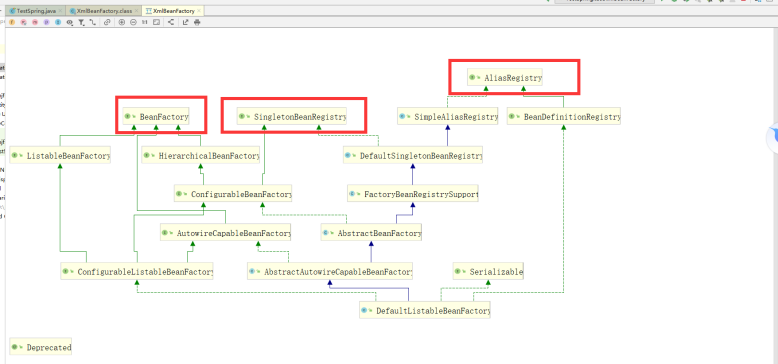
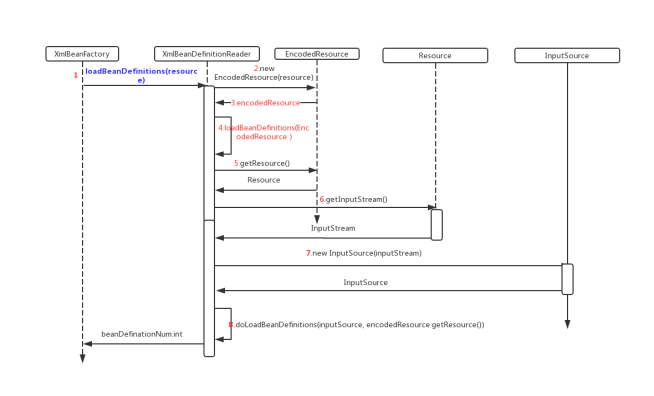
spring源码剖析

# BeanFactory

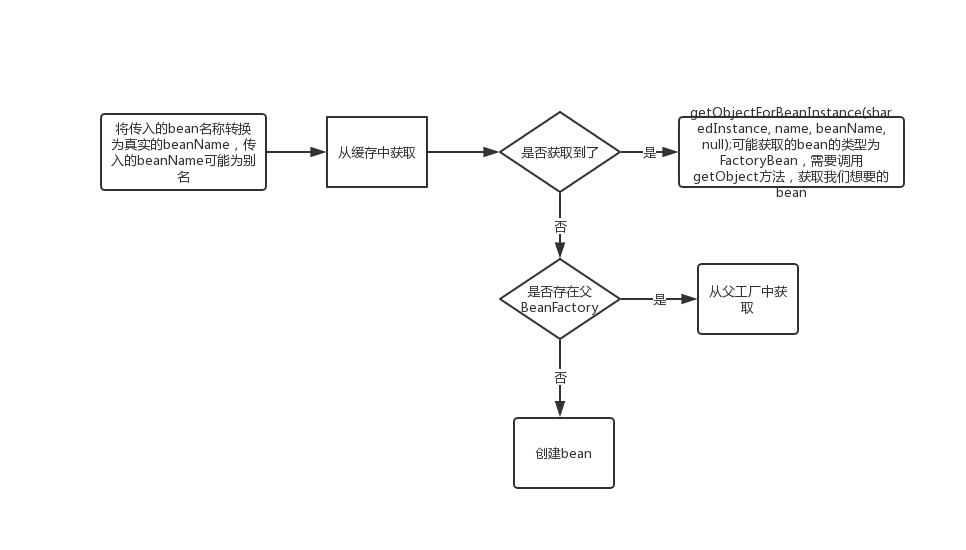
## 类图



## 时序图

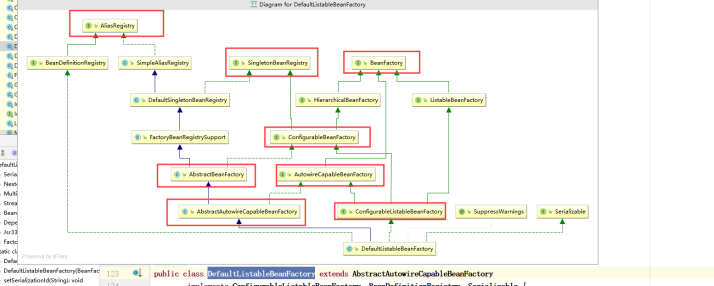


## getBean的流程图



## DefaultListableBeanFactory

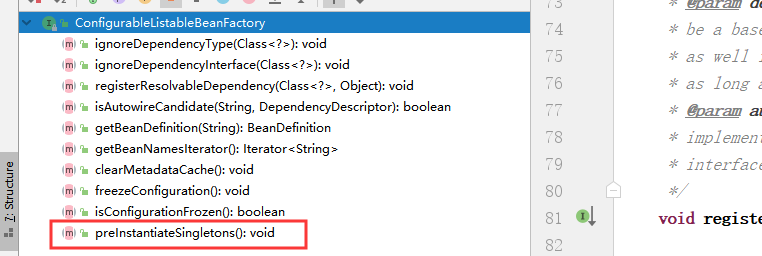
### 类图



### 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

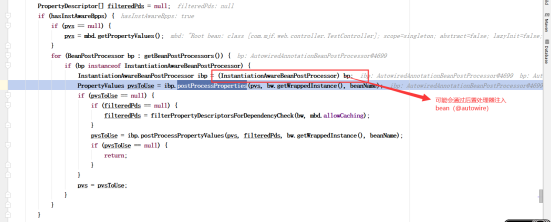
* Springmvc的注解

### 方法

|  |
| --- |
| **@Nullable default Object postProcessBeforeInitialization(Object bean, String beanName) throws BeansException {  return bean; }**  **@Nullable default Object postProcessAfterInitialization(Object bean, String beanName) throws BeansException {  return bean; }** |

## InstantiationAwareBeanPostProcessor

### 使用场景



* Aop
* Autowire注解

### AutowiredAnnotationBeanPostProcessor

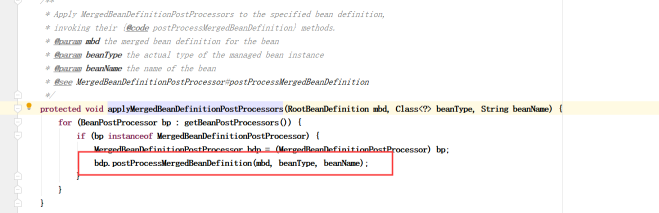
#### 作用

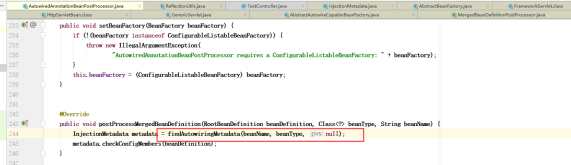
用来处理Autowired注解的

#### 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);  
}**

注入bean

|  |
| --- |
| @Override **protected void inject(Object bean, @Nullable String beanName, @Nullable PropertyValues pvs) throws Throwable {  Field field = (Field) this.member;  Object value;  if (this.cached) {  value = resolvedCachedArgument(beanName, this.cachedFieldValue);  }  else {  DependencyDescriptor desc = new DependencyDescriptor(field, this.required);  desc.setContainingClass(bean.getClass());  Set<String> autowiredBeanNames = new LinkedHashSet<>(1);  Assert.*state*(beanFactory != null, "No BeanFactory available");  TypeConverter typeConverter = beanFactory.getTypeConverter();  try {  value = beanFactory.resolveDependency(desc, beanName, autowiredBeanNames, typeConverter);  }  catch (BeansException ex) {  throw new UnsatisfiedDependencyException(null, beanName, new InjectionPoint(field), ex);  }  synchronized (this) {  if (!this.cached) {  if (value != null || this.required) {  this.cachedFieldValue = desc;  registerDependentBeans(beanName, autowiredBeanNames);  if (autowiredBeanNames.size() == 1) {  String autowiredBeanName = autowiredBeanNames.iterator().next();  if (beanFactory.containsBean(autowiredBeanName) &&  beanFactory.isTypeMatch(autowiredBeanName, field.getType())) {  this.cachedFieldValue = new ShortcutDependencyDescriptor(  desc, autowiredBeanName, field.getType());  }  }  }  else {  this.cachedFieldValue = null;  }  this.cached = true;  }  }  }  if (value != null) {  ReflectionUtils.*makeAccessible*(field);  field.set(bean, value);  } }** |

#### 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

# BeanFactoryPostProcessor体系

|  |
| --- |
| **void postProcessBeanFactory(ConfigurableListableBeanFactory beanFactory) throws BeansException;** |

# Aware体系

## BeanFactoryAware

## ApplicationContextAware

BeanNameAware

# Spring集成到web中

## 配置web.xml

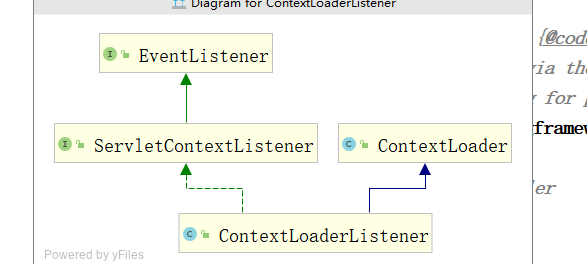
|  |
| --- |
| <**context-param**>  <**param-name**>contextConfigLocation</**param-name**>  <**param-value**>classpath:application.xml</**param-value**> </**context-param**> <**listener**>  <**listener-class**>org.springframework.web.context.ContextLoaderListener</**listener-class**> </**listener**>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

## ContextLoaderListener

### 作用

用来初始化spring容器

### 类结构

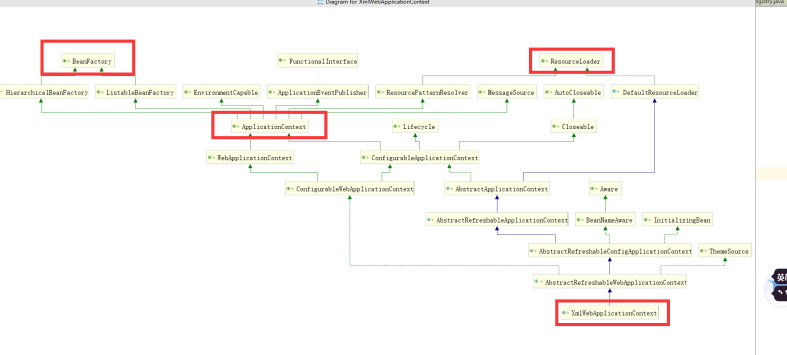


### 核心代码

|  |
| --- |
| ***/\*\*  \* 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

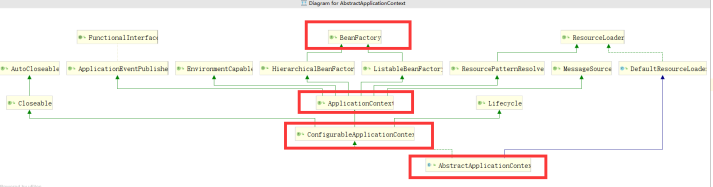
如果不是maven项目，一定要将jar包放到WEB-INF下面



# 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 } |

# spring的生命周期

**1,调用Bean的构造函数(或者工厂方法)实例化Bean.**

**2,对Bean的成员变量赋值.**

**3,如果Bean实现了BeanNameAware,调用Bean的setBeanName方法.**

**4,如果Bean实现了BeanFactoryAware,调用Bean的setBeanFactory方法.**

**5,如果Bean实现了ApplicationContextAware,调用Bean的setApplicationContext方法.**

**6,如果容器中配置了BeanPostProcessor,调用BeanPostProcessor的postProcessBeforeInitialization方法(如果有多个BeanPostProcessor,调用每一个BeanPostProcessor的postProcessBeforeInitialization方法).**

**6,如果Bean实现了InitializingBean,调用Bean的afterPropertiesSet方法.**

**7,如果Bean配置了init-method方法,调用init-method配置的Bean方法.**

**8,如果容器中配置了BeanPostProcessor,调用BeanPostProcessor的postProcessAfterInitialization方法.(如果有多个BeanPostProcessor,调用每一个BeanPostProcessor的postProcessAfterInitialization方法).**

**9,Bean处于可以使用的状态.**

**10,Spring容器关闭.**

**11,4,如果Bean实现了DisposableBean,调用Bean的destroy方法.**

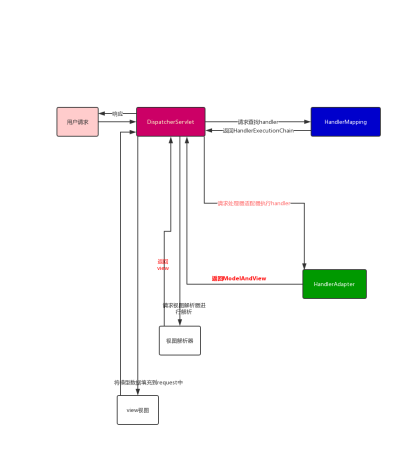
**12,如果Bean配置了destroy-method方法,调用destroy-method配置的Bean的方法.**

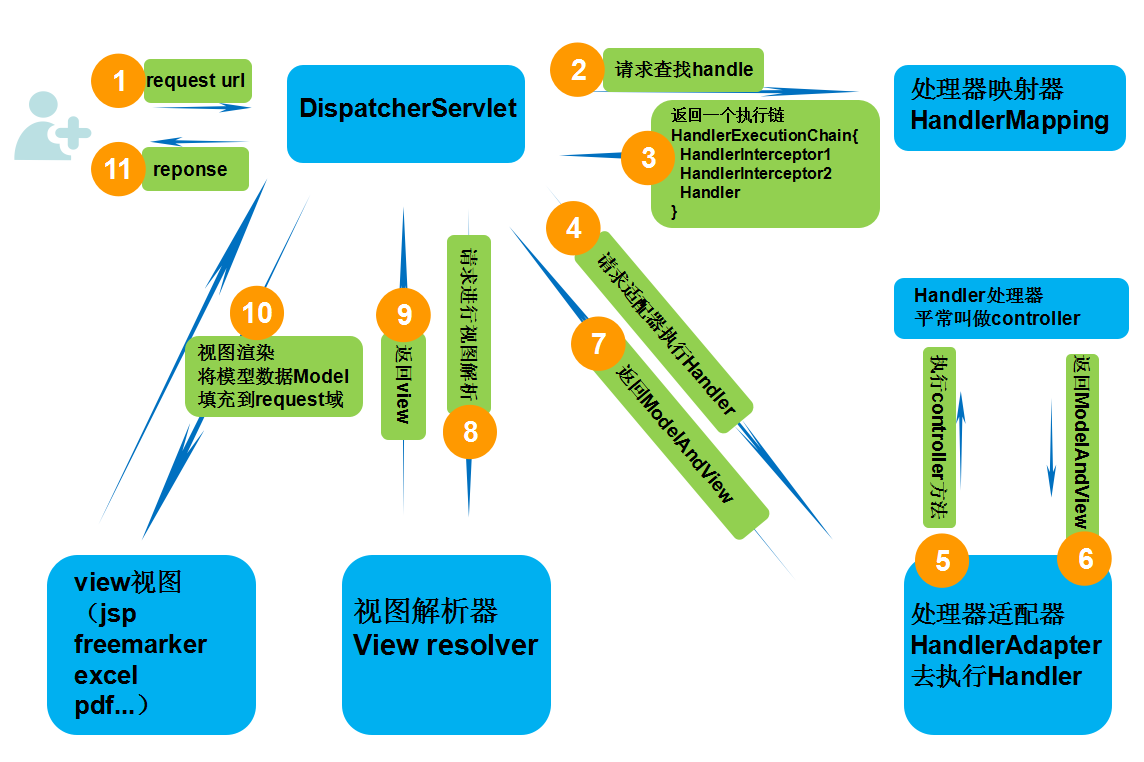
# 处理@auwired注解的后置处理器是什么时候添加到BeanFactory中



# Springmvc

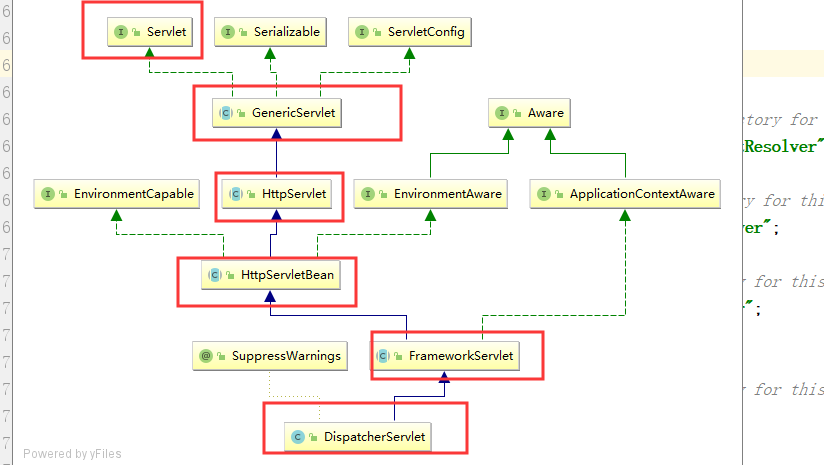
## 执行流程





## DispatcherServlet

### 类图



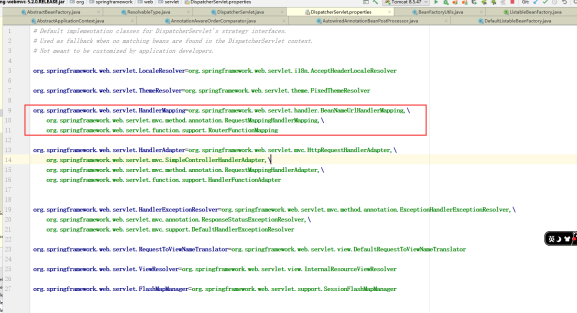
### 什么时候初始化mvc容器

在第一次访问url的时候，才去初始化

### 初始化

|  |
| --- |
| ***/\*\*  \* Initialize the strategy objects that this servlet uses.  \* <p>May be overridden in subclasses in order to initialize further strategy objects.  \*/* protected void initStrategies(ApplicationContext context) {  initMultipartResolver(context);  initLocaleResolver(context);  initThemeResolver(context);**  **// 初始化处理器映射  initHandlerMappings(context);  initHandlerAdapters(context);  initHandlerExceptionResolvers(context);  initRequestToViewNameTranslator(context);  initViewResolvers(context);  initFlashMapManager(context); }** |

如果没有配置处理器映射，就使用默认的



|  |
| --- |
| **protected <T> List<T> getDefaultStrategies(ApplicationContext context, Class<T> strategyInterface) {  String key = strategyInterface.getName();  String value = *defaultStrategies*.getProperty(key);  if (value != null) {  String[] classNames = StringUtils.*commaDelimitedListToStringArray*(value);  List<T> strategies = new ArrayList<>(classNames.length);  for (String className : classNames) {  try {  Class<?> clazz = ClassUtils.*forName*(className, DispatcherServlet.class.getClassLoader());**  **// 创建实例  Object strategy = createDefaultStrategy(context, clazz);  strategies.add((T) strategy);  }  catch (ClassNotFoundException ex) {  throw new BeanInitializationException(  "Could not find DispatcherServlet's default strategy class [" + className +  "] for interface [" + key + "]", ex);  }  catch (LinkageError err) {  throw new BeanInitializationException(  "Unresolvable class definition for DispatcherServlet's default strategy class [" +  className + "] for interface [" + key + "]", err);  }  }  return strategies;  }  else {  return new LinkedList<>();  } }** |

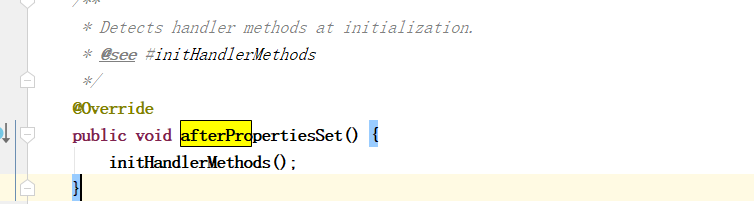
### 转发

|  |
| --- |
| ***/\*\*  \* Process the actual dispatching to the handler.  \* <p>The handler will be obtained by applying the servlet's HandlerMappings in order.  \* The HandlerAdapter will be obtained by querying the servlet's installed HandlerAdapters  \* to find the first that supports the handler class.  \* <p>All HTTP methods are handled by this method. It's up to HandlerAdapters or handlers  \* themselves to decide which methods are acceptable.  \* @param request current HTTP request  \* @param response current HTTP response  \* @throws Exception in case of any kind of processing failure  \*/* protected void doDispatch(HttpServletRequest request, HttpServletResponse response) throws Exception {  HttpServletRequest processedRequest = request;  HandlerExecutionChain mappedHandler = null;  boolean multipartRequestParsed = false;   WebAsyncManager asyncManager = WebAsyncUtils.*getAsyncManager*(request);   try {  ModelAndView mv = null;  Exception dispatchException = null;   try {  processedRequest = checkMultipart(request);  multipartRequestParsed = (processedRequest != request);   *// Determine handler for the current request.* mappedHandler = getHandler(processedRequest);  if (mappedHandler == null) {  noHandlerFound(processedRequest, response);  return;  }   *// Determine handler adapter for the current request.* HandlerAdapter ha = getHandlerAdapter(mappedHandler.getHandler());   *// Process last-modified header, if supported by the handler.* String method = request.getMethod();  boolean isGet = "GET".equals(method);  if (isGet || "HEAD".equals(method)) {  long lastModified = ha.getLastModified(request, mappedHandler.getHandler());  if (new ServletWebRequest(request, response).checkNotModified(lastModified) && isGet) {  return;  }  }   if (!mappedHandler.applyPreHandle(processedRequest, response)) {  return;  }   *// Actually invoke the handler.* mv = ha.handle(processedRequest, response, mappedHandler.getHandler());   if (asyncManager.isConcurrentHandlingStarted()) {  return;  }   applyDefaultViewName(processedRequest, mv);  mappedHandler.applyPostHandle(processedRequest, response, mv);  }  catch (Exception ex) {  dispatchException = ex;  }  catch (Throwable err) {  *// As of 4.3, we're processing Errors thrown from handler methods as well,  // making them available for @ExceptionHandler methods and other scenarios.* dispatchException = new NestedServletException("Handler dispatch failed", err);  }  processDispatchResult(processedRequest, response, mappedHandler, mv, dispatchException);  }  catch (Exception ex) {  triggerAfterCompletion(processedRequest, response, mappedHandler, ex);  }  catch (Throwable err) {  triggerAfterCompletion(processedRequest, response, mappedHandler,  new NestedServletException("Handler processing failed", err));  }  finally {  if (asyncManager.isConcurrentHandlingStarted()) {  *// Instead of postHandle and afterCompletion* if (mappedHandler != null) {  mappedHandler.applyAfterConcurrentHandlingStarted(processedRequest, response);  }  }  else {  *// Clean up any resources used by a multipart request.* if (multipartRequestParsed) {  cleanupMultipart(processedRequest);  }  }  } }** |

#### AbstractHandlerMethodMapping

实现了InitializingBean

会去调用afterPropertiesSet方法



初始化HandlerMethod

##### 查找处理器

|  |
| --- |
| ***/\*\*  \* Look up the best-matching handler method for the current request.  \* If multiple matches are found, the best match is selected.  \* @param lookupPath mapping lookup path within the current servlet mapping  \* @param request the current request  \* @return the best-matching handler method, or {@code null} if no match  \* @see #handleMatch(Object, String, HttpServletRequest)  \* @see #handleNoMatch(Set, String, HttpServletRequest)  \*/* @Nullable protected HandlerMethod lookupHandlerMethod(String lookupPath, HttpServletRequest request) throws Exception {  List<Match> matches = new ArrayList<>();  List<T> directPathMatches = this.mappingRegistry.getMappingsByUrl(lookupPath);  if (directPathMatches != null) {  addMatchingMappings(directPathMatches, matches, request);  }  if (matches.isEmpty()) {  *// No choice but to go through all mappings...* addMatchingMappings(this.mappingRegistry.getMappings().keySet(), matches, request);  }   if (!matches.isEmpty()) {  Comparator<Match> comparator = new MatchComparator(getMappingComparator(request));  matches.sort(comparator);**  **// 取第一个  Match bestMatch = matches.get(0);  if (matches.size() > 1) {  if (logger.isTraceEnabled()) {  logger.trace(matches.size() + " matching mappings: " + matches);  }  if (CorsUtils.*isPreFlightRequest*(request)) {  return *PREFLIGHT\_AMBIGUOUS\_MATCH*;  }  Match secondBestMatch = matches.get(1);  if (comparator.compare(bestMatch, secondBestMatch) == 0) {  Method m1 = bestMatch.handlerMethod.getMethod();  Method m2 = secondBestMatch.handlerMethod.getMethod();  String uri = request.getRequestURI();  throw new IllegalStateException(  "Ambiguous handler methods mapped for '" + uri + "': {" + m1 + ", " + m2 + "}");  }  }  request.setAttribute(*BEST\_MATCHING\_HANDLER\_ATTRIBUTE*, bestMatch.handlerMethod);  handleMatch(bestMatch.mapping, lookupPath, request);  return bestMatch.handlerMethod;  }  else {  return handleNoMatch(this.mappingRegistry.getMappings().keySet(), lookupPath, request);  } }** |

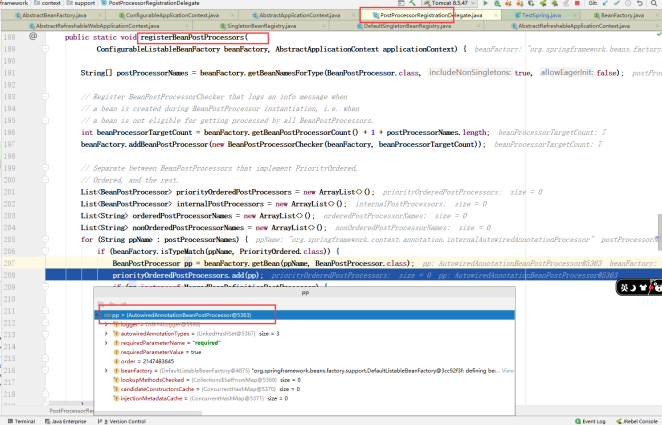
## 调用处理器并且处理返回值

|  |
| --- |
| ***/\*\*  \* Invoke the method and handle the return value through one of the  \* configured {@link HandlerMethodReturnValueHandler HandlerMethodReturnValueHandlers}.  \* @param webRequest the current request  \* @param mavContainer the ModelAndViewContainer for this request  \* @param providedArgs "given" arguments matched by type (not resolved)  \*/* public void invokeAndHandle(ServletWebRequest webRequest, ModelAndViewContainer mavContainer,  Object... providedArgs) throws Exception {**  **// 调用处理器  Object returnValue = invokeForRequest(webRequest, mavContainer, providedArgs);  setResponseStatus(webRequest);   if (returnValue == null) {  if (isRequestNotModified(webRequest) || getResponseStatus() != null || mavContainer.isRequestHandled()) {  disableContentCachingIfNecessary(webRequest);  mavContainer.setRequestHandled(true);  return;  }  }  else if (StringUtils.*hasText*(getResponseStatusReason())) {  mavContainer.setRequestHandled(true);  return;  }   mavContainer.setRequestHandled(false);  Assert.*state*(this.returnValueHandlers != null, "No return value handlers");  try {  this.returnValueHandlers.handleReturnValue(  returnValue, getReturnValueType(returnValue), mavContainer, webRequest);  }  catch (Exception ex) {  if (logger.isTraceEnabled()) {  logger.trace(formatErrorForReturnValue(returnValue), ex);  }  throw ex;  } }** |

### 创建WebApplicationContenxt

|  |
| --- |
| **// ApplicationContext parent是指spring容器**  **protected WebApplicationContext createWebApplicationContext(@Nullable ApplicationContext parent) {  Class<?> contextClass = getContextClass();  if (!ConfigurableWebApplicationContext.class.isAssignableFrom(contextClass)) {  throw new ApplicationContextException(  "Fatal initialization error in servlet with name '" + getServletName() +  "': custom WebApplicationContext class [" + contextClass.getName() +  "] is not of type ConfigurableWebApplicationContext");  }**  **// 实例化XmlWebApplicationContentext  ConfigurableWebApplicationContext wac =  (ConfigurableWebApplicationContext) BeanUtils.*instantiateClass*(contextClass);   wac.setEnvironment(getEnvironment());  // 设置父容器**  **wac.setParent(parent);**  **String configLocation = getContextConfigLocation();  if (configLocation != null) {  wac.setConfigLocation(configLocation);  }**  **// ioc容器和di  configureAndRefreshWebApplicationContext(wac);   return wac; }** |

### AutowiredAnnotationBeanPostProcessor的注册



## RequestMappingHandlerAdapter

### 默认的返回值处理器

|  |
| --- |
| ***/\*\*  \* Return the list of return value handlers to use including built-in and  \* custom handlers provided via {@link #setReturnValueHandlers}.  \*/* private List<HandlerMethodReturnValueHandler> getDefaultReturnValueHandlers() {  List<HandlerMethodReturnValueHandler> handlers = new ArrayList<>();   *// Single-purpose return value types* handlers.add(new ModelAndViewMethodReturnValueHandler());  handlers.add(new ModelMethodProcessor());  handlers.add(new ViewMethodReturnValueHandler());  handlers.add(new ResponseBodyEmitterReturnValueHandler(getMessageConverters(),  this.reactiveAdapterRegistry, this.taskExecutor, this.contentNegotiationManager));  handlers.add(new StreamingResponseBodyReturnValueHandler());  handlers.add(new HttpEntityMethodProcessor(getMessageConverters(),  this.contentNegotiationManager, this.requestResponseBodyAdvice));  handlers.add(new HttpHeadersReturnValueHandler());  handlers.add(new CallableMethodReturnValueHandler());  handlers.add(new DeferredResultMethodReturnValueHandler());  handlers.add(new AsyncTaskMethodReturnValueHandler(this.beanFactory));   *// Annotation-based return value types* handlers.add(new ModelAttributeMethodProcessor(false));**  **// handler上面加@ResponseBody注解，就使用该处理器  handlers.add(new RequestResponseBodyMethodProcessor(getMessageConverters(),  this.contentNegotiationManager, this.requestResponseBodyAdvice));   *// Multi-purpose return value types* handlers.add(new ViewNameMethodReturnValueHandler());  handlers.add(new MapMethodProcessor());   *// Custom return value types* if (getCustomReturnValueHandlers() != null) {  handlers.addAll(getCustomReturnValueHandlers());  }   *// Catch-all* if (!CollectionUtils.*isEmpty*(getModelAndViewResolvers())) {  handlers.add(new ModelAndViewResolverMethodReturnValueHandler(getModelAndViewResolvers()));  }  else {  handlers.add(new ModelAttributeMethodProcessor(true));  }   return handlers; }** |

## RequestMappingInfo

|  |
| --- |
| ***/\*\*  \* Checks if all conditions in this request mapping info match the provided request and returns  \* a potentially new request mapping info with conditions tailored to the current request.  \* <p>For example the returned instance may contain the subset of URL patterns that match to  \* the current request, sorted with best matching patterns on top.  \* @return a new instance in case all conditions match; or {@code null} otherwise  \*/* @Override @Nullable public RequestMappingInfo getMatchingCondition(HttpServletRequest request) {  RequestMethodsRequestCondition methods = this.methodsCondition.getMatchingCondition(request);  if (methods == null) {  return null;  }  ParamsRequestCondition params = this.paramsCondition.getMatchingCondition(request);  if (params == null) {  return null;  }  HeadersRequestCondition headers = this.headersCondition.getMatchingCondition(request);  if (headers == null) {  return null;  }  ConsumesRequestCondition consumes = this.consumesCondition.getMatchingCondition(request);  if (consumes == null) {  return null;  }  ProducesRequestCondition produces = this.producesCondition.getMatchingCondition(request);  if (produces == null) {  return null;  }  PatternsRequestCondition patterns = this.patternsCondition.getMatchingCondition(request);  if (patterns == null) {  return null;  }  RequestConditionHolder custom = this.customConditionHolder.getMatchingCondition(request);  if (custom == null) {  return null;  }   return new RequestMappingInfo(this.name, patterns,  methods, params, headers, consumes, produces, custom.getCondition()); }** |

# Aop

## 解析aspectj-autoproxy

AspectJAutoProxyBeanDefinitionParser

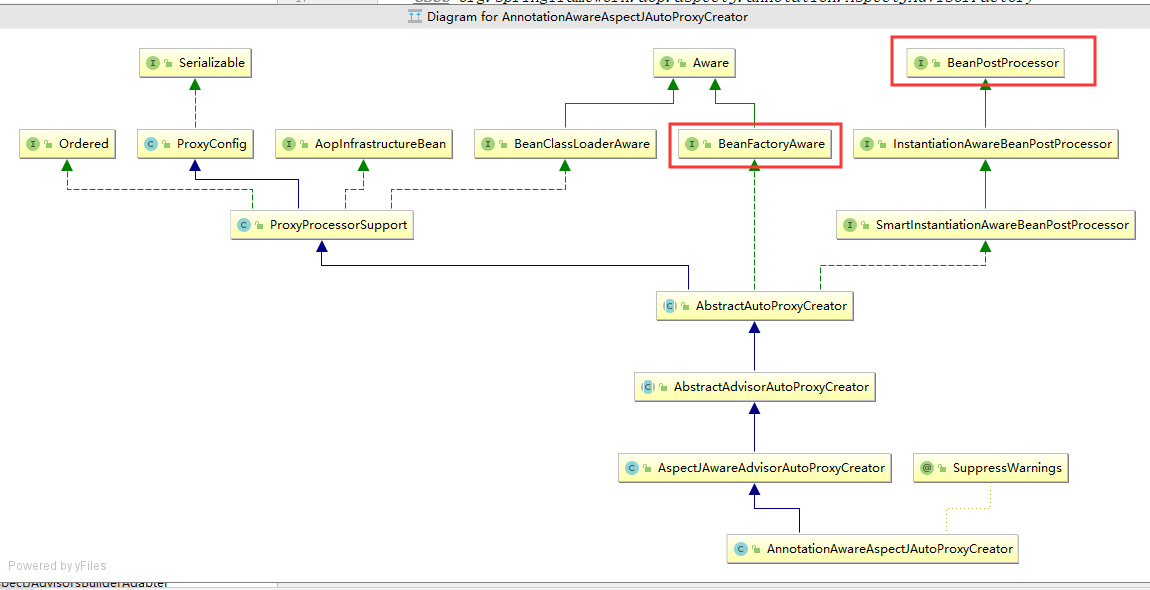
|  |
| --- |
| **@Override @Nullable public BeanDefinition parse(Element element, ParserContext parserContext) {  AopNamespaceUtils.*registerAspectJAnnotationAutoProxyCreatorIfNecessary*(parserContext, element);  extendBeanDefinition(element, parserContext);  return null; }** |

## AnnotationAwareAspectJAutoProxyCreator

## AopConfigUtils

工具类

### 类图



### 获取通知

|  |
| --- |
| **@Override @Nullable protected Object[] getAdvicesAndAdvisorsForBean(  Class<?> beanClass, String beanName, @Nullable TargetSource targetSource) {   List<Advisor> advisors = findEligibleAdvisors(beanClass, beanName);  if (advisors.isEmpty()) {  return *DO\_NOT\_PROXY*;  }  return advisors.toArray(); }** |

### 获取通知的真正实现

BeanFactoryAspectJAdvisorsBuilder

|  |
| --- |
| ***/\*\*  \* Look for AspectJ-annotated aspect beans in the current bean factory,  \* and return to a list of Spring AOP Advisors representing them.  \* <p>Creates a Spring Advisor for each AspectJ advice method.  \* @return the list of {@link org.springframework.aop.Advisor} beans  \* @see #isEligibleBean  \*/* public List<Advisor> buildAspectJAdvisors() {  List<String> aspectNames = this.aspectBeanNames;   if (aspectNames == null) {  synchronized (this) {  aspectNames = this.aspectBeanNames;  if (aspectNames == null) {  List<Advisor> advisors = new ArrayList<>();  aspectNames = new ArrayList<>();  String[] beanNames = BeanFactoryUtils.*beanNamesForTypeIncludingAncestors*(  this.beanFactory, Object.class, true, false);  for (String beanName : beanNames) {  if (!isEligibleBean(beanName)) {  continue;  }  *// We must be careful not to instantiate beans eagerly as in this case they  // would be cached by the Spring container but would not have been weaved.* Class<?> beanType = this.beanFactory.getType(beanName);  if (beanType == null) {  continue;  }**  **// 判断是否有@Aspect注解  if (this.advisorFactory.isAspect(beanType)) {  aspectNames.add(beanName);  AspectMetadata amd = new AspectMetadata(beanType, beanName);  if (amd.getAjType().getPerClause().getKind() == PerClauseKind.*SINGLETON*) {  MetadataAwareAspectInstanceFactory factory =  new BeanFactoryAspectInstanceFactory(this.beanFactory, beanName);  List<Advisor> classAdvisors = this.advisorFactory.getAdvisors(factory);  if (this.beanFactory.isSingleton(beanName)) {  this.advisorsCache.put(beanName, classAdvisors);  }  else {  this.aspectFactoryCache.put(beanName, factory);  }  advisors.addAll(classAdvisors);  }  else {  *// Per target or per this.* if (this.beanFactory.isSingleton(beanName)) {  throw new IllegalArgumentException("Bean with name '" + beanName +  "' is a singleton, but aspect instantiation model is not singleton");  }  MetadataAwareAspectInstanceFactory factory =  new PrototypeAspectInstanceFactory(this.beanFactory, beanName);  this.aspectFactoryCache.put(beanName, factory);  advisors.addAll(this.advisorFactory.getAdvisors(factory));  }  }  }  this.aspectBeanNames = aspectNames;  return advisors;  }  }  }   if (aspectNames.isEmpty()) {  return Collections.*emptyList*();  }  List<Advisor> advisors = new ArrayList<>();  for (String aspectName : aspectNames) {  List<Advisor> cachedAdvisors = this.advisorsCache.get(aspectName);  if (cachedAdvisors != null) {  advisors.addAll(cachedAdvisors);  }  else {  MetadataAwareAspectInstanceFactory factory = this.aspectFactoryCache.get(aspectName);  advisors.addAll(this.advisorFactory.getAdvisors(factory));  }  }  return advisors; }** |

### 获取通知方法

|  |
| --- |
| **private List<Method> getAdvisorMethods(Class<?> aspectClass) {  final List<Method> methods = new ArrayList<>();  ReflectionUtils.*doWithMethods*(aspectClass, method -> {  *// Exclude pointcuts* if (AnnotationUtils.*getAnnotation*(method, Pointcut.class) == null) {  methods.add(method);  }  }, ReflectionUtils.*USER\_DECLARED\_METHODS*);  methods.sort(*METHOD\_COMPARATOR*);  return methods; }** |

## Aop始终没有生效的原因

由于我是在@service层加的aop，这些bean是存在于springmvc容器中的，如果把aop的配置放到spring的配置文件中，那么springmvc容器是不存在这个后置处理器的，导致无法给加入aop的bean生成代理，那么aop自然不会生效。

## @EnableAspectJAutoProxy



## 某个方法被多个切点拦截

可以定义不同的切面，在切面上添加@Order注解，就可以对切面进行排序，这样该方法就可以被多个切面拦截。

如果一个切面中既有@around，又有@before，那么该方法就会执行两个通知，本来@around就有包括@before，然后又定义了一个@before通知，多次一举，不要这样做。

## Aop的入口

|  |
| --- |
| ***/\*\*  \* Initialize the given bean instance, applying factory callbacks  \* as well as init methods and bean post processors.  \* <p>Called from {@link #createBean} for traditionally defined beans,  \* and from {@link #initializeBean} for existing bean instances.  \* @param beanName the bean name in the factory (for debugging purposes)  \* @param bean the new bean instance we may need to initialize  \* @param mbd the bean definition that the bean was created with  \* (can also be {@code null}, if given an existing bean instance)  \* @return the initialized bean instance (potentially wrapped)  \* @see BeanNameAware  \* @see BeanClassLoaderAware  \* @see BeanFactoryAware  \* @see #applyBeanPostProcessorsBeforeInitialization  \* @see #invokeInitMethods  \* @see #applyBeanPostProcessorsAfterInitialization  \*/* 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()) {**  **// 对类中某些特殊方法的调用，比如@PostContrut，Aware接口  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()) {**  **// aop  wrappedBean = applyBeanPostProcessorsAfterInitialization(wrappedBean, beanName);  }   return wrappedBean; }** |

# 事务

## @Transactional，@Before，@After执行顺序

先执行事务通知，然后执行前置通知，后置通知，再去执行事务通知，提交事务

## 实现

通过aop实现的

## 注入事务的方法



## tx:annotation-driven标签的解析器

AnnotationDrivenBeanDefinitionParser

## AnnotationDrivenBeanDefinitionParser

### 注册事务管理器

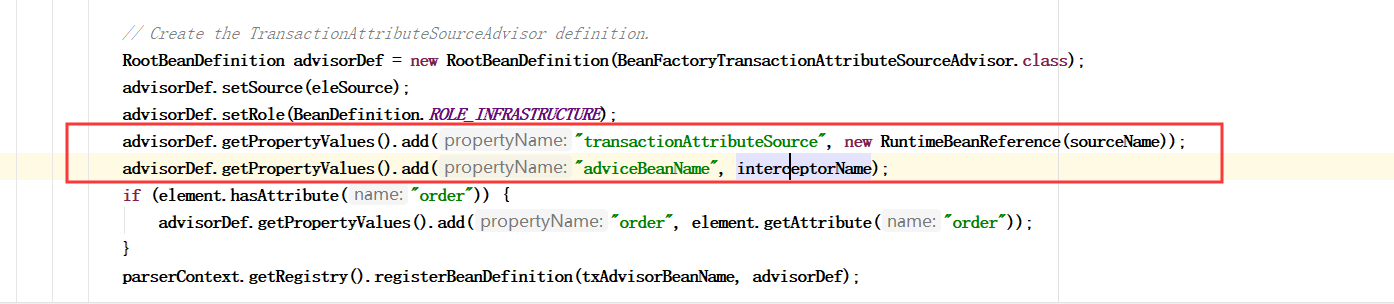
给TransactionInterceptor添加transactionManagerBeanName

|  |
| --- |
| **private static void registerTransactionManager(Element element, BeanDefinition def) {  def.getPropertyValues().add("transactionManagerBeanName",  TxNamespaceHandler.*getTransactionManagerName*(element)); }** |

## AnnotationTransactionAttributeSource

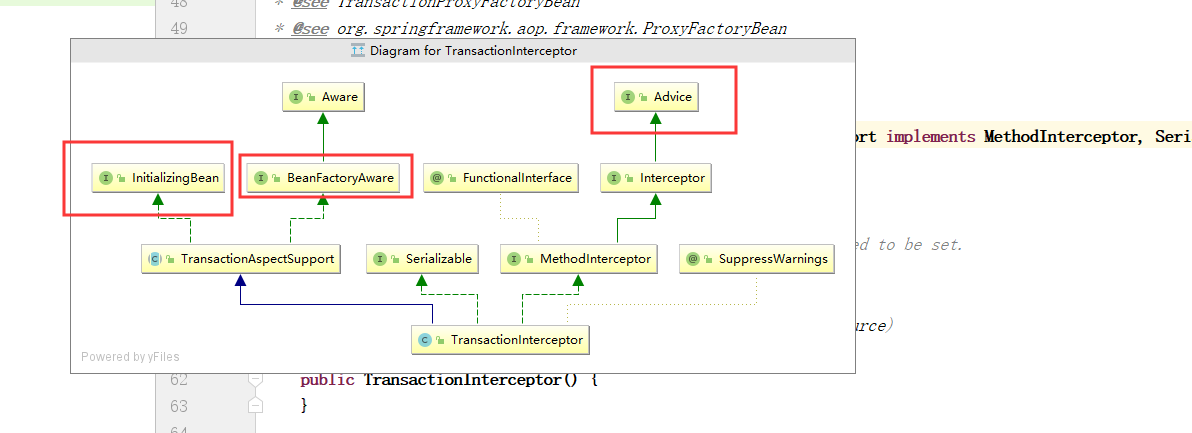
### 什么时候实例化

在给BeanFactoryTransactionAttributeSourceAdvisor设置属性的时候实例化的。



## TransactionInterceptor

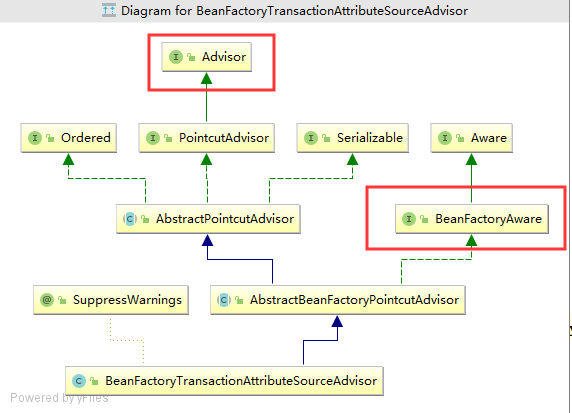
### 类图



## BeanFactoryTransactionAttributeSourceAdvisor

事务切面

### 类图



## SpringTransactionAnnotationParser

### 作用

解析@Transactional注解的

|  |
| --- |
| **@Override @Nullable public TransactionAttribute parseTransactionAnnotation(AnnotatedElement element) {  AnnotationAttributes attributes = AnnotatedElementUtils.*findMergedAnnotationAttributes*(  element, Transactional.class, false, false);  if (attributes != null) {  return parseTransactionAnnotation(attributes);  }  else {  return null;  } }** |

## DataSourceTransactionManager

### 作用

用来 获取事务，提交事务，和回滚事务的

### 处理已经存在的事务

|  |
| --- |
| **private TransactionStatus handleExistingTransaction(  TransactionDefinition definition, Object transaction, boolean debugEnabled)  throws TransactionException {**  **// 不需要事务  if (definition.getPropagationBehavior() == TransactionDefinition.*PROPAGATION\_NEVER*) {  throw new IllegalTransactionStateException(  "Existing transaction found for transaction marked with propagation 'never'");  }   if (definition.getPropagationBehavior() == TransactionDefinition.*PROPAGATION\_NOT\_SUPPORTED*) {  if (debugEnabled) {  logger.debug("Suspending current transaction");  }**  **//挂起事务  Object suspendedResources = suspend(transaction);  boolean newSynchronization = (getTransactionSynchronization() == *SYNCHRONIZATION\_ALWAYS*);  return prepareTransactionStatus(  definition, null, false, newSynchronization, debugEnabled, suspendedResources);  }  // 新建事务  if (definition.getPropagationBehavior() == TransactionDefinition.*PROPAGATION\_REQUIRES\_NEW*) {  if (debugEnabled) {  logger.debug("Suspending current transaction, creating new transaction with name [" +  definition.getName() + "]");  }  SuspendedResourcesHolder suspendedResources = suspend(transaction);  try {  boolean newSynchronization = (getTransactionSynchronization() != *SYNCHRONIZATION\_NEVER*);  DefaultTransactionStatus status = newTransactionStatus(  definition, transaction, true, newSynchronization, debugEnabled, suspendedResources);  doBegin(transaction, definition);  prepareSynchronization(status, definition);  return status;  }  catch (RuntimeException | Error beginEx) {  resumeAfterBeginException(transaction, suspendedResources, beginEx);  throw beginEx;  }  }   if (definition.getPropagationBehavior() == TransactionDefinition.*PROPAGATION\_NESTED*) {  if (!isNestedTransactionAllowed()) {  throw new NestedTransactionNotSupportedException(  "Transaction manager does not allow nested transactions by default - " +  "specify 'nestedTransactionAllowed' property with value 'true'");  }  if (debugEnabled) {  logger.debug("Creating nested transaction with name [" + definition.getName() + "]");  }  if (useSavepointForNestedTransaction()) {  *// Create savepoint within existing Spring-managed transaction,  // through the SavepointManager API implemented by TransactionStatus.  // Usually uses JDBC 3.0 savepoints. Never activates Spring synchronization.* DefaultTransactionStatus status =  prepareTransactionStatus(definition, transaction, false, false, debugEnabled, null);  status.createAndHoldSavepoint();  return status;  }  else {  *// Nested transaction through nested begin and commit/rollback calls.  // Usually only for JTA: Spring synchronization might get activated here  // in case of a pre-existing JTA transaction.* boolean newSynchronization = (getTransactionSynchronization() != *SYNCHRONIZATION\_NEVER*);  DefaultTransactionStatus status = newTransactionStatus(  definition, transaction, true, newSynchronization, debugEnabled, null);  doBegin(transaction, definition);  prepareSynchronization(status, definition);  return status;  }  }   *// Assumably PROPAGATION\_SUPPORTS or PROPAGATION\_REQUIRED.* if (debugEnabled) {  logger.debug("Participating in existing transaction");  }  if (isValidateExistingTransaction()) {  if (definition.getIsolationLevel() != TransactionDefinition.*ISOLATION\_DEFAULT*) {  Integer currentIsolationLevel = TransactionSynchronizationManager.*getCurrentTransactionIsolationLevel*();  if (currentIsolationLevel == null || currentIsolationLevel != definition.getIsolationLevel()) {  Constants isoConstants = DefaultTransactionDefinition.*constants*;  throw new IllegalTransactionStateException("Participating transaction with definition [" +  definition + "] specifies isolation level which is incompatible with existing transaction: " +  (currentIsolationLevel != null ?  isoConstants.toCode(currentIsolationLevel, DefaultTransactionDefinition.*PREFIX\_ISOLATION*) :  "(unknown)"));  }  }  if (!definition.isReadOnly()) {  if (TransactionSynchronizationManager.*isCurrentTransactionReadOnly*()) {  throw new IllegalTransactionStateException("Participating transaction with definition [" +  definition + "] is not marked as read-only but existing transaction is");  }  }  }  boolean newSynchronization = (getTransactionSynchronization() != *SYNCHRONIZATION\_NEVER*);  return prepareTransactionStatus(definition, transaction, false, newSynchronization, debugEnabled, null); }** |

### 开启事务

|  |
| --- |
| **protected void doBegin(Object transaction, TransactionDefinition definition) {  DataSourceTransactionObject txObject = (DataSourceTransactionObject) transaction;  Connection con = null;   try {  if (!txObject.hasConnectionHolder() ||  txObject.getConnectionHolder().isSynchronizedWithTransaction()) {**  **// 获取连接  Connection newCon = obtainDataSource().getConnection();  if (logger.isDebugEnabled()) {  logger.debug("Acquired Connection [" + newCon + "] for JDBC transaction");  }  txObject.setConnectionHolder(new ConnectionHolder(newCon), true);  }   txObject.getConnectionHolder().setSynchronizedWithTransaction(true);  con = txObject.getConnectionHolder().getConnection();   Integer previousIsolationLevel = DataSourceUtils.*prepareConnectionForTransaction*(con, definition);  txObject.setPreviousIsolationLevel(previousIsolationLevel);   *// Switch to manual commit if necessary. This is very expensive in some JDBC drivers,  // so we don't want to do it unnecessarily (for example if we've explicitly  // configured the connection pool to set it already).* if (con.getAutoCommit()) {  txObject.setMustRestoreAutoCommit(true);  if (logger.isDebugEnabled()) {  logger.debug("Switching JDBC Connection [" + con + "] to manual commit");  }**  **// 设置为手动提交事务  con.setAutoCommit(false);  }   prepareTransactionalConnection(con, definition);  txObject.getConnectionHolder().setTransactionActive(true);   int timeout = determineTimeout(definition);  if (timeout != TransactionDefinition.*TIMEOUT\_DEFAULT*) {  txObject.getConnectionHolder().setTimeoutInSeconds(timeout);  }   *// Bind the connection holder to the thread.* if (txObject.isNewConnectionHolder()) {  TransactionSynchronizationManager.*bindResource*(obtainDataSource(), txObject.getConnectionHolder());  }  }   catch (Throwable ex) {  if (txObject.isNewConnectionHolder()) {  DataSourceUtils.*releaseConnection*(con, obtainDataSource());  txObject.setConnectionHolder(null, false);  }  throw new CannotCreateTransactionException("Could not open JDBC Connection for transaction", ex);  } }** |

### 提交事务

|  |
| --- |
| **@Override protected void doCommit(DefaultTransactionStatus status) {  DataSourceTransactionObject txObject = (DataSourceTransactionObject) status.getTransaction();  Connection con = txObject.getConnectionHolder().getConnection();  if (status.isDebug()) {  logger.debug("Committing JDBC transaction on Connection [" + con + "]");  }  try {  con.commit();  }  catch (SQLException ex) {  throw new TransactionSystemException("Could not commit JDBC transaction", ex);  } }** |

## 事务的隔离级别

### 事务只读

只读事务不存在数据的修改，因此数据库会对只读事务提供一些优化

## 事务的传播特性

### REQUIRED



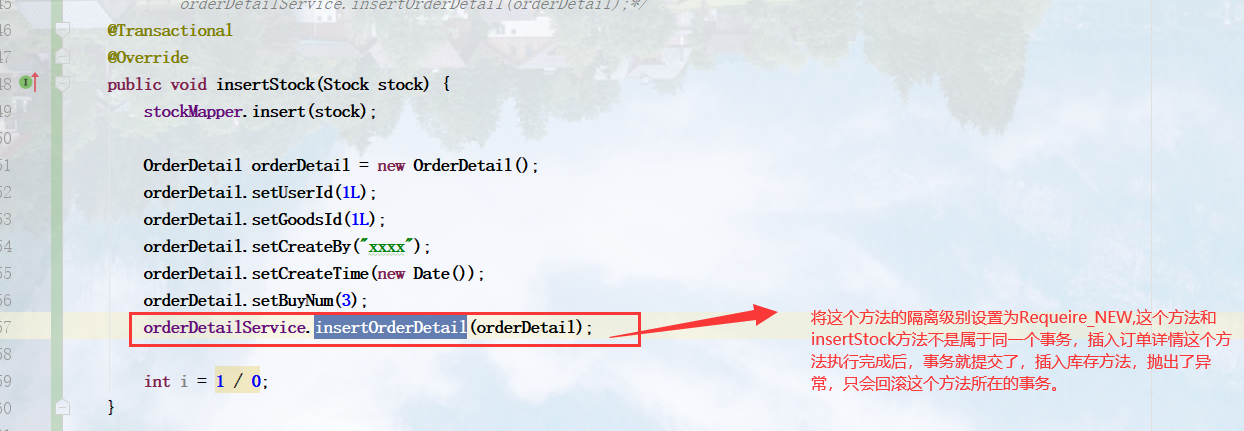


### SUPPORTS

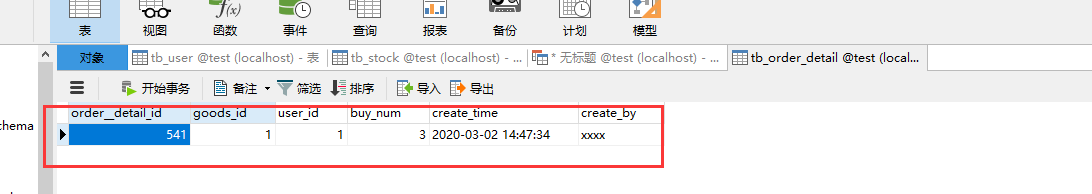


### MANDATORY

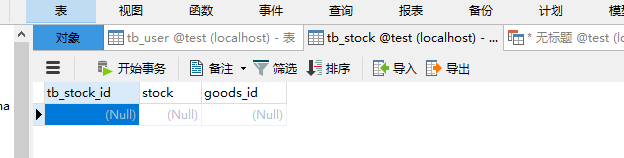
### REQUIRES\_NEW



结果：订单详情表



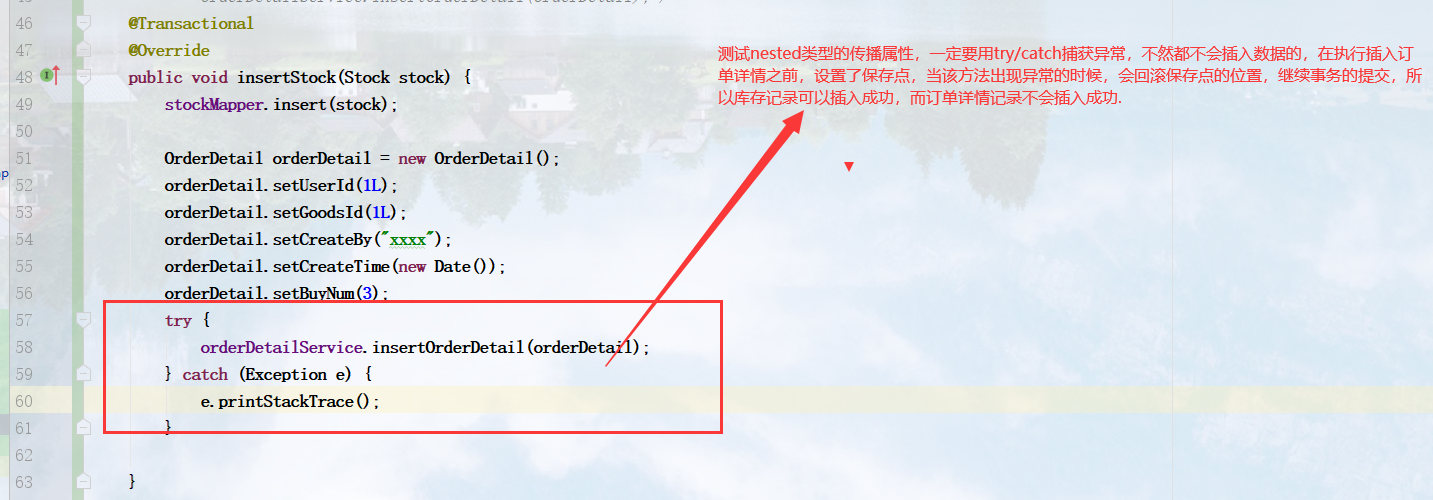
库存表



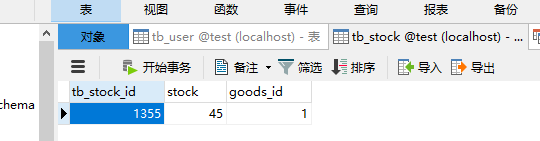
#### NOT\_SUPPORTED

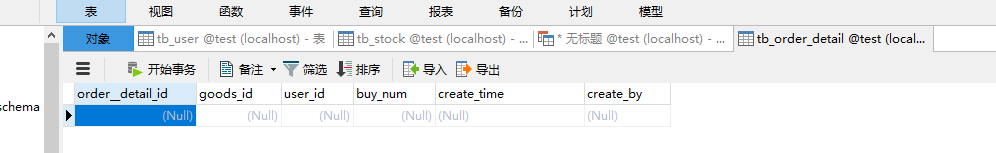
### NEVER

### NESTED



结果：



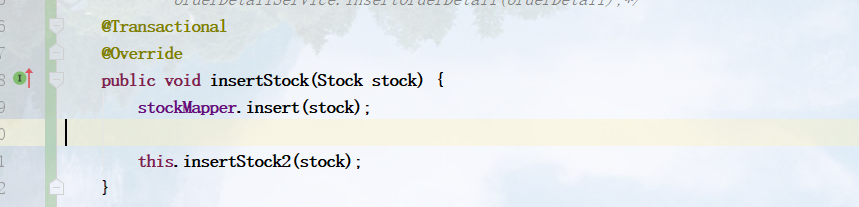


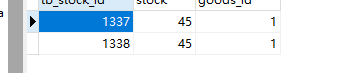
## 工作中常**遇到**的事务问题

### 在同一个类中调用带有@transacrional的方法

第一种情况:正常执行

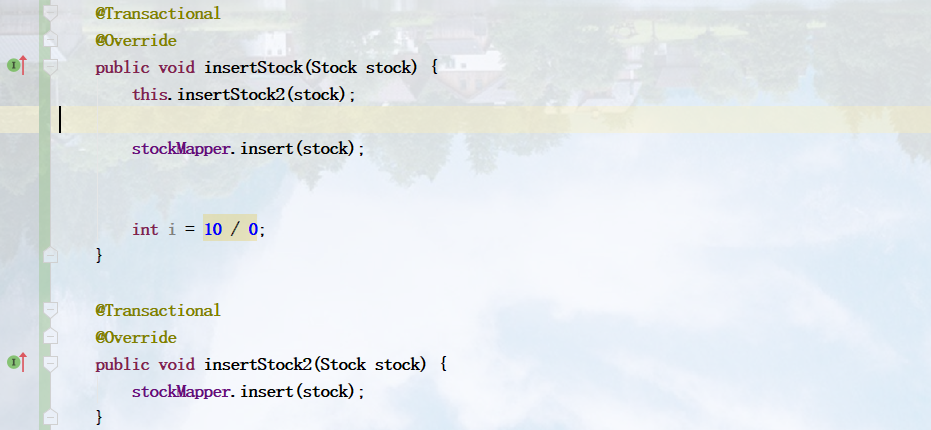
结果:





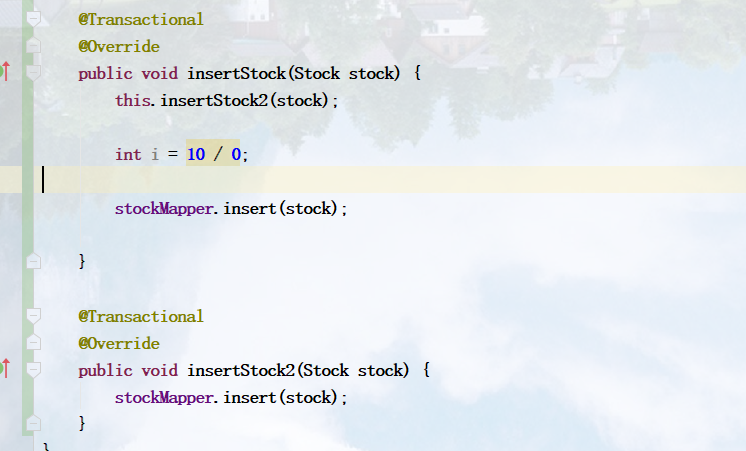
数据库中添加了两条记录

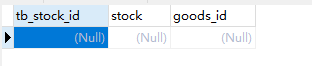
第二种情况:在结束之前添加一行异常





第三种情况:





### 在并发情况下重复添加多条记录，虽然已经添加synchronized

这是因为在方法结束的时候，事务并没有提交，但是其他线程已经获取到了锁，进入了该方法中，由于事务没有提交，数据也就没有添加进去，所以会再次创建一条数据，然后另一个线程将数据提交了，这个线程也将数据提交了，这就造成了数据重复的问题，解决方法，可以再写一个方法，然后去调用这个添加事务的方法，但是新写的方法不要添加事务注解

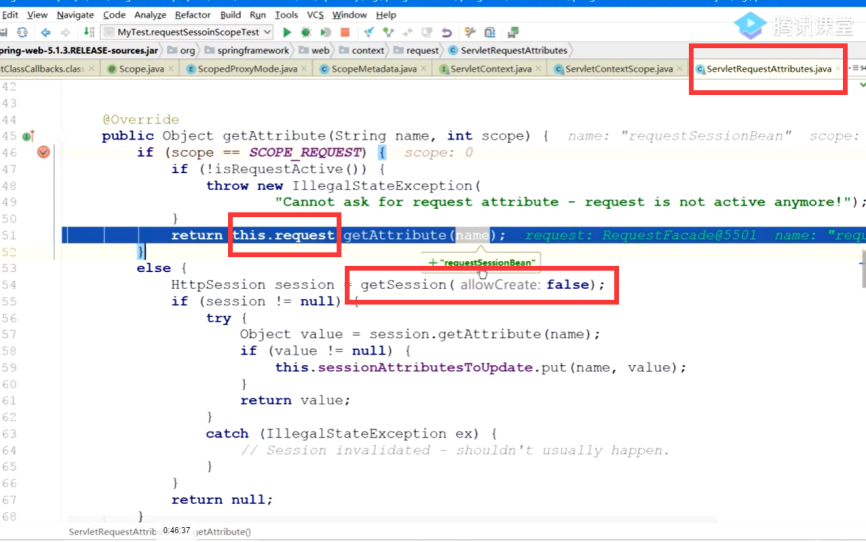


# Scope

Request、session作用域在web环境中才有效

Request作用域使用request来缓存的

Session作用域是缓存在session中的



注册scope，可以自定义scope

在哪里注册，可以实现BeanFactoryProcessor接口，调用registerScope

# @import

可能扫描包路径扫描不到某一个类，就可以通过@import导入到spring IOC容器中去。

# ConfigurationClassBeanDefinitionReader

可以用来处理@bean @configuration等注解

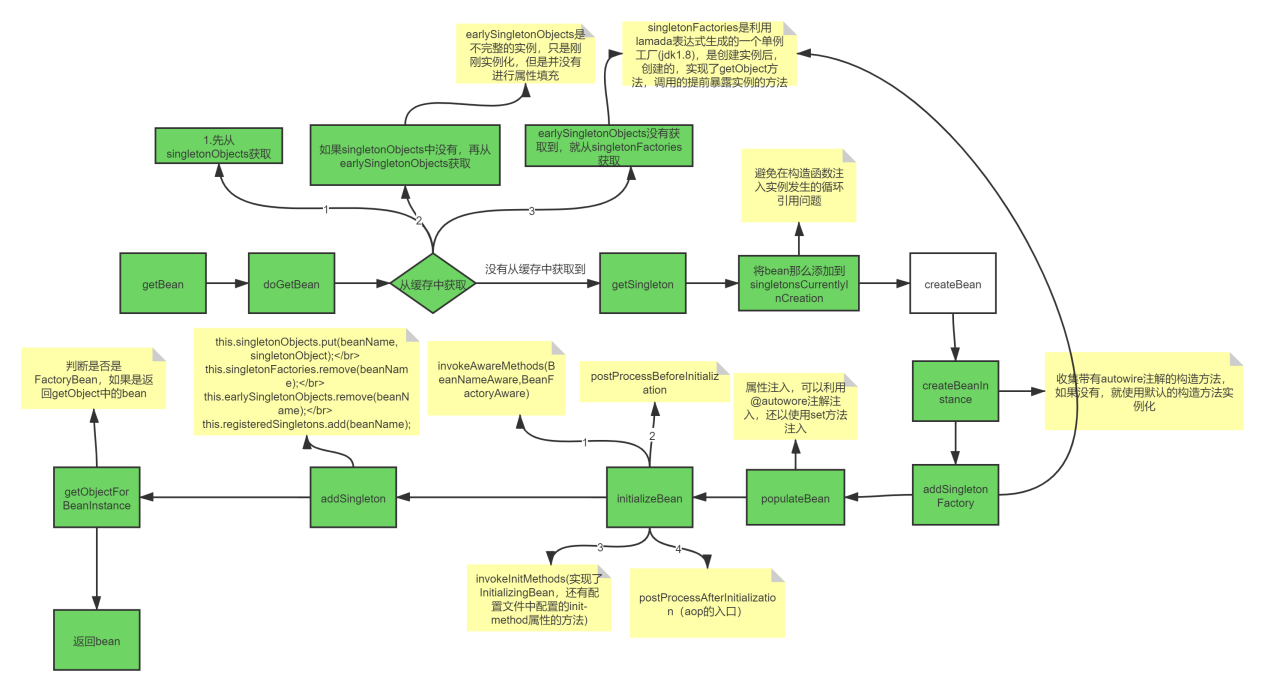
# LocalVariableTableParameterNameDiscoverer

可以拿到参数名称

|  |
| --- |
| **public void** getParameterName() {  LocalVariableTableParameterNameDiscoverer discoverer = **new** LocalVariableTableParameterNameDiscoverer();  // 必须不能是抽象类  Method[] declaredMethods = OrderServiceImpl.**class**.getDeclaredMethods();  **for** (Method method : declaredMethods) {  String[] parameterNames = discoverer.getParameterNames(method);  **if** (parameterNames != **null**) {  **for** (String str : parameterNames) {  System.***out***.println(str);  }  }   } } |

# 流程图

## IOC的流程图



## AOP的流程图

