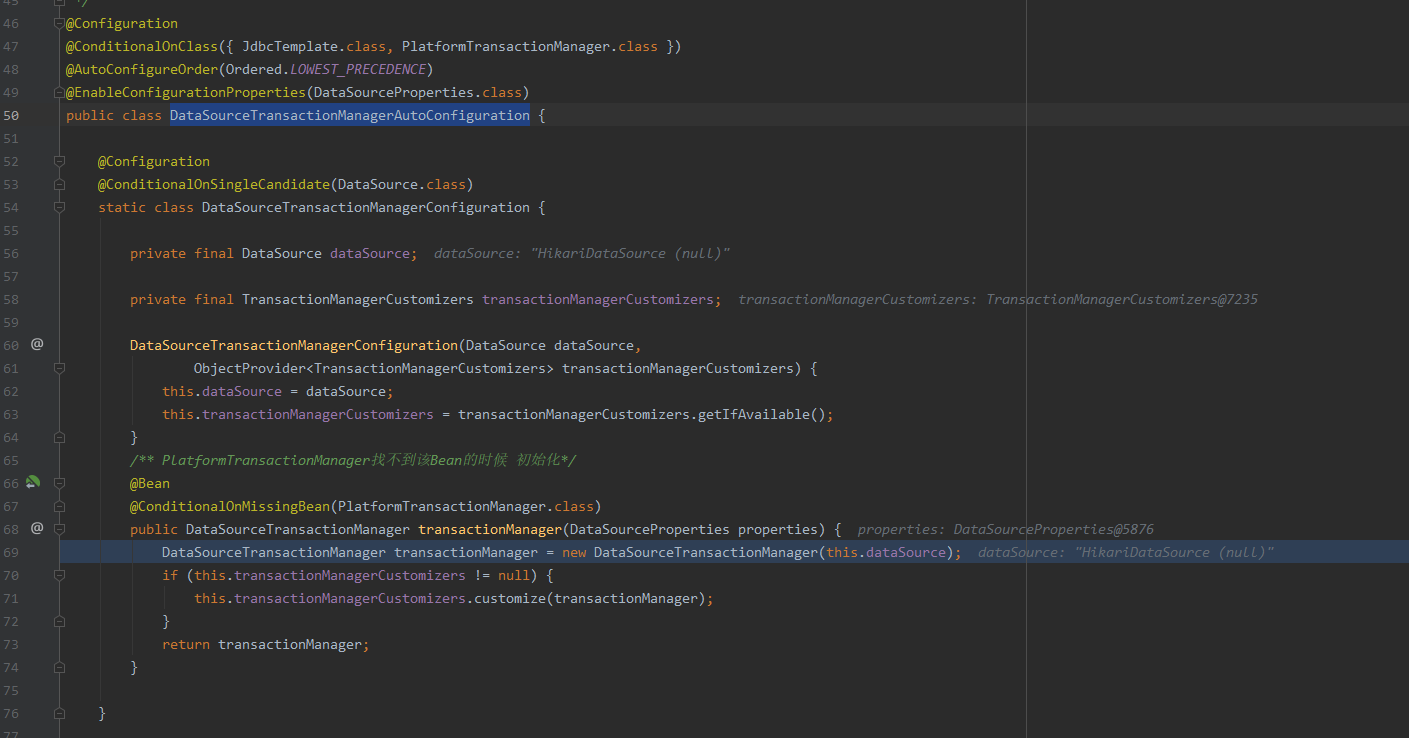


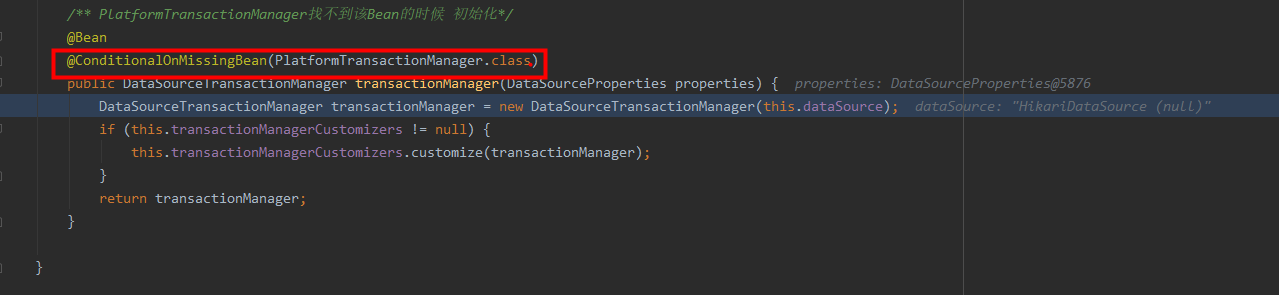
# Transaction

## 类图



## Spring boot 自动配置



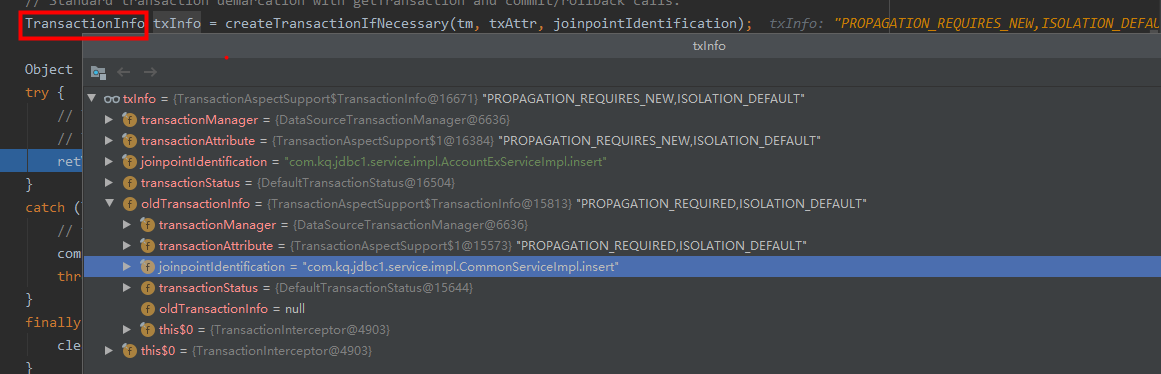


## CglibAopProxy

## AbstractPlatformTransactionManager

## DataSourceTransactionManager

## TransactionInfo



# Spring MVC

## DispatcherServlet

doService(HttpServletRequest request, HttpServletResponse response)

doDispatch(HttpServletRequest request, HttpServletResponse response)

# Spring boot

## Config.file

### ConfigFileApplicationListener（application.yml）

处理 classpath:/,classpath:/config/,file:./,file:./config/