**Spring Framework**

**学习手册**

**番石榴工作室**

2014年10月16

目录

[1. Spring Framework介绍 4](#_Toc402212493)

[2. IoC容器 5](#_Toc402212494)

[2.1 基本原理 容器和bean 5](#_Toc402212495)

[2.1.1 Bean 5](#_Toc402212496)

[2.1.2 容器 6](#_Toc402212497)

[2.1.3 实例化Spring容器 6](#_Toc402212498)

[2.2 源码分析 6](#_Toc402212499)

[2.2.1 prepareRefresh 8](#_Toc402212500)

[2.2.2 obtainFreshBeanFactory 8](#_Toc402212501)

[2.2.3 prepareBeanFactory(ConfigurableListableBeanFactory) 21](#_Toc402212502)

[2.2.4 postProcessBeanFactory(ConfigurableListableBeanFactory) 22](#_Toc402212503)

[2.2.5 invokeBeanFactoryPostProcessors 22](#_Toc402212504)

[2.2.6 registerBeanPostProcessors 26](#_Toc402212505)

[2.2.7 initMessageSource 27](#_Toc402212506)

[2.2.8 initApplicationEventMulticaster 27](#_Toc402212507)

[2.2.9 onRefresh 27](#_Toc402212508)

[2.2.10 registerListeners 28](#_Toc402212509)

[2.2.11 finishBeanFactoryInitialization 28](#_Toc402212510)

[2.2.12 finishRefresh 40](#_Toc402212511)

[3. Spring MVC 41](#_Toc402212512)

[3.1 简介 41](#_Toc402212513)

[3.2 源码分析 42](#_Toc402212514)

[3.2.1 init() 42](#_Toc402212515)

[4. Spring AOP 43](#_Toc402212516)

[附录 接口/类解析 43](#_Toc402212517)

[BeanFactory 43](#_Toc402212518)

[ApplicationContext 43](#_Toc402212519)

[Environment 43](#_Toc402212520)

[PropertyResolver 44](#_Toc402212521)

[PropertySource 44](#_Toc402212522)

[BeanDefinitionReader 45](#_Toc402212523)

[AliasRegistry 45](#_Toc402212524)

[ResourceLoader 46](#_Toc402212525)

[EnvironmentCapable 46](#_Toc402212526)

[EntityResolver 47](#_Toc402212527)

[Resource 47](#_Toc402212528)

[DocumentLoader 48](#_Toc402212529)

[BeanDefinition 48](#_Toc402212530)

[BeanMetadataElement 49](#_Toc402212531)

[BeanExpressionResolver 49](#_Toc402212532)

[PropertyEditorRegistrar 49](#_Toc402212533)

[ClassLoader 50](#_Toc402212534)

[ClassMetadata 50](#_Toc402212535)

[Scope 51](#_Toc402212536)

[BeanFactoryPostProcessor 51](#_Toc402212537)

[BeanPostProcessor 52](#_Toc402212538)

[InstantiationAwareBeanPostProcessor 52](#_Toc402212539)

[MergedBeanDefinitionPostProcessor 52](#_Toc402212540)

[DestructionAwareBeanPostProcessor 53](#_Toc402212541)

[MessageSource 53](#_Toc402212542)

[ApplicationEventMulticaster 54](#_Toc402212543)

[ThemeSource 54](#_Toc402212544)

[ApplicationListener 55](#_Toc402212545)

[BeanWrapper 55](#_Toc402212546)

[ConversionService 56](#_Toc402212547)

[BeanPostProcessor 56](#_Toc402212548)

[PropertyEditorRegistry 57](#_Toc402212549)

[TypeConverter 57](#_Toc402212550)

[PropertyEditor 58](#_Toc402212551)

[InstantiationStrategy 58](#_Toc402212552)

[PropertyValues 59](#_Toc402212553)

[FeatureDescriptor/PropertyDescriptor 59](#_Toc402212554)

[Lifecycle 60](#_Toc402212555)

[ApplicationEvent 60](#_Toc402212556)

[DispatcherServlet 61](#_Toc402212557)

# Spring Framework介绍

Spring是一个开源框架，Spring是于2003 年兴起的一个轻量级的Java 开发框架，由[Rod Johnson](http://baike.baidu.com/view/2192255.htm) 在其著作Expert One-On-One J2EE Development and Design中[阐述](http://baike.baidu.com/view/620953.htm)的部分理念和原型衍生而来。它是为了解决企业应用开发的复杂性而创建的。Spring使用基本的[JavaBean](http://baike.baidu.com/view/28155.htm)来完成以前只可能由EJB完成的事情。然而，Spring的用途不仅限于[服务器](http://baike.baidu.com/view/899.htm)端的开发。从简单性、可测试性和松耦合的角度而言，任何Java应用都可以从Spring中受益。

目的：解决企业应用开发的复杂性

功能：使用基本的JavaBean代替EJB，并提供了更多的企业[应用](http://baike.baidu.com/view/220910.htm)功能

范围：任何Java应用

简单来说，Spring是一个[轻量级](http://baike.baidu.com/view/1318763.htm)的[控制反转](http://baike.baidu.com/view/1486379.htm)（[IoC](http://baike.baidu.com/view/146665.htm)）和面向切面（[AOP](http://baike.baidu.com/view/73626.htm)）的[容器](http://baike.baidu.com/view/864334.htm)框架。

轻量——从大小与开销两方面而言Spring都是轻量的。完整的Spring[框架](http://baike.baidu.com/view/66971.htm)可以在一个大小只有1MB多的JAR文件里发布。并且Spring所需的处理开销也是微不足道的。此外，Spring是非侵入式的：典型地，Spring应用中的[对象](http://baike.baidu.com/view/2387.htm)不依赖于Spring的特定类。

[控制反转](http://baike.baidu.com/view/1486379.htm)——Spring通过一种称作[控制反转](http://baike.baidu.com/view/1486379.htm)（[IoC](http://baike.baidu.com/subview/146665/146665.htm)）的技术促进了低耦合。当应用了IoC，一个[对象](http://baike.baidu.com/view/2387.htm)依赖的其它对象会通过被动的方式传递进来，而不是这个对象自己创建或者查找依赖对象。你可以认为IoC与JNDI相反——不是[对象](http://baike.baidu.com/view/2387.htm)从容器中查找依赖，而是容器在对象初始化时不等对象请求就主动将依赖传递给它。

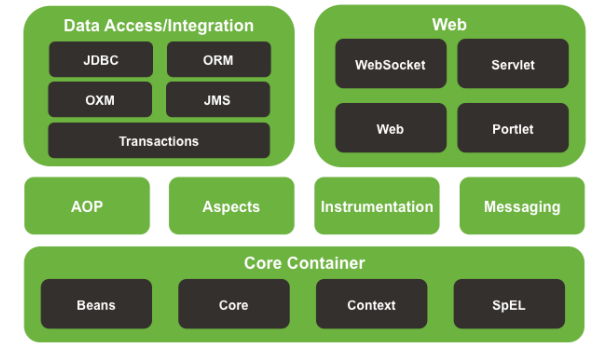
面向切面——Spring提供了[面向切面编程](http://baike.baidu.com/view/1865230.htm)的丰富支持，允许通过分离应用的业务[逻辑](http://baike.baidu.com/view/1838.htm)与[系统](http://baike.baidu.com/view/25302.htm)级[服务](http://baike.baidu.com/view/133203.htm)（例如审计（auditing）和[事务](http://baike.baidu.com/view/121511.htm)（[transaction](http://baike.baidu.com/view/2368136.htm)）管理）进行[内聚性](http://baike.baidu.com/view/2424328.htm)的开发。[应用对象](http://baike.baidu.com/view/5314991.htm)只实现它们应该做的——完成业务逻辑——仅此而已。它们并不负责（甚至是意识）其它的系统级关注点，例如日志或[事务](http://baike.baidu.com/view/121511.htm)支持。

容器——Spring包含并管理[应用对象](http://baike.baidu.com/view/5314991.htm)的配置和生命周期，在这个意义上它是一种容器，你可以配置你的每个bean如何被创建——基于一个可配置[原型](http://baike.baidu.com/view/228368.htm)（[prototype](http://baike.baidu.com/subview/1217697/14146221.htm)），你的bean可以创建一个单独的实例或者每次需要时都生成一个新的实例——以及它们是如何相互关联的。然而，Spring不应该被混同于传统的重量级的EJB容器，它们经常是庞大与笨重的，难以使用。

[框架](http://baike.baidu.com/view/66971.htm)——Spring可以将简单的[组件](http://baike.baidu.com/view/379950.htm)配置、组合成为复杂的应用。在Spring中，[应用对象](http://baike.baidu.com/view/5314991.htm)被声明式地组合，典型地是在一个XML文件里。Spring也提供了很多基础功能（[事务管理](http://baike.baidu.com/view/3871410.htm)、持久化框架集成等等），将应用逻辑的开发留给了你。

MVC——Spring的作用是整合，但不仅仅限于整合，Spring 框架可以被看做是一个企业解决方案级别的框架。客户端发送请求，服务器控制器（由DispatcherServlet实现的)完成请求的转发，控制器调用一个用于映射的类HandlerMapping，该类用于将请求映射到对应的处理器来处理请求。HandlerMapping 将请求映射到对应的处理器Controller（相当于Action）在Spring 当中如果写一些处理器组件，一般实现Controller 接口，在Controller 中就可以调用一些Service 或DAO 来进行数据操作 ModelAndView 用于存放从DAO 中取出的数据，还可以存放响应视图的一些数据。 如果想将处理结果返回给用户，那么在Spring 框架中还提供一个视图组件ViewResolver，该组件根据Controller 返回的标示，找到对应的视图，将响应response 返回给用户。

所有Spring的这些特征使你能够编写更干净、更可管理、并且更易于测试的代码。它们也为Spring中的各种模块提供了基础支持。



# IoC容器

org.springframework.beans及org.springframework.context包是Spring IoC容器的基础。BeanFactory提供了高级配置机制，使得管理各种对象称为可能。ApplicationContext是BeanFactory的扩展，功能得到了进一步增强，比如更易与Spring AOP集成、资源处理（国际化处理）、事件传递及各种不同应用层的context实现（如针对Web应用的WebApplicationContext）。

简而言之，BeanFactory提供了配置框架及基本功能，而ApplicationContext则增加了更多支持企业核心内容功能。ApplicationContext完全由BeanFactory扩展而来，因而BeanFactory所具备的能力和行为也适用于ApplicationContext。

## 基本原理 容器和bean

### Bean

在Spring中，那些组成你应用程序的主体（backbone）及由Spring Ioc容器所管理的对象，被称之为bean。简单地讲，bean就是由Spring容器初始化、装配及管理的对象，除此之外，bean就与应用程序中的其他对象没有什么区别了。而bean定义以及bean相互间的依赖关系将通过元数据来描述。

为什么使用bean?

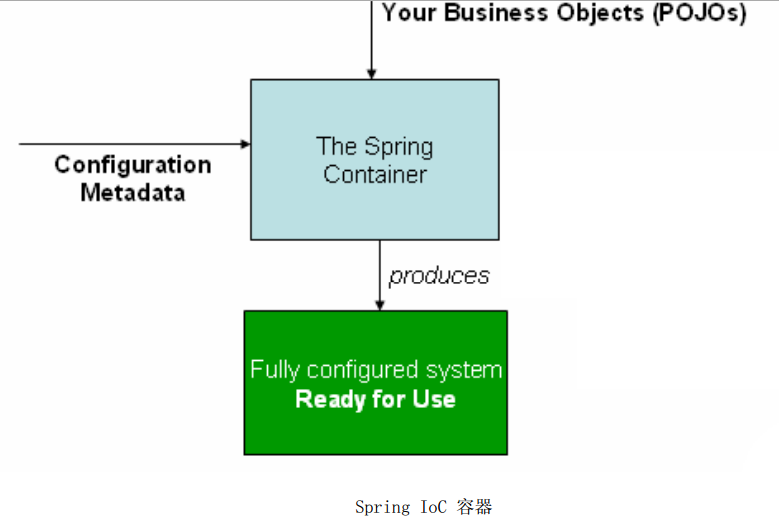
使用”bean”这个名字而不是”组件(component)”或”对象(Object)”的动机源于Spring框架本身(它产生的部分原因就是对Enterprise Java Beans复杂性的一次解构)。

### 容器

org.springframework.beans.factory.BeanFactory是Spring IoC容器的实际代表者，IoC容器负责容纳此前所描述的bean，并对bean进行管理。

在Spring中，BeanFactory是IoC容器的核心接口。它的责任包括：实例化、定位、配置应用程序中的对象及建立这些对象间的依赖。

Spring为我们提供了许多易用的BeanFactory实现，XmlBeanFactory就是最常用的一个。该实现将以XML方式描述组成应用的对象以及对象间的依赖关系。XmlBeanFactory类将获得此XML配置元数据，并讲它来构建一个完全可配置的系统或应用。



### 实例化Spring容器

Spring IoC容器的实例化非常简单，如下：

ApplicationContext ac = new ClassPathXmlApplicationContext(new String[]{”services.xml”,”dao.xml”});

关于ApplicationContext详细信息见<<附录 接口/类解析>> [ApplicationContext](#_ApplicationContext)章节。

## 源码分析

本章节通过实例化Spring容器来引入Spring IoC源码分析，源码分析中涉及的接口/类介绍请参考<< [附录 接口/类解析](#_附录_接口/类解析)>>章节。

如果想获得一个ApplicationContext实例，一般是new一个ClassPathXmlApplicationContext、FileSystemXmlApplicationContext、XmlWebApplicationContext、AnnotationConfigWebApplicationContext。

ClassPathXmlApplicationContext实例化方式：

ApplicationContext ac = new ClassPathXmlApplicationContext(“configuration.xml”);

XmlWebApplicationContext实例化方式：

XmlWebApplicationContext适合Web工程，在Spring web包中有此方法的调用。

方法说明：实例化ApplicationContext

接口/类路径：org.springframework.web.servlet.FrameworkServlet.java

方法：protected WebApplicationContext createWebApplicationContext(ApplicationContext parent);

源代码：

ConfigurableWebApplicationContext wac = (ConfigurableWebApplicationContext) BeanUtils.instantiateClass(contextClass); // 实例化XmlWebApplicationContext

wac.setEnvironment(getEnvironment());// 设置标准[Environment](#_Environment)信息，StandardServletEnvironment

wac.setParent(parent);// 设置parent Spring容器上下文

wac.setConfigLocation(getContextConfigLocation());// 配置在web.xml中的init-param[]中，param-name为contextConfigLocation.

方法说明:实例化StandardServletEnvironment

接口/类路径：org.springframework.web.context.support.StandardServletEnvironment

方法：protected void customizePropertySources(MutablePropertySources propertySources);

源代码：

protected void customizePropertySources(MutablePropertySources propertySources) {

propertySources.addLast(new StubPropertySource(SERVLET\_CONFIG\_PROPERTY\_SOURCE\_NAME));

propertySources.addLast(new StubPropertySource(SERVLET\_CONTEXT\_PROPERTY\_SOURCE\_NAME));

if (JndiLocatorDelegate.isDefaultJndiEnvironmentAvailable()) {

propertySources.addLast(new JndiPropertySource(JNDI\_PROPERTY\_SOURCE\_NAME));

}

super.customizePropertySources(propertySources);

}

代码解读：

在上述createWebApplicationContext中，需要通过getEnvironment()方法获取Environment信息，如果获取不到则创建StandardServletEnvironment实例，由于StandardServletEnvironment继承于AbstractEnvironment抽象类，实例化StandardServletEnvironment同时也会实例化其父类，在AbstractEnvironment实例化中，会调用customizePropertySources初始化一些属性源信息。属性信息包括：

servletConfigInitParams Servlet配置属性

servletContextInitParams Servlet上下文属性

jndiProperties JNDI属性

systemProperties 当前的系统属性

systemEnvironment 系统环境属性

但不管采用的是何种ApplicationContext的实现，最后还是需要调用org.springframework.context.support.AbstractApplicationContext的refresh()方法来读取配置文件，并实例化bean。

接口/类路径：org.springframework.context.support.AbstractApplicationContext

代码方法：refresh()

源代码：

public void refresh() throws BeansException, IllegalStateException {

synchronized (this.startupShutdownMonitor) {// 上下文刷新和销毁的同步监听器

prepareRefresh();

ConfigurableListableBeanFactory beanFactory = obtainFreshBeanFactory();

prepareBeanFactory(beanFactory);

try {

postProcessBeanFactory(beanFactory);

invokeBeanFactoryPostProcessors(beanFactory);

registerBeanPostProcessors(beanFactory);

initMessageSource();

initApplicationEventMulticaster();

onRefresh();

registerListeners();

finishBeanFactoryInitialization(beanFactory);

finishRefresh();

} catch (BeansException ex) {

destroyBeans();

cancelRefresh(ex);

throw ex;

}

}

}

接下来我们通过解析refresh()方法来剖析整个Spring容器上下文的实例化过程。

### prepareRefresh

方法说明：上下文刷新前的准备，主要是设置启动时间、活动标识、获取属性来源。

接口/类路径：org.springframework.context.support.AbstractApplicationContext

代码方法：prepareRefresh ()

源代码：

protected void prepareRefresh() {

this.startupDate = System.currentTimeMillis();

synchronized (this.activeMonitor) {

this.active = true;

}

initPropertySources();

getEnvironment().validateRequiredProperties();

}

代码解读：

initPropertySources()方法重新设置**StandardServletEnvironment** 中的propertySources队列中StubPropertySource中的类型.如果是Web工程，会将StubPropertySource([PropertySource](#_PropertySource))转换为ServletContextPropertySource类型。

validateRequiredProperties()方法校验ApplicationContext初始化过程中必须存在的配置, 此方法最终调用org.springframework.core.env.AbstractPropertyResolver中的validateRequiredProperties()方法完成校验.

### obtainFreshBeanFactory

方法说明：创建Spring IoC容器BeanFactory，读取上下文配置文件，并根据配置文件构建BeanDefinition。

接口/类路径：org.springframework.context.support.AbstractApplicationContext

代码方法：obtainFreshBeanFactory ()

源代码

protected ConfigurableListableBeanFactory obtainFreshBeanFactory() {

refreshBeanFactory();// 详细见[refreshBeanFactory](#_refreshBeanFactory)

ConfigurableListableBeanFactory beanFactory = getBeanFactory();

return beanFactory;

}

代码解读

此方法将调用org.springframework.context.support.AbstractRefreshableApplicationContext 的refreshBeanFactory()方法真正完成BeanFactory的刷新，详细见[refreshBeanFactory](#_refreshBeanFactory)。

#### refreshBeanFactory

方法说明：上下文刷新前的准备，主要是设置启动时间、活动标识、获取属性来源。

接口/类路径：org.springframework.context.support.AbstractRefreshableApplicationContext

代码方法：refreshBeanFactory()

源代码：

protected final void refreshBeanFactory() throws BeansException {

if (hasBeanFactory()) {// 判断当前Spring上下文中是否已经实例化了IoC容器BeanFactory

destroyBeans();// 销毁单例Bean对象

closeBeanFactory();// 关闭已有的IoC容器BeanFactory，其实就是将上下文的BeanFactory设置为null

}

try {

//为当前上下文创建IoC容器BeanFactory

DefaultListableBeanFactory beanFactory = createBeanFactory();

//指定一个id用于序列化，允许通过这个id反序列化会这个BeanFactory对象

beanFactory.setSerializationId(getId());

// 为BeanFactory设置自定义相关信息

// 涉及allowBeanDefinitionOverriding和allowCircularReferences字段。

customizeBeanFactory(beanFactory);

// 根据上下文配置获取Bean信息BeanDefiition

loadBeanDefinitions(beanFactory);

synchronized (this.beanFactoryMonitor) {

this.beanFactory = beanFactory;

}

} catch (IOException ex) {

}

}

代码解读

hasBeanFactory()方法判断ApplicationContext上下文中是否初始化了一个IoC工厂BeanFactory，在此方法中通过同步 beanFactoryMonitor对象来保证工厂被完全初始化。如果上下文已经初始化了IoC工厂BeanFactory，则调用destroyBeans()销毁BeanFactory中所有的SingletonsBean，调用closeBeanFactory方法销毁BeanFactory工厂。

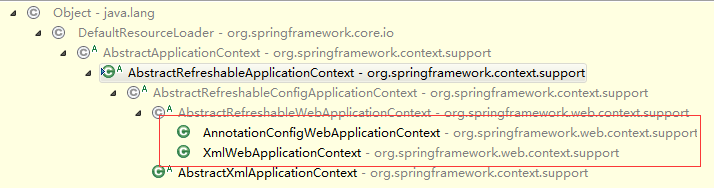
createBeanFactory()方法用来创建BeanFactory，创建BeanFactory的类型为DefaultListableBeanFactory子类，DefaultListableBeanFactory继承了AbstractAutowireCapableBeanFactory抽象类，在初始化AbstractAutowireCapableBeanFactory时，会将BeanNameAware、BeanFactoryAware和BeanClassLoaderAware三个接口加入到ignoredDependencyInterfaces(依赖接口依赖检查和自动装配,忽略类对象的集合。默认情况下,只有BeanFactory接口将被忽略)。

beanFactory.setSerializationId()目的是指定一个id为进行序列化, 如果必要，允许这BeanFactory从这个id反序列化回BeanFactory对象。

loadBeanDefinitions()方法用来从上下文配置文件中加载出所有的bean定义。详细见[loadBeanDefinitions(DefaultListableBeanFactory beanFactory)](#_loadBeanDefinitions(DefaultListable)。

#### loadBeanDefinitions(DefaultListableBeanFactory beanFactory)

loadBeanDefinitions是AbstractRefreshableApplicationContext提供的一个钩子方法，由其实现类实现，Spring中主要存在基于Xml的上下文和基于注解的上下文，如下图：



我们这里从基于Xml配置的上下文进行解析，如下:

方法说明：通过XmlBeanDefinitionReader加载出Bean定义BeanDefiition。

接口/类路径：org.springframework.web.context.support.XmlWebApplicationContext

代码方法：loadBeanDefinitions(DefaultListableBeanFactory beanFactory)

源代码：

protected void loadBeanDefinitions(DefaultListableBeanFactory beanFactory) throws BeansException, IOException {

// 为BeanFactory创建bean定义阅读器

XmlBeanDefinitionReader beanDefinitionReader = new XmlBeanDefinitionReader(beanFactory);

// 设置XmlBeanDefinitionReader的Environment环境信息，与上下文ApplicationContext保持一致

beanDefinitionReader.setEnvironment(getEnvironment());

// 设置ResourceLoader信息，this告诉我们上下文XmlWebApplicationContext其实就是一个ResourceLoader。

beanDefinitionReader.setResourceLoader(this);

// 设置EntityResolver，ResourceEntityResolver中包含了BeansDtdResolver和PluggableSchemaResolver

beanDefinitionReader.setEntityResolver(new ResourceEntityResolver(this));

initBeanDefinitionReader(beanDefinitionReader);

// 使用XmlBeanDefinitionReader根据上下文配置加载出所有的BeanDefiition

loadBeanDefinitions(beanDefinitionReader);

}

代码解读

XmlBeanDefinitionReader：用于解析XML上下文配置文件加载出所有的bean定义的工具类。

beanDefinitionReader.setEnvironment(getEnvironment()); 设置XmlBeanDefinitionReader的环境信息，与ApplicationContext保持一致。

beanDefinitionReader.setResourceLoader(this); 设置XmlBeanDefinitionReader的ResourceLoader（详细见[ResourceLoader](#_ResourceLoader)和[ApplicationContext](#_ApplicationContext)），与ApplicationContext保持一致，web工程为PathMatchingResourcePatternResolver。

beanDefinitionReader.setEntityResolver(new ResourceEntityResolver(this))；设置XmlBeanDefinitionReader的EntityResolver[ResourceEntityResolver]， ResourceEntityResolver继承了DelegatingEntityResolver类，**DelegatingEntityResolver**中包含了BeansDtdResolver和PluggableSchemaResolver。BeansDtdResolver用于解析.Dtd文档模式的xml文件; PluggableSchemaResolver用于解析.xsd文档模式的xml文件。详细见[EntityResolver](#_EntityResolver)。

loadBeanDefinitions(beanDefinitionReader); 解析上下文配置文件，获取bean定义信息。详细见[loadBeanDefinitions(XmlBeanDefinitionReader reader)](#_loadBeanDefinitions(XmlBeanDefiniti)

#### loadBeanDefinitions(XmlBeanDefinitionReader reader)

方法说明：通过指定的XmlBeanDefinitionReader加载出上下文配置的BeanDefiition，通过ResourcePatternResolver[ServletContextResourcePatternResolver]解析ConfigLocations，并生成相应的Resource。详细见[loadBeanDefinitions(String, Set<Resource>)](#_loadBeanDefinitions(String,_Set<Res)

接口/类路径：org.springframework.web.context.support.XmlWebApplicationContext

代码方法：loadBeanDefinitions(XmlBeanDefinitionReader reader)

源代码：

protected void loadBeanDefinitions(XmlBeanDefinitionReader reader) throws IOException {

String[] configLocations = getConfigLocations();

if (configLocations != null) {

for (String configLocation : configLocations) {

reader.loadBeanDefinitions(configLocation);

}

}

}

#### loadBeanDefinitions(String, Set<Resource>)

方法说明：根据上下文配置文件，构建Resource，调用XmlBeanDefinitionReader. loadBeanDefinitions(EncodedResource encodedResource)

接口/类路径：org.springframework.beans.factory.support.AbstractBeanDefinitionReader

代码方法：loadBeanDefinitions(String, Set<Resource>)

源代码：

**public** **int** loadBeanDefinitions(String location, Set<Resource> actualResources) **throws** BeanDefinitionStoreException {

ResourceLoader resourceLoader = getResourceLoader();

**if** (resourceLoader == **null**) {

**throw** **new** BeanDefinitionStoreException("Cannot import bean definitions from location [" + location + "]: no ResourceLoader available");

}

// 在上一步创建XmlBeanDefinitionReader已经实例化了PathMatchingResourcePatternResolver。

**if** (resourceLoader **instanceof** ResourcePatternResolver) {

**try** {

// 将上下文配置文件解析成Resource

// 详细见：[PathMatchingResourcePatternResolver.getResources](#_PathMatchingResourcePatternResolver)

Resource[] resources = ((ResourcePatternResolver) resourceLoader).getResources(location);

**int** loadCount = loadBeanDefinitions(resources);

**if** (actualResources != **null**) {

**for** (Resource resource : resources) {

actualResources.add(resource);

}

}

**return** loadCount;

} **catch** (IOException ex) {

**throw** **new** BeanDefinitionStoreException("Could not resolve bean definition resource pattern [" + location + "]", ex);

}

} **else** {

// Can only load single resources by absolute URL.

Resource resource = resourceLoader.getResource(location);

**int** loadCount = loadBeanDefinitions(resource);

**if** (actualResources != **null**) {

actualResources.add(resource);

}

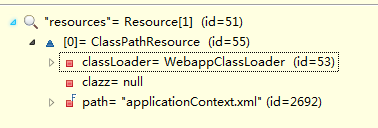
**return** loadCount;

}

}

代码解读

Resource[] resources = ((ResourcePatternResolver) resourceLoader).getResources(location);此部分逻辑将最终调用PathMatchingResourcePatternResolver的getResources(String)方法加载出工程下所有的.class文件以及其他类型的文件资源信息，并将这些信息存放在Resource中的classes字段中，Resource结构见下图。



Resource类图见[Resource](#_Resource)

int loadCount = loadBeanDefinitions(resources); 将Resource封装成EncodedResource，调用[loadBeanDefinitions(EncodedResource encodedResource)](#_loadBeanDefinitions(EncodedResource)方法

#### PathMatchingResourcePatternResolver.getResources

**public** Resource[] getResources(String locationPattern) **throws** IOException {

// locationPattern 🡪 classpath:applicationContext.xml

// CLASSPATH\_ALL\_URL\_PREFIX 🡪 classpath\*:

**if** (locationPattern.startsWith(***CLASSPATH\_ALL\_URL\_PREFIX***)) {

**if** (getPathMatcher().isPattern(locationPattern.substring(***CLASSPATH\_ALL\_URL\_PREFIX***.length()))) {

**return** findPathMatchingResources(locationPattern);

} **else** {

**return** findAllClassPathResources(locationPattern.substring(***CLASSPATH\_ALL\_URL\_PREFIX***.length()));

}

} **else** {

**int** prefixEnd = locationPattern.indexOf(":") + 1;

**if** (getPathMatcher().isPattern(locationPattern.substring(prefixEnd))) {

**return** findPathMatchingResources(locationPattern);

} **else** { // contextConfigLocation配置为”classpath:applicationContext.xml”

**return** **new** Resource[] {getResourceLoader().getResource(locationPattern)};

}

}

}

根据web.xml中配置的contextConfigLocation路径(如：classpath:applicationContext.xml)加载资源信息，如果contextConfigLocation配置为”classpath:applicationContext.xml”则会由DefaultResourceLoader. getResource(String)方法获取Resource信息：

**public** Resource getResource(String location) {

Assert.*notNull*(location, "Location must not be null");

**if** (location.startsWith("/")) {

**return** getResourceByPath(location);

} **else** **if** (location.startsWith(***CLASSPATH\_URL\_PREFIX***)) {// classpath:applicationContext.xml

**return** **new** ClassPathResource(location.substring(***CLASSPATH\_URL\_PREFIX***.length()), getClassLoader());

} **else** {

**try** {

URL url = **new** URL(location);

**return** **new** UrlResource(url);

} **catch** (MalformedURLException ex) {

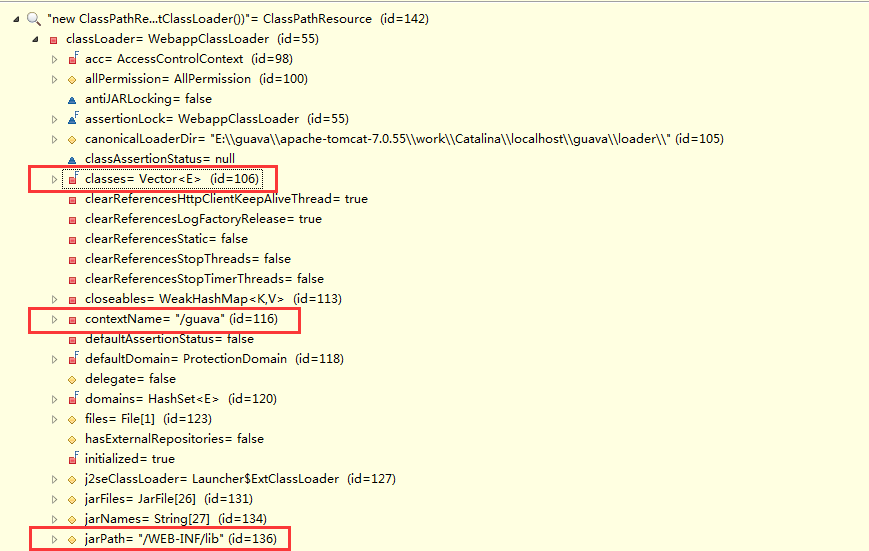
**return** getResourceByPath(location);

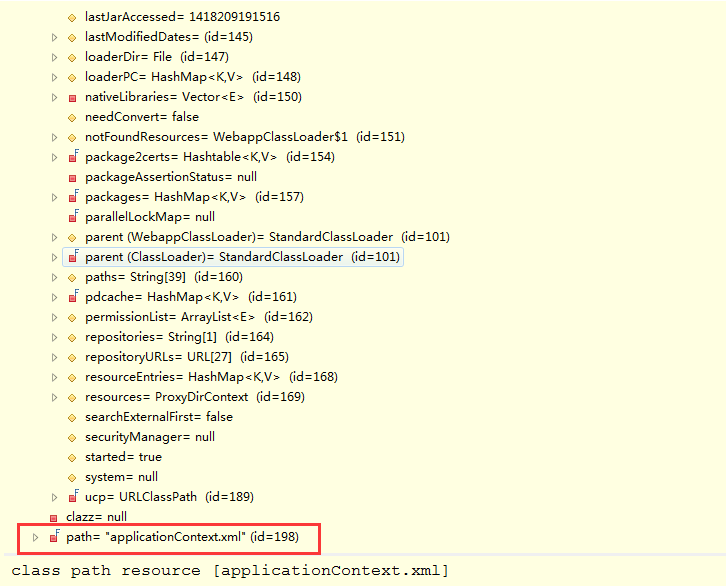
}

}

}

ClassPathResource的结构如下：





ClassPathResource结构

#### loadBeanDefinitions(EncodedResource encodedResource)

方法说明：根据上下文配置文件，构建Resource，调用XmlBeanDefinitionReader. loadBeanDefinitions(EncodedResource encodedResource)

接口/类路径：org.springframework.beans.factory.xml.XmlBeanDefinitionReader

代码方法：loadBeanDefinitions(EncodedResource encodedResource)

源代码：

public int loadBeanDefinitions(EncodedResource encodedResource) throws BeanDefinitionStoreException {

// …

try {

// 获取文件输入流

InputStream inputStream = encodedResource.getResource().getInputStream();

try {

InputSource inputSource = new InputSource(inputStream);

if (encodedResource.getEncoding() != null) {

inputSource.setEncoding(encodedResource.getEncoding());

}

return doLoadBeanDefinitions(inputSource, encodedResource.getResource());

}

}

代码解读

// encodedResource.getResource().getInputStream() 🡪 此处以ClassPathResource为例

**public** InputStream getInputStream() **throws** IOException {

InputStream is;

**if** (**this**.clazz != **null**) {

is = **this**.clazz.getResourceAsStream(**this**.path);

} **else** **if** (**this**.classLoader != **null**) {

is = **this**.classLoader.getResourceAsStream(**this**.path);

} **else** {

is = ClassLoader.*getSystemResourceAsStream*(**this**.path);

}

**if** (is == **null**) {

**throw** **new** FileNotFoundException(getDescription() + " cannot be opened because it does not exist");

}

**return** is;

}

读取上下文配置文件，读取文件输入流，并将其封装成InputSource流，调用[doLoadBeanDefinitions(InputSource, Resource)](#_doLoadBeanDefinitions(InputSource,_)完成解析工作

#### doLoadBeanDefinitions(InputSource, Resource)

方法说明： 将文件流解析成Document对象，再从Document对象中解析出bean定义信息

接口/类路径：org.springframework.beans.factory.xml.XmlBeanDefinitionReader

代码方法：doLoadBeanDefinitions(InputSource, Resource)

源代码：

protected int doLoadBeanDefinitions(InputSource inputSource, Resource resource) throws Exception {

try {

// 方法实现见[doLoadDocument(InputSource, Resource)](#_doLoadDocument(InputSource,_Resourc)

Document doc = doLoadDocument(inputSource, resource);

// 方法实现见[registerBeanDefinitions(Document, Resource)](#_registerBeanDefinitions(Document,_R)

return registerBeanDefinitions(doc, resource);

} catch (Exception ex) {

throw ex;

}

}

代码解读

Document doc = doLoadDocument(inputSource, resource); 将文件流解析成Document对象。

registerBeanDefinitions(doc, resource); // 从Document中解析出bean定义

#### doLoadDocument(InputSource, Resource)

方法说明：获取DefaultDocumentLoader.loadDocument方法所需要的相关数据。[loadDocument(InputSource, EntityResolver,ErrorHandler, int, boolean)](#_loadDocument(InputSource,_EntityRes)

接口/类路径：org.springframework.beans.factory.xml.XmlBeanDefinitionReader

代码方法：doLoadDocument(InputSource, Resource)

源代码：

protected Document doLoadDocument(InputSource inputSource, Resource resource) throws Exception{

return this.documentLoader.loadDocument(inputSource, getEntityResolver(), this.errorHandler, getValidationModeForResource(resource), isNamespaceAware());

}

代码解读

documentLoader为DefaultDocumentLoader

EntityResolver为DelegatingEntityResolver

此值的设值在[loadBeanDefinitions(DefaultListableBeanFactory beanFactory)](#_loadBeanDefinitions)

errorHandler 为SimpleSaxErrorHandler

getValidationModeForResource(resource) ： 根据xml文档类型定义获取校验模式，.dtd校验模式为2；.xsd校验模式为3.

**protected** **int** getValidationModeForResource(Resource resource) {

**int** validationModeToUse = getValidationMode();

**if** (validationModeToUse != ***VALIDATION\_AUTO***) {

**return** validationModeToUse;

}

// 根据输入流InputStream，读取输入流，跟后判断xml文档声明决定使用那种文档模式

**int** detectedMode = detectValidationMode(resource);

**if** (detectedMode != ***VALIDATION\_AUTO***) {

**return** detectedMode;

}

**return** ***VALIDATION\_XSD***;

}

isNamespaceAware()为false

#### loadDocument(InputSource, EntityResolver,ErrorHandler, int, boolean)

方法说明：获取DefaultDocumentLoader.loadDocument方法所需要的相关数据。[loadDocument(InputSource, EntityResolver,ErrorHandler, int, boolean)](#_loadDocument(InputSource,_EntityRes)

接口/类路径：org.springframework.beans.factory.xml.DefaultDocumentLoader

代码方法：loadDocument(InputSource, EntityResolver,ErrorHandler, int, boolean)

源代码：

public Document loadDocument(InputSource inputSource, EntityResolver entityResolver,

ErrorHandler errorHandler, int validationMode, boolean namespaceAware) throws Exception {

DocumentBuilderFactory factory = createDocumentBuilderFactory(validationMode, namespaceAware);

DocumentBuilder builder = createDocumentBuilder(factory, entityResolver, errorHandler);

return builder.parse(inputSource);

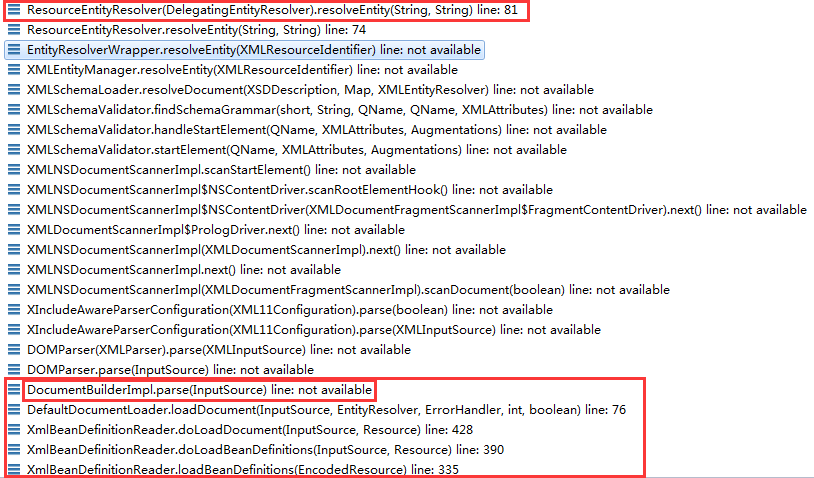
}

代码解读

DocumentBuilderFactory factory = createDocumentBuilderFactory(validationMode, namespaceAware); // 创建DocumentBuilderFactory，如果.xsd文档类型，NamespaceAware会被设置为true并将设置DocumentBuilderFactory的http://java.sun.com/xml/jaxp/properties/schemaLanguage属性为http://www.w3.org/2001/XMLSchema。

DocumentBuilder builder = createDocumentBuilder(factory, entityResolver, errorHandler);// DocumentBuilderFactory创建DocumentBuilder，并设置DocumentBuilder的EntityResolver[ResourceEntityResolver]和ErrorHandler[SimpleSaxErrorHandler]。ResourceEntityResolver 中会包含PluggableSchemaResolver，PluggableSchemaResolver 中的DEFAULT\_SCHEMA\_MAPPINGS\_LOCATION[META-INF/spring.schemas]的定义模式映射的文件的位置。可以出现在多个JAR文件。当此类的getSchemaMappings()方法被调用时，会加载classpath下所有META-INF/spring.schemas文件，组成uri与.xsd文件的键值对关系，存放在schemaMappings中。

我们可以从”builder.parse(InputSource)”的调用方法栈中可以看出



最后还是会回调到DelegatingEntityResolver的resolveEntity(String, String)方法

public InputSource resolveEntity(String publicId, String systemId) throws SAXException, IOException {

if (systemId != null) {

if (systemId.endsWith(DTD\_SUFFIX)) {// DTD文档模式

return this.dtdResolver.resolveEntity(publicId, systemId);

} else if (systemId.endsWith(XSD\_SUFFIX)) {// XSD文档模式

return this.schemaResolver.resolveEntity(publicId, systemId);

}

}

return null;

}

从这段方法的逻辑我们就可以看出，spring委派DelegatingEntityResolver来解析配置文件，由DelegatingEntityResolver根据上下文xml配置文件的文档模式来决定是BeansDtdResolver还是PluggableSchemaResolver解析xml配置文件。

从PluggableSchemaResolver对resolveEntity的实现可以看出，此方法会通过getSchemaMappings()获取所有的SchemaMapping信息。在getSchemaMappings()方法中会加载工程下所有DEFAULT\_SCHEMA\_MAPPINGS\_LOCATION[META-INF/spring.schemas]信息。

#### registerBeanDefinitions(Document, Resource)

方法说明：注册包含在Document中的Bean定义

接口/类路径：org.springframework.beans.factory.xml.XmlBeanDefinitionReader

代码方法：registerBeanDefinitions(Document, Resource)

源代码：

public int registerBeanDefinitions(Document doc, Resource resource) throws BeanDefinitionStoreException {

BeanDefinitionDocumentReader documentReader = createBeanDefinitionDocumentReader();

documentReader.setEnvironment(getEnvironment());

int countBefore = getRegistry().getBeanDefinitionCount();

documentReader.registerBeanDefinitions(doc, createReaderContext(resource));

return getRegistry().getBeanDefinitionCount() - countBefore;

}

代码解读

BeanDefinitionDocumentReader documentReader = createBeanDefinitionDocumentReader();

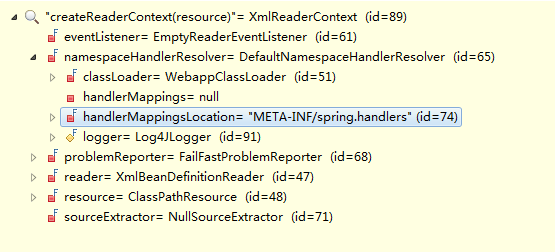
//初始化BeanDefinitionDocumentReader[DefaultBeanDefinitionDocumentReader]。

documentReader.setEnvironment(getEnvironment());

// 设置BeanDefinitionDocumentReader环境信息，与XmlBeanDefinitionReader保持一致，也就是间接的与ApplicationContext保持一致。

createReaderContext(resource);

// 创建XmlReaderContext实例，需要设置eventListener[EmptyReaderEventListener], problemReporter[FailFastProblemReporter]、sourceExtractor[NullSourceExtractor]、NamespaceHandlerResolver[DefaultNamespaceHandlerResolver]。DefaultNamespaceHandlerResolver中需要设置” 寻找位置的映射文件”，默认为META-INF/spring.handlers，可以在classpath下出现多个.



documentReader.registerBeanDefinitions(doc, createReaderContext(resource));//详细见[registerBeanDefinitions(Document, XmlReaderContext)](#_registerBeanDefinitions(Document,_X)。

#### registerBeanDefinitions(Document, XmlReaderContext)

方法说明：获取DefaultDocumentLoader.loadDocument方法所需要的相关数据。[loadDocument(InputSource, EntityResolver,ErrorHandler, int, boolean)](#_loadDocument(InputSource,_EntityRes)

接口/类路径：org.springframework.beans.factory.xml.DefaultBeanDefinitionDocumentReader

代码方法：registerBeanDefinitions(Document, XmlReaderContext)

源代码：

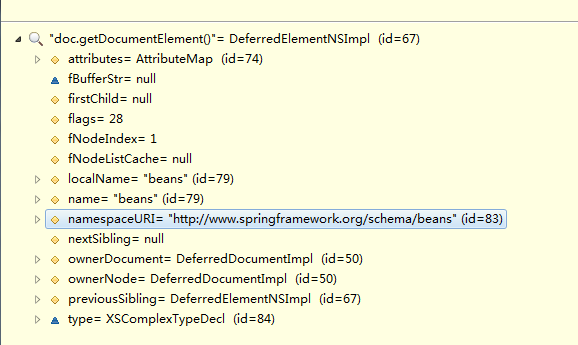
public void registerBeanDefinitions(Document doc, XmlReaderContext readerContext) {

this.readerContext = readerContext;

Element root = doc.getDocumentElement();

doRegisterBeanDefinitions(root); // 详细见[doRegisterBeanDefinitions(Element)](#_doRegisterBeanDefinitions(Element))

}



attributes[AttributeMap]:

[default-autowire="default",

default-lazy-init="default",

default-merge="default",

xmlns="http://www.springframework.org/schema/beans",

xmlns:aop="http://www.springframework.org/schema/aop",

xmlns:context="http://www.springframework.org/schema/context",

xmlns:jee="http://www.springframework.org/schema/jee",

xmlns:mvc="http://www.springframework.org/schema/mvc",

xmlns:p="http://www.springframework.org/schema/p",

xmlns:tx="http://www.springframework.org/schema/tx",

xmlns:util="http://www.springframework.org/schema/util",

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance",

xsi:schemaLocation="http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans-4.0.xsd

http://www.springframework.org/schema/aop

http://www.springframework.org/schema/aop/spring-aop-4.0.xsd

http://www.springframework.org/schema/jee

http://www.springframework.org/schema/jee/spring-jee-4.0.xsd

http://www.springframework.org/schema/context

http://www.springframework.org/schema/context/spring-context-4.0.xsd

http://www.springframework.org/schema/tx

http://www.springframework.org/schema/tx/spring-tx-4.0.xsd

http://www.springframework.org/schema/util

http://www.springframework.org/schema/util/spring-util-4.0.xsd

http://www.springframework.org/schema/mvc

http://www.springframework.org/schema/mvc/spring-mvc-4.0.xsd"]

namespaceURI: http://www.springframework.org/schema/beans

#### doRegisterBeanDefinitions(Element)

方法说明：

接口/类路径：org.springframework.beans.factory.xml.DefaultBeanDefinitionDocumentReader

代码方法：doRegisterBeanDefinitions(Element)

源代码：

protected void doRegisterBeanDefinitions(Element root) {

String profileSpec = root.getAttribute(PROFILE\_ATTRIBUTE);

if (StringUtils.hasText(profileSpec)) {

String[] specifiedProfiles = StringUtils.tokenizeToStringArray(profileSpec, BeanDefinitionParserDelegate.MULTI\_VALUE\_ATTRIBUTE\_DELIMITERS);

if (!getEnvironment().acceptsProfiles(specifiedProfiles)) {

return;

}

} // 判断beans下面是否存在profile属性，对profile属性值进行校验。

BeanDefinitionParserDelegate parent = this.delegate;

this.delegate = createDelegate(this.readerContext, root, parent);

preProcessXml(root);

parseBeanDefinitions(root, this.delegate);

postProcessXml(root);

this.delegate = parent;

}

源码解读

this.delegate = createDelegate(this.readerContext, root, parent); // 创建BeanDefinitionParserDelegate实例后，调用initDefaults方法初始化BeanDefinitionParserDelegate中defaults变量的值。初始化defaults值时，需要根据Element父节点属性，当前节点[Element]属性来设置defaults的值。详细见org.springframework.beans.factory.xml.BeanDefinitionParserDelegate中的populateDefaults方法。

parseBeanDefinitions(root, this.delegate); // 解析root中的元素，构建出BeanDefinition。详细见[parseBeanDefinitions(Element,BeanDefinitionParserDelegate)](#_parseBeanDefinitions(Element,BeanDe)

#### parseBeanDefinitions(Element,BeanDefinitionParserDelegate)

方法说明： 此方法是用来解析Document中的Element元素的，然后构建出BeanDefiition

接口/类路径：org.springframework.beans.factory.xml.DefaultBeanDefinitionDocumentReader

代码方法：parseBeanDefinitions(Element)

源代码：

protected void parseBeanDefinitions(Element root, BeanDefinitionParserDelegate delegate) {

if (delegate.isDefaultNamespace(root)) { // 通过root元素的namespaceURI判断是否为beans节点

NodeList nl = root.getChildNodes();// 获取根节点下的子节点

for (int i = 0; i < nl.getLength(); i++) {

Node node = nl.item(i);

if (node instanceof Element) {

Element ele = (Element) node;

if (delegate.isDefaultNamespace(ele)) {

parseDefaultElement(ele, delegate);

} else {

delegate.parseCustomElement(ele);

}

}

}

} else {

delegate.parseCustomElement(root);

}

}

代码解读

parseDefaultElement(ele, delegate);// 解析名称为”import”、”alias”、”bean”和”beans”节点元素。详细见[parseDefaultElement(Element,BeanDefinitionParserDelegate)](#_parseDefaultElement(Element,BeanDef)。

delegate.parseCustomElement(ele);// 非”import”、”alias”、”bean”、”beans”节点在此方法中解析。详细见[parseCustomElement(Element,BeanDefinition)](#_parseCustomElement(Element,BeanDefi)。

#### parseDefaultElement(Element,BeanDefinitionParserDelegate)

方法说明： 解析名称为”import”、”alias”、”bean”和”beans”的节点元素

接口/类路径：org.springframework.beans.factory.xml.DefaultBeanDefinitionDocumentReader

代码方法：parseDefaultElement (Element)

源代码：

private void parseDefaultElement(Element ele, BeanDefinitionParserDelegate delegate) {

if (delegate.nodeNameEquals(ele, IMPORT\_ELEMENT)) {

importBeanDefinitionResource(ele);

} else if (delegate.nodeNameEquals(ele, ALIAS\_ELEMENT)) {

processAliasRegistration(ele);

} else if (delegate.nodeNameEquals(ele, BEAN\_ELEMENT)) {

processBeanDefinition(ele, delegate);

}else if (delegate.nodeNameEquals(ele, NESTED\_BEANS\_ELEMENT)) {

// recurse

doRegisterBeanDefinitions(ele);

}

}

代码解读

importBeanDefinitionResource(ele);// 解析节点名称为”import”的元素，获取元素属性”resource”,获取需要解析的spring配置文件进行bean定义解析，见[loadBeanDefinitions(EncodedResource encodedResource)](#_loadBeanDefinitions(EncodedResource)。

processAliasRegistration(ele);// 解析节点名称为”alias”的元素，用来支持bean别名

doRegisterBeanDefinitions(ele);// 解析节点名称为”beans”的元素，此段逻辑将重新调回[doRegisterBeanDefinitions(Element)](#_doRegisterBeanDefinitions(Element))方法。

processBeanDefinition(ele, delegate);// 解析节点名称为”bean”的元素，实际处理解析bean节点元素的逻辑在processBeanDefinition方法中实现。源码如下：

protected void processBeanDefinition(Element ele, BeanDefinitionParserDelegate delegate) {

BeanDefinitionHolder bdHolder = delegate.parseBeanDefinitionElement(ele);

if (bdHolder != null) {

bdHolder = delegate.decorateBeanDefinitionIfRequired(ele, bdHolder);

try {

BeanDefinitionReaderUtils.registerBeanDefinition(bdHolder,getReaderContext().getRegistry());

} catch (BeanDefinitionStoreException ex) {}

// Send registration event.

getReaderContext().fireComponentRegistered(new BeanComponentDefinition(bdHolder));

}

}

delegate.parseBeanDefinitionElement(ele);// 根据bean节点属性构建AbstractBeanDefinition，并设置AbstractBeanDefinition相关的默认属性，解析元素的name属性，用”,;”进行切割处理，切割处理后当作alias，构建BeanDefinitionHolder。

bdHolder = delegate.decorateBeanDefinitionIfRequired(ele, bdHolder); // 此段逻辑主要是处理bean下面还有子节点<util:[constant|property-path|list|set|map|properties]>的情况。

#### parseCustomElement(Element,BeanDefinition)

接口/类路径：org.springframework.beans.factory.xml.BeanDefinitionParserDelegate

代码方法：parseCustomElement

源代码：

public BeanDefinition parseCustomElement(Element ele, BeanDefinition containingBd) {

String namespaceUri = getNamespaceURI(ele);

NamespaceHandler handler = this.readerContext.getNamespaceHandlerResolver().resolve(namespaceUri);

if (handler == null) {

error("Unable to locate Spring NamespaceHandler for XML schema namespace [" + namespaceUri + "]", ele);

return null;

}

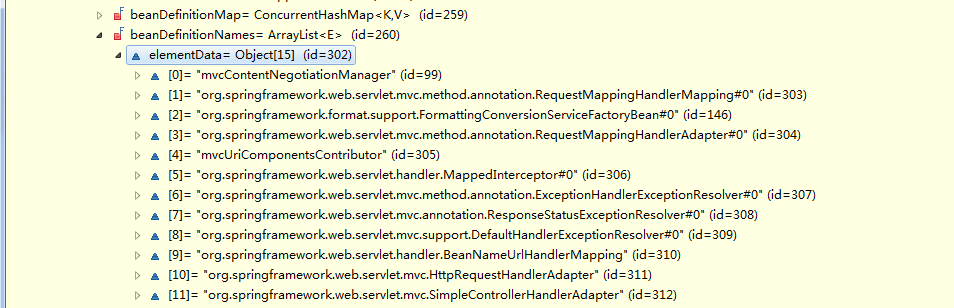
return handler.parse(ele, new ParserContext(this.readerContext, this, containingBd));

}

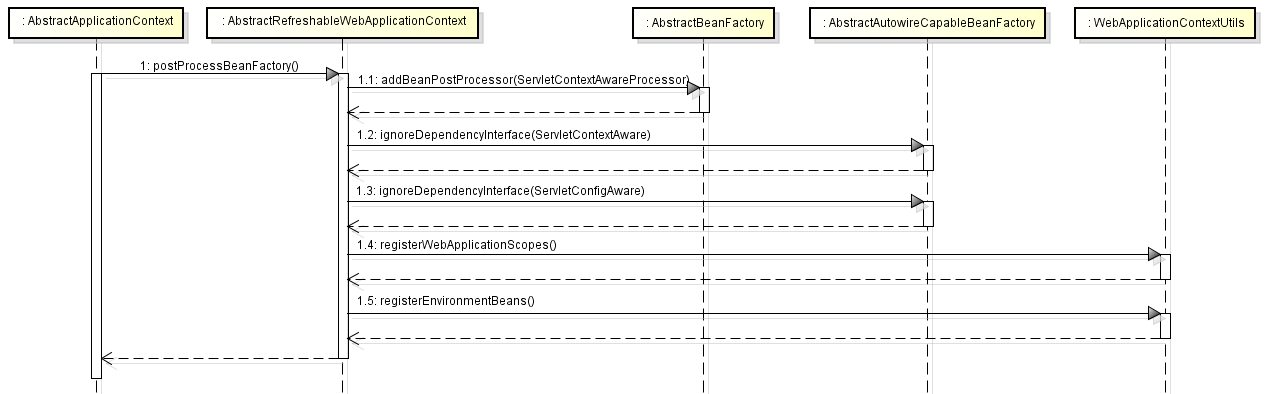
代码解读：

NamespaceHandler handler = this.readerContext.getNamespaceHandlerResolver().resolve(namespaceUri);// 调用DefaultNamespaceHandlerResolver中的resolve方法，方法内部通过加载classpath下”META-INF/spring.handlers”文件，获取namespaceURI与NamespaceHandler映射关系，再根据根据元素的namespaceURI获取对应的NamespaceHandler，并为NamespaceHandler注册BeanDefinitionParser。然后进行解析bean的定义。例如<mvc:annotation-driven>元素。

首先：通过<mvc:annotation-driven>的namespaceURI获取对应的NamespaceHandler为” MvcNamespaceHandler”(关系在spring-webmvc/src/main/resources/META-INF/spring.handlers)中体现，在MvcNamespaceHandler中的init方法中，将注册AnnotationDrivenBeanDefinitionParser[annotation-driven]、DefaultServletHandlerBeanDefinitionParser[default-servlet-handler]、InterceptorsBeanDefinitionParser[interceptors]、ResourcesBeanDefinitionParser [resources]、ViewControllerBeanDefinitionParser [view-controller]等BeanDefinitionParser。通过” annotation-driven”可获得解析器为AnnotationDrivenBeanDefinitionParser。在AnnotationDrivenBeanDefinitionParser中将默认创建如下Bean定义：



### prepareBeanFactory(ConfigurableListableBeanFactory)



方法说明：配置BeanFactory的上下文特性，如ClassLoader、后置处理器等。

接口/类路径：org.springframework.context.support.AbstractApplicationContext

代码方法：prepareBeanFactory(ConfigurableListableBeanFactory)

代码解读

设置BeanFactory的ClassLoader(与ApplicationContext保持一致)、设置Bean表达式解析类(详细见[BeanExpressionResolver](#_BeanExpressionResolver))、设置资源编辑注册器(ResourceEditorRegistrar，详细见[PropertyEditorRegistrar](#_PropertyEditorRegistrar))、添加Bean后置处理器(ApplicationContextAwareProcessor，详细见[BeanPostProcessor](#_BeanPostProcessor))、添加忽略依赖接口ResourceLoaderAware、ApplicationEventPublisherAware、MessageSourceAware、ApplicationContextAware、EnvironmentAware.

BeanFactory注册可分解的依赖，代码逻辑如下：

beanFactory.registerResolvableDependency(BeanFactory.class, beanFactory);

beanFactory.registerResolvableDependency(ResourceLoader.class, this);

beanFactory.registerResolvableDependency(ApplicationEventPublisher.class, this);

beanFactory.registerResolvableDependency(ApplicationContext.class, this);

如果BeanFactory中包含了beanName为”loadTimeWeaver”的bean，需要添加bean后置处理器LoadTimeWeaverAwareProcessor和临时ClassLoader[ContextTypeMatchClassLoader]。

### postProcessBeanFactory(ConfigurableListableBeanFactory)

方法说明：配置BeanFactory的上下文特性，如ClassLoader、后置处理器等。

接口/类路径：org.springframework.web.context.support.AbstractRefreshableWebApplicationContext

代码方法：postProcessBeanFactory(ConfigurableListableBeanFactory)

源代码

beanFactory.addBeanPostProcessor(new ServletContextAwareProcessor(this.servletContext, this.servletConfig));

beanFactory.ignoreDependencyInterface(ServletContextAware.class);

beanFactory.ignoreDependencyInterface(ServletConfigAware.class);

WebApplicationContextUtils.registerWebApplicationScopes(beanFactory, this.servletContext);

WebApplicationContextUtils.registerEnvironmentBeans(beanFactory, this.servletContext, this.servletConfig);

代码解读

添加bean后置处理器ServletContextAwareProcessor，添加忽略依赖接口ServletContextAware、ServletConfigAware。

WebApplicationContextUtils.registerWebApplicationScopes(beanFactory, this.servletContext);// 添加request、session、globalSession和application类别的scope。

为BeanFactory添加可分解的依赖，如下：

beanFactory.registerResolvableDependency(ServletRequest.class, new RequestObjectFactory());

beanFactory.registerResolvableDependency(HttpSession.class, new SessionObjectFactory());

beanFactory.registerResolvableDependency(WebRequest.class, new WebRequestObjectFactory());

WebApplicationContextUtils.registerEnvironmentBeans(beanFactory, this.servletContext, this.servletConfig); // 注册web环境下的bean，这些bean包含ServletContext、ServletConfig、ServletContext对应的InitParameter和AttributeName.

### invokeBeanFactoryPostProcessors

方法说明：调用处理器工厂注册的bean上下文。

接口/类路径：org.springframework.context.support.AbstractApplicationContext

代码方法：invokeBeanFactoryPostProcessors

PostProcessorRegistrationDelegate.invokeBeanFactoryPostProcessors(beanFactory, getBeanFactoryPostProcessors());// 详细见[invokeBeanFactoryPostProcessors(ConfigurableListableBeanFactory, List<BeanFactoryPostProcessor>)](#_invokeBeanFactoryPostProcessors(Con)

#### invokeBeanFactoryPostProcessors(ConfigurableListableBeanFactory, List<BeanFactoryPostProcessor>)

// 实例化并调用所有的BeanFactoryPostProcessor

// 一、**如果BeanFactory类型为BeanDefinitionRegistry**

for (BeanFactoryPostProcessor postProcessor : beanFactoryPostProcessors) {

if (postProcessor instanceof BeanDefinitionRegistryPostProcessor) {

BeanDefinitionRegistryPostProcessor registryPostProcessor =

(BeanDefinitionRegistryPostProcessor) postProcessor;

registryPostProcessor.postProcessBeanDefinitionRegistry(registry);

registryPostProcessors.add(registryPostProcessor);

} else {

regularPostProcessors.add(postProcessor);

}

}

// 迭代上下文中已存在的BeanFactoryPostProcessor。如果BeanFactoryPostProcessor为

BeanDefinitionRegistryPostProcessor[ConfigurationClassPostProcessor]，需要在BeanFactory添加ImportAwareBeanPostProcessor和EnhancedConfigurationBeanPostProcessor的bean定义,并创建和验证基于@Configuration的classes.详细见[postProcessBeanDefinitionRegistry(BeanDefinitionRegistry)](#_postProcessBeanDefinitionRegistry(B)，将这些BeanFactoryPostProcessor加入到registryPostProcessors；否则加入regularPostProcessors中。

postProcessorNames=beanFactory.getBeanNamesForType(BeanDefinitionRegistryPostProcessor.class, true, false);

List<BeanDefinitionRegistryPostProcessor> orderedPostProcessors = new ArrayList<BeanDefinitionRegistryPostProcessor>();

for (String ppName : postProcessorNames) {

if (!processedBeans.contains(ppName) && beanFactory.isTypeMatch(ppName, Ordered.class)) {

orderedPostProcessors.add(beanFactory.getBean(ppName,BeanDefinitionRegistryPostProcessor.class));

processedBeans.add(ppName);

}

}

OrderComparator.sort(orderedPostProcessors);

registryPostProcessors.addAll(orderedPostProcessors);

invokeBeanDefinitionRegistryPostProcessors(orderedPostProcessors, registry);

// 获取BeanFactory中所有的Class类型为BeanDefinitionRegistryPostProcessor的BeanDefiition，并且这些BeanDefinitionRegistryPostProcessor实现了PriorityOrdered接口，对其进行实例化实例化。实例化后加入到priorityOrderedPostProcessors列表中并排序，调用invokeBeanDefinitionRegistryPostProcessors方法。在invokeBeanDefinitionRegistryPostProcessors方法中其实是迭代所有的priorityOrderedPostProcessors，并调用BeanDefinitionRegistryPostProcessor的postProcessBeanDefinitionRegistry方法，详细见[postProcessBeanDefinitionRegistry(BeanDefinitionRegistry)](#_postProcessBeanDefinitionRegistry(B)。

postProcessorNames=beanFactory.getBeanNamesForType(BeanDefinitionRegistryPostProcessor.class, true, false);

List<BeanDefinitionRegistryPostProcessor> orderedPostProcessors = new ArrayList<BeanDefinitionRegistryPostProcessor>();

for (String ppName : postProcessorNames) {

if (!processedBeans.contains(ppName) && beanFactory.isTypeMatch(ppName, Ordered.class)) {

orderedPostProcessors.add(beanFactory.getBean(ppName,BeanDefinitionRegistryPostProcessor.class));

processedBeans.add(ppName);

}

}

OrderComparator.sort(orderedPostProcessors);

registryPostProcessors.addAll(orderedPostProcessors);

invokeBeanDefinitionRegistryPostProcessors(orderedPostProcessors, registry);

// 与上一步类似，只是这里处理实现了Order接口的BeanDefinitionRegistryPostProcessor。

boolean reiterate = true;

while (reiterate) {

reiterate = false;

postProcessorNames=beanFactory.getBeanNamesForType(BeanDefinitionRegistryPostProcessor.class, true, false);

for (String ppName : postProcessorNames) {

if (!processedBeans.contains(ppName)) {

BeanDefinitionRegistryPostProcessor pp = beanFactory.getBean(ppName, BeanDefinitionRegistryPostProcessor.class);

registryPostProcessors.add(pp);

processedBeans.add(ppName);

pp.postProcessBeanDefinitionRegistry(registry);

reiterate = true;

}

}

}

// 处理其他类型的BeanDefinitionRegistryPostProcessor。

// 调用invokeBeanFactoryPostProcessors方法。

// 一、**如果BeanFactory类型不为BeanDefinitionRegistry**

调用invokeBeanFactoryPostProcessors方法。

List<BeanFactoryPostProcessor> priorityOrderedPostProcessors = new ArrayList<BeanFactoryPostProcessor>();

List<String> orderedPostProcessorNames = new ArrayList<String>();

List<String> nonOrderedPostProcessorNames = new ArrayList<String>();

for (String ppName : postProcessorNames) {

if (processedBeans.contains(ppName)) {

// skip - already processed in first phase above

} else if (beanFactory.isTypeMatch(ppName, PriorityOrdered.class)) {

priorityOrderedPostProcessors.add(beanFactory.getBean(ppName,BeanFactoryPostProcessor.class));

} else if (beanFactory.isTypeMatch(ppName, Ordered.class)) {

orderedPostProcessorNames.add(ppName);

} else {

nonOrderedPostProcessorNames.add(ppName);

}

}

// First, invoke the BeanFactoryPostProcessors that implement PriorityOrdered.

OrderComparator.sort(priorityOrderedPostProcessors);

invokeBeanFactoryPostProcessors(priorityOrderedPostProcessors, beanFactory);

// Next, invoke the BeanFactoryPostProcessors that implement Ordered.

List<BeanFactoryPostProcessor> orderedPostProcessors = new ArrayList<BeanFactoryPostProcessor>();

for (String postProcessorName : orderedPostProcessorNames) {

orderedPostProcessors.add(beanFactory.getBean(postProcessorName, BeanFactoryPostProcessor.class));

}

OrderComparator.sort(orderedPostProcessors);

invokeBeanFactoryPostProcessors(orderedPostProcessors, beanFactory);

// Finally, invoke all other BeanFactoryPostProcessors.

List<BeanFactoryPostProcessor> nonOrderedPostProcessors = new ArrayList<BeanFactoryPostProcessor>();

for (String postProcessorName : nonOrderedPostProcessorNames) {

nonOrderedPostProcessors.add(beanFactory.getBean(postProcessorName, BeanFactoryPostProcessor.class));

}

invokeBeanFactoryPostProcessors(nonOrderedPostProcessors, beanFactory);

// 此部分处理实现了BeanFactoryPostProcessor接口的类，最后也是要调用[postProcessBeanDefinitionRegistry(BeanDefinitionRegistry)](#_postProcessBeanDefinitionRegistry(B)方法。

#### postProcessBeanDefinitionRegistry(BeanDefinitionRegistry)

### registerBeanPostProcessors

// 注册并实例化BeanPostProcessors,并把BeanPostProcessors加入到BeanFactory中。需要对BeanPostProcessors进行排序处理，如下：

1、最开始实例化实现了PriorityOrdered接口的BeanPostProcessors，并将其加入到priorityOrderedPostProcessors中，并注册到BeanFactory中。

for (String ppName : postProcessorNames) {

if (beanFactory.isTypeMatch(ppName, PriorityOrdered.class)) {

BeanPostProcessor pp = beanFactory.getBean(ppName, BeanPostProcessor.class);//实例化

priorityOrderedPostProcessors.add(pp);

if (pp instanceof MergedBeanDefinitionPostProcessor) {

internalPostProcessors.add(pp);

}

} else if (beanFactory.isTypeMatch(ppName, Ordered.class)) {

orderedPostProcessorNames.add(ppName);

} else {

nonOrderedPostProcessorNames.add(ppName);

}

}

OrderComparator.sort(priorityOrderedPostProcessors);

registerBeanPostProcessors(beanFactory, priorityOrderedPostProcessors);

2、实例化实现了Ordered接口的BeanPostProcessor，并注册到BeanFactory中

List<BeanPostProcessor> orderedPostProcessors = new ArrayList<BeanPostProcessor>();

for (String ppName : orderedPostProcessorNames) {

BeanPostProcessor pp = beanFactory.getBean(ppName, BeanPostProcessor.class);

orderedPostProcessors.add(pp);

if (pp instanceof MergedBeanDefinitionPostProcessor) {

internalPostProcessors.add(pp);

}

}

OrderComparator.sort(orderedPostProcessors);

registerBeanPostProcessors(beanFactory, orderedPostProcessors);

3、实例化普通的BeanPostProcessor，并注册到BeanFactory中。

List<BeanPostProcessor> nonOrderedPostProcessors = new ArrayList<BeanPostProcessor>();

for (String ppName : nonOrderedPostProcessorNames) {

BeanPostProcessor pp = beanFactory.getBean(ppName, BeanPostProcessor.class);

nonOrderedPostProcessors.add(pp);

if (pp instanceof MergedBeanDefinitionPostProcessor) {

internalPostProcessors.add(pp);

}

}

registerBeanPostProcessors(beanFactory, nonOrderedPostProcessors);

4、对实现了MergedBeanDefinitionPostProcessor接口的BeanPostProcessor重新排序。

OrderComparator.sort(internalPostProcessors);

registerBeanPostProcessors(beanFactory, internalPostProcessors);

5、添加ApplicationListenerDetector

beanFactory.addBeanPostProcessor(new ApplicationListenerDetector(applicationContext));

### initMessageSource

源代码：

protected void initMessageSource() {

ConfigurableListableBeanFactory beanFactory = getBeanFactory();

if (beanFactory.containsLocalBean(MESSAGE\_SOURCE\_BEAN\_NAME)) {

this.messageSource = beanFactory.getBean(MESSAGE\_SOURCE\_BEAN\_NAME, MessageSource.class);

if (this.parent != null && this.messageSource instanceof HierarchicalMessageSource) {

HierarchicalMessageSource hms = (HierarchicalMessageSource) this.messageSource;

if (hms.getParentMessageSource() == null) { hms.setParentMessageSource(getInternalParentMessageSource());

}

}

} else {

DelegatingMessageSource dms = new DelegatingMessageSource();

dms.setParentMessageSource(getInternalParentMessageSource());

this.messageSource = dms;

beanFactory.registerSingleton(MESSAGE\_SOURCE\_BEAN\_NAME, this.messageSource);

}

}

代码解读

在BeanFactory中注册beanName为” messageSource”的Bean，如果有则将其设置到ApplicationContext的messageSource字段；否则单独创建DelegatingMessageSource实例，并设置到ApplicationContext的messageSource字段。详细见[MessageSource](#_MessageSource)。

### initApplicationEventMulticaster

在BeanFactory中注册beanName为” applicationEventMulticaster”的Bean，如果有则将其设置到ApplicationContext的applicationEventMulticaster字段；否则单独创建SimpleApplicationEventMulticaster实例，设置到applicationEventMulticaster字段。详细见[ApplicationEventMulticaster](#_ApplicationEventMulticaster)。

### onRefresh

该方法内部调用org.springframework.ui.context.support.UiApplicationContextUtils中的initThemeSource，此方法将判断xml中是否定义了beanName为” themeSource”的Bean，如果没有则创建名为ResourceBundleThemeSource的bean实例，并将其加入上下文ApplicationContext中。详细见

### registerListeners

为ApplicationEventMulticaster注册监听器(详细见[ApplicationListener](#_ApplicationListener))。

### finishBeanFactoryInitialization

源代码：org/springframework/context/support/AbstractApplicationContext.java

protected void finishBeanFactoryInitialization(ConfigurableListableBeanFactory beanFactory) {

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));

}

String[] weaverAwareNames = beanFactory.getBeanNamesForType(LoadTimeWeaverAware.class, false, false);

for (String weaverAwareName : weaverAwareNames) {

getBean(weaverAwareName);

}

beanFactory.setTempClassLoader(null);

beanFactory.freezeConfiguration();

beanFactory.preInstantiateSingletons();

}

源码解读：

beanFactory.freezeConfiguration();// 将beanDefinitionNames备份到BeanFactory中的frozenBeanDefinitionNames中。

beanFactory.preInstantiateSingletons();// 详细见[preInstantiateSingletons()](#_preInstantiateSingletons())

#### preInstantiateSingletons()

**public** **void** preInstantiateSingletons() **throws** BeansException {

**if** (**this**.logger.isDebugEnabled()) {

**this**.logger.debug("Pre-instantiating singletons in " + **this**);

}

List<String> beanNames;

**synchronized** (**this**.beanDefinitionMap) {

beanNames = **new** ArrayList<String>(**this**.beanDefinitionNames);

}

**for** (String beanName : beanNames) {

RootBeanDefinition bd = getMergedLocalBeanDefinition(beanName);

**if** (!bd.isAbstract() && bd.isSingleton() && !bd.isLazyInit()) {

**if** (isFactoryBean(beanName)) {

**final** FactoryBean<?> factory = (FactoryBean<?>) getBean(***FACTORY\_BEAN\_PREFIX*** + beanName);

**boolean** isEagerInit;

**if** (System.*getSecurityManager*() != **null** && factory **instanceof** SmartFactoryBean) {

isEagerInit = AccessController.*doPrivileged*(**new** PrivilegedAction<Boolean>() {

@Override

**public** Boolean run() {

**return** ((SmartFactoryBean<?>) factory).isEagerInit();

}

}, getAccessControlContext());

}

**else** {

isEagerInit = (factory **instanceof** SmartFactoryBean && ((SmartFactoryBean<?>) factory).isEagerInit());

}

**if** (isEagerInit) {

getBean(beanName);

}

}

**else** {

getBean(beanName);

}

}

}

}

接口/类路径：org.springframework.beans.factory.support.DefaultListableBeanFactory

代码方法：preInstantiateSingletons()

源码解读：

DefaultListableBeanFactory的beanDefinitionMap字段保存了所有的bean定义信息，beanDefinitionNames报错了所有bean的名称。

从if (!bd.isAbstract() && bd.isSingleton() && !bd.isLazyInit()) {//…}逻辑可以看出，此方法只实例化分抽象的&&单例bean&&非懒惰实例化bean。

如果定义的bean类型为SmartFactoryBean，调用其isEagerInit()方法判断是否需要初始化，如果isEagerInit()返回false表示不需要初始化。

接下来调用[doGetBean(String,Class<T>,Object[],boolean)](#_doGetBean(String,Class<T>,Object[],).

#### doGetBean(String,Class<T>,Object[],boolean)

接口/类路径：org.springframework.beans.factory.support.AbstractBeanFactory

代码方法：doGetBean(String,Class<T>,Object[],boolean)

代码解读：

final String beanName = transformedBeanName(name);// 处理别名问题，需要别名对应的源。

Object sharedInstance = getSingleton(beanName);

if (sharedInstance != null && args == null) {

bean = getObjectForBeanInstance(sharedInstance, name, beanName, null);// 详细见[getObjectForBeanInstance(Object, String, String, RootBeanDefinition)](#_getObjectForBeanInstance(Object,_St)。

} // 逐步从JVM缓存的singletonObjects[已实例化的单例对象缓存]、earlySingletonObjects[早期已实例化的单例对象缓存]、singletonFactories[单例工厂]中获取bean。如果从singletonFactories[单例工厂]中获取到了bean定义，需要将bean加入到earlySingletonObjects并从singletonFactories移除。

// 如果从JVM缓存中无法获取bean实例，进行实例化操作。

BeanFactory parentBeanFactory = getParentBeanFactory();

if (parentBeanFactory != null && !containsBeanDefinition(beanName)) {

String nameToLookup = originalBeanName(name);

if (args != null) {

return (T) parentBeanFactory.getBean(nameToLookup, args);

} else {

return parentBeanFactory.getBean(nameToLookup, requiredType);

}

}// 如果当前BeanFactory存在ParentBeanFactory，相当于去上一级IoC容器中获取bean实例

// 处理依赖Bean初始化。

final RootBeanDefinition mbd = getMergedLocalBeanDefinition(beanName);

checkMergedBeanDefinition(mbd, beanName, args);

String[] dependsOn = mbd.getDependsOn();

if (dependsOn != null) {

for (String dependsOnBean : dependsOn) {

if (isDependent(beanName, dependsOnBean)) {

throw new BeanCreationException(mbd.getResourceDescription(), beanName, "Circular depends-on relationship between '" + beanName + "' and '" + dependsOnBean + "'");

}

registerDependentBean(dependsOnBean, beanName);

getBean(dependsOnBean);

}

}

// 对BeanDefiition的实例话，需要分清楚三种情况，分别如下：

场景一：

if (mbd.isSingleton()) {

sharedInstance = getSingleton(beanName, new ObjectFactory<Object>() {

public Object getObject() throws BeansException { // 在getSingleton中回调此方法

try {

return createBean(beanName, mbd, args);

} catch (BeansException ex) {

destroySingleton(beanName);

throw ex;

}

}

});

bean = getObjectForBeanInstance(sharedInstance, name, beanName, mbd);

}

// 这里会调用getSingleton来获得单例bean实例，这里创建了一个内部ObjectFactory，这个ObjectFactory实现了getObject()， getSingleton将回调ObjectFactory的getObject()方法完成bean实例化。详细见[getSingleton(String,ObjectFactory<?>)](#_getSingleton(String,ObjectFactory<?)。

场景二：

if (mbd.isPrototype()) {

Object prototypeInstance = null;

try {

beforePrototypeCreation(beanName);

prototypeInstance = createBean(beanName, mbd, args);//请参考[createBean(String,RootBeanDefinition,Object[])](#_createBean(String,RootBeanDefinitio)

} finally {

afterPrototypeCreation(beanName);

}

bean = getObjectForBeanInstance(prototypeInstance, name, beanName, mbd);

}

场景三：

String scopeName = mbd.getScope();

final Scope scope = this.scopes.get(scopeName);

if (scope == null) {

throw new IllegalStateException("No Scope registered for scope '" + scopeName + "'");

}

try {

Object scopedInstance = scope.get(beanName, new ObjectFactory<Object>() {

public Object getObject() throws BeansException {

beforePrototypeCreation(beanName);

try {

return createBean(beanName, mbd, args);

} finally {

afterPrototypeCreation(beanName);

}

}

});

bean = getObjectForBeanInstance(scopedInstance, name, beanName, mbd);

} catch (IllegalStateException ex) {// …}

// 在[postProcessBeanFactory(ConfigurableListableBeanFactory)](#_postProcessBeanFactory(Configurable)方法中为IoC容器添加了RequestScope、SessionScope等Scope。然后从对应的类别属性中获取Bean缓存，如果没有获取则回调ObjectFactory的getObject()方法完成bean实例化。// 详细见[get()](#_get())

#### getObjectForBeanInstance(Object, String, String, RootBeanDefinition)

#### getSingleton(String,ObjectFactory<?>)

接口/类路径：org.springframework.beans.factory.support.DefaultSingletonBeanRegistry

代码方法：getSingleton

public Object getSingleton(String beanName, ObjectFactory<?> singletonFactory) {

synchronized (this.singletonObjects) {

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

if (singletonObject == null) {

if (this.singletonsCurrentlyInDestruction) {

throw new BeanCreationNotAllowedException(beanName,"...");

}

beforeSingletonCreation(beanName);

boolean recordSuppressedExceptions = (this.suppressedExceptions == null);

if (recordSuppressedExceptions) {

this.suppressedExceptions = new LinkedHashSet<Exception>();

try {

singletonObject = singletonFactory.getObject();

} catch (BeanCreationException ex) {

throw ex;

} finally {

afterSingletonCreation(beanName);

}

addSingleton(beanName, singletonObject);

}

return (singletonObject != NULL\_OBJECT ? singletonObject : null);

}

}

// 此段逻辑主要是为创建bean实例做准备，里面最重要的一段逻辑就是回调ObjectFactory中的getObject()方法。请参考[getObject()](#_getObject())

#### get()

接口/类路径：org.springframework.web.context.request.AbstractRequestAttributesScope

代码方法：get()

public Object get(String name, ObjectFactory<?> objectFactory) {

RequestAttributes attributes = RequestContextHolder.currentRequestAttributes();

Object scopedObject = attributes.getAttribute(name, getScope());

if (scopedObject == null) {

scopedObject = objectFactory.getObject();

attributes.setAttribute(name, scopedObject, getScope());

}

return scopedObject;

}

// 将RequestAttributes 保存在ThreadLocal中，然后再将单例bean存放在RequestAttributes。如果获取不到，则回调ObjectFactory的getObject()方法完成bean实例化。参考[getObject()](#_getObject())

#### getObject()

接口/类路径：ObjectFactory

代码方法：getObject()

在[get()](#_get())和[getSingleton(String,ObjectFactory<?>)](#_getSingleton(String,ObjectFactory<?)方法中都需要回调ObjectFactory的getObject()方法，此方法撒都没做，就是直接调用[createBean(String,RootBeanDefinition,Object[])](#_createBean(String,RootBeanDefinitio)完成bean实例化。

#### createBean(String,RootBeanDefinition,Object[])

接口/类路径：org.springframework.beans.factory.support.AbstractAutowireCapableBeanFactory

代码方法：createBean

protected Object createBean(final String beanName, final RootBeanDefinition mbd, final Object[] args) throws BeanCreationException {

resolveBeanClass(mbd, beanName); // 根据BeanDefiition的className获取bean的Class，保证ClassLoader与ApplicationContext的ClassLoader一致。

try {

mbd.prepareMethodOverrides(); // 处理方法重写

} catch (BeanDefinitionValidationException ex) {

//...

}

try {

Object bean = resolveBeforeInstantiation(beanName, mbd);

if (bean != null) {

return bean;

}

} catch (Throwable ex) {

// ...

}

Object beanInstance = doCreateBean(beanName, mbd, args);

return beanInstance;

}

Object bean = resolveBeforeInstantiation(beanName, mbd); // 给BeanPostProcessors机会返回一个代理对象，里面会涉及到AOP的处理，详细见[resolveBeforeInstantiation](#_resolveBeforeInstantiation).

Object beanInstance = doCreateBean(beanName, mbd, args);// 如果没有代理对象，则创建bean，详细见[doCreateBean(String, RootBeanDefinition,Object[])](#_doCreateBean(String,_RootBeanDefini)

#### resolveBeforeInstantiation

接口/类路径：org.springframework.beans.factory.support.AbstractAutowireCapableBeanFactory

代码方法：resolveBeforeInstantiation

protected Object resolveBeforeInstantiation(String beanName, RootBeanDefinition mbd) {

Object bean = null;

if (!Boolean.FALSE.equals(mbd.beforeInstantiationResolved)) {

// Make sure bean class is actually resolved at this point.

if (mbd.hasBeanClass() && !mbd.isSynthetic() && hasInstantiationAwareBeanPostProcessors()) {

bean = applyBeanPostProcessorsBeforeInstantiation(mbd.getBeanClass(), beanName);

if (bean != null) {

bean = applyBeanPostProcessorsAfterInitialization(bean, beanName);

}

}

mbd.beforeInstantiationResolved = (bean != null);

}

return bean;

}

// 详细见[Spring AOP](#_Spring_AOP)

// [InstantiationAwareBeanPostProcessor](#_InstantiationAwareBeanPostProcessor)

#### doCreateBean(String, RootBeanDefinition,Object[])

接口/类路径：org.springframework.beans.factory.support.AbstractAutowireCapableBeanFactory

代码方法：doCreateBean

**protected** Object doCreateBean(**final** String beanName, **final** RootBeanDefinition mbd, **final** Object[] args) {

// 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 != **null** ? instanceWrapper.getWrappedInstance() : **null**);

Class<?> beanType = (instanceWrapper != **null** ? instanceWrapper.getWrappedClass() : **null**);

// Allow post-processors to modify the merged bean definition.

**synchronized** (mbd.postProcessingLock) {

**if** (!mbd.postProcessed) {

applyMergedBeanDefinitionPostProcessors(mbd, beanType, beanName);

mbd.postProcessed = **true**;

}

}

**boolean** earlySingletonExposure = (mbd.isSingleton() && **this**.allowCircularReferences &&isSingletonCurrentlyInCreation(beanName));

**if** (earlySingletonExposure) {

addSingletonFactory(beanName, **new** ObjectFactory<Object>() {

@Override

**public** Object getObject() **throws** BeansException {

**return** getEarlyBeanReference(beanName, mbd, bean);

}

});

}

Object exposedObject = bean;

**try** {

populateBean(beanName, mbd, instanceWrapper);

**if** (exposedObject != **null**) {

exposedObject = initializeBean(beanName, exposedObject, mbd);

}

} **catch** (Throwable ex) {

**//…**

}

**if** (earlySingletonExposure) {

Object earlySingletonReference = getSingleton(beanName, **false**);

**if** (earlySingletonReference != **null**) {

**if** (exposedObject == bean) {

exposedObject = earlySingletonReference;

}**else** **if** (!**this**.allowRawInjectionDespiteWrapping && hasDependentBean(beanName)) { String[] dependentBeans = getDependentBeans(beanName);

Set<String> actualDependentBeans = **new** LinkedHashSet<String>(dependentBeans.length);

**for** (String dependentBean : dependentBeans) {

**if** (!removeSingletonIfCreatedForTypeCheckOnly(dependentBean)) {

actualDependentBeans.add(dependentBean);

}

}

**if** (!actualDependentBeans.isEmpty()) {

**// …**

}

}

}

**try** {

registerDisposableBeanIfNecessary(beanName, bean, mbd);

} **catch** (BeanDefinitionValidationException ex) {

**//…**

}

**return** exposedObject;

}

1、调用[createBeanInstance(String,RootBeanDefinition,Object[])](#_createBeanInstance(String,RootBeanD)实例化bean。

2、根据BeanDefiition中的postProcessed字段判断bean是否已经进行了后置处理，如果没有被后置处理过，则迭代BeanFactory中的MergedBeanDefinitionPostProcessor，然后调用MergedBeanDefinitionPostProcessor的postProcessMergedBeanDefinition方法后置处理相关bean。

3、调用[populateBean(String,Object,RootBeanDefinition)](#_populateBean(String,Object,RootBean)完成IoC的依赖注入。

4、调用[initializeBean(String,Object,RootBeanDefinition)](#_initializeBean(String,Object,RootBe)完成一些初始化工作。

5、如果有必要将bean注册为一次性bean，将其加入相关列表中。[registerDisposableBeanIfNecessary(String,Object,RootBeanDefinition)](#_registerDisposableBeanIfNecessary(S).

#### registerDisposableBeanIfNecessary(String,Object,RootBeanDefinition)

#### initializeBean(String,Object,RootBeanDefinition)

接口/类路径：org.springframework.beans.factory.support.AbstractAutowireCapableBeanFactory

代码方法：initializeBean

**protected** Object initializeBean(**final** String beanName, **final** Object bean, RootBeanDefinition mbd){

**if** (System.*getSecurityManager*() != **null**) {

AccessController.*doPrivileged*(**new** PrivilegedAction<Object>() {

@Override

**public** Object run() {

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;

}

1、如果bean的类型是Aware，这里需要处理一下逻辑：

**private** **void** invokeAwareMethods(**final** String beanName, **final** Object bean) {

**if** (bean **instanceof** Aware) {

**if** (bean **instanceof** BeanNameAware) {

((BeanNameAware) bean).setBeanName(beanName);

}

**if** (bean **instanceof** BeanClassLoaderAware) {

((BeanClassLoaderAware) bean).setBeanClassLoader(getBeanClassLoader());

}

**if** (bean **instanceof** BeanFactoryAware) {

((BeanFactoryAware) bean).setBeanFactory(AbstractAutowireCapableBeanFactory.**this**);

}

}

}

2、如果BeanDefiition为null，或者当前BeanDefiition不为合成bean，则需要获取BeanFactory中所有的BeanPostProcessor，并迭代它们，调用其postProcessAfterInitialization方法。

3、调用bean的实例化方法，这里需要分为一下两步处理。

3.1，如果bean实现了InitializingBean接口，则调用其afterPropertiesSet方法。

3.2，如果定义bean的时候，存在init-method，则通过发射调用其对应的方法。

4、如果BeanDefiition为null，或者当前BeanDefiition不为合成bean，则需要获取BeanFactory中所有的BeanPostProcessor，并迭代它们，调用其postProcessAfterInitialization。

#### populateBean(String,Object,RootBeanDefinition)

接口/类路径：org.springframework.beans.factory.support.AbstractAutowireCapableBeanFactory

代码方法：populateBean

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

PropertyValues pvs = mbd.getPropertyValues();

**if** (bw == **null**) {

**if** (!pvs.isEmpty()) {

**throw** **new** BeanCreationException(mbd.getResourceDescription(), beanName, "Cannot apply property values to null instance");

} **else** {// Skip property population phase for null instance.

**return**;

}

}

**boolean** continueWithPropertyPopulation = **true**;

**if** (!mbd.isSynthetic() && hasInstantiationAwareBeanPostProcessors()) {

**for** (BeanPostProcessor bp : getBeanPostProcessors()) {

**if** (bp **instanceof** InstantiationAwareBeanPostProcessor) {

InstantiationAwareBeanPostProcessor ibp = (InstantiationAwareBeanPostProcessor) bp;

**if** (!ibp.postProcessAfterInstantiation(bw.getWrappedInstance(), beanName)) {

continueWithPropertyPopulation = **false**;

**break**;

}

}

}

}

**if** (!continueWithPropertyPopulation) {

**return**;

}

**if** (mbd.getResolvedAutowireMode() == RootBeanDefinition.***AUTOWIRE\_BY\_NAME*** || mbd.getResolvedAutowireMode() == RootBeanDefinition.***AUTOWIRE\_BY\_TYPE***) {

MutablePropertyValues newPvs = **new** MutablePropertyValues(pvs);

// Add property values based on autowire by name if applicable.

**if** (mbd.getResolvedAutowireMode() == RootBeanDefinition.***AUTOWIRE\_BY\_NAME***) {

autowireByName(beanName, mbd, bw, newPvs);

}

// Add property values based on autowire by type if applicable.

**if** (mbd.getResolvedAutowireMode() == RootBeanDefinition.***AUTOWIRE\_BY\_TYPE***) {

autowireByType(beanName, mbd, bw, newPvs);

}

pvs = newPvs;

}

**boolean** hasInstAwareBpps = hasInstantiationAwareBeanPostProcessors();

**boolean** needsDepCheck = (mbd.getDependencyCheck() != RootBeanDefinition.***DEPENDENCY\_CHECK\_NONE***);

**if** (hasInstAwareBpps || needsDepCheck) {

PropertyDescriptor[] filteredPds = filterPropertyDescriptorsForDependencyCheck(bw, mbd.allowCaching);

**if** (hasInstAwareBpps) {

**for** (BeanPostProcessor bp : getBeanPostProcessors()) {

**if** (bp **instanceof** InstantiationAwareBeanPostProcessor) {

InstantiationAwareBeanPostProcessor ibp = (InstantiationAwareBeanPostProcessor) bp;

pvs = ibp.postProcessPropertyValues(pvs, filteredPds, bw.getWrappedInstance(), beanName);

**if** (pvs == **null**) {

**return**;

}

}

}

}

**if** (needsDepCheck) {

checkDependencies(beanName, mbd, filteredPds, pvs);

}

}

applyPropertyValues(beanName, mbd, bw, pvs);

}

1、判断BeanDefiition是否为合成bean，根据BeanDefiition中的synthetic字段判断，如果不是合成bean，则迭代BeanFactory中的所有InstantiationAwareBeanPostProcessor的postProcessAfterInstantiation方法。如果此方法返回false则返回。

2、根据BeanDefiition中的autowireMode字段，判断IoC依赖注入的自动装配的类型，如下是Spring定义的自动装配类型：

AUTOWIRE\_NO = 0;

AUTOWIRE\_BY\_NAME = 1;

AUTOWIRE\_BY\_TYPE = 2;

AUTOWIRE\_CONSTRUCTOR = 3;

在populateBean方法中需要处理按名称和类型自动状态的处理，详细见[autowireByName(String,AbstractBeanDefinition,BeanWrapper,MutablePropertyValues)](#_autowireByName(String,AbstractBeanD)和[autowireByType(String,AbstractBeanDefinition,BeanWrapper, MutablePropertyValues)](#_autowireByType(String,AbstractBeanD)

3、如果BeanFactory存在InstantiationAwareBeanPostProcessor和对应的bean需要进行依赖检查[BeanDefiition的dependencyCheck字段]则需要进行依赖处理。

3.1 如果存在InstantiationAwareBeanPostProcessor，则调用其上的postProcessPropertyValues方法处理依赖属性。

3.2 调用AbstractAutowireCapableBeanFactory的checkDependencies方法校验。

4、将属性值通过setter方法注入，详细见AbstractAutowireCapableBeanFactory的applyPropertyValues方法。注入的方式还是通过Java的发射机制，详细见BeanWrapperImpl的setPropertyValue(PropertyTokenHolder, PropertyValue)

#### autowireByName(String,AbstractBeanDefinition,BeanWrapper,MutablePropertyValues)

#### autowireByType(String,AbstractBeanDefinition,BeanWrapper, MutablePropertyValues)

#### createBeanInstance(String,RootBeanDefinition,Object[])

源代码文件：org/springframework/beans/factory/support/AbstractAutowireCapableBeanFactory.java

**protected** BeanWrapper createBeanInstance(String beanName, RootBeanDefinition mbd, Object[] args){

Class<?> beanClass = resolveBeanClass(mbd, beanName);

**if** (beanClass != **null** && !Modifier.*isPublic*(beanClass.getModifiers()) && !mbd.isNonPublicAccessAllowed()) {

**throw** **new** BeanCreationException(mbd.getResourceDescription(), beanName, "Bean class isn't public, and non-public access not allowed: " + beanClass.getName());

}

**if** (mbd.getFactoryMethodName() != **null**) {

**return** instantiateUsingFactoryMethod(beanName, mbd, args);

}

**boolean** resolved = **false**;

**boolean** autowireNecessary = **false**;

**if** (args == **null**) {

**synchronized** (mbd.constructorArgumentLock) {

**if** (mbd.resolvedConstructorOrFactoryMethod != **null**) {

resolved = **true**;

autowireNecessary = mbd.constructorArgumentsResolved;

}

}

}

**if** (resolved) {

**if** (autowireNecessary) {

**return** autowireConstructor(beanName, mbd, **null**, **null**);

} **else** {

**return** instantiateBean(beanName, mbd);

}

}

Constructor<?>[] ctors = determineConstructorsFromBeanPostProcessors(beanClass, beanName);

**if** (ctors != **null** || mbd.getResolvedAutowireMode() == RootBeanDefinition.***AUTOWIRE\_CONSTRUCTOR*** || mbd.hasConstructorArgumentValues() || !ObjectUtils.*isEmpty*(args)) {

**return** autowireConstructor(beanName, mbd, ctors, args);

}

**return** instantiateBean(beanName, mbd);

}

1、resolveBeanClass主要是获取到bean对应的Class信息，这里需要保证ClassLoader与BeanFactory的ClassLoader一致。

2、mbd.getFactoryMethodName()判断是否存在工厂方法，如果则存在工厂方法则使用工厂方法实例化bean，实例化的工作由[instantiateUsingFactoryMethod(String,RootBeanDefinition,Object[])](#_instantiateUsingFactoryMethod(Strin)完成。

3、除了工厂方法初始化Bean外，Spring还提供了根据构造器初始化bean，我们可以从上面的逻辑中看出，分别由如下两个方法完成构造器实例化bean。

3.1 根据参数自动装配构造器实例化bean，见[autowireConstructor(String,RootBeanDefinition,Constructor<?>[],Object[])](#_autowireConstructor(String,RootBean)

3.2 如果根据参数没有匹配到构造器，则根据默认构造器实例化bean，见[instantiateBean(String,RootBeanDefinition)](#_instantiateBean(String,RootBeanDefi)。

#### instantiateUsingFactoryMethod(String,RootBeanDefinition,Object[])

源代码文件：org/springframework/beans/factory/support/AbstractAutowireCapableBeanFactory.java

**protected** BeanWrapper instantiateUsingFactoryMethod(String beanName, RootBeanDefinition mbd, Object[] explicitArgs) {

**return** **new** ConstructorResolver(**this**).instantiateUsingFactoryMethod(beanName, mbd, explicitArgs);

}

// 从上面的逻辑可以看出，instantiateUsingFactoryMethod将所有的逻辑都委派给ConstructorResolver的instantiateUsingFactoryMethod方法完成。

// 我们可以分解一下ConstructorResolver中instantiateUsingFactoryMethod方法的处理逻辑。

1、调用AbstractBeanFactory.initBeanWrapper方法为BeanWrapper注册ConversionService和初始化自定义编辑器。

2、工厂方法判断逻辑：

String factoryBeanName = mbd.getFactoryBeanName();

**if** (factoryBeanName != **null**) {

**if** (factoryBeanName.equals(beanName)) {

**throw** **new** BeanDefinitionStoreException(mbd.getResourceDescription(), beanName, "factory-bean reference points back to the same bean definition");

}

factoryBean = **this**.beanFactory.getBean(factoryBeanName);

**if** (factoryBean == **null**) {

**throw** **new** BeanCreationException(mbd.getResourceDescription(), beanName, "factory-bean '" + factoryBeanName + "' (or a BeanPostProcessor involved) returned null");

}

factoryClass = factoryBean.getClass();

isStatic = **false**;

} **else** {

**if** (!mbd.hasBeanClass()) {

**throw** **new** BeanDefinitionStoreException(mbd.getResourceDescription(), beanName, "bean definition declares neither a bean class nor a factory-bean reference");

}

factoryBean = **null**;

factoryClass = mbd.getBeanClass();

isStatic = **true**;

}

从上面来看，Spring支持静态工厂方法和工厂方法两种形式。

2.1 工厂方法需要其他类提供，不能为本身类提供。

2.2 静态工厂方法为本身类提供的static方法。

3、根据配置参数自动匹配最合适的工厂方法，然后采用java反射机制调用工厂方法，有工厂方法完成bean的实例化。、

反射调用逻辑见SimpleInstantiationStrategy.instantiate方法。

#### autowireConstructor(String,RootBeanDefinition,Constructor<?>[],Object[])

代码源文件：org.springframework.beans.factory.support.ConstructorResolver

代码方法：autowireConstructor

自动匹配构造器，最后会根据bean class的方法覆盖情况采用反射或cglib形式初始化bean。

#### instantiateBean(String,RootBeanDefinition)

代码源文件：org/springframework/beans/factory/support/AbstractAutowireCapableBeanFactory.java

**protected** BeanWrapper instantiateBean(**final** String beanName, **final** RootBeanDefinition mbd) {

**try** {

Object beanInstance;

**final** BeanFactory parent = **this**;

**if** (System.*getSecurityManager*() != **null**) {

beanInstance = AccessController.*doPrivileged*(**new** PrivilegedAction<Object>() {

@Override

**public** Object run() {

**return** getInstantiationStrategy().instantiate(mbd, beanName, parent);

}

}, getAccessControlContext());

} **else** {

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

}

BeanWrapper bw = **new** BeanWrapperImpl(beanInstance);

initBeanWrapper(bw);

**return** bw;

} **catch** (Throwable ex) {

**throw** **new** BeanCreationException(mbd.getResourceDescription(), beanName, "Instantiation of bean failed", ex);

}

}

此方法分为两步，如下：

1、将实例化工作交给SimpleInstantiationStrategy的[instantiate(RootBeanDefinition,String,BeanFactory)](#_instantiate(RootBeanDefinition,Stri)

2、将实例化后的bean封装成BeanWrapper。

#### instantiate(RootBeanDefinition,String,BeanFactory)

代码源文件：org/springframework/beans/factory/support/SimpleInstantiationStrategy.java

**public** Object instantiate(RootBeanDefinition beanDefinition,String beanName,BeanFactory owner){

**if** (beanDefinition.getMethodOverrides().isEmpty()) {

Constructor<?> constructorToUse;

**synchronized** (beanDefinition.constructorArgumentLock) {

constructorToUse = (Constructor<?>) beanDefinition.resolvedConstructorOrFactoryMethod;

**if** (constructorToUse == **null**) {

**final** Class<?> clazz = beanDefinition.getBeanClass();

**if** (clazz.isInterface()) {

**throw** **new** BeanInstantiationException(clazz, "Specified class is an interface");

}

**try** {

**if** (System.*getSecurityManager*() != **null**) {

constructorToUse = AccessController.*doPrivileged*(**new** PrivilegedExceptionAction<Constructor<?>>() {

@Override

**public** Constructor<?> run() **throws** Exception {

**return** clazz.getDeclaredConstructor((Class[]) **null**);

}

});

} **else** {

constructorToUse = clazz.getDeclaredConstructor((Class[]) **null**);

}

beanDefinition.resolvedConstructorOrFactoryMethod = constructorToUse;

} **catch** (Exception ex) {

**throw** **new** BeanInstantiationException(clazz, "No default constructor found", ex);

}

}

}

**return** BeanUtils.*instantiateClass*(constructorToUse);

} **else** {

// Must generate CGLIB subclass.

**return** instantiateWithMethodInjection(beanDefinition, beanName, owner);

}

}

从此段逻辑看以看出，如果bean对应的类不存在override方法，则采用默认的无参构造器实例化；否则采用cglib实例化bean。

### finishRefresh

1. initLifecycleProcessor

protected void initLifecycleProcessor() {

ConfigurableListableBeanFactory beanFactory = getBeanFactory();

if (beanFactory.containsLocalBean(LIFECYCLE\_PROCESSOR\_BEAN\_NAME)) {

this.lifecycleProcessor = beanFactory.getBean(LIFECYCLE\_PROCESSOR\_BEAN\_NAME, LifecycleProcessor.class);

} else {

DefaultLifecycleProcessor defaultProcessor = new DefaultLifecycleProcessor();

defaultProcessor.setBeanFactory(beanFactory);

this.lifecycleProcessor = defaultProcessor;

beanFactory.registerSingleton(LIFECYCLE\_PROCESSOR\_BEAN\_NAME, this.lifecycleProcessor);

}

}

// 注册生命周期处理器，详细见[Lifecycle](#_Lifecycle)

2、getLifecycleProcessor().onRefresh(); 启动满足条件的Lifecycle

private void startBeans(boolean autoStartupOnly) {

Map<String, Lifecycle> lifecycleBeans = getLifecycleBeans();

Map<Integer, LifecycleGroup> phases = new HashMap<Integer, LifecycleGroup>();

for (Map.Entry<String, ? extends Lifecycle> entry : lifecycleBeans.entrySet()) {

Lifecycle bean = entry.getValue();

if (!autoStartupOnly || (bean instanceof SmartLifecycle && ((SmartLifecycle) bean).isAutoStartup())) {

int phase = getPhase(bean);

LifecycleGroup group = phases.get(phase);

if (group == null) {

group = new LifecycleGroup(phase, this.timeoutPerShutdownPhase, lifecycleBeans, autoStartupOnly);

phases.put(phase, group);

}

group.add(entry.getKey(), bean);

}

}

if (phases.size() > 0) {

List<Integer> keys = new ArrayList<Integer>(phases.keySet());

Collections.sort(keys);

for (Integer key : keys) {

phases.get(key).start();

}

}}

# Spring MVC

## 简介

一、概述

1、Spring MVC是围绕DispatcherServlet设计。DispatcherServlet的作用是将请求分发到不同的处理器。Spring的web框架包括可配置的处理器映射（handler mapping）、视图解析（view resolver）、本地化解析（local resolver）、主图解析（theme resolver）以及对文件上传的支持。

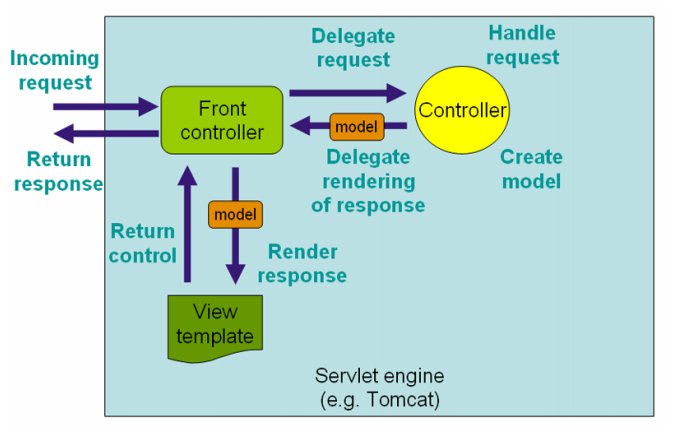
    2、Spring的web框架中缺省的处理器是Controller接口，仅包含ModelAndView handleRequest(request,response)方法。可以通过实现这个接口来创建自己的控制器，但是更推荐继承Spring提供的一系列控制器，比如AbstractController、AbstractCommandController和SimpleFormController。

    3、Spring Web MVC允许使用任何对象作为命令对象（或表单对象）-不必实现某个特定于框架的接口或某个基类继承。Spring的数据绑定相当灵活，例如，他认为类型不匹配这样的错误应该是应用级的验证错误,而不是系统错误。

二、DispatcherServlet

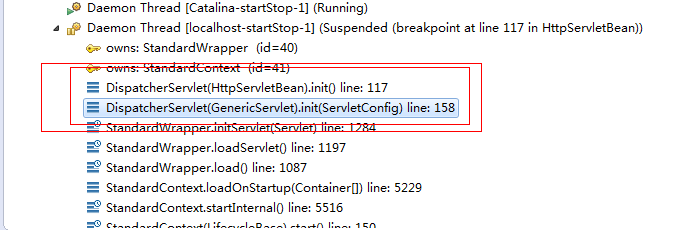
1、与其它web MVC框架一样，Spring的web MVC框架是一个请求驱动的web框架，其设计围绕一个中心的servlet进行，它能将请求分发给控制器，并提供其他功能帮助web应用开发。然而，Spring的DispatcherServlet所做的不仅仅是这些，它和Spring的IOC容器完全集成在一起，从而允许使用Spring的其它功能。

    2、Spring Web MVC请求处理流程



## 源码分析

Spring MVC是围绕DispatcherServlet设计，我们可以从[DispatcherServlet](#_DispatcherServlet)看到其是继承GenericServlet而来，Tomcat在启动时，我们可以通过Debug看出堆栈方法的调用层次关系，如下：

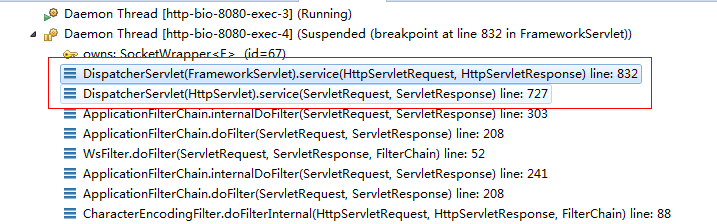


所以我们从HttpServletBean的init()方法开始分析SpringMVC的源码。会在初始化ApplicationContext，详细见[实例化Spring容器](#_实例化Spring容器)。在实例化Spring容器的时候，如果上下文配置文件applicationContext.xml[举例]定义了mvc前缀的标签元素，如<mvc:annotation-driven/>、<mvc:resources/>、<default-servlet-handler/>、<interceptors/>、<view-controller/>，在上下文配置解析的时候[详细见[parseCustomElement(Element,BeanDefinition)](#_parseCustomElement(Element,BeanDefi)]。在” /spring-framework/spring-webmvc/src/main/resources/META-INF/spring.handlers”中定义mvc前缀对应的namespace与handler的映射关系，为” http://www.springframework.org/schema/mvc=org.springframework.web.servlet.config.MvcNamespaceHandler”.在[parseCustomElement(Element,BeanDefinition)](#_parseCustomElement(Element,BeanDefi)中有如下逻辑：

NamespaceHandler handler = this.readerContext.getNamespaceHandlerResolver().resolve(namespaceUri);

在resolve中会调用NamespaceHandler的init()方法，详细见[NamespaceHandler.init()/parse()](#_NamespaceHandler.init()).

初始化WebApplicationContext后，当客户端发送http请求到服务器(如Tomcat)，通过下图的程序方法调用图可以看出，处理请求首先到HttpServlet的service(ServletRequest, ServletResponse)方法，然后将Request和Response做一个下转型，转换为HttpServletRequest和HttpServletResponse，然后交给HttpServlet子类FrameworkServlet的service(HttpServletRequest, HttpServletResponse)方法完成http请求处理。



### NamespaceHandler.init()/parse()

接口/类路径： org.springframework.web.servlet.config.MvcNamespaceHandler

代码方法：init

public void init() {

registerBeanDefinitionParser("annotation-driven", new AnnotationDrivenBeanDefinitionParser());

registerBeanDefinitionParser("default-servlet-handler", new DefaultServletHandlerBeanDefinitionParser());

registerBeanDefinitionParser("interceptors", new InterceptorsBeanDefinitionParser());

registerBeanDefinitionParser("resources", new ResourcesBeanDefinitionParser());

registerBeanDefinitionParser("view-controller", new ViewControllerBeanDefinitionParser());

}

我们可以从[NamespaceHandler](#_NamespaceHandler)中看出MvcNamespaceHandler继承了NamespaceHandlerSupport类，重写了init()方法，在init方法定义了解析mvc前缀不同元素的BeanDefinitionParser[详细见[BeanDefinitionParser](#_BeanDefinitionParser)].

注册BeanDefinitionParser后，NamespaceHandler会根据元素的namespace获取对应的BeanDefinitionParser的parse()方法解析xml元素。

#### <mvc:annotation-driven/>

<mvc:annotation-driven>对应的解析类AnnotationDrivenBeanDefinitionParser。

1、

Object source = parserContext.extractSource(element);

CompositeComponentDefinition compDefinition = new CompositeComponentDefinition(element.getTagName(), source);

parserContext.pushContainingComponent(compDefinition);

// 创建name为” mvc:annotation-driven”CompositeComponentDefinition[详细见[ComponentDefinition](#_ComponentDefinition)]实例，并将实例设置到ParserContext中的containingComponents栈中。

2、

RuntimeBeanReference contentNegotiationManager = getContentNegotiationManager(element, source, parserContext); // getContentNegotiationManager逻辑如下：

private RuntimeBeanReference getContentNegotiationManager(Element element, Object source, ParserContext parserContext) {

RuntimeBeanReference contentNegotiationManagerRef;

if (element.hasAttribute("content-negotiation-manager")) {

contentNegotiationManagerRef = new RuntimeBeanReference(element.getAttribute("content-negotiation-manager"));

} else {

RootBeanDefinition factoryBeanDef = new RootBeanDefinition(ContentNegotiationManagerFactoryBean.class);

factoryBeanDef.setSource(source);

factoryBeanDef.setRole(BeanDefinition.ROLE\_INFRASTRUCTURE);

factoryBeanDef.getPropertyValues().add("mediaTypes", getDefaultMediaTypes());

String beanName = "mvcContentNegotiationManager";

parserContext.getReaderContext().getRegistry().registerBeanDefinition(beanName , factoryBeanDef);

parserContext.registerComponent(new BeanComponentDefinition(factoryBeanDef, beanName));

contentNegotiationManagerRef = new RuntimeBeanReference(beanName);

}

return contentNegotiationManagerRef;

}

// 需要判断”mvc:annotation-driven”是否包含”content-negotiation-manager”属性，如果包含则创建对应属性值的RuntimeBeanReference；如果不包含则创建名为”mvcContentNegotiationManager”的RootBeanDefinition、BeanComponentDefinition和RuntimeBeanReference。

// PS : ContentNegotiationManager详细见[ContentNegotiationManager](#_ContentNegotiationManager)

3、

RootBeanDefinition handlerMappingDef = new RootBeanDefinition(RequestMappingHandlerMapping.class);

handlerMappingDef.setSource(source);

handlerMappingDef.setRole(BeanDefinition.ROLE\_INFRASTRUCTURE);

handlerMappingDef.getPropertyValues().add("order", 0);

handlerMappingDef.getPropertyValues().add("contentNegotiationManager",contentNegotiationManager);

String methodMappingName = parserContext.getReaderContext().registerWithGeneratedName(handlerMappingDef);

// 注册RequestMappingHandlerMapping对应的BeanDefiition，名字自动生成。并加入order属性进行排序，加入contentNegotiationManager属性引用” mvcContentNegotiationManager”。

// PS : RequestMappingHandlerMapping详细见[HandlerMapping](#_HandlerMapping)

4、

if (element.hasAttribute("enable-matrix-variables")) {

Boolean enableMatrixVariables = Boolean.valueOf(element.getAttribute("enable-matrix-variables"));

handlerMappingDef.getPropertyValues().add("removeSemicolonContent", !enableMatrixVariables);

} else if (element.hasAttribute("enableMatrixVariables")) {

Boolean enableMatrixVariables = Boolean.valueOf(element.getAttribute("enableMatrixVariables"));

handlerMappingDef.getPropertyValues().add("removeSemicolonContent", !enableMatrixVariables);

}

// 判断是否启动矩阵变量。

5、

configurePathMatchingProperties(handlerMappingDef, element);// 配置路径匹配属性

// 举例配置如下：

<mvc:path-matching path-helper="" registered-suffixes-only="true" suffix-pattern="true" trailing-slash="true" path-matcher=""/>

说明,需要在”mvc:annotation-driven”中添加”mvc:path-matching”标签元素，path-helper和path-matcher配置帮助类和匹配类对应的beanName.

6、

RuntimeBeanReference conversionService = getConversionService(element, source, parserContext);

getConversionService方法的实现逻辑如下：

private RuntimeBeanReference getConversionService(Element element, Object source, ParserContext parserContext) {

RuntimeBeanReference conversionServiceRef;

if (element.hasAttribute("conversion-service")) {

conversionServiceRef = new RuntimeBeanReference(element.getAttribute("conversion-service"));

} else {

RootBeanDefinition conversionDef = new RootBeanDefinition(FormattingConversionServiceFactoryBean.class);

conversionDef.setSource(source);

conversionDef.setRole(BeanDefinition.ROLE\_INFRASTRUCTURE);

String conversionName = parserContext.getReaderContext().registerWithGeneratedName(conversionDef);

parserContext.registerComponent(new BeanComponentDefinition(conversionDef, conversionName));

conversionServiceRef = new RuntimeBeanReference(conversionName);

}

return conversionServiceRef;

}

// 示例配置

<mvc:annotation-driven conversion-service="beanName">//…</mvc:annotation-driven>

如果<mvc:annotation-driven />标签元素下有”conversion-service”属性，此属性配置的值为ConversionService对应的BeanName;如果没有”conversion-service”属性,则定义FormattingConversionServiceFactoryBean对应的BeanDefiition.并创建对应的RuntimeBeanReference。

// PS : ConversionService详细见[ConversionService](#_ConversionService)。

7、 RuntimeBeanReference validator = getValidator(element, source, parserContext);

实现逻辑如下：

private RuntimeBeanReference getValidator(Element element, Object source, ParserContext parserContext) {

if (element.hasAttribute("validator")) {

return new RuntimeBeanReference(element.getAttribute("validator"));

} else if (javaxValidationPresent) {

RootBeanDefinition validatorDef = new RootBeanDefinition("org.springframework.validation.beanvalidation.OptionalValidatorFactoryBean");

validatorDef.setSource(source);

validatorDef.setRole(BeanDefinition.ROLE\_INFRASTRUCTURE);

String validatorName = parserContext.getReaderContext().registerWithGeneratedName(validatorDef);

parserContext.registerComponent(new BeanComponentDefinition(validatorDef, validatorName));

return new RuntimeBeanReference(validatorName);

} else {

return null;

}

// 如果”mvc:annotation-driven”配置了validator属性，则创建名为{ validatorBeanName }的引用，举例配置如下

<mvc:annotation-driven enable-matrix-variables="true" validator="{validatorBeanName}"/>

// 否则判断javaxValidationPresent,

javaxValidationPresent = ClassUtils.isPresent("javax.validation.Validator", AnnotationDrivenBeanDefinitionParser.class.getClassLoader())

8、RuntimeBeanReference messageCodesResolver = getMessageCodesResolver(element);

实现逻辑如下：

private RuntimeBeanReference getMessageCodesResolver(Element element) {

if (element.hasAttribute("message-codes-resolver")) {

return new RuntimeBeanReference(element.getAttribute("message-codes-resolver"));

} else {

return null;

}

}

// 如果”mvc:annotation-driven”配置了message-codes-resolver属性，则创建名为{ messageCodesResolverBeanName }的引用，举例配置如下

<mvc:annotation-driven enable-matrix-variables="true" message-codes-resolver ="{messageCodesResolverBeanName}"/>

MessageCodesResolver详细信息见[MessageCodesResolver](#_MessageCodesResolver)

9、

RootBeanDefinition bindingDef = new RootBeanDefinition(ConfigurableWebBindingInitializer.class);

bindingDef.setSource(source);

bindingDef.setRole(BeanDefinition.ROLE\_INFRASTRUCTURE);

bindingDef.getPropertyValues().add("conversionService", conversionService);

bindingDef.getPropertyValues().add("validator", validator);

bindingDef.getPropertyValues().add("messageCodesResolver", messageCodesResolver);

添加ConfigurableWebBindingInitializer类的BeanDefiition，并为其注册ConversionService、Validator和MessageCodesResolver属性引用。

10、ManagedList<?> messageConverters = getMessageConverters(element, source, parserContext);

getMessageConverters方法的实现逻辑如下：

Element convertersElement = DomUtils.getChildElementByTagName(element, "message-converters");

ManagedList<? super Object> messageConverters = new ManagedList<Object>();

if (convertersElement != null) {

messageConverters.setSource(source);

for (Element beanElement : DomUtils.getChildElementsByTagName(convertersElement, "bean", "ref")) {

Object object = parserContext.getDelegate().parsePropertySubElement(beanElement, null);

messageConverters.add(object);

}

}

// 如果”<mvc:annotation-driven/>”标签元素下存在”<mvc:message-converters/>”自标签元素，则迭代”<mvc:message-converters>”下的<bean/>和<ref/>标签元素,例如：

<mvc:message-converters>

<bean class="org.springframework.http.converter.json.MappingJackson2HttpMessageConverter">

<property name="supportedMediaTypes">

<list>

<value>application/json; charset=UTF-8</value>

</list>

</property>

</bean>

</mvc:message-converters>

// 如果”<mvc:annotation-driven/>”标签元素下不存在”<mvc:message-converters/>”标签元素或者” <mvc:message-converters/>”标签元素”register-defaults”的值为true.则添加ByteArrayHttpMessageConverter、StringHttpMessageConverter、ResourceHttpMessageConverter、SourceHttpMessageConverter、AllEncompassingFormHttpMessageConverter到messageConverters中。

除了默认的MessageConverter外，还会根据相关条件添加MessageConverter，如下

if (romePresent) {

messageConverters.add(createConverterDefinition(AtomFeedHttpMessageConverter.class, source));

messageConverters.add(createConverterDefinition(RssChannelHttpMessageConverter.class, source));

}

if (jaxb2Present) {

messageConverters.add(createConverterDefinition(Jaxb2RootElementHttpMessageConverter.class, source));

}

if (jackson2Present) {

messageConverters.add(createConverterDefinition(MappingJackson2HttpMessageConverter.class, source));

} else if (jacksonPresent) {

messageConverters.add(createConverterDefinition(

org.springframework.http.converter.json.MappingJacksonHttpMessageConverter.class, source));

}

MessageConverters详细信息见[HttpMessageConverter](#_HttpMessageConverter)。

11、ManagedList<?> argumentResolvers = getArgumentResolvers(element, parserContext);

getArgumentResolvers方法的实现逻辑

private ManagedList<?> getArgumentResolvers(Element element, ParserContext parserContext) {

Element resolversElement = DomUtils.getChildElementByTagName(element, "argument-resolvers");

if (resolversElement != null) {

ManagedList<BeanDefinitionHolder> argumentResolvers = extractBeanSubElements(resolversElement, parserContext);

return wrapWebArgumentResolverBeanDefs(argumentResolvers, parserContext);

}

return null;

}

// 判断”mvc:annotation-driven”是否存在”argument-resolvers”自标签元素，如果促在，则调用extractBeanSubElements方法获取其下面的”<bean/>”标签，根据标签内容构建BeanDefinitionHolder。

<mvc:annotation-driven>

<mvc:argument-resolvers>

<bean></bean>

</mvc:argument-resolvers>

</mvc:annotation-driven>

// wrapWebArgumentResolverBeanDefs接口实现如下：

ManagedList<BeanDefinitionHolder> result = new ManagedList<BeanDefinitionHolder>();

for (BeanDefinitionHolder beanDef : beanDefs) {

String className = beanDef.getBeanDefinition().getBeanClassName();

Class<?> clazz = ClassUtils.resolveClassName(className, parserContext.getReaderContext().getBeanClassLoader());

if (WebArgumentResolver.class.isAssignableFrom(clazz)) {

RootBeanDefinition adapter = new RootBeanDefinition(ServletWebArgumentResolverAdapter.class);

adapter.getConstructorArgumentValues().addIndexedArgumentValue(0, beanDef);

result.add(new BeanDefinitionHolder(adapter, beanDef.getBeanName() + "Adapter"));

} else {

result.add(beanDef);

}

}

return result;

// 如果<bean>定义的class实现了WebArgumentResolver接口，需要构建ServletWebArgumentResolverAdapter对应的BeanDefiition，并将WebArgumentResolver加入到ServletWebArgumentResolverAdapter的constructorArgumentValues字段；

// ServletWebArgumentResolverAdapter最终实现了HandlerMethodArgumentResolver接口，HandlerMethodArgumentResolver详细信息见[HandlerMethodArgumentResolver](#_HandlerMethodArgumentResolver)。

12、ManagedList<?> returnValueHandlers = getReturnValueHandlers(element, parserContext);

getReturnValueHandlers方法的实现逻辑如下：

Element handlersElement = DomUtils.getChildElementByTagName(element, "return-value-handlers");

if (handlersElement != null) {

return extractBeanSubElements(handlersElement, parserContext);

}

return null;

// 如果” mvc:annotation-driven”标签元素下存在” <mvc:return-value-handlers/>”子标签元素，将其下面所有<bean/>子元素便签加载出来，并构建对应的BeanDefiition。

13、String asyncTimeout = getAsyncTimeout(element);

getAsyncTimeout实现逻辑如下，

Element asyncElement = DomUtils.getChildElementByTagName(element, "async-support");

return (asyncElement != null) ? asyncElement.getAttribute("default-timeout") : null;

//如果” mvc:annotation-driven”标签元素下存在”async-support”子标签元素，获取其default-timeout属性值。

14、RuntimeBeanReference asyncExecutor = getAsyncExecutor(element);

配置异步执行器

<mvc:async-support default-timeout="12" task-executor="{taskExecutorBeanName}">

</mvc:async-support>

15、

ManagedList<?> callableInterceptors = getCallableInterceptors(element, source, parserContext);

ManagedList<?> deferredResultInterceptors = getDeferredResultInterceptors(element, source, parserContext);

配置回调拦截器和延期结果拦截器

<mvc:async-support default-timeout="12" task-executor="taskExecutorBeanName">

<mvc:callable-interceptors>

<bean></bean>

</mvc:callable-interceptors>

<mvc:deferred-result-interceptors>

<bean></bean>

</mvc:deferred-result-interceptors>

</mvc:async-support>

16、构建RequestMappingHandlerAdapter，并将ContentNegotiationManager、ConfigurableWebBindingInitializer和MessageConverters作为属性注入RequestMappingHandlerAdapter(详细见[HandlerAdapter](#_HandlerAdapter))中。

RootBeanDefinition handlerAdapterDef = new RootBeanDefinition(RequestMappingHandlerAdapter.class);

handlerAdapterDef.setSource(source);

handlerAdapterDef.setRole(BeanDefinition.ROLE\_INFRASTRUCTURE);

handlerAdapterDef.getPropertyValues().add("contentNegotiationManager", contentNegotiationManager);

handlerAdapterDef.getPropertyValues().add("webBindingInitializer", bindingDef);

handlerAdapterDef.getPropertyValues().add("messageConverters", messageConverters);

if (element.hasAttribute("ignore-default-model-on-redirect")) {

Boolean ignoreDefaultModel = Boolean.valueOf(element.getAttribute("ignore-default-model-on-redirect"));

handlerAdapterDef.getPropertyValues().add("ignoreDefaultModelOnRedirect", ignoreDefaultModel);

} else if (element.hasAttribute("ignoreDefaultModelOnRedirect")) {

// "ignoreDefaultModelOnRedirect" spelling is deprecated

Boolean ignoreDefaultModel = Boolean.valueOf(element.getAttribute("ignoreDefaultModelOnRedirect"));

handlerAdapterDef.getPropertyValues().add("ignoreDefaultModelOnRedirect", ignoreDefaultModel);

}

// 判断是否” annotation-driven”是否存在”ignore-default-model-on-redirect”属性，有的话就将其注入到RequestMappingHandlerAdapter中的“ignoreDefaultModelOnRedirect”;没有则判断是否包含” ignoreDefaultModelOnRedirect”属性，有的话就将其注入到RequestMappingHandlerAdapter中的“ignoreDefaultModelOnRedirect”

17、

if (argumentResolvers != null) {

handlerAdapterDef.getPropertyValues().add("customArgumentResolvers", argumentResolvers);

}

if (returnValueHandlers != null) {

handlerAdapterDef.getPropertyValues().add("customReturnValueHandlers", returnValueHandlers);

}

if (asyncTimeout != null) {

handlerAdapterDef.getPropertyValues().add("asyncRequestTimeout", asyncTimeout);

}

if (asyncExecutor != null) {

handlerAdapterDef.getPropertyValues().add("taskExecutor", asyncExecutor);

}

handlerAdapterDef.getPropertyValues().add("callableInterceptors", callableInterceptors);

handlerAdapterDef.getPropertyValues().add("deferredResultInterceptors", deferredResultInterceptors);

String handlerAdapterName = parserContext.getReaderContext().registerWithGeneratedName(handlerAdapterDef);

// 注入customArgumentResolvers、asyncRequestTimeout、asyncRequestTimeout、taskExecutor、callableInterceptors和deferredResultInterceptors属性。并将RequestMappingHandlerAdapter加入到BeanFactory中。

18、

String uriCompContribName = MvcUriComponentsBuilder.MVC\_URI\_COMPONENTS\_CONTRIBUTOR\_BEAN\_NAME;

RootBeanDefinition uriCompContribDef = new RootBeanDefinition(CompositeUriComponentsContributorFactoryBean.class);

uriCompContribDef.setSource(source);

uriCompContribDef.getPropertyValues().addPropertyValue("handlerAdapter", handlerAdapterDef);

uriCompContribDef.getPropertyValues().addPropertyValue("conversionService", conversionService);

parserContext.getReaderContext().getRegistry().registerBeanDefinition(uriCompContribName, uriCompContribDef);

// 构建CompositeUriComponentsContributorFactoryBean，beanName为” mvcUriComponentsContributor” ，将RequestMappingHandlerAdapter和ConversionService分别注入其名为” handlerAdapter”和” conversionService”的属性中。并将其加入到BeanFactory中。

19、

RootBeanDefinition csInterceptorDef = new RootBeanDefinition(ConversionServiceExposingInterceptor.class);

csInterceptorDef.setSource(source);

csInterceptorDef.getConstructorArgumentValues().addIndexedArgumentValue(0, conversionService);

RootBeanDefinition mappedCsInterceptorDef = new RootBeanDefinition(MappedInterceptor.class);

mappedCsInterceptorDef.setSource(source);

mappedCsInterceptorDef.setRole(BeanDefinition.ROLE\_INFRASTRUCTURE);

mappedCsInterceptorDef.getConstructorArgumentValues().addIndexedArgumentValue(0, (Object) null);

mappedCsInterceptorDef.getConstructorArgumentValues().addIndexedArgumentValue(1, csInterceptorDef);

String mappedInterceptorName = parserContext.getReaderContext().registerWithGeneratedName(mappedCsInterceptorDef);

// 创建ConversionServiceExposingInterceptor[详细见[HandlerInterceptor](#_HandlerInterceptor)]对应的BeanDefiition，同时创建MappedInterceptor对应的BeanDefiition，并将ConversionServiceExposingInterceptor以构造器参数的形式注入到MappedInterceptor。

20、

RootBeanDefinition exceptionHandlerExceptionResolver = new RootBeanDefinition(ExceptionHandlerExceptionResolver.class);

exceptionHandlerExceptionResolver.setSource(source);

exceptionHandlerExceptionResolver.setRole(BeanDefinition.ROLE\_INFRASTRUCTURE);

exceptionHandlerExceptionResolver.getPropertyValues().add("contentNegotiationManager", contentNegotiationManager);

exceptionHandlerExceptionResolver.getPropertyValues().add("messageConverters", messageConverters);

exceptionHandlerExceptionResolver.getPropertyValues().add("order", 0);

String methodExceptionResolverName = parserContext.getReaderContext().registerWithGeneratedName(exceptionHandlerExceptionResolver);

RootBeanDefinition responseStatusExceptionResolver = new RootBeanDefinition(ResponseStatusExceptionResolver.class);

responseStatusExceptionResolver.setSource(source);

responseStatusExceptionResolver.setRole(BeanDefinition.ROLE\_INFRASTRUCTURE);

responseStatusExceptionResolver.getPropertyValues().add("order", 1);

String responseStatusExceptionResolverName = parserContext.getReaderContext().registerWithGeneratedName(responseStatusExceptionResolver);

RootBeanDefinition defaultExceptionResolver = new RootBeanDefinition(DefaultHandlerExceptionResolver.class);

defaultExceptionResolver.setSource(source);

defaultExceptionResolver.setRole(BeanDefinition.ROLE\_INFRASTRUCTURE);

defaultExceptionResolver.getPropertyValues().add("order", 2);

String defaultExceptionResolverName = parserContext.getReaderContext().registerWithGeneratedName(defaultExceptionResolver);

// 构建ExceptionHandlerExceptionResolver、ResponseStatusExceptionResolver、DefaultHandlerExceptionResolver对应的BeanDefiition，并定义好排序编号，分别为0，1，2.

// 将ContentNegotiationManager、MessageConverters注入ExceptionHandlerExceptionResolver的contentNegotiationManager、messageConverters属性。

// 详细见[HandlerExceptionResolver](#_HandlerExceptionResolver)

21、 MvcNamespaceUtils.registerDefaultComponents(parserContext, source);

registerDefaultComponents实现逻辑如下

registerBeanNameUrlHandlerMapping(parserContext, source);

registerHttpRequestHandlerAdapter(parserContext, source);

registerSimpleControllerHandlerAdapter(parserContext, source);

定义BeanNameUrlHandlerMapping、HttpRequestHandlerAdapter和SimpleControllerHandlerAdapter对应的BeanDefiition，并将其注入BeanFactory中。

总结：如果Spring上下文配置中存在”<mvc:annotation-driven/>”标签元素，就相当于BeanFactory会自动装配RequestMappingHandlerMapping、RequestMappingHandlerAdapter、CompositeUriComponentsContributorFactoryBean、ConversionServiceExposingInterceptor、ExceptionHandlerExceptionResolver、ResponseStatusExceptionResolver、DefaultHandlerExceptionResolver、BeanNameUrlHandlerMapping、HttpRequestHandlerAdapter和SimpleControllerHandlerAdapter类型的Bean。并会根据相关配置往对应Bean中注入相关属性。

#### <mvc:default-servlet-handler/>

<mvc:default-servlet-handler/>对应的解析类DefaultServletHandlerBeanDefinitionParser.

String defaultServletName = element.getAttribute("default-servlet-name");

RootBeanDefinition defaultServletHandlerDef = new RootBeanDefinition(DefaultServletHttpRequestHandler.class);

defaultServletHandlerDef.setSource(source);

defaultServletHandlerDef.setRole(BeanDefinition.ROLE\_INFRASTRUCTURE);

if (StringUtils.hasText(defaultServletName)) {

defaultServletHandlerDef.getPropertyValues().add("defaultServletName", defaultServletName);

}

String defaultServletHandlerName = parserContext.getReaderContext().generateBeanName(defaultServletHandlerDef);

parserContext.getRegistry().registerBeanDefinition(defaultServletHandlerName, defaultServletHandlerDef);

parserContext.registerComponent(new BeanComponentDefinition(defaultServletHandlerDef, defaultServletHandlerName));

// 创建DefaultServletHttpRequestHandler对应的BeanDefiition，判断<mvc:default-servlet-handler/>是否存在”default-servlet-name”属性，如果存在将其注入DefaultServletHttpRequestHandler的defaultServletName属性。将DefaultServletHttpRequestHandler注册到BeanFactory中。

//DefaultServletHttpRequestHandler详细见[HttpRequestHandler](#_HttpRequestHandler)。

Map<String, String> urlMap = new ManagedMap<String, String>();

urlMap.put("/\*\*", defaultServletHandlerName);

RootBeanDefinition handlerMappingDef = new RootBeanDefinition(SimpleUrlHandlerMapping.class);

handlerMappingDef.setSource(source);

handlerMappingDef.setRole(BeanDefinition.ROLE\_INFRASTRUCTURE);

handlerMappingDef.getPropertyValues().add("urlMap", urlMap);

String handlerMappingBeanName = parserContext.getReaderContext().generateBeanName(handlerMappingDef);

parserContext.getRegistry().registerBeanDefinition(handlerMappingBeanName, handlerMappingDef);

parserContext.registerComponent(new BeanComponentDefinition(handlerMappingDef, handlerMappingBeanName));

// SimpleUrlHandlerMapping对应的BeanDefiition，并指定其urlMap属性值为”/\*\*”

MvcNamespaceUtils.registerDefaultComponents(parserContext, source);

// 与<mvc:annotation-driven/>一样，也需要在BeanFactory为mvc注册BeanNameUrlHandlerMapping、HttpRequestHandlerAdapter和SimpleControllerHandlerAdapter对应的BeanDefiition。

#### <interceptors/>

<mvc:interceptors />对应的解析类InterceptorsBeanDefinitionParser.

<mvc:interceptors path-matcher="">

<bean></bean>

<ref />

<mvc:interceptor>

<mvc:mapping path="" />

<mvc:exclude-mapping path="" />

<ref />

</mvc:interceptor>

</mvc:interceptors>

#### <mvc:resources/>

<mvc:resources />对应的解析类ResourcesBeanDefinitionParser.

String resourceHandlerName = registerResourceHandler(parserContext, element, source);

if (resourceHandlerName == null) {

return null;

}

实现逻辑如下：

String locationAttr = element.getAttribute("location");

if (!StringUtils.hasText(locationAttr)) {

parserContext.getReaderContext().error("The 'location' attribute is required.", parserContext.extractSource(element));

return null;

}

ManagedList<String> locations = new ManagedList<String>();

locations.addAll(Arrays.asList(StringUtils.commaDelimitedListToStringArray(locationAttr)));

RootBeanDefinition resourceHandlerDef = new RootBeanDefinition(ResourceHttpRequestHandler.class);

resourceHandlerDef.setSource(source);

resourceHandlerDef.setRole(BeanDefinition.ROLE\_INFRASTRUCTURE);

resourceHandlerDef.getPropertyValues().add("locations", locations);

String cacheSeconds = element.getAttribute("cache-period");

if (StringUtils.hasText(cacheSeconds)) {

resourceHandlerDef.getPropertyValues().add("cacheSeconds", cacheSeconds);

}

String beanName = parserContext.getReaderContext().generateBeanName(resourceHandlerDef);

parserContext.getRegistry().registerBeanDefinition(beanName, resourceHandlerDef);

parserContext.registerComponent(new BeanComponentDefinition(resourceHandlerDef, beanName));

return beanName;

// <mvc:resources/>标签元素的locations属性必填，且支持逗号[,]分隔，从而支持多个location对应一个mapping。在BeanFactory注册ResourceHttpRequestHandler对应的BeanDefiition，BeanName根据规则生成，如果元素存在”cache-period”属性，则将其注入ResourceHttpRequestHandler的cacheSeconds属性。

Map<String, String> urlMap = new ManagedMap<String, String>();

String resourceRequestPath = element.getAttribute("mapping");

if (!StringUtils.hasText(resourceRequestPath)) {

parserContext.getReaderContext().error("The 'mapping' attribute is required.", parserContext.extractSource(element));

return null;

}

urlMap.put(resourceRequestPath, resourceHandlerName);

RootBeanDefinition handlerMappingDef = new RootBeanDefinition(SimpleUrlHandlerMapping.class);

handlerMappingDef.setSource(source);

handlerMappingDef.setRole(BeanDefinition.ROLE\_INFRASTRUCTURE);

handlerMappingDef.getPropertyValues().add("urlMap", urlMap);

String order = element.getAttribute("order");

handlerMappingDef.getPropertyValues().add("order", StringUtils.hasText(order) ? order : Ordered.LOWEST\_PRECEDENCE - 1);

String beanName = parserContext.getReaderContext().generateBeanName(handlerMappingDef);

parserContext.getRegistry().registerBeanDefinition(beanName, handlerMappingDef);

parserContext.registerComponent(new BeanComponentDefinition(handlerMappingDef, beanName));

// 获取便签元素的mapping属性，并创建mapping与ResourceHttpRequestHandler对应的beanName的映射关系urlMap，创建SimpleUrlHandlerMapping对应BeanDefiition，BeanName自增创建，并将映射关系urlMap注入到SimpleUrlHandlerMapping的urlMap属性中。

#### <mvc:view-controller/>

<mvc:view-controller />对应的解析类ViewControllerBeanDefinitionParser.

### FrameworkServlet.service(HttpServletRequest, HttpServletResponse)

protected void service(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {

String method = request.getMethod();

if (method.equalsIgnoreCase(RequestMethod.PATCH.name())) {

processRequest(request, response);

} else {

super.service(request, response);

}

}

// 在FrameworkServlet中的service方法可以看出，除了父类支持的http method之外，Spring还额外支持request method为”PATCH”的请求。

父类service方法逻辑

String method = req.getMethod();

if (method.equals("GET")) {

long lastModified = getLastModified(req);

if (lastModified == -1L) {

doGet(req, resp);

} else {

long ifModifiedSince;

try {

ifModifiedSince = req.getDateHeader("If-Modified-Since");

} catch (IllegalArgumentException iae) {

ifModifiedSince = -1L;

}

if (ifModifiedSince < lastModified / 1000L \* 1000L) {

maybeSetLastModified(resp, lastModified);

doGet(req, resp);

} else {

resp.setStatus(304);

}

}

} else if (method.equals("HEAD")) {

long lastModified = getLastModified(req);

maybeSetLastModified(resp, lastModified);

doHead(req, resp);

} else if (method.equals("POST")) {

doPost(req, resp);

} else if (method.equals("PUT")) {

doPut(req, resp);

} else if (method.equals("DELETE")) {

doDelete(req, resp);

} else if (method.equals("OPTIONS")) {

doOptions(req, resp);

} else if (method.equals("TRACE")) {

doTrace(req, resp);

} else {

String errMsg = lStrings.getString("http.method\_not\_implemented");

Object[] errArgs = new Object[1];

errArgs[0] = method;

errMsg = MessageFormat.format(errMsg, errArgs);

resp.sendError(501, errMsg);

}

// 从上面的逻辑来看，HttpServlet支持的request method有”GET”、”HEAD”、”POST”、”PUT”、”DELETE”、”OPTIONS”、”TRACE”.如果传入的方法不正确，将返回501错误。

#### processRequest(HttpServletRequest, HttpServletResponse)

接口/类路径：org.springframework.web.servlet.FrameworkServlet

代码方法：processRequest

long startTime = System.currentTimeMillis();

Throwable failureCause = null;

LocaleContext previousLocaleContext = LocaleContextHolder.getLocaleContext();// 返回当前线程相关联的LocaleContext。详细见[LocaleContext](#_LocaleContext)。

LocaleContext localeContext = buildLocaleContext(request);// 根据当前请求创建一个LocaleContext。

RequestAttributes previousAttributes = RequestContextHolder.getRequestAttributes();// 返回当前当前线程已绑定的RequestAttributes。详细见[RequestAttributes](#_RequestAttributes)

ServletRequestAttributes requestAttributes = buildRequestAttributes(request, response, previousAttributes);// 根据当前request创建RequestAttributes，如果前一个RequestAttributes[previousAttributes]为null或者类型为ServletRequestAttributes，则创建一个ServletRequestAttributes实例，否则不创建新的RequestAttributes。

WebAsyncManager asyncManager = WebAsyncUtils.getAsyncManager(request);// 从request属性中获取一个WebAsyncManager，对应的属性KEY为：WebAsyncManager.class.getName() + ".WEB\_ASYNC\_MANAGER"。如果无法从request属性中获取到，则创建一个WebAsyncManager，并设置到request属性，key为：WebAsyncManager.class.getName() + ".WEB\_ASYNC\_MANAGER"。

asyncManager.registerCallableInterceptor(FrameworkServlet.class.getName(), new RequestBindingInterceptor()); //为WebAsyncManager注册CallableProcessingInterceptor，存放在WebAsyncManager实例中的callableInterceptors中，KEY为：FrameworkServlet.class.getName()

// CallableProcessingInterceptor详细见[CallableProcessingInterceptor](#_CallableProcessingInterceptor)。

initContextHolders(request, localeContext, requestAttributes);// 将LocaleContext 和RequestAttributes 存放到ThreadLocal中。

try {

doService(request, response); // 调用doService处理请求,详细见[doService(HttpServletRequest, HttpServletResponse)](#_doService(HttpServletRequest,_HttpS)

} catch (ServletException ex) {

failureCause = ex;

throw ex;

} catch (IOException ex) {

failureCause = ex;

throw ex;

} catch (Throwable ex) {

failureCause = ex;

throw new NestedServletException("Request processing failed", ex);

} finally {

resetContextHolders(request, previousLocaleContext, previousAttributes);

if (requestAttributes != null) {

requestAttributes.requestCompleted();

}

publishRequestHandledEvent(request, startTime, failureCause);

}

#### doService(HttpServletRequest, HttpServletResponse)

接口/类路径：org.springframework.web.servlet.DispatcherServlet

代码方法：doService

Map<String, Object> attributesSnapshot = null;

// 如果request中存在”javax.servlet.include.request\_uri”名字的属性，则把所有的属性信息当作快照保存到attributesSnapshot中。

if (WebUtils.isIncludeRequest(request)) {

attributesSnapshot = new HashMap<String, Object>();

Enumeration<?> attrNames = request.getAttributeNames();

while (attrNames.hasMoreElements()) {

String attrName = (String) attrNames.nextElement();

if (this.cleanupAfterInclude || attrName.startsWith("org.springframework.web.servlet")) {

attributesSnapshot.put(attrName, request.getAttribute(attrName));

}

}

}

// WebApplicationContext、LocaleResolver、ThemeResolver、ThemeSource、FlashMap和FlashMapManager等信息设置到request属性中。

request.setAttribute(WEB\_APPLICATION\_CONTEXT\_ATTRIBUTE, getWebApplicationContext());

request.setAttribute(LOCALE\_RESOLVER\_ATTRIBUTE, this.localeResolver);

request.setAttribute(THEME\_RESOLVER\_ATTRIBUTE, this.themeResolver);

request.setAttribute(THEME\_SOURCE\_ATTRIBUTE, getThemeSource());

FlashMap inputFlashMap = this.flashMapManager.retrieveAndUpdate(request, response);

if (inputFlashMap != null) {

request.setAttribute(INPUT\_FLASH\_MAP\_ATTRIBUTE, Collections.unmodifiableMap(inputFlashMap));

}

request.setAttribute(OUTPUT\_FLASH\_MAP\_ATTRIBUTE, new FlashMap());

request.setAttribute(FLASH\_MAP\_MANAGER\_ATTRIBUTE, this.flashMapManager);

try {

doDispatch(request, response);// 详细见[doDispatch(HttpServletRequest, HttpServletResponse)](#_doDispatch(HttpServletRequest,_Http)

} finally {

if (WebAsyncUtils.getAsyncManager(request).isConcurrentHandlingStarted()) {// 开启异步请求处理

return;

}

if (attributesSnapshot != null) {

restoreAttributesAfterInclude(request, attributesSnapshot);//根据快照中的request属性信息去修改处理完请求后的request属性信息。

}

}

#### doDispatch(HttpServletRequest, HttpServletResponse)

接口/类路径：org.springframework.web.servlet.DispatcherServlet

代码方法：doDispatch

HttpServletRequest processedRequest = request;

**HandlerExecutionChain** mappedHandler = null;

boolean multipartRequestParsed = false;

WebAsyncManager asyncManager = WebAsyncUtils.getAsyncManager(request);// 从request中获取WebAsyncManager。

try {

ModelAndView mv = null;

Exception dispatchException = null;

try {

processedRequest = checkMultipart(request);// 判断请求是否为多部件请求，文件上传会用到。详细见[checkMultipart(HttpServletRequest request)](#_checkMultipart(HttpServletRequest_r)

multipartRequestParsed = (processedRequest != request);// 如果是多部件请求，则multipartRequestParsed为true。

mappedHandler = getHandler(processedRequest); // 根据reuqest信息获取**HandlerExecutionChain** ，详细见[getHandler(HttpServletRequest)](#_getHandler(HttpServletRequest)_1)

if (mappedHandler == null || mappedHandler.getHandler() == null) {

noHandlerFound(processedRequest, response);

return;

}// 如果没有处理的HandlerMethod，则终止业务受理，并返回404或者抛出异常。

HandlerAdapter ha = getHandlerAdapter(mappedHandler.getHandler());// 根据当前请求请求获得最靠前，最适合的HandlerAdapter。

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;

}

}

/\*\*

\* 执行**HandlerExecutionChain**中的所有的HandlerInterceptor对应的**preHandle**方法。如果执行triggerAfterCompletion方法，详细见[triggerAfterCompletion(HttpServletRequest,HttpServletResponse,Exception)](#_triggerAfterCompletion(HttpServletR)。

\*/

if (!mappedHandler.applyPreHandle(processedRequest, response)) {

return;

}

try {

mv = ha.handle(processedRequest, response, mappedHandler.getHandler());//处理request请求。不同的HandlerAdapter有不同的实现，这里以” RequestMappingHandlerAdapter”为例，详细见[handleInternal(HttpServletRequest,HttpServletResponse, HandlerMethod)](#_handleInternal(HttpServletRequest,H)

} finally { // 判断异步管理器是否启动，如果启动则不进行后续处理

if (asyncManager.isConcurrentHandlingStarted()) {

return;

}

}

applyDefaultViewName(request, mv); // 详细见[applyDefaultViewName(HttpServletRequest, ModelAndView)](#_applyDefaultViewName(HttpServletReq)

mappedHandler.applyPostHandle(processedRequest, response, mv);// 迭代HandlerExecutionChain中所有的HandlerInterceptor，执行其postHandle方法。

} catch (Exception ex) {

dispatchException = ex;

}

processDispatchResult(processedRequest, response, mappedHandler, mv, dispatchException);// 处理调度接口，详细见[processDispatchResult(HttpServletRequest,HttpServletResponse,HandlerExecutionChain, ModelAndView, Exception)](#_processDispatchResult(HttpServletRe)

} catch (Exception ex) {

triggerAfterCompletion(processedRequest, response, mappedHandler, ex);

} catch (Error err) {

triggerAfterCompletionWithError(processedRequest, response, mappedHandler, err);

} finally {

if (asyncManager.isConcurrentHandlingStarted()) {

/\*\*

如果启动了异步管理器，那么在执行玩HandlerAdapter的handle()方法后，则不会去执行HandlerInterceptor的postHandle方法，这里就会去迭代HandlerExecutionChain中所有的AsyncHandlerInterceptor，并执行其afterConcurrentHandlingStarted方法

\*/

mappedHandler.applyAfterConcurrentHandlingStarted(processedRequest, response);

return;

}

if (multipartRequestParsed) {

cleanupMultipart(processedRequest);

}

}

#### checkMultipart(HttpServletRequest request)

接口/类路径：org.springframework.web.servlet.DispatcherServlet

代码方法：doDispatch

if (this.multipartResolver != null && this.multipartResolver.isMultipart(request)) {

if (WebUtils.getNativeRequest(request, MultipartHttpServletRequest.class) != null) {

logger.debug("Request is already a MultipartHttpServletRequest - if not in a forward, " + "this typically results from an additional MultipartFilter in web.xml");

} else if (request.getAttribute(WebUtils.ERROR\_EXCEPTION\_ATTRIBUTE) instanceof MultipartException) {

logger.debug("Multipart resolution failed for current request before - " + "skipping re-resolution for undisturbed error rendering");

} else {

return this.multipartResolver.resolveMultipart(request);

}

}

return request;

// this.multipartResolver.isMultipart(request) // 判断请求是否为多部件请求

if (!"post".equals(request.getMethod().toLowerCase())) {

return false;

}

String contentType = request.getContentType();

return (contentType != null && contentType.toLowerCase().startsWith("multipart/"));

// request method is “POST”且contentType以“multipart/”开头。

//this.multipartResolver.resolveMultipart(request); 将request包装成StandardServletMultipartResolver，并接起请求头中的”content-disposition”部分获取文件信息。

#### getHandler(HttpServletRequest)

接口/类路径：org.springframework.web.servlet.DispatcherServlet

代码方法：getHandler

for (HandlerMapping hm : this.handlerMappings) {

HandlerExecutionChain handler = hm.getHandler(request);

if (handler != null) {

return handler;

}

}

return null;

// 迭代上下文总所有的HandlerMapping，然后调用getHandler获取HandlerExecutionChain，详细见[getHandler(HttpServletRequest)](#_getHandler(HttpServletRequest))

#### getHandler(HttpServletRequest)

接口/类路径：org.springframework.web.servlet.handler.AbstractHandlerMapping

代码方法：getHandler

Object handler = getHandlerInternal(request);// 根据request获取HandlerMethod,详细见[getHandlerInternal(HttpServletRequest)](#_getHandlerInternal(HttpServletReque)

if (handler == null) {

handler = getDefaultHandler();

}

if (handler == null) {

return null;

}

// Bean name or resolved handler?

if (handler instanceof String) {

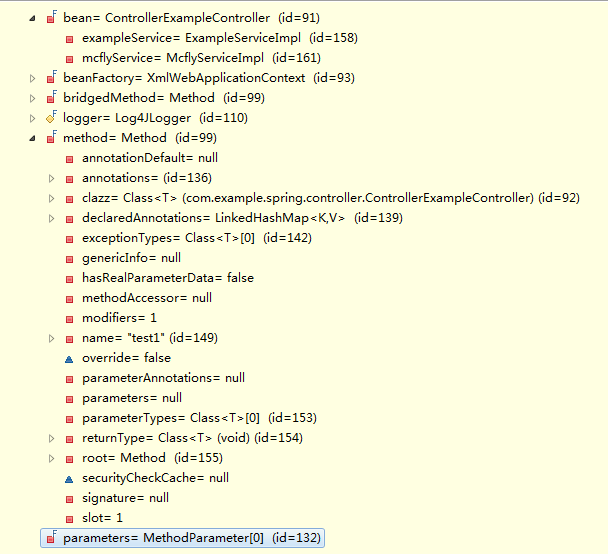
String handlerName = (String) handler;

handler = getApplicationContext().getBean(handlerName);

}

return getHandlerExecutionChain(handler, request); // 将HandlerMethod和匹配的拦截器加入到HandlerExecutionChain中。

// HandlerMethod 详细见[HandlerMethod](#_HandlerMethod)，所有的HandlerMethod会在[initializeBean(String,Object,RootBeanDefinition)](#_initializeBean(String,Object,RootBe)中初始化。



#### getHandlerInternal(HttpServletRequest)

接口/类路径：org.springframework.web.servlet.handler.AbstractHandlerMapping

代码方法：getHandlerInternal

String lookupPath = getUrlPathHelper().getLookupPathForRequest(request);

HandlerMethod handlerMethod = lookupHandlerMethod(lookupPath, request);

return (handlerMethod != null ? handlerMethod.createWithResolvedBean() : null);

// 根据路径匹配出HandlerMethod，所有的HandlerMethod会在initializeBean方法中初始化加载，详细见[initializeBean(String,Object,RootBeanDefinition)](#_initializeBean(String,Object,RootBe)。

#### triggerAfterCompletion(HttpServletRequest,HttpServletResponse,Exception)

接口/类路径：org.springframework.web.servlet.HandlerExecutionChain

代码方法：triggerAfterCompletion

if (getInterceptors() == null) {

return;

}

for (int i = this.interceptorIndex; i >= 0; i--) {

HandlerInterceptor interceptor = getInterceptors()[i];

try {

interceptor.afterCompletion(request, response, this.handler, ex);

} catch (Throwable ex2) {

logger.error("HandlerInterceptor.afterCompletion threw exception", ex2);

}

}

#### handleInternal(HttpServletRequest,HttpServletResponse, HandlerMethod)

接口/类路径：org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerAdapter

代码方法：handleInternal

if (getSessionAttributesHandler(handlerMethod).hasSessionAttributes()) {

checkAndPrepare(request, response, this.cacheSecondsForSessionAttributeHandlers, true);

} else {

checkAndPrepare(request, response, true);

}

if (this.synchronizeOnSession) {

HttpSession session = request.getSession(false);

if (session != null) {

Object mutex = WebUtils.getSessionMutex(session);

synchronized (mutex) {

return invokeHandleMethod(request, response, handlerMethod);

}

}

}

return invokeHandleMethod(request, response, handlerMethod);

#### invokeHandleMethod(HttpServletRequest,HttpServletResponse, HandlerMethod)

接口/类路径：org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerAdapter

代码方法：invokeHandleMethod

ServletWebRequest webRequest = new ServletWebRequest(request, response);

// 获取DataBeanFactory

WebDataBinderFactory binderFactory = getDataBinderFactory(handlerMethod);// 详细见[getDataBinderFactory(HandlerMethod)](#_getDataBinderFactory(HandlerMethod))

// 获取getModelFactory

ModelFactory modelFactory = getModelFactory(handlerMethod, binderFactory);// 详细见[getModelFactory(HandlerMethod,WebDataBinderFactory)](#_getModelFactory(HandlerMethod,WebDa)

// 根据HandlerMethod和DataBeanFactory构建ServletInvocableHandlerMethod

ServletInvocableHandlerMethod requestMappingMethod = createRequestMappingMethod(handlerMethod, binderFactory);// 详细见[createRequestMappingMethod(HandlerMethod, WebDataBinderFactory)](#_createRequestMappingMethod(HandlerM)

// 创建模型和视图容器ModelAndViewContainer

ModelAndViewContainer mavContainer = new ModelAndViewContainer();

mavContainer.addAllAttributes(RequestContextUtils.getInputFlashMap(request));// 将FlashMap设置到ModelAndViewContainer中。

modelFactory.initModel(webRequest, mavContainer, requestMappingMethod);// 详细见[initModel(NativeWebRequest,ModelAndViewContainer,HandlerMethod)](#_initModel(NativeWebRequest,ModelAnd)

mavContainer.setIgnoreDefaultModelOnRedirect(this.ignoreDefaultModelOnRedirect);

// 获取异步WebRequest，需要Servlet容器[如Tomcat]支持，

AsyncWebRequest asyncWebRequest = WebAsyncUtils.createAsyncWebRequest(request, response);// 详细见[createAsyncWebRequest(HttpServletRequest,HttpServletResponse)](#_createAsyncWebRequest(HttpServletRe)

asyncWebRequest.setTimeout(this.asyncRequestTimeout);

// 获取WebAsyncManager，并配置相关数据

final WebAsyncManager asyncManager = WebAsyncUtils.getAsyncManager(request);

asyncManager.setTaskExecutor(this.taskExecutor);

asyncManager.setAsyncWebRequest(asyncWebRequest);

asyncManager.registerCallableInterceptors(this.callableInterceptors);

asyncManager.registerDeferredResultInterceptors(this.deferredResultInterceptors);

if (asyncManager.hasConcurrentResult()) {

Object result = asyncManager.getConcurrentResult();

mavContainer = (ModelAndViewContainer) asyncManager.getConcurrentResultContext()[0];

asyncManager.clearConcurrentResult();

requestMappingMethod = requestMappingMethod.wrapConcurrentResult(result);

}

//调用方法并通过注册处理返回值“HandlerMethodReturnValueHandler”

requestMappingMethod.invokeAndHandle(webRequest, mavContainer);// 详细见[invokeAndHandle(ServletWebRequest webRequest,ModelAndViewContainer, Object...)](#_invokeAndHandle(ServletWebRequest_w)

if (asyncManager.isConcurrentHandlingStarted()) { // 如果启动了异步管理器，则不返回ModelAndView

return null;

}

// 处理结果，返回ModelAndView

return getModelAndView(mavContainer, modelFactory, webRequest);// 详细见[getModelAndView(ModelAndViewContainer,ModelFactory,NativeWebRequest)](#_getModelAndView(ModelAndViewContain)

#### getDataBinderFactory(HandlerMethod)

// 此方法主要是加载出在@Controller和@ControllerAdvice中@InitBinder的方法。

Class<?> handlerType = handlerMethod.getBeanType();

Set<Method> methods = this.initBinderCache.get(handlerType);// 获取initBinder缓存数据

if (methods == null) { // 如果没有，则去handlerType中获取匹配@InitBinder注解的方法。

methods = HandlerMethodSelector.selectMethods(handlerType, INIT\_BINDER\_METHODS);

this.initBinderCache.put(handlerType, methods);

}

// initBinderAdviceCache全局变量会在[initializeBean(String,Object,RootBeanDefinition)](#_initializeBean(String,Object,RootBe)中触发初始化。此方法内部会调用具体类需要实现InitializingBean接口中的**afterPropertiesSet**方法。

// initBinderAdviceCache回去使用了@ControllerAdvice中的类中获取@InitBinder的方法。

List<InvocableHandlerMethod> initBinderMethods = new ArrayList<InvocableHandlerMethod>();

// Global methods first

for (Entry<ControllerAdviceBean, Set<Method>> entry : this.initBinderAdviceCache .entrySet()){

if (entry.getKey().isApplicableToBeanType(handlerType)) {

Object bean = entry.getKey().resolveBean();

for (Method method : entry.getValue()) {

initBinderMethods.add(createInitBinderMethod(bean, method));

}

}

}

for (Method method : methods) {

Object bean = handlerMethod.getBean();

// 根据@InitBinder注解方法，创建InvocableHandlerMethod对象，需要初始化设值HandlerMethodArgumentResolver、DataBinderFactory和ParameterNameDiscoverer等信息。

initBinderMethods.add(createInitBinderMethod(bean, method));

}

// 创建ServletRequestDataBinderFactory，并将所有匹配的@InitBinder方法设置到其中。

return createDataBinderFactory(initBinderMethods);

#### getModelFactory(HandlerMethod,WebDataBinderFactory)

// 加载出@Controller和@ControllerAdvice中存在@ModelAttribute但没有@RequestMapping注解的方法。

// 通过

// ModelFactory(List<InvocableHandlerMethod>,WebDataBinderFactory,SessionAttributesHandler)构// 造器生成ModelFactory实例。

SessionAttributesHandler sessionAttrHandler = getSessionAttributesHandler(handlerMethod);

Class<?> handlerType = handlerMethod.getBeanType();

Set<Method> methods = this.modelAttributeCache.get(handlerType);

if (methods == null) {

methods = HandlerMethodSelector.selectMethods(handlerType, MODEL\_ATTRIBUTE\_METHODS);

this.modelAttributeCache.put(handlerType, methods);

}

List<InvocableHandlerMethod> attrMethods = new ArrayList<InvocableHandlerMethod>();

// Global methods first

for (Entry<ControllerAdviceBean, Set<Method>> entry : this.modelAttributeAdviceCache.entrySet()){

if (entry.getKey().isApplicableToBeanType(handlerType)) {

Object bean = entry.getKey().resolveBean();

for (Method method : entry.getValue()) {

attrMethods.add(createModelAttributeMethod(binderFactory, bean, method));

}

}

}

for (Method method : methods) {

Object bean = handlerMethod.getBean();

attrMethods.add(createModelAttributeMethod(binderFactory, bean, method));

}

return new ModelFactory(attrMethods, binderFactory, sessionAttrHandler);

#### createRequestMappingMethod(HandlerMethod,WebDataBinderFactory)

// 通过HandlerMethod实例化ServletInvocableHandlerMethod，需要注意的这里会涉及到@ResponseStatus注// 解

ServletInvocableHandlerMethod requestMethod;

requestMethod = new ServletInvocableHandlerMethod(handlerMethod);

requestMethod.setHandlerMethodArgumentResolvers(this.argumentResolvers);

requestMethod.setHandlerMethodReturnValueHandlers(this.returnValueHandlers);

requestMethod.setDataBinderFactory(binderFactory);

requestMethod.setParameterNameDiscoverer(this.parameterNameDiscoverer);

return requestMethod;

#### initModel(NativeWebRequest,ModelAndViewContainer,HandlerMethod)

Map<String,?> attributesInSession = this.sessionAttributesHandler.retrieveAttributes(request);

mavContainer.mergeAttributes(attributesInSession);

invokeModelAttributeMethods(request, mavContainer);// 调用@ModelAttribute注解方法，将结果设置到

// ModelAndViewContainer中

for (String name : findSessionAttributeArguments(handlerMethod)) {

if (!mavContainer.containsAttribute(name)) {

Object value = this.sessionAttributesHandler.retrieveAttribute(request, name);

if (value == null) {

throw new HttpSessionRequiredException("Expected session attribute '" + name + "'");

}

mavContainer.addAttribute(name, value);

}}

#### createAsyncWebRequest(HttpServletRequest,HttpServletResponse)

public static AsyncWebRequest createAsyncWebRequest(HttpServletRequest request, HttpServletResponse response) {

return ClassUtils.hasMethod(ServletRequest.class, "startAsync") ? createStandardServletAsyncWebRequest(request, response) : new NoSupportAsyncWebRequest(request, response);

}

// 需要判断javax.servlet.ServletRequest中是否包含startAsync方法，此方法是servlet3.0提供的方法，用于支持request异步处理。如果当前服务器[如Tomcat] 支持servlet3.0，则创建StandardServletAsyncWebRequest用于处理异步请求。

#### invokeAndHandle(ServletWebRequest webRequest,ModelAndViewContainer, Object...)

Object returnValue = invokeForRequest(webRequest, mavContainer, providedArgs);

setResponseStatus(webRequest); // 处理请求，调用具体的控制器方法[invokeForRequest(NativeWebRequest,ModelAndViewContainer,Object... providedArgs)](#_invokeForRequest(NativeWebRequest,M)

if (returnValue == null) {

if (isRequestNotModified(webRequest) || hasResponseStatus() || mavContainer.isRequestHandled()) {

mavContainer.setRequestHandled(true);

return;

}

} else if (StringUtils.hasText(this.responseReason)) {

mavContainer.setRequestHandled(true);

return;

}

mavContainer.setRequestHandled(false);

try {

this.returnValueHandlers.handleReturnValue(returnValue, getReturnValueType(returnValue), mavContainer, webRequest);// 处理返回值,详细见[handleReturnValue(Object,MethodParameter,ModelAndViewContainer,NativeWebRequest)](#_handleReturnValue(Object,MethodPara)

} catch (Exception ex) {

// …

}

#### invokeForRequest(NativeWebRequest,ModelAndViewContainer,Object... providedArgs)

接口/类路径：org.springframework.web.method.support.InvocableHandlerMethod

代码方法：invokeForRequest

Object[] args = getMethodArgumentValues(request, mavContainer, providedArgs);// 根据请求，获取方法调用参数，详细见[getMethodArgumentValues(NativeWebRequest,ModelAndViewContainer,Object... providedArgs)](#_getMethodArgumentValues(NativeWebRe)

Object returnValue = invoke(args);

return returnValue;

#### getMethodArgumentValues(NativeWebRequest,ModelAndViewContainer,Object... providedArgs)

private Object[] getMethodArgumentValues(NativeWebRequest request, ModelAndViewContainer mavContainer, Object... providedArgs) throws Exception {

MethodParameter[] parameters = getMethodParameters();

Object[] args = new Object[parameters.length];

for (int i = 0; i < parameters.length; i++) {

MethodParameter parameter = parameters[i];

// 设置参数名字发现器，详细见[ParameterNameDiscoverer](#_ParameterNameDiscoverer)

parameter.initParameterNameDiscovery(this.parameterNameDiscoverer);

GenericTypeResolver.resolveParameterType(parameter, getBean().getClass());

args[i] = resolveProvidedArgument(parameter, providedArgs);

if (args[i] != null) {

continue;

}

// 迭代HandlerMethod中所有的HandlerMethodArgumentResolver，这些数据会在RequestMappingHandlerAdapter的afterPropertiesSet方法中初始化。

if (this.argumentResolvers.supportsParameter(parameter)) {// 判断是否支持存在支持的参数解析器。

try {

args[i] = this.argumentResolvers.resolveArgument(parameter, mavContainer, request, this.dataBinderFactory);// 解析参数

continue;

} catch (Exception ex) {

throw ex;

}

}

if (args[i] == null) {

String msg = getArgumentResolutionErrorMessage("No suitable resolver for argument", i);

throw new IllegalStateException(msg);

}

}

return args;

}

#### handleReturnValue(Object,MethodParameter,ModelAndViewContainer,NativeWebRequest)

接口/类路径：org.springframework.web.servlet.DispatcherServlet

代码方法：handleReturnValue

HandlerMethodReturnValueHandler handler = getReturnValueHandler(returnType);//获取到能够处理返回值的HandlerMethodReturnValueHandler，详细见[HandlerMethodReturnValueHandler](#_HandlerMethodReturnValueHandler)。所有的[HandlerMethodReturnValueHandler](#_HandlerMethodReturnValueHandler)会在RequestMappingHandlerAdapter初始化的时候调用afterPropertiesSet方法进行初始化。

handler.handleReturnValue(returnValue, returnType, mavContainer, webRequest);// 处理返回值



#### getModelAndView(ModelAndViewContainer,ModelFactory,NativeWebRequest)

modelFactory.updateModel(webRequest, mavContainer);//更新sessionAttribute

if (mavContainer.isRequestHandled()) {

return null;

}

ModelMap model = mavContainer.getModel();

ModelAndView mav = new ModelAndView(mavContainer.getViewName(), model);

if (!mavContainer.isViewReference()) {

mav.setView((View) mavContainer.getView());

}

if (model instanceof RedirectAttributes) {

Map<String, ?> flashAttributes = ((RedirectAttributes) model).getFlashAttributes();

HttpServletRequest request = webRequest.getNativeRequest(HttpServletRequest.class);

RequestContextUtils.getOutputFlashMap(request).putAll(flashAttributes);

}

return mav;

#### applyDefaultViewName(HttpServletRequest, ModelAndView)

接口/类路径：org.springframework.web.servlet.DispatcherServlet

代码方法：applyDefaultViewName

if (mv != null && !mv.hasView()) {

mv.setViewName(getDefaultViewName(request));

}// 如果ModelAndView没有返回视图，则通过RequestToViewNameTranslator从request中解析出默认视图名字。

#### processDispatchResult(HttpServletRequest,HttpServletResponse,HandlerExecutionChain, ModelAndView, Exception)

接口/类路径：org.springframework.web.servlet.DispatcherServlet

代码方法：processDispatchResult

boolean errorView = false;

if (exception != null) {

if (exception instanceof ModelAndViewDefiningException) {

mv = ((ModelAndViewDefiningException) exception).getModelAndView();

} else {

Object handler = (mappedHandler != null ? mappedHandler.getHandler() : null);

mv = processHandlerException(request, response, handler, exception);

errorView = (mv != null);

}

}

if (mv != null && !mv.wasCleared()) {

render(mv, request, response);

if (errorView) {

WebUtils.clearErrorRequestAttributes(request);

}

} else {}

if (WebAsyncUtils.getAsyncManager(request).isConcurrentHandlingStarted()) {

return;

}

if (mappedHandler != null) {

mappedHandler.triggerAfterCompletion(request, response, null);

}

# Spring AOP

## AOP相关介绍

面向切面编程(AOP)通过提供另外一种思考程序结构的途径来弥补面向对象编程(OOP)的不足。在OOP中模块化的关键单元是类(class)，而在AOP中模块化的单元则是切面。切面能对关注点进行模块化，例如横切多个类型和对象的事务管理。

AOP在Spring framework中的作用：

1、提供声明式企业服务，特别是为了替代EJB声明式服务.最重要的服务是声明式事务管理。

2、允许用户实现自定义切面，用AOP来完善OOP的使用。

### Spring AOP的功能和目标

1、Spring AOP使用纯Java实现，它不需要专门的编译过程。Spring AOP不需要控制类装载器层次，因此适用于J2EE web容器或应用服务器。

2、Spring目前仅支持使用方法调用作为连接点(join point)。虽然可以在不影响到Spring AOP核心API的情况下加入对成员变量拦截器支持，但Spring并没有实现成员变量拦截器。如果你需要把对成员变量的访问和更新也作为通知的连接点，可以考虑其它的语言，如AspectJ。

Spring实现AOP的方法跟其他的框架不同，Spring并不是要提供最完整的AOP实现(尽管Spring AOP有这个能力)，相反的，它其实侧重于提供一种AOP实现和Spring IoC容器之前的整合,用于帮助解决在企业级开发中的常见问题。

因此,Spring的AOP功能通常和Spring IoC容器一起使用。切面使用普通的bean定义语法来配置(尽管Spring提供了强大的”自动代理(autoproxying)”功能)：与其他的AOP实现相比这是一个显著的区别。有些事使用Spring AOP是无法轻松或者高效完成的,比如说通知一个细粒度的对象(例如典型的域对象)：这时候，使用AspectJ是最好的选择。

Spring AOP从来没有打算通过提供一种全面的AOP解决方案来与AspectJ竞争。我们相信无论是基于代理(proxy-based)的框架如Spring AOP或者是成熟的框架如AspectJ都是很有价值的,他们之间应该是互补而不是竞争的关系。Spring 2.0以上可以无缝的整合Spring AOP，IoC和AspectJ，使得所有AOP应用完全融合于基于Spring的应用体系。

**Spring AOP还是完全用AspectJ**

Spring AOP比完全使用AspectJ更加简单，因为它不需要引入AspectJ的编译器和织入器到你开发和构建过程中。

如果你需要通知domain对象或其他没有在Spring容器中管理的任意对象，那么你需要使用AspectJ。

### AOP代理

Spring缺省使用J2SE动态代理（dynamic proxies）来作为AOP的代理。这样任何接口（或者接口集）都可以被代理。

Spring也可以使用CGLIB代理，对于需要代理类而不是代理接口的时候CGLIB代理是很有必要的。如果一个业务对象并没有实现一个接口，默认就会使用CGLIB。

**代理机制**

Spring AOP部分使用JDK动态代理或者CGLIB来为目标对象创建代理。(建议优先使用JDK的动态代理)。

如果被代理的目标对象实现了至少一个接口,则会使用JDK动态代理。所有该目标类型实现的接口都将被代理.若该目标对象没有实现任何接口，则创建一个CGLIB代理。

如果你想强制使用CGLIB代理(例如：希望代理目标对象的所有方法,而不只是实现自接口的方法)那也可以。但是需要考虑以下问题：

1、无法通知(advise)final方法，因为它们不能被覆盖。

2、你需要将CGLIB 二进制发行包放在classpath下面,与之相较JDK本身就提供了动态代理。当需要CGLIB而在classpath下又没有找到CGLIB类库的话，Spring会自动提醒。

3、代理对象的构造器会被调用两次。这是很自然的结果，因为在CGLIB代理模式下每一个代理对象都会产生一个子类。每一个代理实例会生成两个对象：实际代理对象和它的一个实现了通知的子类实例，而使用JDK代理时不会出现这样的行为。通常情况下，调用代理类型的构造器两次并不是问题，因此除了会发生指派外没有任何真正的逻辑被实现。

强制使用CGLIB代理需要将<aop:config proxy-target-class=”true”/>

<aop:config proxy-target-class=”true”>

<!-- other beans defined here -->

</aop:config>

当使用@AspectJ自动代理时需要强制使用CGLIB，请将<aop:aspectj-autoproxy>的proxy-target-class属性设置为true。

<aop:aspectj-autoproxy proxy-target-class=”true” />

注意：在<tx:annotation-driven/>、<aop:aspectj-autoproxy/>或者<aop:config/>元素上使用proxy-target-class=”true”会导致将CGLIB代理应用于三者之上。

**理解AOP代理**

Spring AOP是基于代理机制的。实际上在你编写自己的切面或使用任何有Spring框架提供的基于Spring AOP切面之前，深刻领会这一句话的意思是非常重要的。

考虑如下场景,当你拿到一个无代理的、无任何特殊之处的POJO对象引用时，如下代码段所示：

public class SimplePojo implements Pojo{

public void foo() {

this.bar();

}

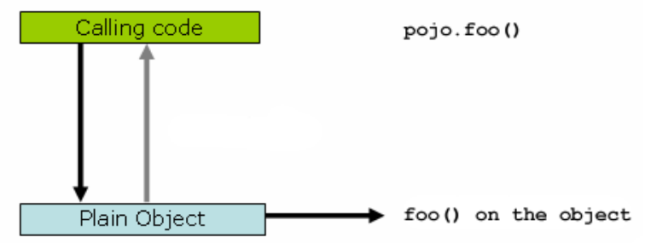
public void bar(){

// ,,,

}

}

当你调用一个对象引用的方法时，此对象引用上的方法直接被调用，如下所示：



public class Main {

public static void main(String[] args){

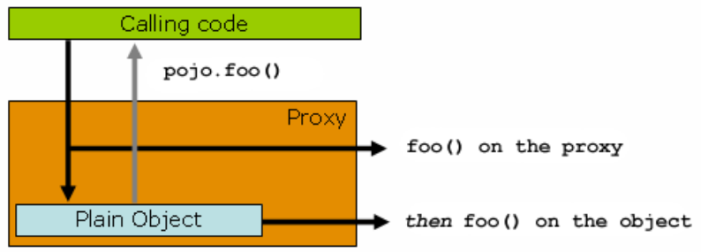
Pojo pojo = new SimplePojo();

pojo.foo();

}

}

当客户代码所持的引用是一个代理的时候则略有不同了，请考虑如下图和代码片段。



public class Main{

public static void main(String args[]){

ProxyFactory factory = new proxyFactory(new SimplePojo());

factory.addInterface(Pojo.class);

factory.addAdvice(new RetryAdvice());

Pojo pojo = (Pojo)factory.getProxy();

pojo.foo();

}

}

理解此处的关键是Main类main(…)方法中的客户代码拥有一个代理的引用，这意味着对这个对象引用中方法的调用就是对代理的调用，而这个代理能够代理所有跟特定方法调用相关的拦截器。不过，一旦调用最终抵达了目标对象(此处为SimplePojo类的引用)，任何对自身的调用例如this.bar()或者this.foo()将对this引用进行调用而非代理。这一点意义重大，它意味着自我调用将不会导致和方法调用关联的通知得到执行的机会。

那好,为此要怎么办呢？最好的办法(这里使用最好这个术语不甚精确)就是重构你的代码使自我调用不会出现。当然，这的确需要你做一些工作,但却是最好的，最少侵入性的方法。另一个方法则很可怕，也正因为如此我几乎不愿意之处这种方法。你可以像如下这样完全把业务逻辑写在你的Spring AOP类中：

public class SimplePojo implements Pojo{

public void foo() {

// this.bar();

((Pojo)AopContext.currentProxy()).bar();

}

public void bar(){

// ,,,

}

}

在代理创建的时候也需要一些额外的配置：

public class Main{

public static void main(String args[]){

ProxyFactory factory = new proxyFactory(new SimplePojo());

factory.addInterface(Pojo.class);

factory.addAdvice(new RetryAdvice());

factory.setExposeProxy(true);

Pojo pojo = (Pojo)factory.getProxy();

pojo.foo();

}

}

### AOP概念介绍

1、切面（Aspect）：一个关注点的模块化，这个关注点可能会横切多个对象。

2、连接点（Joinpoint）：在程序执行过程中某个特定的点，比如某方法调用的时候或处理异常的时候。在Spring AOP中，一个连接点总是表示一个方法的执行。

3、通知（Advice）：在切面的某个特定的连接点上执行的动作.许多AOP框架（包括Spring）都是以拦截器做通知模型，并维护一个以连接点为中心的拦截器链。

4、切入点（Pointcut）：匹配连接点的断言。通知和一个切入点表达式关联，并在满足这个切入点的连接点上运行.切入点表达式如何和连接点匹配是AOP的核心。

5、引入（Introduction）：用来给一个类型声明额外的方法或属性（也被称为连接类型声明[inter-type declaration]）。Spring允许引入新的接口（以及一个对应的实现）到任何被代理的对象。

6、目标对象（target object）：被一个或多个切面所通知的对象。也被称做被通知（advised）对象。既然Spring AOP是通过运行时代理实现的，这个对象永远是一个被代理（proxied）对象。

7、AOP代理（AOP proxy）：AOP框架创建的对象，在Spring中，AOP代理可以是JDK动态代理或则CGLIB代理。

8、织入（weaving）：把切面连接到其他的应用程序类型或对象上,并创建一个被通知的对象。这些可以在编译时（例如使用AspectJ编译器），类加载时和运行时完成。Spring和其他纯Java AOP框架一样，在运行时完成织入。

## 源码分析

Spring AOP提供了@AspectJ支持和基于Schema的AOP支持。

### 基于Schema的AOP支持

在IoC章节我们已经了解了关于Spring IoC容器初始化的步骤，我们这里需要关注如下几点：

1、Spring AOP的Schema声明。

2、Spring AOP的xml Schema解析。

3、Spring AOP对应bean的初始化。

#### Spring AOP的Schema声明



#### Spring AOP的xml Schema解析

在Spring IoC中我们了解到关于Spring对Xml的解析，这里我们需要关注对AOP元素的解析，我们可以在” META-INF/spring.handlers”文件中找到关于对aop命名空间的解析。我们可以从AopNamespaceHandler找到Spring为对应的元素注册了对应的BeanDefiitionParser。如下：

**public** **void** init() {

registerBeanDefinitionParser("config", **new** ConfigBeanDefinitionParser());

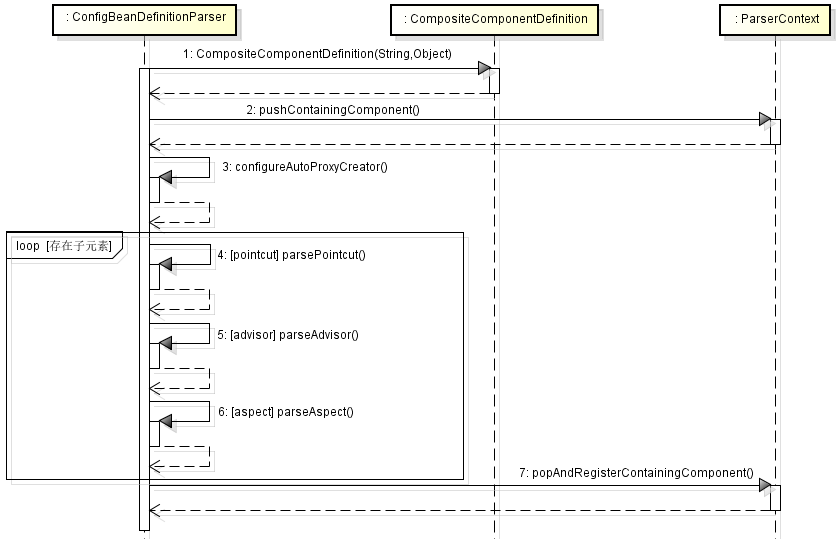
registerBeanDefinitionParser("aspectj-autoproxy", **new** AspectJAutoProxyBeanDefinitionParser());

registerBeanDefinitionDecorator("scoped-proxy", **new** ScopedProxyBeanDefinitionDecorator());

registerBeanDefinitionParser("spring-configured", **new** SpringConfiguredBeanDefinitionParser());

}

##### <aop:config />



ConfigBeanDefinitionParser.parse方法时序图

1、创建便签元素对应的复合容器定义[CompositeComponentDefinition],并将其存放到PageContext中，保存在containingComponents字段中。

2、调用configureAutoProxyCreator方法配置自动代理创建器。详细见[configureAutoProxyCreator(ParserContext, Element)](#_configureAutoProxyCreator(ParserCon)。

3、迭代<aop:config/>下面的子元素，需要对<aop:pointcut />、<aop:advisor />、<aop:aspect/>三中子元素，对应的解析方法[parsePointcut(Element, ParserContext)](#_parsePointcut(Element,_ParserContex)、[parseAdvisor(Element, ParserContext)](#_parseAdvisor(Element,_ParserContext)和[parseAspect(Element, ParserContext)](#_parseAspect(Element,_ParserContext))。

##### configureAutoProxyCreator(ParserContext, Element)

文件位置：org/springframework/aop/config/ConfigBeanDefinitionParser.java

1、为Spring AOP在IoC容器中注册**AspectJAwareAdvisorAutoProxyCreator**（详细见[AbstractAutoProxyCreator](#_AbstractAutoProxyCreator)）对应的Bean定义，如果在BeanFactory中已经存在了名字为”org.springframework.aop.config.internalAutoProxyCreator”的定义，但是对应的class不为AspectJAwareAdvisorAutoProxyCreator，需要根据AspectJAwareAdvisorAutoProxyCreator 的优先级高于现在的beanName对应的class，则替换他。否则保持现有的bean定义。优先级根据APC\_PRIORITY\_LIST的数组下标判断，下标越大的优先级越高。

**static** {

***APC\_PRIORITY\_LIST***.add(InfrastructureAdvisorAutoProxyCreator.**class**);

***APC\_PRIORITY\_LIST***.add(AspectJAwareAdvisorAutoProxyCreator.**class**);

***APC\_PRIORITY\_LIST***.add(AnnotationAwareAspectJAutoProxyCreator.**class**);

}

如果BeanFactory中不存在”org.springframework.aop.config.internalAutoProxyCreator”的定义，则创建AspectJAwareAdvisorAutoProxyCreator对应的Bean定义，如下：

RootBeanDefinition beanDefinition = **new** RootBeanDefinition(cls);

beanDefinition.setSource(source);

beanDefinition.getPropertyValues().add("order", Ordered.***HIGHEST\_PRECEDENCE***);

beanDefinition.setRole(BeanDefinition.***ROLE\_INFRASTRUCTURE***); // 角色为2

registry.registerBeanDefinition(***AUTO\_PROXY\_CREATOR\_BEAN\_NAME***, beanDefinition);

2、判断**<aop:config/>**是否存在proxy-target-class和expose-proxy属性，如果存在则将AspectJAwareAdvisorAutoProxyCreator中的proxyTargetClass和exposeProxy域字段设置为true。

3、如果AspectJAwareAdvisorAutoProxyCreator对应的Bean定义不为空，则将对应的BeanDefinition封装成BeanComponentDefinition。

##### parsePointcut(Element, ParserContext)

**private** AbstractBeanDefinition parsePointcut(Element pointcutElement, ParserContext parserContext) {

String id = pointcutElement.getAttribute(***ID***);

String expression = pointcutElement.getAttribute(***EXPRESSION***); // 获取切面表达式

AbstractBeanDefinition pointcutDefinition = **null**;

**try** {

**this**.parseState.push(**new** PointcutEntry(id));

pointcutDefinition = createPointcutDefinition(expression);

pointcutDefinition.setSource(parserContext.extractSource(pointcutElement));

String pointcutBeanName = id;

**if** (StringUtils.*hasText*(pointcutBeanName)) {

parserContext.getRegistry().registerBeanDefinition(pointcutBeanName, pointcutDefinition);

} **else** {

pointcutBeanName = parserContext.getReaderContext().registerWithGeneratedName(pointcutDefinition);

}

parserContext.registerComponent(**new** PointcutComponentDefinition(pointcutBeanName, pointcutDefinition, expression));

} **finally** {

**this**.parseState.pop();

}

**return** pointcutDefinition;

}

1、获取id和expression属性值。并创建AspectJExpressionPointcut类对应的Bean定义。

**protected** AbstractBeanDefinition createPointcutDefinition(String expression) {

RootBeanDefinition beanDefinition = **new** RootBeanDefinition(**AspectJExpressionPointcut**.**class**);

beanDefinition.setScope(BeanDefinition.***SCOPE\_PROTOTYPE***);

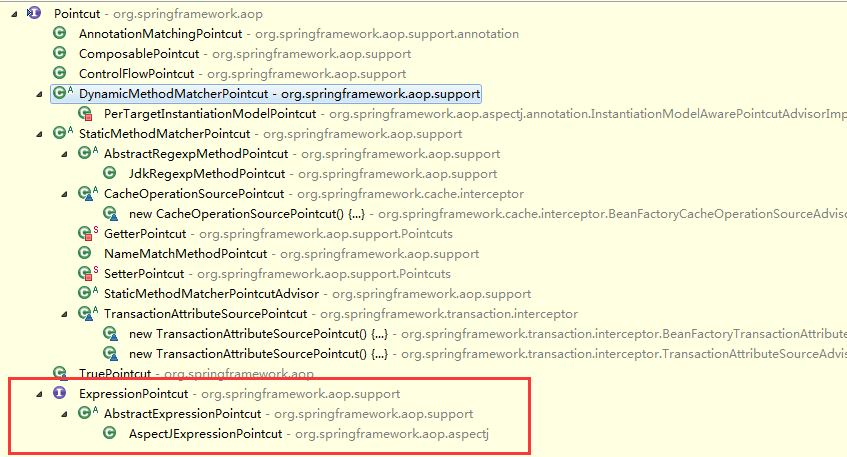
beanDefinition.setSynthetic(**true**); // 表名此bean定义为合成物

beanDefinition.getPropertyValues().add(***EXPRESSION***, expression); // 将表达式设置到bean定义的expression字段

**return** beanDefinition;

}

2、将**AspectJExpressionPointcut**类对应的bean定义注册到BeanFactory中，并根据AspectJExpressionPointcut的bean定义构建PointcutComponentDefinition。



##### parseAdvisor(Element, ParserContext)

文件位置：org/springframework/aop/config/ConfigBeanDefinitionParser.java

**private** **void** parseAdvisor(Element advisorElement, ParserContext parserContext) {

AbstractBeanDefinition advisorDef = createAdvisorBeanDefinition(advisorElement, parserContext);

String id = advisorElement.getAttribute(***ID***);

**try** {

**this**.parseState.push(**new** AdvisorEntry(id));

String advisorBeanName = id;

**if** (StringUtils.*hasText*(advisorBeanName)) {

parserContext.getRegistry().registerBeanDefinition(advisorBeanName, advisorDef);

} **else** {

advisorBeanName = parserContext.getReaderContext().registerWithGeneratedName(advisorDef);

}

Object pointcut = parsePointcutProperty(advisorElement, parserContext);

**if** (pointcut **instanceof** BeanDefinition) {

advisorDef.getPropertyValues().add(***POINTCUT***, pointcut);

parserContext.registerComponent(**new** AdvisorComponentDefinition(advisorBeanName, advisorDef, (BeanDefinition) pointcut));

} **else** **if** (pointcut **instanceof** String) {

advisorDef.getPropertyValues().add(***POINTCUT***, **new** RuntimeBeanReference((String) pointcut));

parserContext.registerComponent(**new** AdvisorComponentDefinition(advisorBeanName, advisorDef));

}

} **finally** {

**this**.parseState.pop();

}

}

1、调用createAdvisorBeanDefinition方法创建<aop:advisor/>对应的bean[**DefaultBeanFactoryPointcutAdvisor**，详细见[Advisor](#_Advisor)]定义，获取” advice-ref”属性，并根据属性值构建RuntimeBeanNameReference，最后将RuntimeBeanNameReference设置到**DefaultBeanFactoryPointcutAdvisor**的adviceBeanName字段；获取order属性值，设置到**DefaultBeanFactoryPointcutAdvisor**的order字段。

2、判断<aop:advisor/>是否定义有id属性，如果有，**DefaultBeanFactoryPointcutAdvisor**对应的beanName为id属性值，否则自动生成name。

3、解析pointcut和pointcut-ref属性，两者不能同时存在。如果是pointcut属性，则想[parsePointcut(Element, ParserContext)](#_parsePointcut(Element,_ParserContex)解析，并创建**AspectJExpressionPointcut**对应的bean定义，并将bean定义添加到DefaultBeanFactoryPointcutAdvisor的pointcut字段；如果是pointcut-ref，则返回引用的pointcut对应的id值，并根据id值构建RuntimeBeanReference.

4、最后将**AspectJExpressionPointcut**对应的bean定义包装成AdvisorComponentDefinition添加到BeanFactory中。

##### parseAspect(Element, ParserContext)

文件位置：org/springframework/aop/config/ConfigBeanDefinitionParser.java

**private** **void** parseAspect(Element aspectElement, ParserContext parserContext) {

String aspectId = aspectElement.getAttribute(***ID***);

String aspectName = aspectElement.getAttribute(***REF***);

**try** {

**this**.parseState.push(**new** AspectEntry(aspectId, aspectName));

List<BeanDefinition> beanDefinitions = **new** ArrayList<BeanDefinition>();

List<BeanReference> beanReferences = **new** ArrayList<BeanReference>();

List<Element> declareParents = DomUtils.*getChildElementsByTagName*(aspectElement, ***DECLARE\_PARENTS***);

**for** (**int** i = ***METHOD\_INDEX***; i < declareParents.size(); i++) {

Element declareParentsElement = declareParents.get(i);

beanDefinitions.add(parseDeclareParents(declareParentsElement, parserContext));

}

// We have to parse "advice" and all the advice kinds in one loop, to get the

// ordering semantics right.

NodeList nodeList = aspectElement.getChildNodes();

**boolean** adviceFoundAlready = **false**;

**for** (**int** i = 0; i < nodeList.getLength(); i++) {

Node node = nodeList.item(i);

**if** (isAdviceNode(node, parserContext)) {

**if** (!adviceFoundAlready) {

adviceFoundAlready = **true**;

**if** (!StringUtils.*hasText*(aspectName)) {

parserContext.getReaderContext().error("<aspect> tag needs aspect bean reference via 'ref' attribute when declaring advices.", aspectElement, **this**.parseState.snapshot());

**return**;

}

beanReferences.add(**new** RuntimeBeanReference(aspectName));

}

AbstractBeanDefinition advisorDefinition = parseAdvice(aspectName, i, aspectElement, (Element) node, parserContext, beanDefinitions, beanReferences);

beanDefinitions.add(advisorDefinition);

}

}

AspectComponentDefinition aspectComponentDefinition = createAspectComponentDefinition(

aspectElement, aspectId, beanDefinitions, beanReferences, parserContext);

parserContext.pushContainingComponent(aspectComponentDefinition);

List<Element> pointcuts = DomUtils.*getChildElementsByTagName*(aspectElement, ***POINTCUT***);

**for** (Element pointcutElement : pointcuts) {

parsePointcut(pointcutElement, parserContext);

}

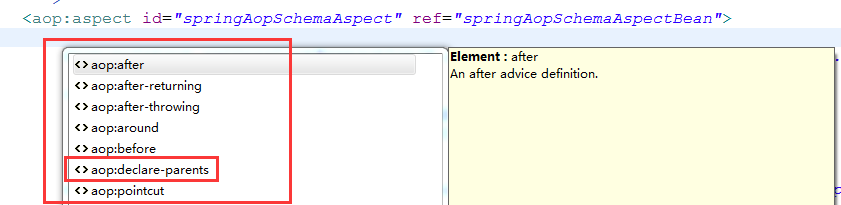
parserContext.popAndRegisterContainingComponent();

} **finally** {

**this**.parseState.pop();

}

}



从上图可以看出，<aop:aspect/>下可以声明<aop:declare-parents/>、<aop:pointcut/>、<aop:before/>、<aop:around/>、<aop:after-throwing/>、<aop:after-returning/>和<aop:after/>七个子元素，上述方法也是对这七个子元素进行解析，然后生成对应的bean定义。

**<aop:declare-parents/>**

**<aop:declare-parents/>**用于定义引用(Introduction)，引用(在ApectJ中被称为inter-type声明)使得一个切面可以定义被通知对象实现给定的接口,并且可以为那些对象提供具体的实现。

List<Element> declareParents = DomUtils.*getChildElementsByTagName*(aspectElement, ***DECLARE\_PARENTS***);

**for** (**int** i = ***METHOD\_INDEX***; i < declareParents.size(); i++) {

Element declareParentsElement = declareParents.get(i);

beanDefinitions.add(parseDeclareParents(declareParentsElement, parserContext));

}

从这段代码可以看出，<aop:aspect/>下可以有多个<aop:declare-parents/>子元素，对<aop:declare-parents/>元素的解析是由parseDeclareParents方法完成，如下：

**private** AbstractBeanDefinition parseDeclareParents(Element declareParentsElement,ParserContext parserContext){

BeanDefinitionBuilder builder = BeanDefinitionBuilder.*rootBeanDefinition*(**DeclareParentsAdvisor**.**class**);

builder.addConstructorArgValue(declareParentsElement.getAttribute(***IMPLEMENT\_INTERFACE***));

builder.addConstructorArgValue(declareParentsElement.getAttribute(***TYPE\_PATTERN***));

String defaultImpl = declareParentsElement.getAttribute(***DEFAULT\_IMPL***);

String delegateRef = declareParentsElement.getAttribute(***DELEGATE\_REF***);

**if** (StringUtils.*hasText*(defaultImpl) && !StringUtils.*hasText*(delegateRef)) {

builder.addConstructorArgValue(defaultImpl);

} **else** **if** (StringUtils.*hasText*(delegateRef) && !StringUtils.*hasText*(defaultImpl)) {

builder.addConstructorArgReference(delegateRef);

} **else** {

parserContext.getReaderContext().error("Exactly one of the " + ***DEFAULT\_IMPL*** + " or " + ***DELEGATE\_REF*** + " attributes must be specified", declareParentsElement, **this**.parseState.snapshot());

}

AbstractBeanDefinition definition = builder.getBeanDefinition();

definition.setSource(parserContext.extractSource(declareParentsElement));

parserContext.getReaderContext().registerWithGeneratedName(definition);

**return** definition;

}

可以看出，Spring为<aop:declare-parents/>元素生成了一个DeclareParentsAdvisor对应的bean定义。<aop:declare-parents/>下存在implement-interface、types-matching、default-impl和delegate-ref四个属性，这四个属性通过构造器的方式注入到DeclareParentsAdvisor对应的bean中。

**<aop:before />、<aop:after/>、<aop:after-returning/>、<aop:after-throwing/>和<aop:around/>**

NodeList nodeList = aspectElement.getChildNodes();

**boolean** adviceFoundAlready = **false**;

**for** (**int** i = 0; i < nodeList.getLength(); i++) {

Node node = nodeList.item(i);

**if** (isAdviceNode(node, parserContext)) {

**if** (!adviceFoundAlready) {

adviceFoundAlready = **true**;

**if** (!StringUtils.*hasText*(aspectName)) {parserContext.getReaderContext().error("<aspect> tag needs aspect bean reference via 'ref' attribute when declaring advices.", aspectElement, **this**.parseState.snapshot());

**return**;

}

beanReferences.add(**new** RuntimeBeanReference(aspectName));

}

AbstractBeanDefinition advisorDefinition = parseAdvice(aspectName, i, aspectElement, (Element) node, parserContext, beanDefinitions, beanReferences);

beanDefinitions.add(advisorDefinition);

}

}

判断<aop:aspect/>是否存在通知标签，如果存在则将”切面名称[aspectName]”封装成RuntimeBeanReference.解析所有通知元素，详细见[parseAdvice(…)](#_parseAdvice(…))

AspectComponentDefinition aspectComponentDefinition = createAspectComponentDefinition(aspectElement, aspectId, **beanDefinitions**, **beanReferences**, parserContext);

parserContext.pushContainingComponent(aspectComponentDefinition);

将面解析出来的BeanDefiition和BeanReference封装成**AspectComponentDefinition**，并添加到BeanFactory中。

**<aop:pointcut/>**

List<Element> pointcuts = DomUtils.*getChildElementsByTagName*(aspectElement, ***POINTCUT***);

**for** (Element pointcutElement : pointcuts) {

parsePointcut(pointcutElement, parserContext);

}

迭代<aop:aspect/>下面所有的内置<aop:pointcut/>元素，详细见[parsePointcut(Element, ParserContext)](#_parsePointcut(Element,_ParserContex)。

##### parseAdvice(…)

文件位置：org/springframework/aop/config/ConfigBeanDefinitionParser.java

**private** AbstractBeanDefinition parseAdvice(String aspectName, **int** order, Element aspectElement, Element adviceElement, ParserContext parserContext, List<BeanDefinition> beanDefinitions, List<BeanReference> beanReferences) {

**try** {

**this**.parseState.push(**new** AdviceEntry(parserContext.getDelegate().getLocalName(adviceElement)));

// create the method factory bean

RootBeanDefinition methodDefinition = **new** RootBeanDefinition(**MethodLocatingFactoryBean**.**class**);

methodDefinition.getPropertyValues().add("targetBeanName", aspectName);

methodDefinition.getPropertyValues().add("methodName", adviceElement.getAttribute("method"));

methodDefinition.setSynthetic(**true**);

// create instance factory definition

RootBeanDefinition aspectFactoryDef = **new** RootBeanDefinition(**SimpleBeanFactoryAwareAspectInstanceFactory**.**class**);

aspectFactoryDef.getPropertyValues().add("aspectBeanName", aspectName);

aspectFactoryDef.setSynthetic(**true**);

// register the pointcut

AbstractBeanDefinition adviceDef = createAdviceDefinition(adviceElement, parserContext, aspectName, order, methodDefinition, aspectFactoryDef, beanDefinitions, beanReferences);

// configure the advisor

RootBeanDefinition advisorDefinition = **new** RootBeanDefinition(**AspectJPointcutAdvisor**.**class**);

advisorDefinition.setSource(parserContext.extractSource(adviceElement));

advisorDefinition.getConstructorArgumentValues().addGenericArgumentValue(adviceDef);

**if** (aspectElement.hasAttribute(***ORDER\_PROPERTY***)) {

advisorDefinition.getPropertyValues().add(***ORDER\_PROPERTY***, aspectElement.getAttribute(***ORDER\_PROPERTY***));

}

// register the final advisor

parserContext.getReaderContext().registerWithGeneratedName(advisorDefinition);

**return** advisorDefinition;

} **finally** {

**this**.parseState.pop();

}

}

1、创建切面通知方法对应的“工厂bean”**MethodLocatingFactoryBean**。需要将**切面aspect**和**通知对应的方法**注入到**MethodLocatingFactoryBean**中的**targetBeanName**、**methodName**字段。

2、创建切面工厂定义**SimpleBeanFactoryAwareAspectInstanceFactory**。需要将切面aspect注入到**MethodLocatingFactoryBean**中的**aspectBeanName**字段。

3、根据第1步的“工厂bean”和第2步“工厂定义”，调用[createAdviceDefinition(…)](#_createAdviceDefinition(…))方法创建通知定义。

4、创建**AspectJPointcutAdvisor**对应的bean定义，并将“第3步”通过构造器注入到**AspectJPointcutAdvisor**中。获取对应<aop:aspect/>的order属性值，对**AspectJPointcutAdvisor**进行排序控制。最后将**AspectJPointcutAdvisor**对应的bean定义注入到BeanFactory中。

Spring会为每一个通知都执行此方法，也就是会针对每一个通知都生成**AspectJPointcutAdvisor**对应的bean定义，这个会在[shouldSkip(Class<?> beanClass, String beanName)](#_shouldSkip(Class<?>_beanClass,_Stri)用到。

##### createAdviceDefinition(…)

**private** AbstractBeanDefinition createAdviceDefinition(Element adviceElement, ParserContext parserContext, String aspectName, **int** order, RootBeanDefinition methodDef, RootBeanDefinition aspectFactoryDef, List<BeanDefinition> beanDefinitions, List<BeanReference> beanReferences) {

RootBeanDefinition adviceDefinition = **new** RootBeanDefinition(getAdviceClass(adviceElement, parserContext));

adviceDefinition.setSource(parserContext.extractSource(adviceElement));

adviceDefinition.getPropertyValues().add(***ASPECT\_NAME\_PROPERTY***, aspectName);

adviceDefinition.getPropertyValues().add(***DECLARATION\_ORDER\_PROPERTY***, order);

**if** (adviceElement.hasAttribute(***RETURNING***)) {

adviceDefinition.getPropertyValues().add(***RETURNING\_PROPERTY***,adviceElement.getAttribute(***RETURNING***));

}

**if** (adviceElement.hasAttribute(***THROWING***)) {

adviceDefinition.getPropertyValues().add(***THROWING\_PROPERTY***, adviceElement.getAttribute(***THROWING***));

}

**if** (adviceElement.hasAttribute(***ARG\_NAMES***)) {

adviceDefinition.getPropertyValues().add(***ARG\_NAMES\_PROPERTY***,adviceElement.getAttribute(***ARG\_NAMES***));

}

ConstructorArgumentValues cav = adviceDefinition.getConstructorArgumentValues();

cav.addIndexedArgumentValue(***METHOD\_INDEX***, methodDef);

Object pointcut = parsePointcutProperty(adviceElement, parserContext);

**if** (pointcut **instanceof** BeanDefinition) {

cav.addIndexedArgumentValue(***POINTCUT\_INDEX***, pointcut);

beanDefinitions.add((BeanDefinition) pointcut);

} **else** **if** (pointcut **instanceof** String) {

RuntimeBeanReference pointcutRef = **new** RuntimeBeanReference((String) pointcut);

cav.addIndexedArgumentValue(***POINTCUT\_INDEX***, pointcutRef);

beanReferences.add(pointcutRef);

}

cav.addIndexedArgumentValue(***ASPECT\_INSTANCE\_FACTORY\_INDEX***, aspectFactoryDef);

**return** adviceDefinition;

}

1. 通过通知类型获取对应的“**通知bean定义**”。

**private** Class<?> getAdviceClass(Element adviceElement, ParserContext parserContext) {

String elementName = parserContext.getDelegate().getLocalName(adviceElement);

**if** (***BEFORE***.equals(elementName)) { // 前置通知

**return** **AspectJMethodBeforeAdvice**.**class**;

} **else** **if** (***AFTER***.equals(elementName)) { // 最终通知

**return** **AspectJAfterAdvice**.**class**;

} **else** **if** (***AFTER\_RETURNING\_ELEMENT***.equals(elementName)) { // 后置通知

**return** **AspectJAfterReturningAdvice**.**class**;

} **else** **if** (***AFTER\_THROWING\_ELEMENT***.equals(elementName)) { // 异常通知

**return** **AspectJAfterThrowingAdvice**.**class**;

} **else** **if** (***AROUND***.equals(elementName)) { // 环绕通知

**return** **AspectJAroundAdvice**.**class**;

} **else** {

**throw** **new** IllegalArgumentException("Unknown advice kind [" + elementName + "].");

}

} // 详细见[Advice](#_Advice)

2、将切面和优先级分别注入到”**通知bean定义**”的aspectName和declarationOrder字段。优先级根据在schema中的定义顺序决定。

3、判断是否存在”returning”属性，如果存在则将属性值注入到”**通知bean定义**”的returningNam字段；判断是否存在”arg-names”属性，如果存在则将属性值注入到”**通知bean定义**”的argumentNames字段；判断是否存在”throwing”属性，如果存在则将属性值注入到”**通知bean定义**”的throwingName字段。

4、获取切入点定义，并连同前面获得的MethodLocatingFactoryBean和**SimpleBeanFactoryAwareAspectInstanceFactory**对应的bean定义注入到”通知bean”的构造器参数中。

### Spring AOP在IoC容器中的实例化过程

通过[基于Schema的AOP支持](#_基于Schema的AOP支持)章节的分析我们可以知道，在Spring对Schema的解析过程中，会产生以下类对应的BeanDefiition。

AspectJAwareAdvisorAutoProxyCreator --> 自动代理代理创建器

AspectJExpressionPointcut --><aop:pointcut>切入点

DefaultBeanFactoryPointcutAdvisor --><aop:advisor> 自包含的切面，这个切面只有一个通知。

<aop:aspect/>

MethodLocatingFactoryBean

SimpleBeanFactoryAwareAspectInstanceFactory

**AspectJPointcutAdvisor**

DeclareParentsAdvisor --> <aop:declare-parents /> 引用

AspectJMethodBeforeAdvice --> <aop:before /> 前置通知

AspectJAfterAdvice --> <aop:after /> 最终通知

AspectJAfterReturningAdvice --> <aop:after-returning/> 后置通知

AspectJAfterThrowingAdvice --> <aop:after-throwing/> 异常通知

AspectJAroundAdvice --> <aop:around/> 环绕通知

在Spring IoC容器初始化的[createBean(String,RootBeanDefinition,Object[])](#_createBean(String,RootBeanDefinitio)方法中会调用[resolveBeforeInstantiation](#_resolveBeforeInstantiation)方法来创建一个基于AOP的代理对象。接下来我们看看[resolveBeforeInstantiation](#_resolveBeforeInstantiation)方法的实现。

**protected** Object resolveBeforeInstantiation(String beanName, RootBeanDefinition mbd) {

Object bean = **null**;

**if** (!Boolean.***FALSE***.equals(mbd.beforeInstantiationResolved)) {

// Make sure bean class is actually resolved at this point.

**if** (mbd.hasBeanClass() && !mbd.isSynthetic() && hasInstantiationAwareBeanPostProcessors()) {

bean = applyBeanPostProcessorsBeforeInstantiation(mbd.getBeanClass(), beanName);

**if** (bean != **null**) {

bean = applyBeanPostProcessorsAfterInitialization(bean, beanName);

}

}

mbd.beforeInstantiationResolved = (bean != **null**);

}

**return** bean;

}

1、从上面的逻辑看，在解析AOP Schema中产生的BeanDefiition中，**MethodLocatingFactoryBean**、**SimpleBeanFactoryAwareAspectInstanceFactory**和**AspectJExpressionPointcut**会不进入此段逻辑，因为它们的BeanDefiition中synthetic字段的值为true。

2、调用applyBeanPostProcessorsBeforeInstantiation方法做实例化的前置处理。

**protected** Object applyBeanPostProcessorsBeforeInstantiation(Class<?> beanClass, String beanName) **throws** BeansException {

**for** (BeanPostProcessor bp : getBeanPostProcessors()) {

**if** (bp **instanceof** InstantiationAwareBeanPostProcessor) {

InstantiationAwareBeanPostProcessor ibp = (InstantiationAwareBeanPostProcessor) bp;

Object result = ibp.postProcessBeforeInstantiation(beanClass, beanName);

**if** (result != **null**) {

**return** result;

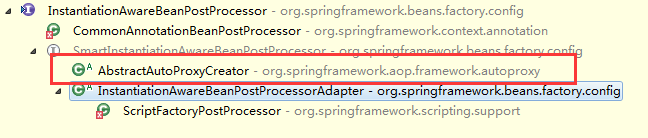
}

}

}

**return** **null**;

}



3、调用applyBeanPostProcessorsAfterInitialization方法做实例化后置处理。

**public** Object applyBeanPostProcessorsAfterInitialization(Object existingBean, String beanName) **throws** BeansException {

Object result = existingBean;

**for** (BeanPostProcessor beanProcessor : getBeanPostProcessors()) {

result = beanProcessor.postProcessAfterInitialization(result, beanName);

**if** (result == **null**) {

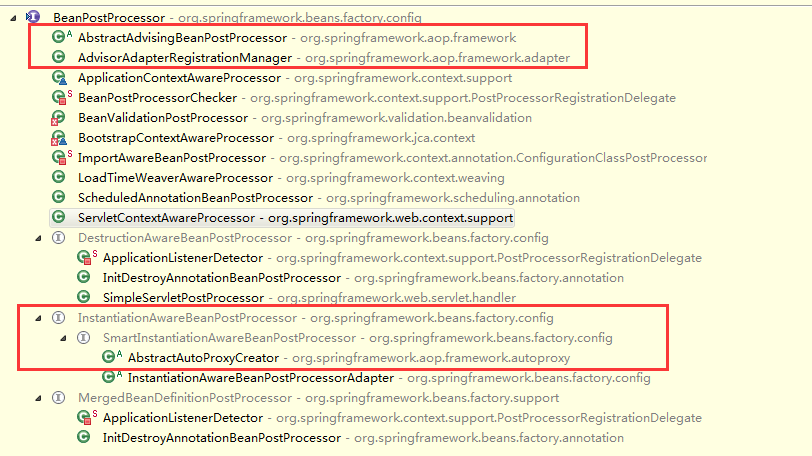
**return** result;

}

}

**return** result;

}



从上面的逻辑可以看出，**AbstractAutoProxyCreator**对postProcessBeforeInstantiation和postProcessAfterInitialization方法做了实现，在[Spring AOP的xml Schema解析](#_Spring_AOP的xml_Schema解析)中有提到会在BeanFactory中会注册**AspectJAwareAdvisorAutoProxyCreator**对应的bean定义，**AspectJAwareAdvisorAutoProxyCreator**实现了**AbstractAutoProxyCreator**抽象类。

#### postProcessBeforeInstantiation(Class<?>, String)

文件位置：org/springframework/aop/framework/autoproxy/AbstractAutoProxyCreator.java

**public** Object postProcessBeforeInstantiation(Class<?> beanClass, String beanName) **throws** BeansException {

Object cacheKey = getCacheKey(beanClass, beanName);

**if** (beanName == **null** || !**this**.targetSourcedBeans.contains(beanName)) {

**if** (**this**.advisedBeans.containsKey(cacheKey)) {

**return** **null**;

}

**if** (isInfrastructureClass(beanClass) || shouldSkip(beanClass, beanName)) {

**this**.advisedBeans.put(cacheKey, Boolean.***FALSE***);

**return** **null**;

}

}

// Create proxy here if we have a custom TargetSource.

// Suppresses unnecessary default instantiation of the target bean:

// The TargetSource will handle target instances in a custom fashion.

**if** (beanName != **null**) {

TargetSource targetSource = getCustomTargetSource(beanClass, beanName);

**if** (targetSource != **null**) {

**this**.targetSourcedBeans.add(beanName);

Object[] specificInterceptors = getAdvicesAndAdvisorsForBean(beanClass, beanName, targetSource);

Object proxy = createProxy(beanClass, beanName, specificInterceptors, targetSource);

**this**.proxyTypes.put(cacheKey, proxy.getClass());

**return** proxy;

}

}

**return** **null**;

}

1、isInfrastructureClass()方法用于判断需要处理的bean对应的class是否为AOP的基础功能类，这些基础功能类对应的类型为Advice、Advisor和AopInfrastructureBean。

**protected** **boolean** isInfrastructureClass(Class<?> beanClass) {

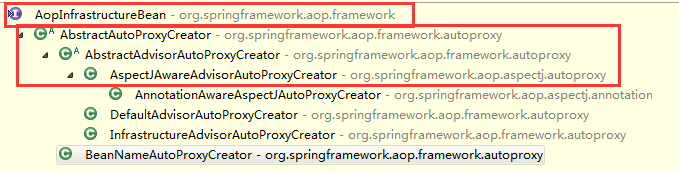
**boolean** retVal = Advice.**class**.isAssignableFrom(beanClass) ||

Advisor.**class**.isAssignableFrom(beanClass) ||

AopInfrastructureBean.**class**.isAssignableFrom(beanClass);

**return** retVal;

}



2、如果不为AOP基础功能类，调用shouldSkip()做进一步判断，AbstractAutoProxyCreator对shouldSkip()实现是默认返回false，但是其子类AspectJAwareAdvisorAutoProxyCreator对其进行了实现。详细见[shouldSkip(Class<?> beanClass, String beanName)](#_shouldSkip(Class<?>_beanClass,_Stri)

如果1和2中其中有一个返回结果为true，则需要将对应的bean信息加入到**AbstractAutoProxyCreator**的**advisedBeans**字段。

3、调用[getAdvicesAndAdvisorsForBean(Class<?>, String, TargetSource)](#_getAdvicesAndAdvisorsForBean(Class<)为目标对象匹配通知。返回通知对应的advisor列表。

4、调用[createProxy(Class<?>,String,Object[],TargetSource)](#_createProxy(Class<?>,String,Object[)创建AOP 代理对象。

#### shouldSkip(Class<?> beanClass, String beanName)

文件位置：org/springframework/aop/aspectj/autoproxy/AspectJAwareAdvisorAutoProxyCreator.java

**protected** **boolean** shouldSkip(Class<?> beanClass, String beanName) {

List<Advisor> candidateAdvisors = findCandidateAdvisors();

**for** (Advisor advisor : candidateAdvisors) {

**if** (advisor **instanceof** AspectJPointcutAdvisor) {

**if** (((AbstractAspectJAdvice) advisor.getAdvice()).getAspectName().equals(beanName)) {

**return** **true**;

}

}

}

**return** **super**.shouldSkip(beanClass, beanName);

}

1、调用findCandidateAdvisors()方法查找候选的Advisor，在[parseAdvice(…)](#_parseAdvice(…))我们谈到，Spring会跌倒Schema中的通知声明，并为每个通知生成一个对应的AspectJPointcutAdvisor，findCandidateAdvisors()方法会调用BeanFactory的getBean()实例化所有的通知(如果没有实例化的话)。

2、如果当前bean对应的beanName为通知对应的beanName,及表示在上一步已经实例化，无需在实例化，返回true。

#### getAdvicesAndAdvisorsForBean(Class<?>, String, TargetSource)

文件位置：org/springframework/aop/framework/autoproxy/AbstractAdvisorAutoProxyCreator.java

**protected** Object[] getAdvicesAndAdvisorsForBean(Class<?> beanClass,String beanName,TargetSource targetSource) {

List<Advisor> advisors = findEligibleAdvisors(beanClass, beanName);

**if** (advisors.isEmpty()) {

**return** ***DO\_NOT\_PROXY***;

}

**return** advisors.toArray();

}

返回给定bean是否需要被代理，返回与之匹配的advices和advisors。

详细见[findEligibleAdvisors(Class<?>, String)](#_findEligibleAdvisors(Class<?>,_Stri)

#### findEligibleAdvisors(Class<?>, String)

文件位置：org/springframework/aop/framework/autoproxy/AbstractAdvisorAutoProxyCreator.java

**protected** List<Advisor> findEligibleAdvisors(Class<?> beanClass, String beanName) {

List<Advisor> candidateAdvisors = findCandidateAdvisors();

List<Advisor> eligibleAdvisors = findAdvisorsThatCanApply(candidateAdvisors, beanClass, beanName);

extendAdvisors(eligibleAdvisors);

**if** (!eligibleAdvisors.isEmpty()) {

eligibleAdvisors = sortAdvisors(eligibleAdvisors);

}

**return** eligibleAdvisors;

}

1、findCandidateAdvisors(…)方法的逻辑是用来获取Spring IoC中所有类型Advisor的bean，在此方法中会实例化所有的Advisor。

2、findAdvisorsThatCanApply(…)方法是根据”步骤1”中获得所有候选Advisor，对预实例化的bean进行根据切入点表达式进行匹配。返回适合此bean的所有Advisor。匹配逻辑见[canApply(Advisor, Class<?>, boolean)](#_canApply(Advisor,_Class<?>,_boolean)

3、调用AspectJProxyUtils.makeAdvisorChainAspectJCapableIfNecessary(…)在”步骤2”的基础上增加DefaultPointcutAdvisor，对应的构造器参数为ExposeInvocationInterceptor。

4、对适合bean的Advisor进行排序，排序规则在**AspectJPrecedenceComparator**中实现。

5、根据AspectJPrecedenceComparator中的排序算法进行排序。

#### canApply(Advisor, Class<?>, boolean)

文件位置：org/springframework/aop/support/AopUtils.java

**public** **static** **boolean** canApply(Advisor advisor, Class<?> targetClass, **boolean** hasIntroductions) {

**if** (advisor **instanceof** IntroductionAdvisor) {

**return** ((IntroductionAdvisor) advisor).getClassFilter().matches(targetClass);

} **else** **if** (advisor **instanceof** PointcutAdvisor) {

PointcutAdvisor pca = (PointcutAdvisor) advisor;

**return** *canApply*(pca.getPointcut(), targetClass, hasIntroductions);

} **else** {

// It doesn't have a pointcut so we assume it applies.

**return** **true**;

}

}

1、如果advisor为IntroductionAdvisor[引用]，在[parseAspect(Element, ParserContext)](#_parseAspect(Element,_ParserContext))中Spring在解析<aop:declare-parents />时注册了**DeclareParentsAdvisor**对应的bean定义(见ConfigBeanDefinitionParser类的parseDeclareParents(…)方法)。从parseDeclareParents(…)逻辑来看，我们需要关注DeclareParentsAdvisor类的构造器，因为在Spring IoC容器实例化时，会通过DeclareParentsAdvisor构造器实例化相关信息。

**构造器一：**

**public** DeclareParentsAdvisor(Class<?> interfaceType, String typePattern, Class<?> defaultImpl) {

**this**(interfaceType, typePattern, defaultImpl, **new** DelegatePerTargetObjectIntroductionInterceptor(defaultImpl, interfaceType));

}

**构造器二：**

**public** DeclareParentsAdvisor(Class<?> interfaceType, String typePattern, Object delegateRef) {

**this**(interfaceType, typePattern, delegateRef.getClass(), **new** DelegatingIntroductionInterceptor(delegateRef));

}

**构造器一和构造器二都会调用如下私有构造器。**

**private** DeclareParentsAdvisor(Class<?> interfaceType, String typePattern, Class<?> implementationClass, Advice advice) {

**this**.introducedInterface = interfaceType;

ClassFilter typePatternFilter = **new** TypePatternClassFilter(typePattern);

// Excludes methods implemented.

ClassFilter exclusion = **new** ClassFilter() {

@Override

**public** **boolean** matches(Class<?> clazz) {

**return** !(introducedInterface.isAssignableFrom(clazz));

}

};

**this**.typePatternClassFilter = ClassFilters.*intersection*(typePatternFilter, exclusion);

**this**.advice = advice;

}

我们再回过头来看匹配逻辑：

((IntroductionAdvisor) advisor).getClassFilter().matches(targetClass);

getClassFilter()方法获取的是DeclareParentsAdvisor的typePatternClassFilter字段值，typePatternClassFilter的实际类型是IntersectionClassFilter，IntersectionClassFilter包含了两个ClassFilter数组，也就是上面**私有构造器中**定义的两个ClassFilter，分别对应<aop:declare-parents />中的types-matching和implement-interface属性值。

2、Spring在解析<aop:aspect />元素的时候，为每一个通知都注册了对应的AspectJPointcutAdvisor bean定义.因为AspectJPointcutAdvisor实现了PointcutAdvisor接口，所里Spring会继续调用

[canApply(Pointcut, Class<?>, boolean)](#_canApply(Pointcut,_Class<?>,_boolea)方法完成匹配逻辑。

#### canApply(Pointcut, Class<?>, boolean)

**public** **static** **boolean** canApply(Pointcut pc, Class<?> targetClass, **boolean** hasIntroductions) {

Assert.*notNull*(pc, "Pointcut must not be null");

**if** (!pc.getClassFilter().matches(targetClass)) {

**return** **false**;

}

MethodMatcher methodMatcher = pc.getMethodMatcher();

IntroductionAwareMethodMatcher introductionAwareMethodMatcher = **null**;

**if** (methodMatcher **instanceof** IntroductionAwareMethodMatcher) {

introductionAwareMethodMatcher = (IntroductionAwareMethodMatcher) methodMatcher;

}

Set<Class<?>> classes = **new** HashSet<Class<?>>(ClassUtils.*getAllInterfacesForClassAsSet*(targetClass));

classes.add(targetClass);

**for** (Class<?> clazz : classes) {

Method[] methods = clazz.getMethods();

**for** (Method method : methods) {

**if** ((introductionAwareMethodMatcher != **null** &&

introductionAwareMethodMatcher.matches(method, targetClass, hasIntroductions))

|| methodMatcher.matches(method, targetClass)) {

**return** **true**;

}

}

}

**return** **false**;

}

1、首先根据Pointcut的matches对应的目标对象class，因为Spring为每一个<aop:pointcut/>都注册了对应的AspectJExpressionPointcut定义，所以这里调用了AspectJExpressionPointcut的matches(Class)方法对目标对象class进行匹配。这是**类级别**的匹配。

2、如果类级别的匹配通过，则**进行方法级别的匹配**，因为AspectJExpressionPointcut实现了IntroductionAwareMethodMatcher接口。所以在方法级匹配的时候首先调用matches(Method, Class<?>, boolean)方法进行匹配，如果匹配不通过则调用matches(Method, Class<?>)方法匹配。

#### createProxy(Class<?>,String,Object[],TargetSource)

文件位置：org/springframework/aop/framework/autoproxy/AbstractAutoProxyCreator.java

**protected** Object createProxy(Class<?> beanClass, String beanName, Object[] specificInterceptors, TargetSource targetSource) {

ProxyFactory proxyFactory = **new** ProxyFactory();

// Copy our properties (proxyTargetClass etc) inherited from ProxyConfig.

proxyFactory.copyFrom(**this**);

**if** (!proxyFactory.isProxyTargetClass()) {

**if** (shouldProxyTargetClass(beanClass, beanName)) {

proxyFactory.setProxyTargetClass(**true**);

} **else** {

evaluateProxyInterfaces(beanClass, proxyFactory);

}

}

Advisor[] advisors = buildAdvisors(beanName, specificInterceptors);

**for** (Advisor advisor : advisors) {

proxyFactory.addAdvisor(advisor);

}

proxyFactory.setTargetSource(targetSource);

customizeProxyFactory(proxyFactory);

proxyFactory.setFrozen(**this**.freezeProxy);

**if** (advisorsPreFiltered()) {

proxyFactory.setPreFiltered(**true**);

}

**return** proxyFactory.getProxy(**this**.proxyClassLoader);

}

1、创建Spring AOP工厂类ProxyFactory,并从AbstractAutoProxyCreator[继承了ProxyConfig类]中复制相关属性。

**public** **void** copyFrom(ProxyConfig other) {

**this**.proxyTargetClass = other.proxyTargetClass;

**this**.optimize = other.optimize;

**this**.exposeProxy = other.exposeProxy;

**this**.frozen = other.frozen;

**this**.opaque = other.opaque;

}

在解析<aop:config/>标签的时候，会注册AspectJAwareAdvisorAutoProxyCreator对应的bean定义，并获取其<aop:config/>中proxy-target-class和expose-proxy属性值，并设置到其父类[ProxyConfig]proxyTargetClass和exposeProxy字段。

2、判断ProxyFactory的proxyTargetClass字段值是否为false，如果为false需要进行一下逻辑。

**if** (!proxyFactory.isProxyTargetClass()) {

**if** (shouldProxyTargetClass(beanClass, beanName)) {

proxyFactory.setProxyTargetClass(**true**);

} **else** {

evaluateProxyInterfaces(beanClass, proxyFactory);

}

}

2.1 shouldProxyTargetClass(…)方法的逻辑主要是判断bean定义信息中” AutoProxyUtils.preserveTargetClass”属性值是否为true。  
2.2 evaluateProxyInterfaces(…)方法逻辑如下：

2.2.1 获取bean对应的class实现的所有接口

2.2.2 如果bean对应的接口存在不是InitializingBean、DisposableBean和Aware接口，且对应接口定义了方法，则将bean实现的所有接口加入到ProxyFactory中，否则将ProxyFactory的proxyTargetClass设置为true。

3、获取所有匹配的Advisor，并将其添加到ProxyFactory中advisors字段。

4、调用[getProxy(ClassLoader)](#_getProxy(ClassLoader))获取代理对象

#### getProxy(ClassLoader)

文件位置：org/springframework/aop/framework/ProxyFactory.java

**public** Object getProxy(ClassLoader classLoader) {

**return** createAopProxy().getProxy(classLoader);

}

1、创建AopProxy,Spring提供了基于JDK动态代理和CGLIB的AopProxy方式.详细见[AopProxy](#_AopProxy)。

**public** AopProxy createAopProxy(AdvisedSupport config) **throws** AopConfigException {

**if** (config.isOptimize() || config.isProxyTargetClass() || hasNoUserSuppliedProxyInterfaces(config)) {

Class<?> targetClass = config.getTargetClass();

**if** (targetClass == **null**) {

**throw** **new** AopConfigException("TargetSource cannot determine target class: " + "Either an interface or a target is required for proxy creation.");

}

**if** (targetClass.isInterface()) {

**return** **new** JdkDynamicAopProxy(config);

}

**return** **new** ObjenesisCglibAopProxy(config);

} **else** {

**return** **new** JdkDynamicAopProxy(config);

}

}

2、基于JDK动态代理和CGLIB的AopProxy方式对getProxy()都有其各自的实现。

**方式一：基于JDK动态代理的AOP 代理实现。**

**public** Object getProxy(ClassLoader classLoader) {

// 获取目标对象实现的接口，需要额外添加SpringProxy和Advised接口

Class<?>[] proxiedInterfaces = AopProxyUtils.*completeProxiedInterfaces*(**this**.advised);

// 判断是否重写了equals和hashCode方法。

findDefinedEqualsAndHashCodeMethods(proxiedInterfaces);

**return** Proxy.*newProxyInstance*(classLoader, proxiedInterfaces, **this**);

}

JdkDynamicAopProxy实现了InvocationHandler接口，所以需要关于invoke()方法的实现。

**public** Object invoke(Object proxy, Method method, Object[] args) **throws** Throwable {

MethodInvocation invocation;

Object oldProxy = **null**;

**boolean** setProxyContext = **false**;

TargetSource targetSource = **this**.advised.targetSource;

Class<?> targetClass = **null**;

Object target = **null**;

**try** {

**if** (!**this**.equalsDefined && AopUtils.*isEqualsMethod*(method)) {

// The target does not implement the equals(Object) method itself.

**return** equals(args[0]);

} // 目标对象没有实现equals方法

**if** (!**this**.hashCodeDefined && AopUtils.*isHashCodeMethod*(method)) {

// The target does not implement the hashCode() method itself.

**return** hashCode();

}// 目标对象没有实现hashCode方法

**if** (!**this**.advised.opaque && method.getDeclaringClass().isInterface() &&

method.getDeclaringClass().isAssignableFrom(Advised.**class**)) {

// 执行目标对象方法

**return** AopUtils.*invokeJoinpointUsingReflection*(**this**.advised, method, args);

}

Object retVal;

**if** (**this**.advised.exposeProxy) {

// Make invocation available if necessary.

oldProxy = AopContext.*setCurrentProxy*(proxy);

setProxyContext = **true**;

}

// May be null. Get as late as possible to minimize the time we "own" the target,

// in case it comes from a pool.

target = targetSource.getTarget();

**if** (target != **null**) {

targetClass = target.getClass();

}

// Get the interception chain for this method.

List<Object> chain = **this**.advised.getInterceptorsAndDynamicInterceptionAdvice(method, targetClass);

// Check whether we have any advice. If we don't, we can fallback on direct

// reflective invocation of the target, and avoid creating a MethodInvocation.

**if** (chain.isEmpty()) { // 没有匹配到通知

// We can skip creating a MethodInvocation: just invoke the target directly

// Note that the final invoker must be an InvokerInterceptor so we know it does

// nothing but a reflective operation on the target, and no hot swapping or fancy proxying.

retVal = AopUtils.*invokeJoinpointUsingReflection*(target, method, args);

} **else** {

// We need to create a method invocation...

invocation = **new** ReflectiveMethodInvocation(proxy, target, method, args, targetClass, chain);

// Proceed to the joinpoint through the interceptor chain.

retVal = **invocation.proceed()**; // 执行

}

// Massage return value if necessary.

Class<?> returnType = method.getReturnType();

**if** (retVal != **null** && retVal == target && returnType.isInstance(proxy) &&

!RawTargetAccess.**class**.isAssignableFrom(method.getDeclaringClass())) {

// Special case: it returned "this" and the return type of the method

// is type-compatible. Note that we can't help if the target sets

// a reference to itself in another returned object.

retVal = proxy;

} **else** **if** (retVal == **null** && returnType != Void.***TYPE*** && returnType.isPrimitive()) {

**throw** **new** AopInvocationException("Null return value from advice does not match primitive return type for: " + method);

}

**return** retVal;

} **finally** {

**if** (target != **null** && !targetSource.isStatic()) {

// Must have come from TargetSource.

targetSource.releaseTarget(target);

}

**if** (setProxyContext) {

// Restore old proxy.

AopContext.*setCurrentProxy*(oldProxy);

}

}

}

**方式二：基于CGLIB的AOP代理实现。**

文件位置：org/springframework/aop/framework/CglibAopProxy.java

**public** Object getProxy(ClassLoader classLoader) {

**try** {

Class<?> rootClass = **this**.advised.getTargetClass();

Assert.*state*(rootClass != **null**, "Target class must be available for creating a CGLIB proxy");

Class<?> proxySuperClass = rootClass;

**if** (ClassUtils.*isCglibProxyClass*(rootClass)) {

proxySuperClass = rootClass.getSuperclass();

Class<?>[] additionalInterfaces = rootClass.getInterfaces();

**for** (Class<?> additionalInterface : additionalInterfaces) {

**this**.advised.addInterface(additionalInterface);

}

}

validateClassIfNecessary(proxySuperClass);

Enhancer enhancer = createEnhancer();

**if** (classLoader != **null**) {

enhancer.setClassLoader(classLoader);

**if** (classLoader **instanceof** SmartClassLoader &&

((SmartClassLoader) classLoader).isClassReloadable(proxySuperClass)) {

enhancer.setUseCache(**false**);

}

}

enhancer.setSuperclass(proxySuperClass);

enhancer.setInterfaces(AopProxyUtils.*completeProxiedInterfaces*(**this**.advised));

enhancer.setNamingPolicy(SpringNamingPolicy.***INSTANCE***);

enhancer.setStrategy(**new** UndeclaredThrowableStrategy(UndeclaredThrowableException.**class**));

Callback[] callbacks = getCallbacks(rootClass);

Class<?>[] types = **new** Class<?>[callbacks.length];

**for** (**int** x = 0; x < types.length; x++) {

types[x] = callbacks[x].getClass();

}

// fixedInterceptorMap only populated at this point, after getCallbacks call above

enhancer.setCallbackFilter(**new** ProxyCallbackFilter(

**this**.advised.getConfigurationOnlyCopy(), **this**.fixedInterceptorMap, **this**.fixedInterceptorOffset));

enhancer.setCallbackTypes(types);

// Generate the proxy class and create a proxy instance.

**return** createProxyClassAndInstance(enhancer, callbacks);

} **catch** (CodeGenerationException ex) {

**throw** **new** AopConfigException("Could not generate CGLIB subclass of class [" +

**this**.advised.getTargetClass() + "]: " +

"Common causes of this problem include using a final class or a non-visible class",

ex);

} **catch** (IllegalArgumentException ex) {

**throw** **new** AopConfigException("Could not generate CGLIB subclass of class [" +

**this**.advised.getTargetClass() + "]: " +

"Common causes of this problem include using a final class or a non-visible class",

ex);

} **catch** (Exception ex) {

// TargetSource.getTarget() failed

**throw** **new** AopConfigException("Unexpected AOP exception", ex);

}

s}

### @AspectJ支持

在Spring容器配置文件中，添加以下标签元素启动@AspectJ支持

<aop:aspectj-autoproxy proxy-target-class=*"false"*/>

在前面我们弹到了，对应Schema的计息，对应的AopNamespaceHandler[见/META-INF/spring.handlers]中会注册aop相关元素的解析类，<aop:aspectj-autoproxy/>对应的解析类为AspectJAutoProxyBeanDefinitionParser。

这里我们从parse方法开始进行解析

#### parse(Element element, ParserContext parserContext)

**public** BeanDefinition parse(Element element, ParserContext parserContext) {

AopNamespaceUtils.*registerAspectJAnnotationAutoProxyCreatorIfNecessary*(parserContext, element);

extendBeanDefinition(element, parserContext);

**return** **null**;

}

1、在Spring BeanFactory中注册AnnotationAwareAspectJAutoProxyCreator对应的bean定义，并获取**<aop:aspectj-autoproxy/>**元素的proxy-target-class和expose-proxy属性值，将值注入到proxyTargetClass和exposeProxy字段上。最后将bean定义注册到BeanFactory中，beanName为” org.springframework.aop.config.internalAutoProxyCreator”。

2、获取**<aop:aspectj-autoproxy/>**下面的子元素<aop:include name=""/>[**存在多个**]，将其name属性值注入到AnnotationAwareAspectJAutoProxyCreator的includePatterns字段上。

**private** **void** addIncludePatterns(Element element, ParserContext parserContext, BeanDefinition beanDef) {

ManagedList<TypedStringValue> includePatterns = **new** ManagedList<TypedStringValue>();

NodeList childNodes = element.getChildNodes();

**for** (**int** i = 0; i < childNodes.getLength(); i++) {

Node node = childNodes.item(i);

**if** (node **instanceof** Element) {

Element includeElement = (Element) node;

TypedStringValue valueHolder = **new** TypedStringValue(includeElement.getAttribute("name"));

valueHolder.setSource(parserContext.extractSource(includeElement));

includePatterns.add(valueHolder);

}

}

**if** (!includePatterns.isEmpty()) {

includePatterns.setSource(parserContext.extractSource(element));

beanDef.getPropertyValues().add("includePatterns", includePatterns);

}

}

[Spring AOP在IoC容器中的实例化过程](#_Spring_AOP在IoC容器中的实例化过程)章节我们了解以基于Schema[<aop:config/>在Spring IoC容器实例化过程中对bean实例话的影响。这里我们来了解一下，基于@AspectJ的AOP在实例化过程中是如何影响Spring IoC bean的实例化。

AnnotationAwareAspectJAutoProxyCreator和AspectJAwareAdvisorAutoProxyCreator都实现了AbstractAdvisorAutoProxyCreator抽象接口，并且AspectJAwareAdvisorAutoProxyCreator是AnnotationAwareAspectJAutoProxyCreator父类。实例化影响的流程是一致的，所以这里我们只了解哪些重写方法的不同之处就可以。

#### findCandidateAdvisors()

**protected** List<Advisor> findCandidateAdvisors() {

// Add all the Spring advisors found according to superclass rules.

List<Advisor> advisors = **super**.findCandidateAdvisors();

// Build Advisors for all AspectJ aspects in the bean factory.

advisors.addAll(**this**.aspectJAdvisorsBuilder.buildAspectJAdvisors());

**return** advisors;

}

1、调用AbstractAdvisorAutoProxyCreator的findCandidateAdvisors()方法从BeanFactory获取已注册的类型为Advisor的bean实例。

2、调用[buildAspectJAdvisors()](#_buildAspectJAdvisors())加载BeanFactory中使用了**@Aspect**注解的类。

#### buildAspectJAdvisors()

文件位置：org/springframework/aop/aspectj/annotation/BeanFactoryAspectJAdvisorsBuilder.java

**public** List<Advisor> buildAspectJAdvisors() {

List<String> aspectNames = **null**;

**synchronized** (**this**) {

aspectNames = **this**.aspectBeanNames;

**if** (aspectNames == **null**) {

List<Advisor> advisors = **new** LinkedList<Advisor>();

aspectNames = **new** LinkedList<String>();

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**;

}

**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** LinkedList<Advisor>();

**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));

}

}

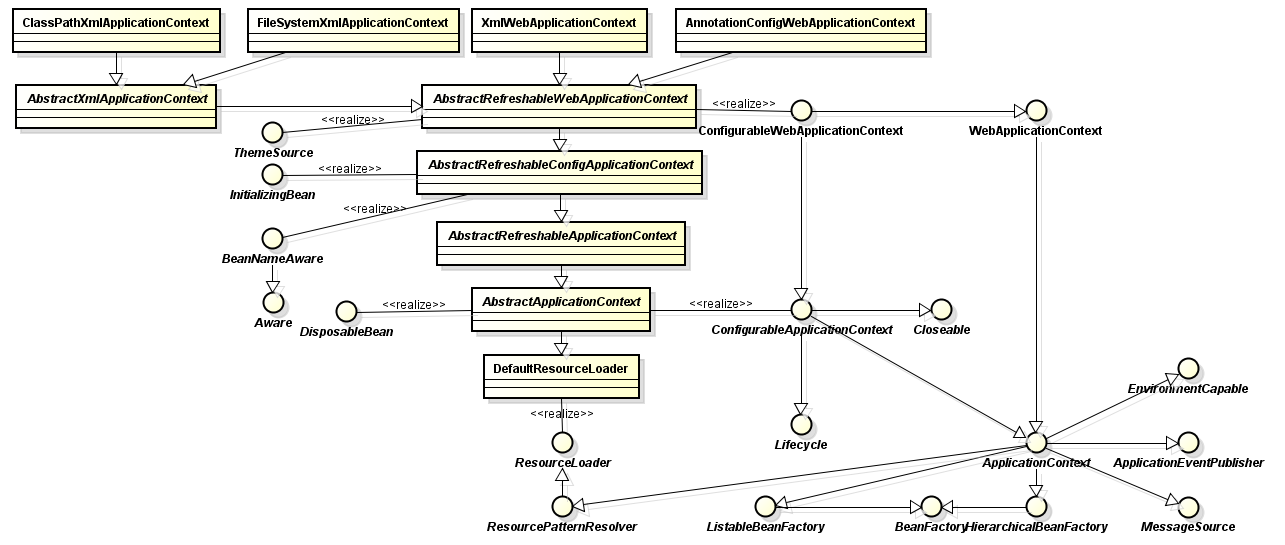
**return** advisors;

}

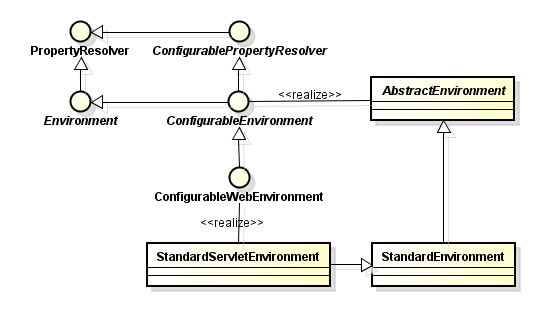
# 附录 接口/类解析

## BeanFactory

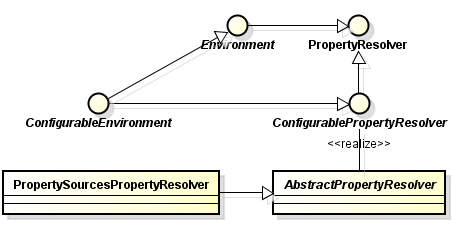
## ApplicationContext



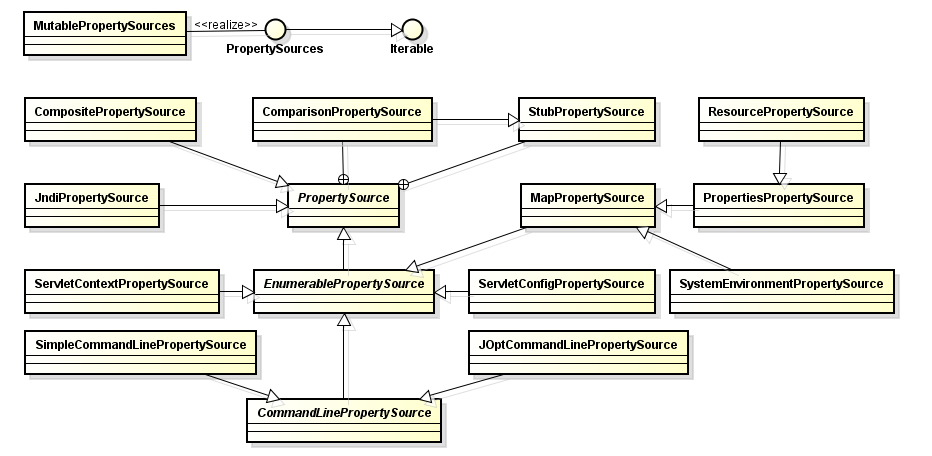
## Environment



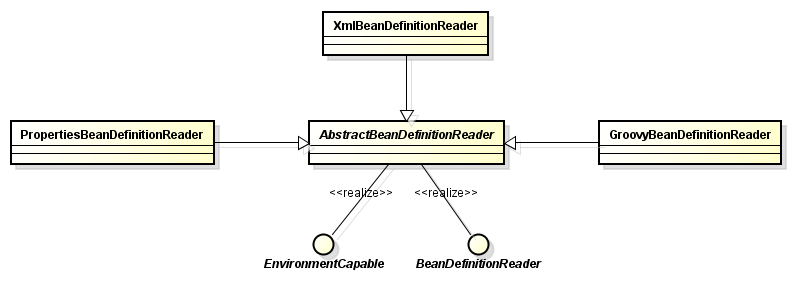
## PropertyResolver



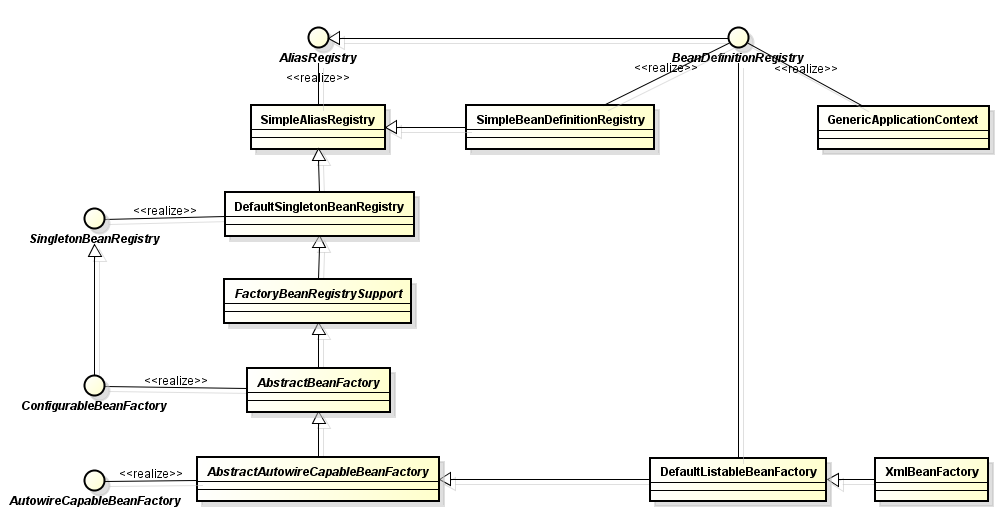
## PropertySource



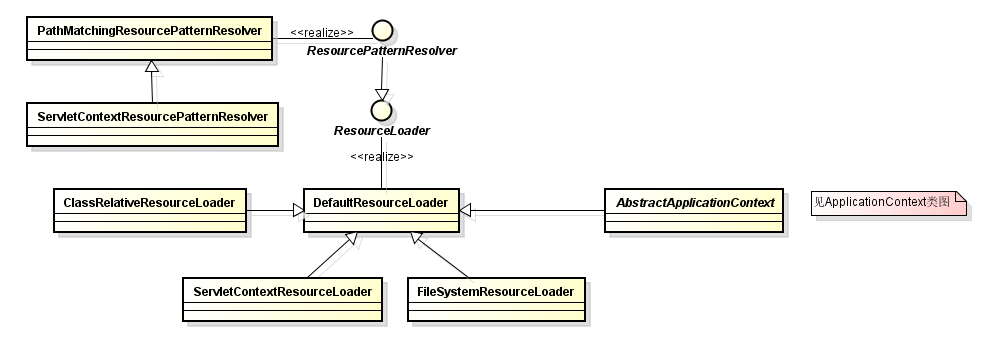
## BeanDefinitionReader



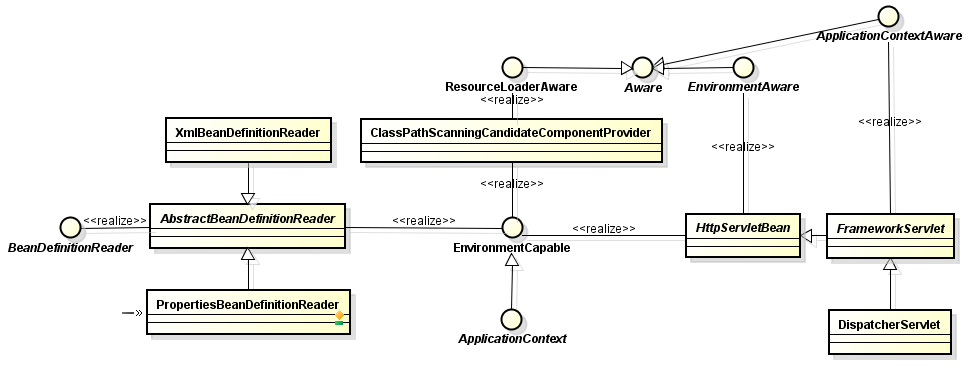
## AliasRegistry



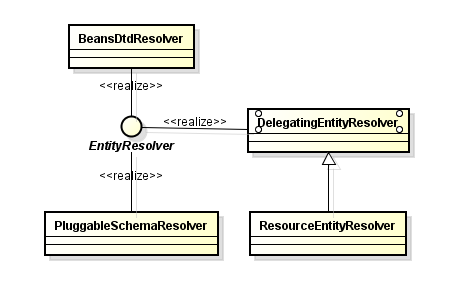
## ResourceLoader



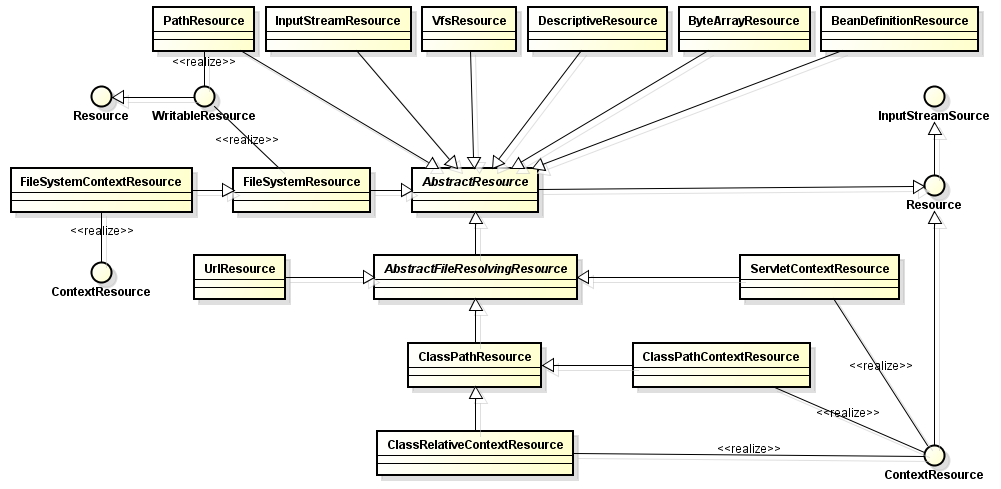
## EnvironmentCapable



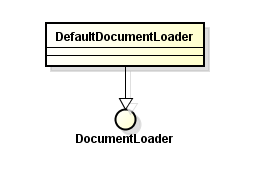
## EntityResolver



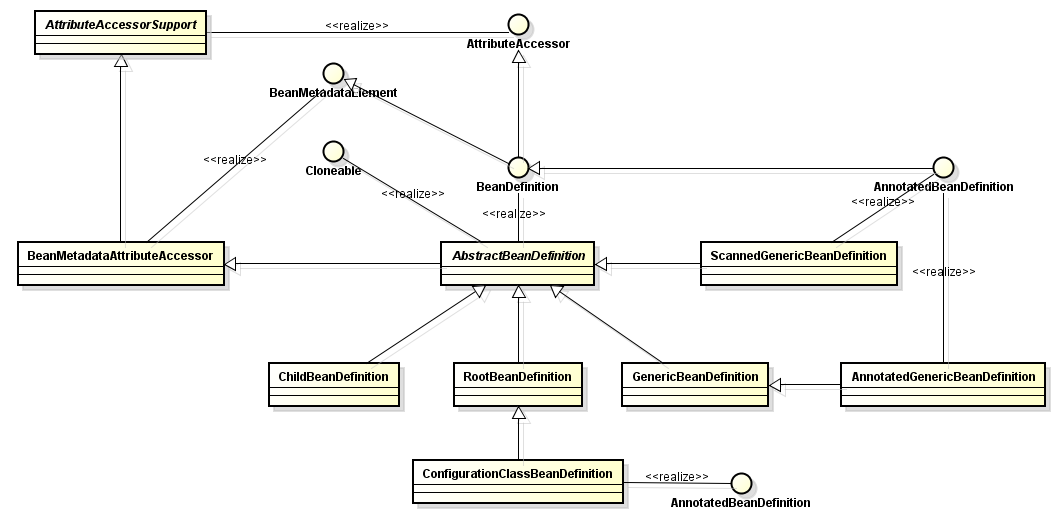
## Resource



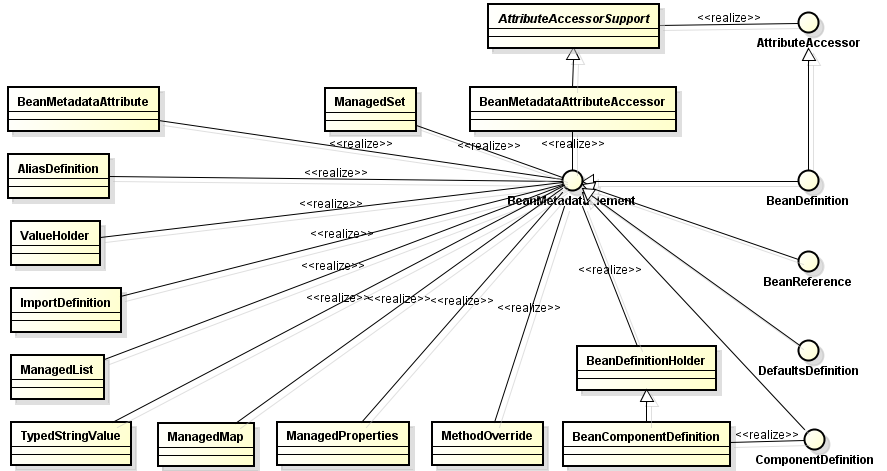
## DocumentLoader



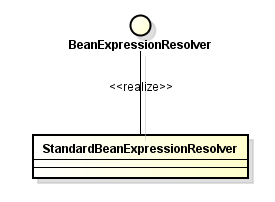
## BeanDefinition



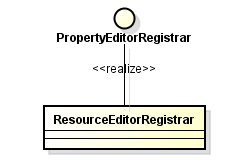
## BeanMetadataElement



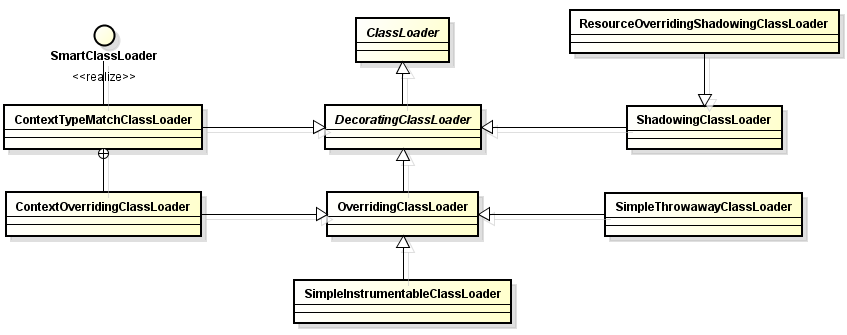
## BeanExpressionResolver



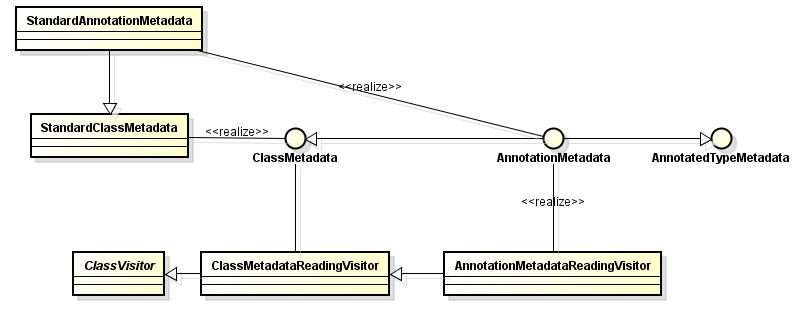
## PropertyEditorRegistrar



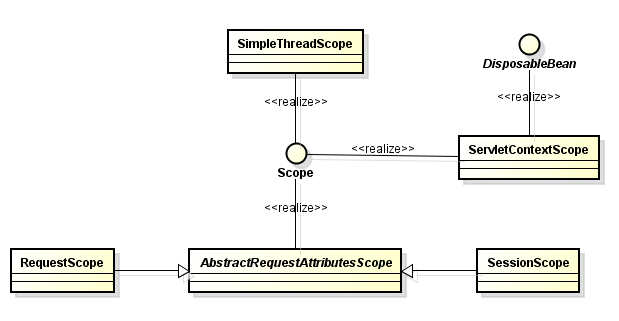
## ClassLoader



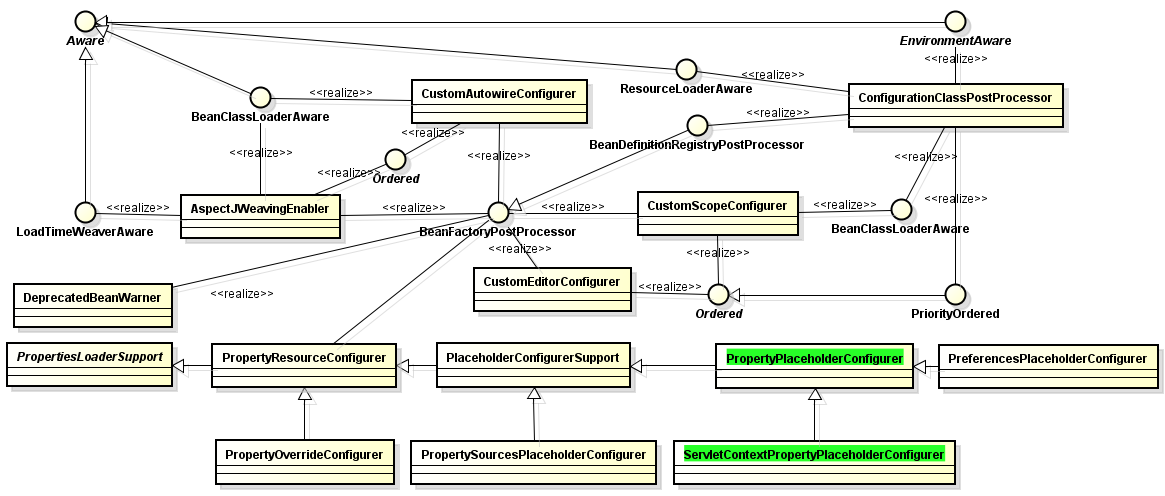
## ClassMetadata



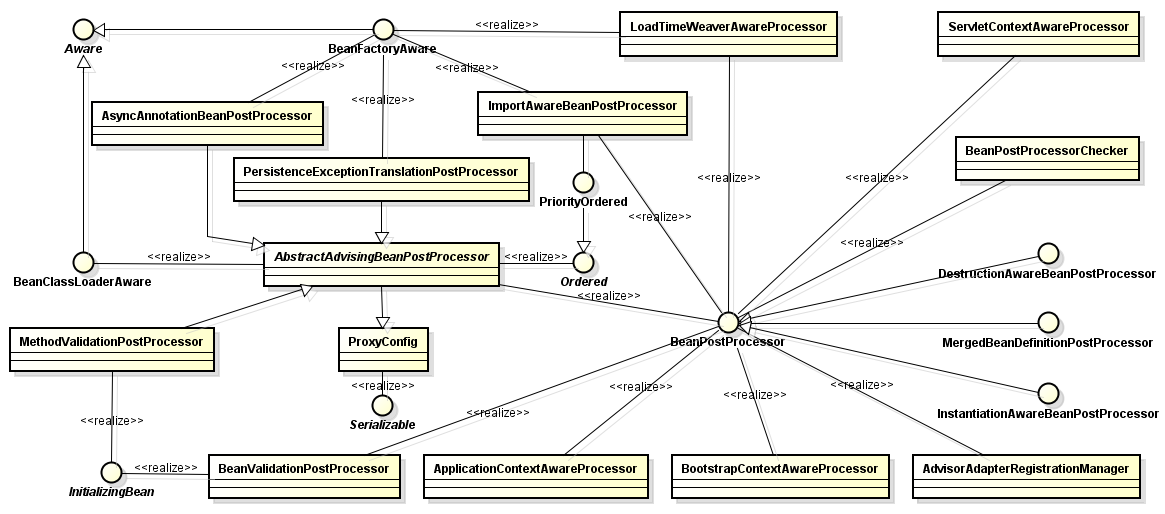
## Scope



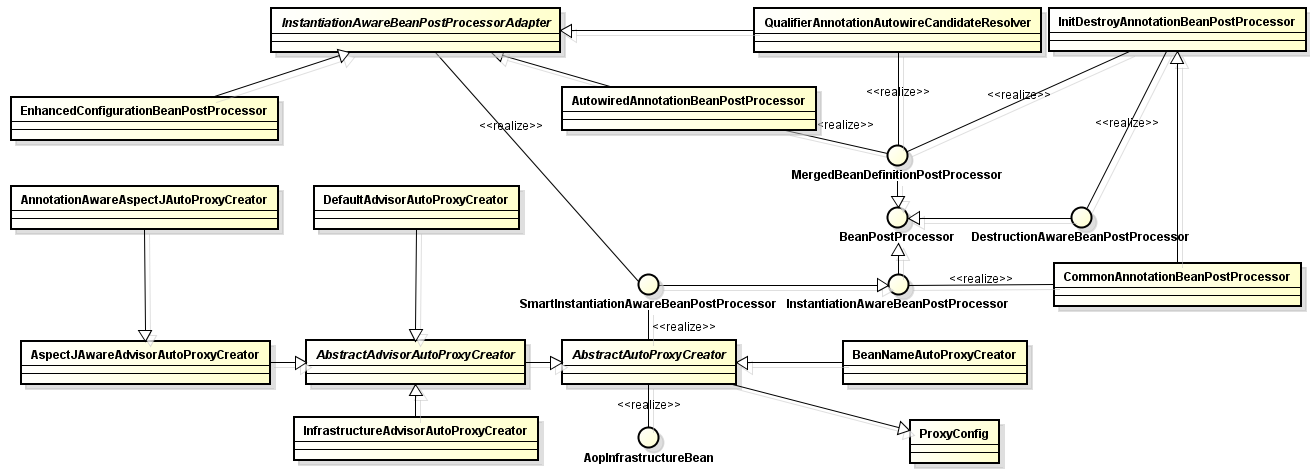
## BeanFactoryPostProcessor



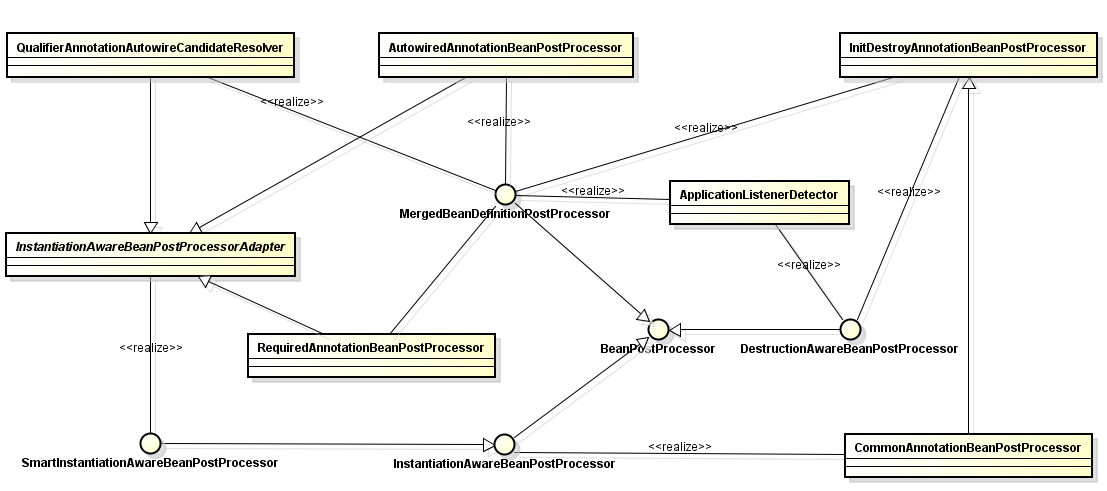
## BeanPostProcessor



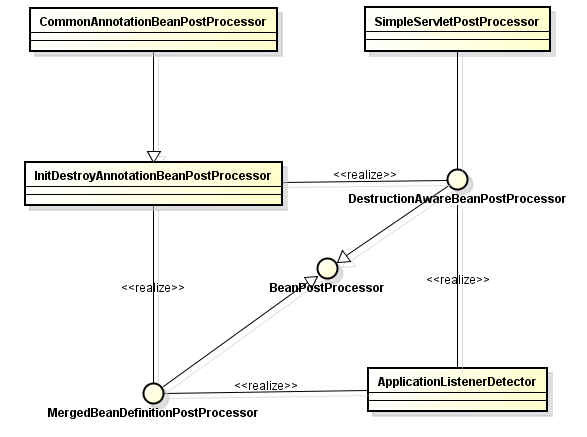
## InstantiationAwareBeanPostProcessor



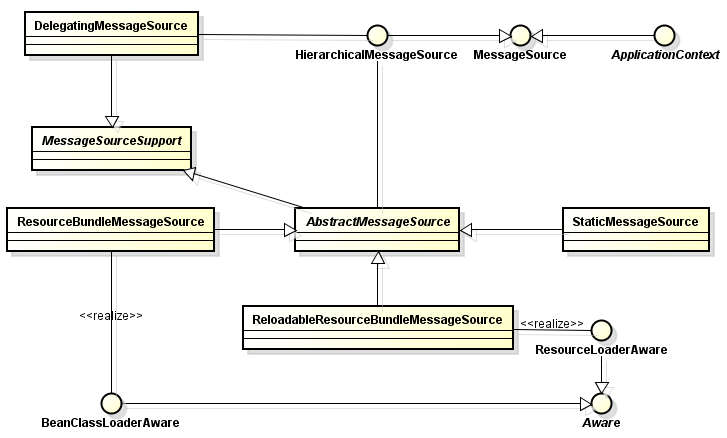
## MergedBeanDefinitionPostProcessor



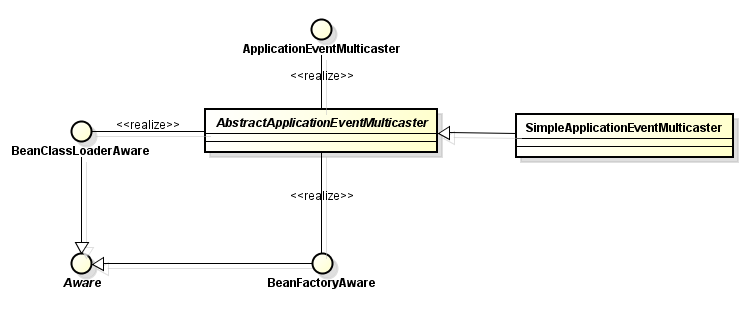
## DestructionAwareBeanPostProcessor



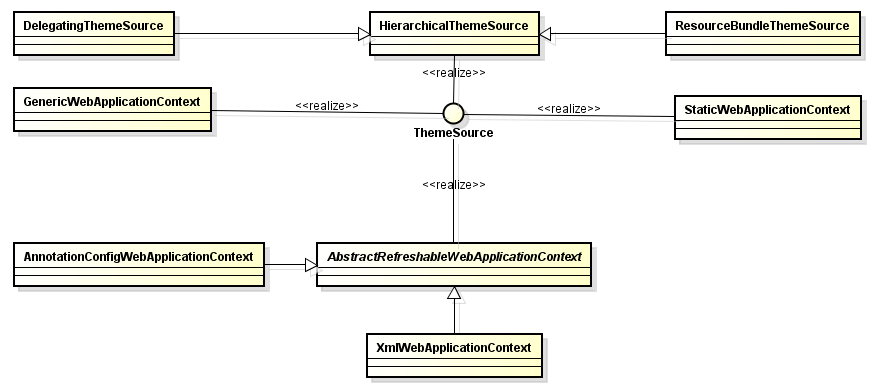
## MessageSource



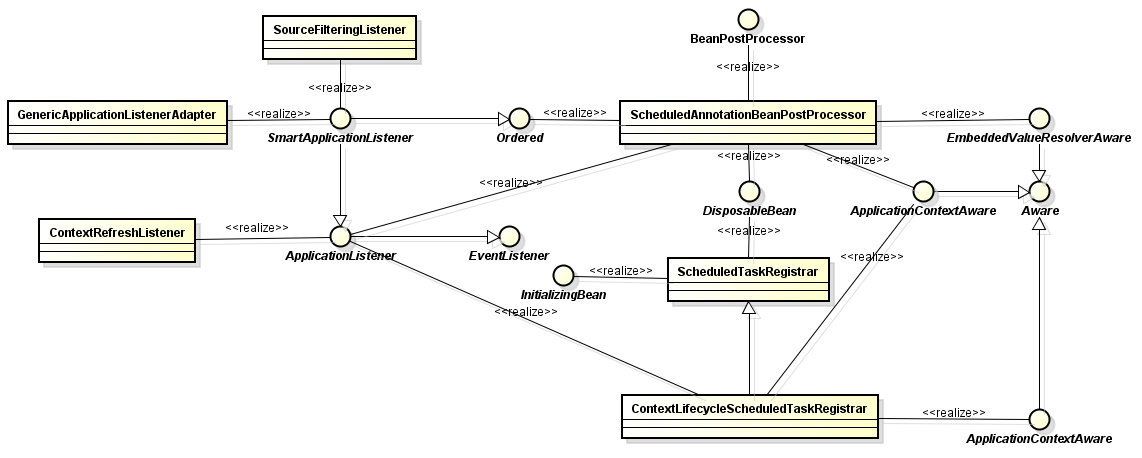
## ApplicationEventMulticaster



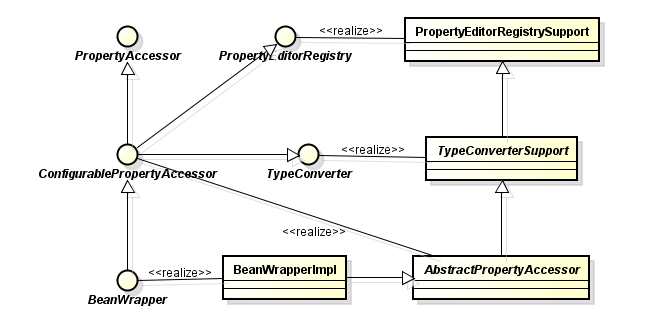
## ThemeSource



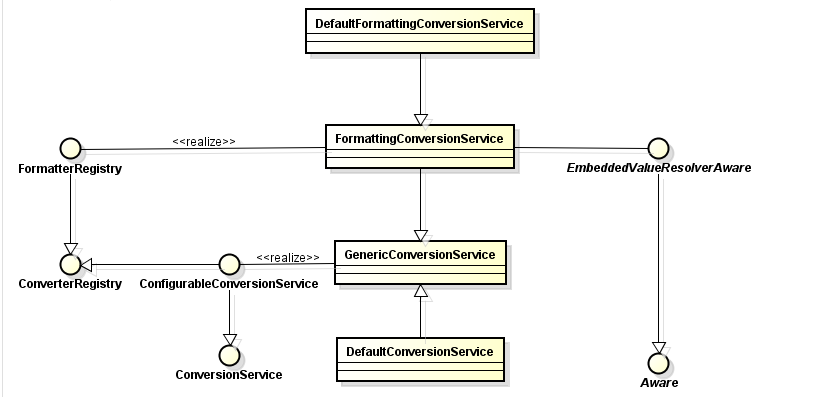
## ApplicationListener



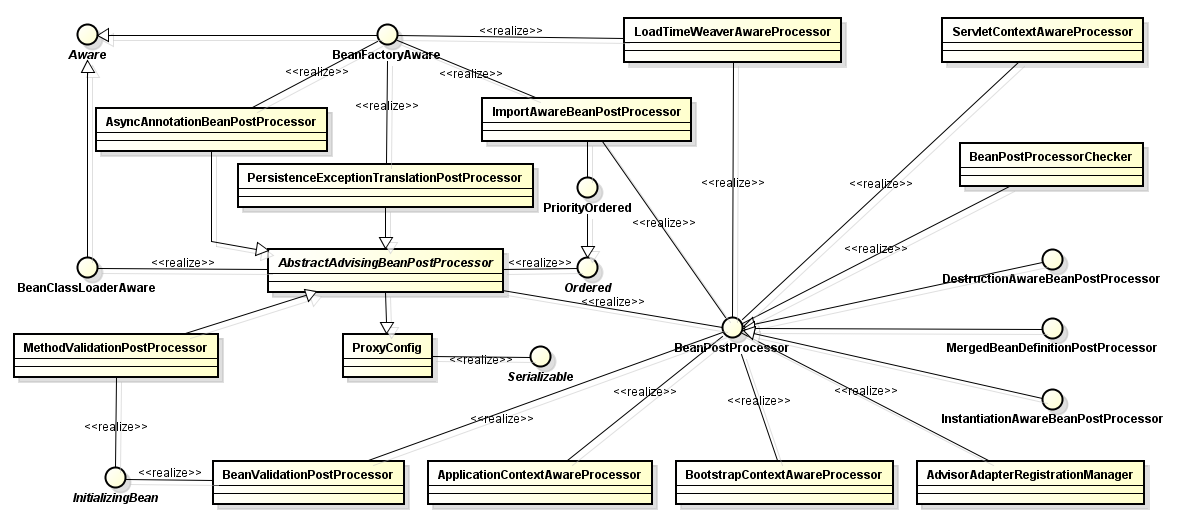
## BeanWrapper



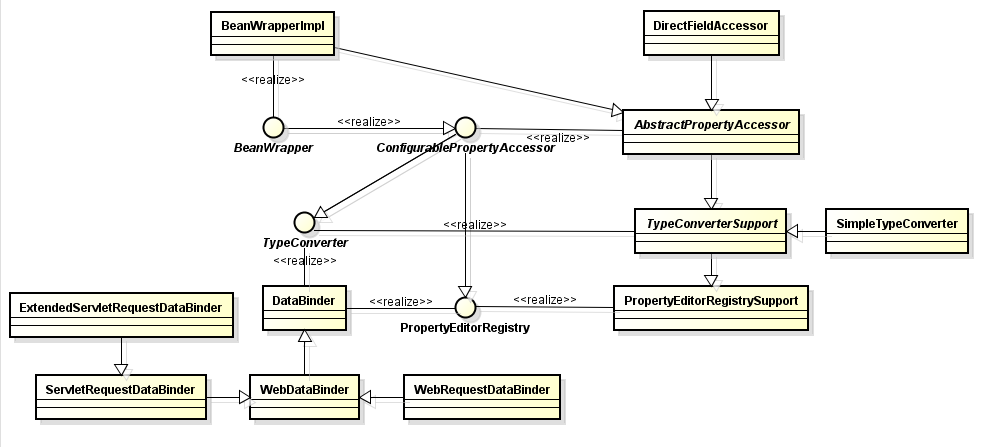
## ConversionService



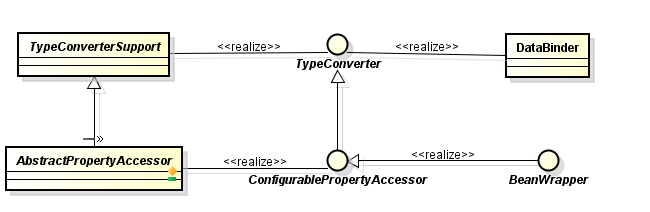
## BeanPostProcessor



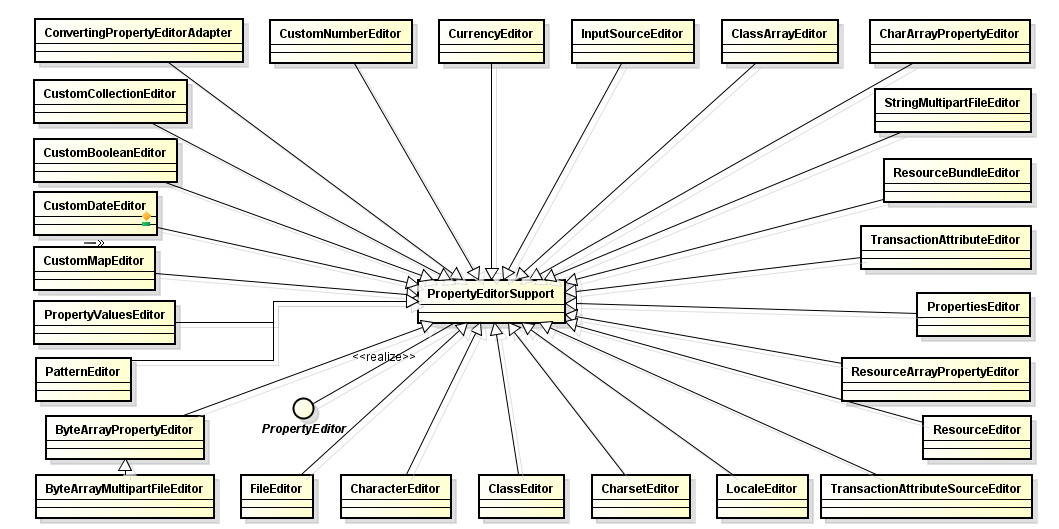
## PropertyEditorRegistry



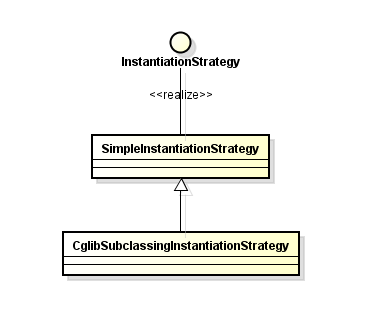
## TypeConverter



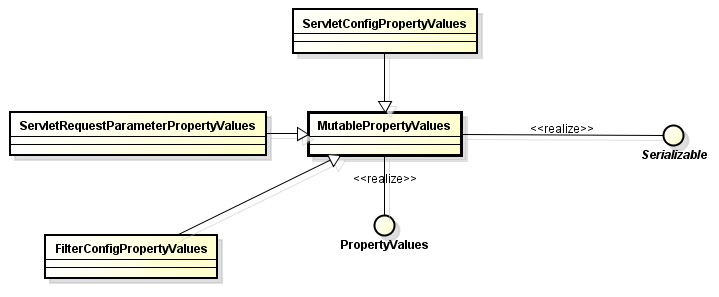
## PropertyEditor



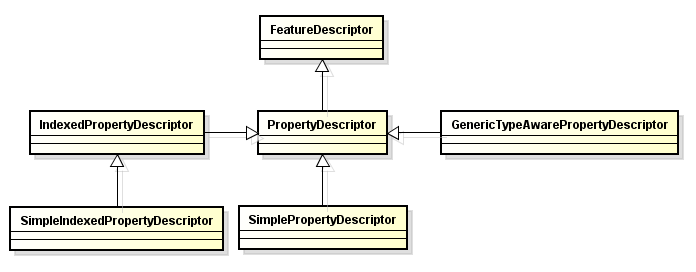
## InstantiationStrategy



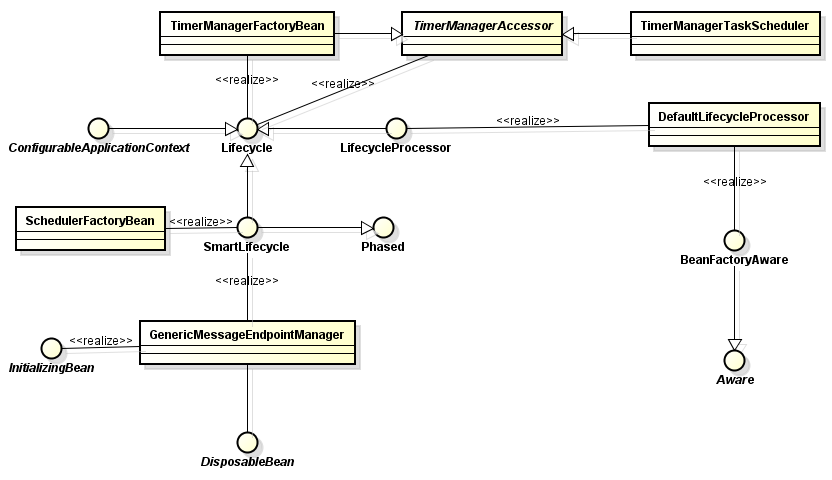
## PropertyValues



## FeatureDescriptor/PropertyDescriptor



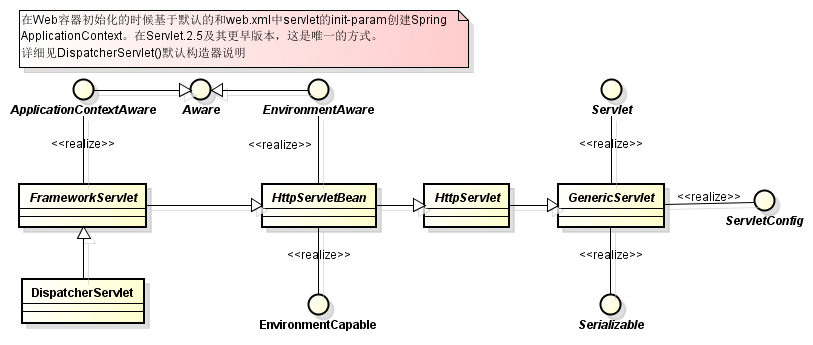
## Lifecycle



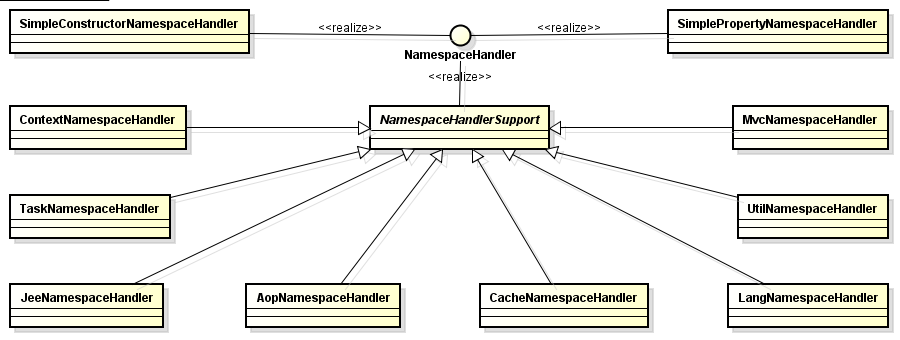
## ApplicationEvent



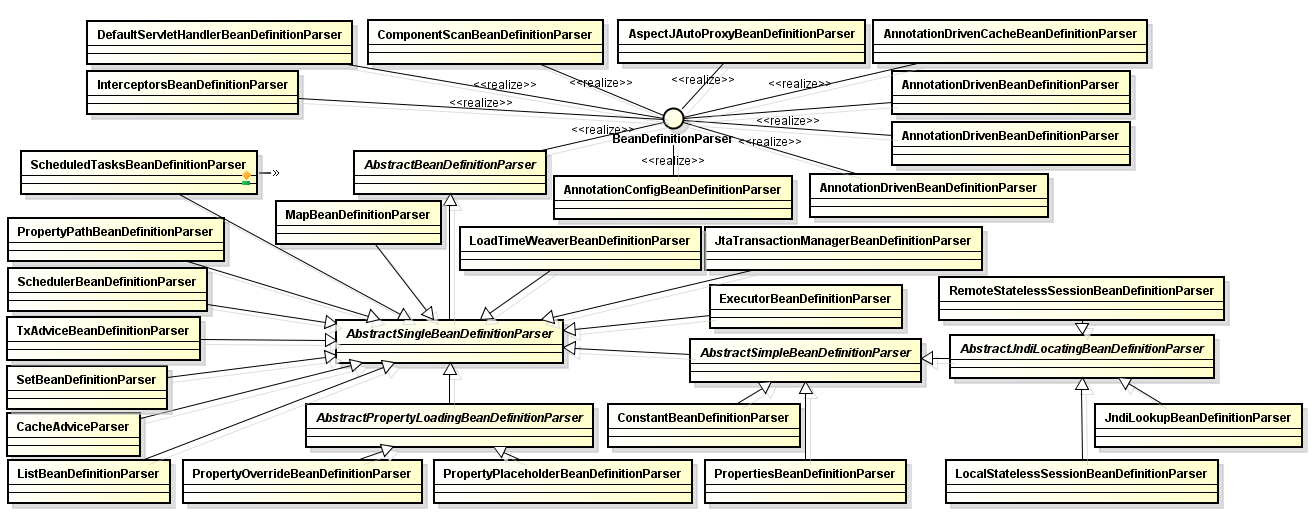
## DispatcherServlet



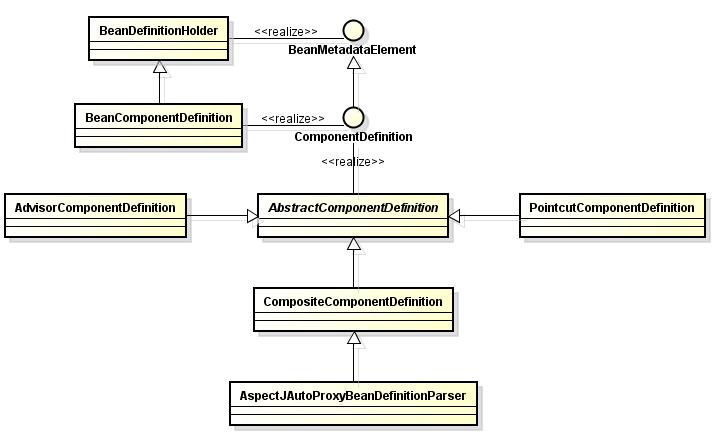
## NamespaceHandler



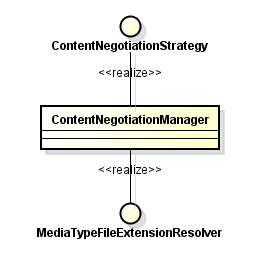
## BeanDefinitionParser



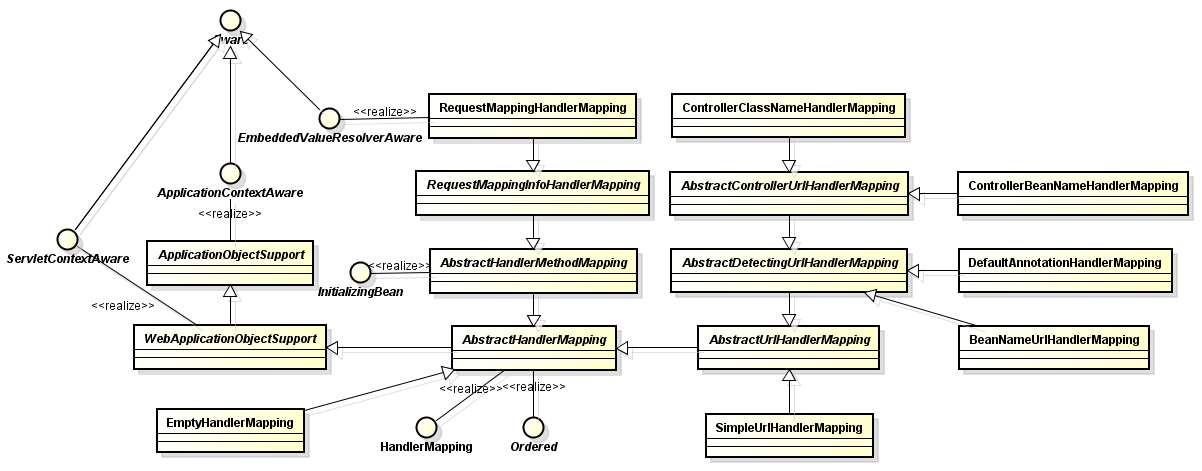
## ComponentDefinition



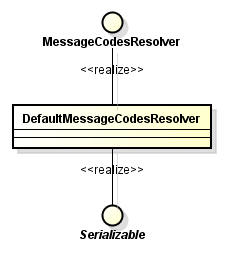
## ContentNegotiationManager



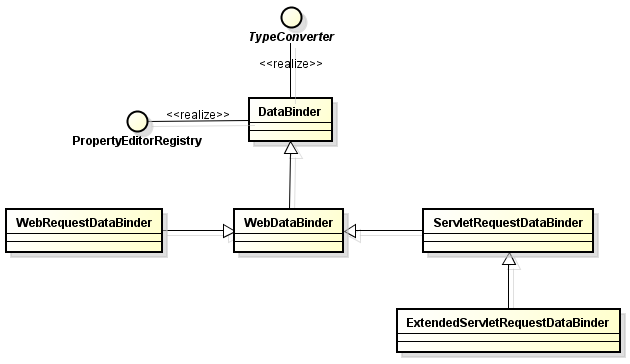
## HandlerMapping



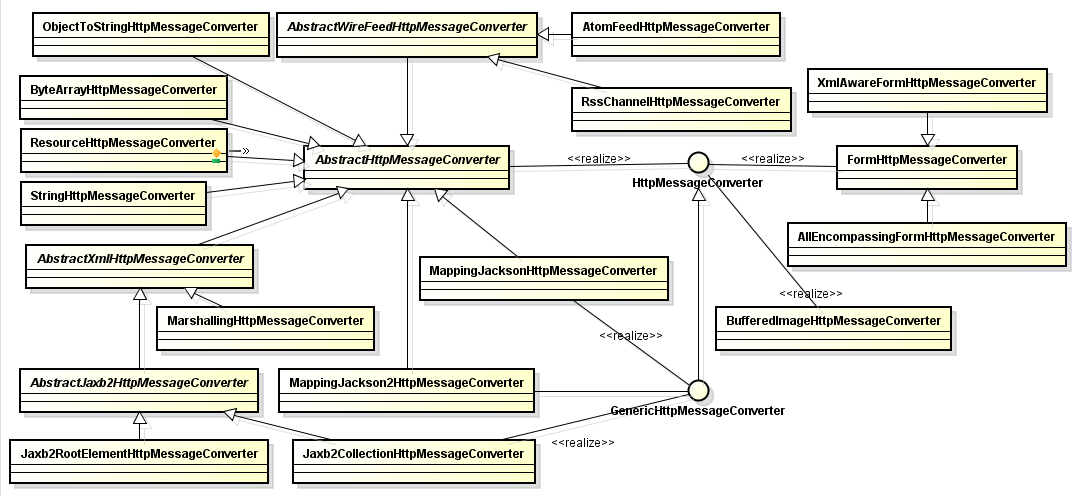
## MessageCodesResolver



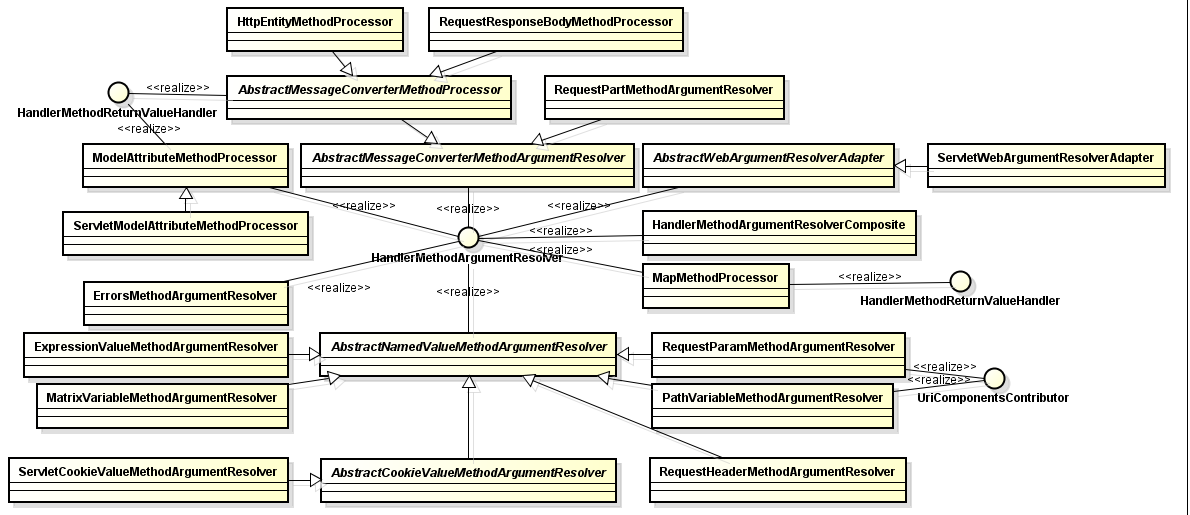
## DataBinder

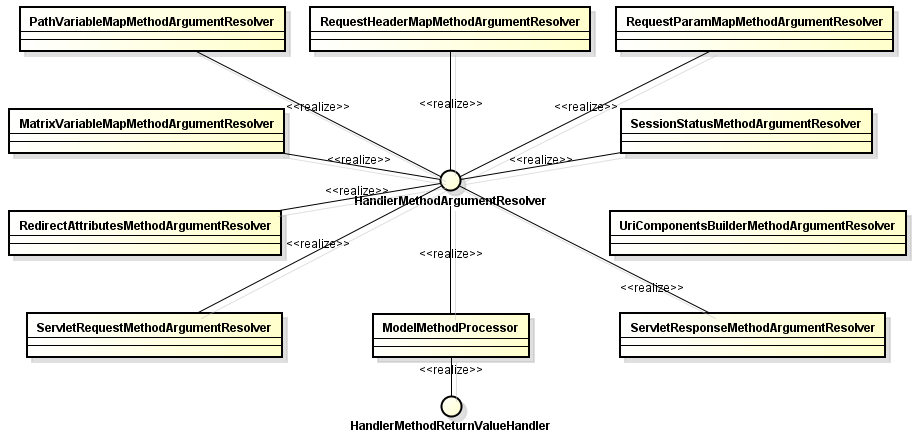


## HttpMessageConverter

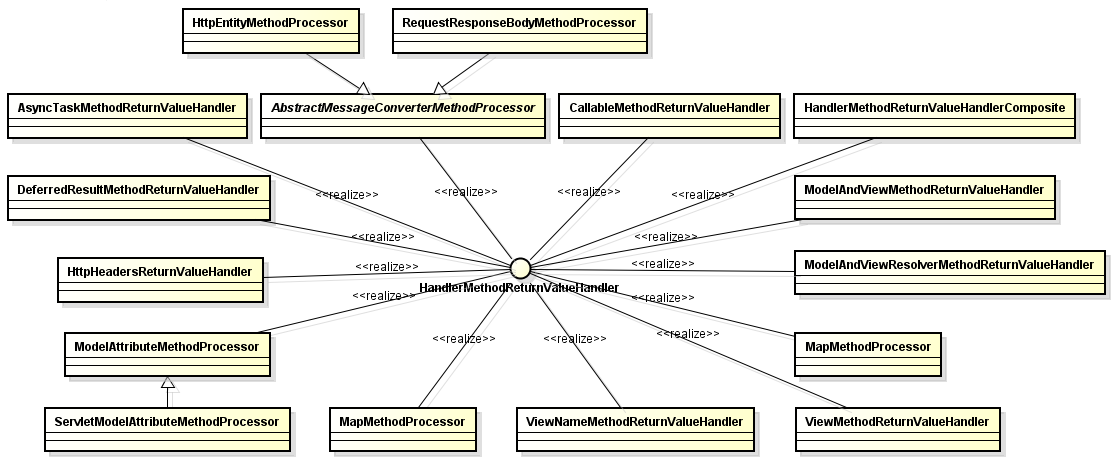


## HandlerMethodArgumentResolver

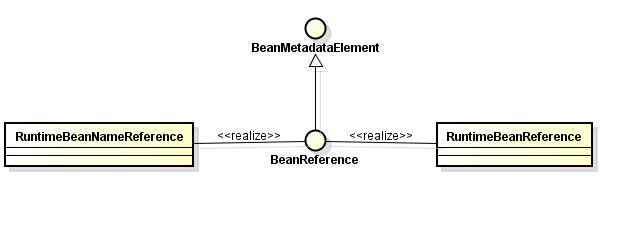




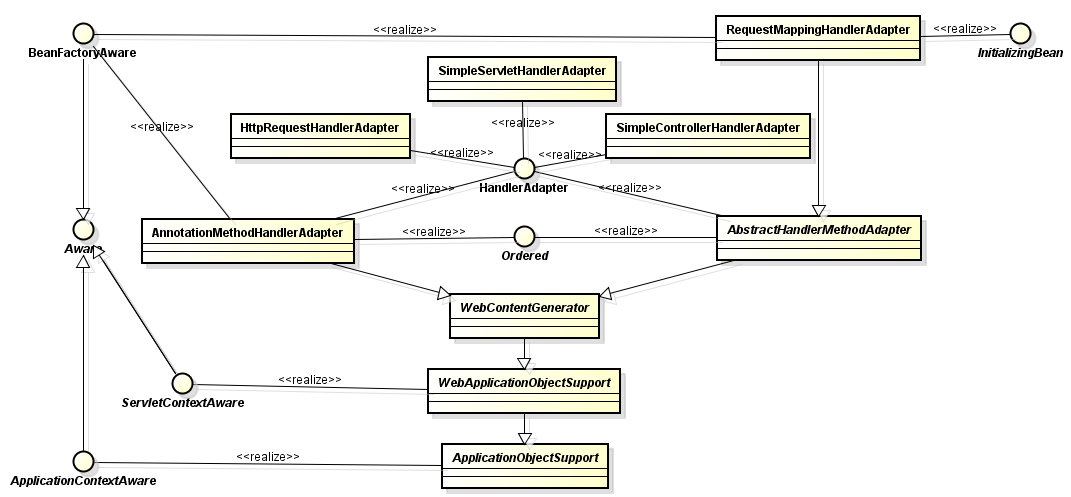
## HandlerMethodReturnValueHandler



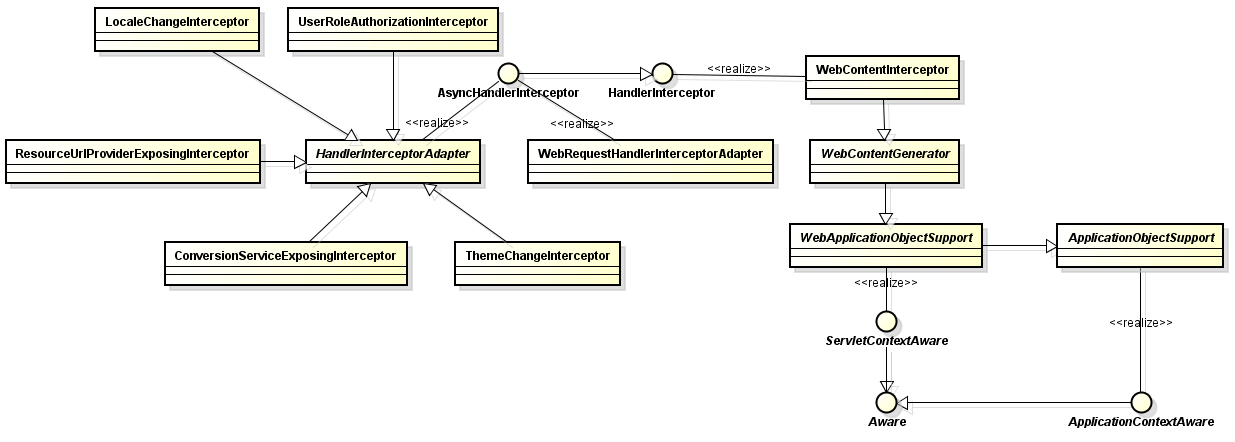
## BeanReference



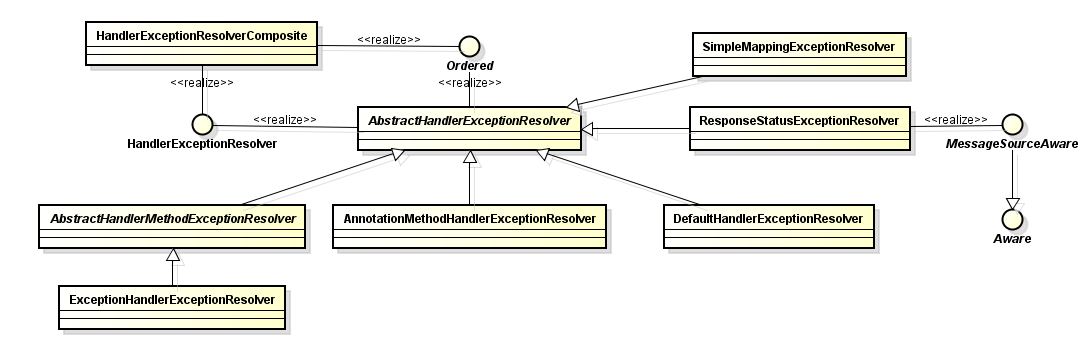
## HandlerAdapter



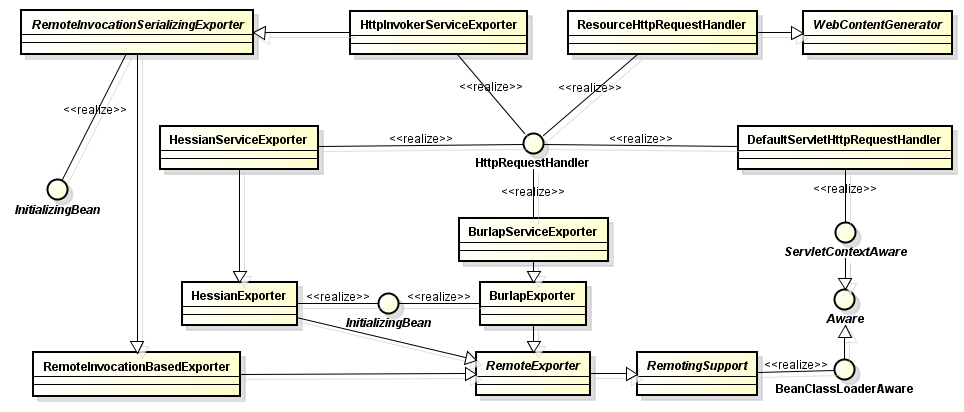
## HandlerInterceptor



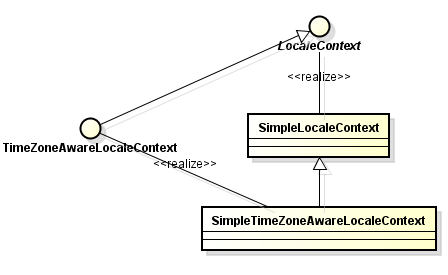
## HandlerExceptionResolver



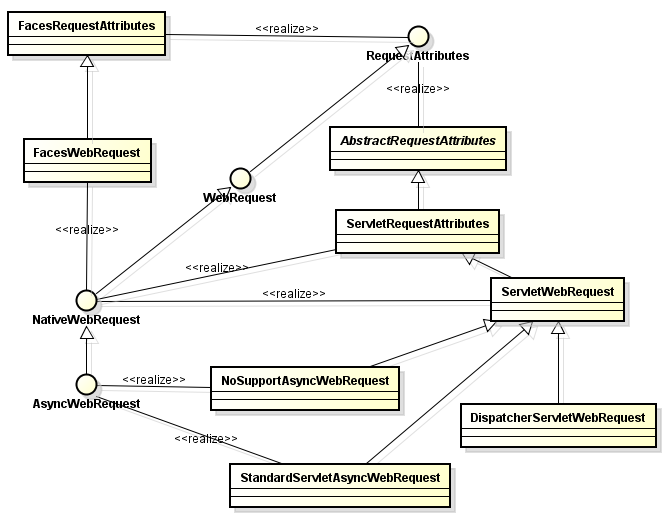
## HttpRequestHandler



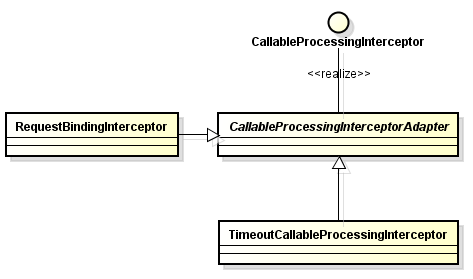
## LocaleContext



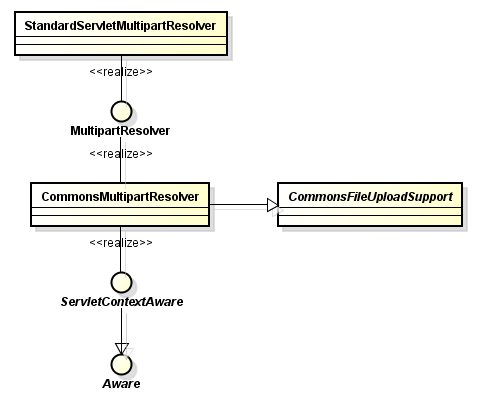
## RequestAttributes



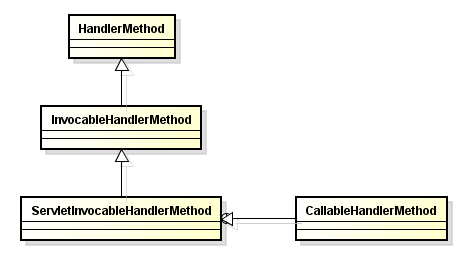
## CallableProcessingInterceptor



## MultipartResolver



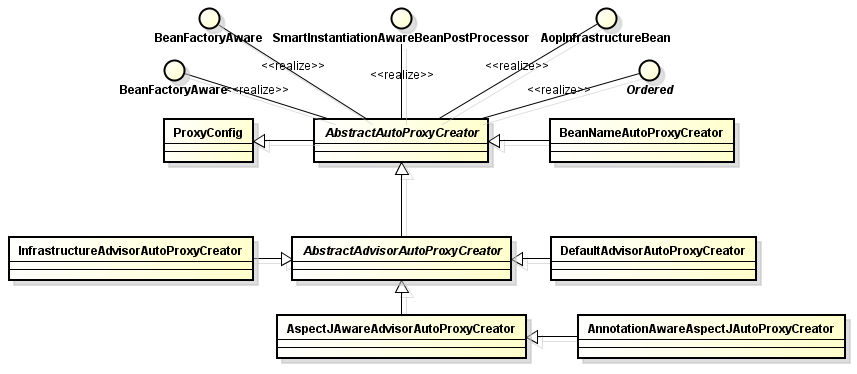
## HandlerMethod



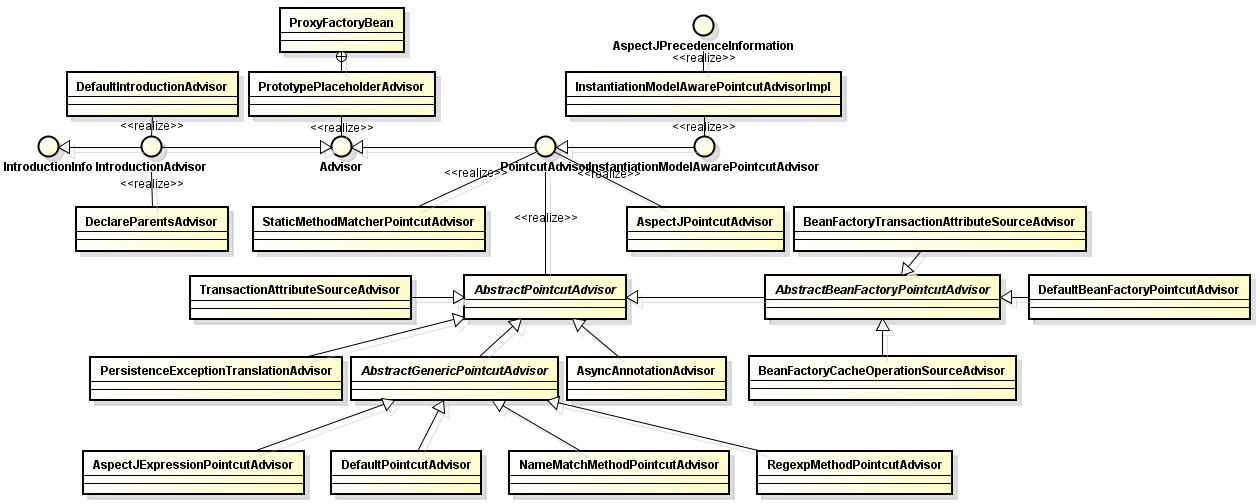
## ParameterNameDiscoverer



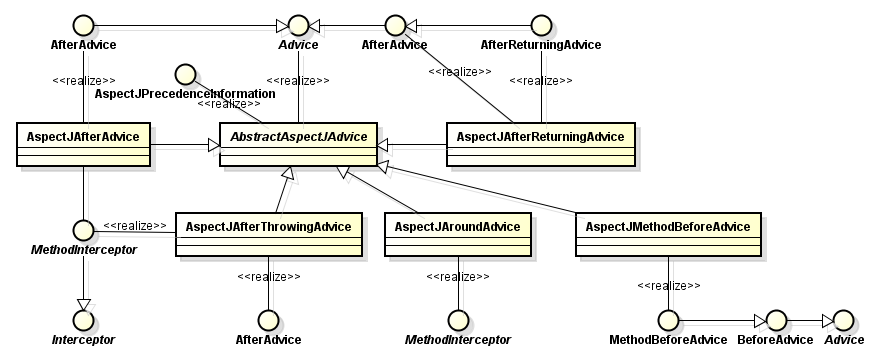
## AbstractAutoProxyCreator



## Advisor



## Advice



## AopProxy

