Spring源码分析解读

# 如何查看源码

Spring源码下载https://github.com/spring-projects/spring-framework/tags?after=v3.1.0.RC1

## 源代码结构组织

Build-spring-framework是整个Spring源代码的构建目录，里面是项目的构建脚本，如果要自己动手构建Spring，可以进入这个目录使用ANT进行构建。

l  org.springframework.context是IoC容器的源代码目录

l  org.springframework.aop是AOP实现的源代码目录

l  org.springframework.jdbc是JDBC的源代码部分

l  org.springframework.orm是O/R Mapping对应的源代码实现部分

# SpringIOC源码分析

## IOC初始化

1、 XmlBeanFactory(屌丝IOC)的整个流程

2、 FileSystemXmlApplicationContext 的IOC容器流程

1、高富帅IOC解剖

2、 设置资源加载器和资源定位

3、AbstractApplicationContext的refresh函数载入Bean定义过程：

4、AbstractApplicationContext子类的refreshBeanFactory()方法：

5、AbstractRefreshableApplicationContext子类的loadBeanDefinitions方法：

6、AbstractBeanDefinitionReader读取Bean定义资源：

7、资源加载器获取要读入的资源：

8、XmlBeanDefinitionReader加载Bean定义资源：

9、DocumentLoader将Bean定义资源转换为Document对象：

10、XmlBeanDefinitionReader解析载入的Bean定义资源文件：

11、DefaultBeanDefinitionDocumentReader对Bean定义的Document对象解析：

12、BeanDefinitionParserDelegate解析Bean定义资源文件中的<Bean>元素：

13、BeanDefinitionParserDelegate解析<property>元素：

14、解析<property>元素的子元素：

15、解析<list>子元素：

16、解析过后的BeanDefinition在IoC容器中的注册：

17、DefaultListableBeanFactory向IoC容器注册解析后的BeanDefinition：

## IOC体系

BeanFactory

         Spring Bean的创建是典型的工厂模式，这一系列的Bean工厂，也即IOC容器为开发者管理对象间的依赖关系提供了很多便利和基础服务，在Spring中有许多的IOC容器的实现供用户选择和使用，其相互关系如下：



## BeanFactory

 BeanFactory定义了 IOC 容器的最基本形式，并提供了 IOC 容器应遵守的的最基本的接口，也就是Spring IOC 所遵守的最底层和最基本的编程规范。在  Spring 代码中， BeanFactory 只是个接口，并不是 IOC容器的具体实现，但是 Spring 容器给出了很多种实现，如 DefaultListableBeanFactory 、 XmlBeanFactory 、ApplicationContext 等，都是附加了某种功能的实现。

|  |
| --- |
| **public** **interface** BeanFactory {  //这里是对FactoryBean的转义定义，因为如果使用bean的名字检索FactoryBean得到的对象是工厂生成的对象，    //如果需要得到工厂本身，需要转义  //转义符“&”用来获取FactoryBean本身  String ***FACTORY\_BEAN\_PREFIX*** = "&";  //根据bean的名字进行获取bean的实例，这是IOC最大的抽象方法  Object getBean(String name) **throws** BeansException;  //根据bean的名字和Class类型进行获取Bean的实例,和上面方法不同的是，bean名字和Bean 的class类型不同时候会爆出异常  <T> T getBean(String name, Class<T> requiredType) **throws** BeansException;  <T> T getBean(Class<T> requiredType) **throws** BeansException;  Object getBean(String name, Object... args) **throws** BeansException;  //检测这个IOC容器中是否含有这个Bean  **boolean** containsBean(String name);  //判断这个Bean是不是单利  **boolean** isSingleton(String name) **throws** NoSuchBeanDefinitionException;  //判断这个Bean是不是原型  **boolean** isPrototype(String name) **throws** NoSuchBeanDefinitionException;  //查询指定的bean的名字和Class类型是不是指定的Class类型  **boolean** isTypeMatch(String name, Class targetType) **throws** NoSuchBeanDefinitionException;  //这里对得到bean实例的Class类型  Class<?> getType(String name) **throws** NoSuchBeanDefinitionException;  //这里得到bean的别名，如果根据别名检索，那么其原名也会被检索出来  String[] getAliases(String name);  } |

## BeanDefinition

|  |
| --- |
| 这个接口，可以理解为xml bean元素的数据载体。通过对比xml bean标签的属性列表和BeanDefinition的属性列表一看便知。  我的理解，是解析XML的过程，就是 xml <bean>元素内容 转换为BeanDefinition对象的过程。而且这个接口，支持层级，对应对象的继承。  有一个类BeanDefinitionHolder,BeanDefinitionHolder，根据名称或者别名持有beanDefinition，它承载了name和BeanDefinition的映射信息。  BeanWarpper:  提供对标准javabean的分析和操作方法：单个或者批量获取和设置属性值，获取属性描述符，查询属性的可读性和可写性等。支持属性的嵌套设置，深度没有限制。  AbstractRefreshableApplicationContext的refreshBeanFactory()这个方法 |

|  |
| --- |
| **protected** **final** **void** refreshBeanFactory() **throws** BeansException {  **if** (hasBeanFactory()) {  destroyBeans();  closeBeanFactory();  }  **try** {  DefaultListableBeanFactory beanFactory = createBeanFactory();//创建IOC容器  beanFactory.setSerializationId(getId());  customizeBeanFactory(beanFactory);  loadBeanDefinitions(beanFactory);//载入loadBeanDefinitions  **synchronized** (**this**.beanFactoryMonitor) {  **this**.beanFactory = beanFactory;  }  }  **catch** (IOException ex) {  **throw** **new** ApplicationContextException("I/O error parsing bean definition source for " + getDisplayName(), ex);  }  }  **public abstract class AbstractXmlApplicationContext extends AbstractRefreshableConfigApplicationContext { 实现**  /\*\*  \* Loads the bean definitions via an XmlBeanDefinitionReader.  \* **@see** org.springframework.beans.factory.xml.XmlBeanDefinitionReader  \* **@see** #initBeanDefinitionReader  \* **@see** #loadBeanDefinitions  \*/  @Override  **protected** **void** loadBeanDefinitions(DefaultListableBeanFactory beanFactory) **throws** BeansException, IOException {  // Create a new XmlBeanDefinitionReader for the given BeanFactory.  XmlBeanDefinitionReader beanDefinitionReader = **new** XmlBeanDefinitionReader(beanFactory);  // Configure the bean definition reader with this context's  // resource loading environment.  beanDefinitionReader.setResourceLoader(**this**);  beanDefinitionReader.setEntityResolver(**new** ResourceEntityResolver(**this**));  // Allow a subclass to provide custom initialization of the reader,  // then proceed with actually loading the bean definitions.  initBeanDefinitionReader(beanDefinitionReader);  loadBeanDefinitions(beanDefinitionReader);  }  先调用本类里面的loadBeanDefinitions  **protected** **void** loadBeanDefinitions(XmlBeanDefinitionReader reader) **throws** BeansException, IOException {  Resource[] configResources = getConfigResources();  **if** (configResources != **null**) {  reader.loadBeanDefinitions(configResources);  }  String[] configLocations = getConfigLocations();  **if** (configLocations != **null**) {  reader.loadBeanDefinitions(configLocations);  }  }  委托给reader.loadBeanDefinitions(configLocation);    XmlBeanDefinitionReader 通过XmlBeanDefinitionReader来读取。下面看一下XmlBeanDefinitionReader这个方法，但其实并不在这个类实现这个方法，而是在它的基类里面AbstractBeanDefinitionReader  **public** **int** loadBeanDefinitions(String... locations) **throws** BeanDefinitionStoreException {  Assert.*notNull*(locations, "Location array must not be null");  **int** counter = 0;  **for** (String location : locations) {  counter += loadBeanDefinitions(location);  }  **return** counter;  }  进入到loadBeanDefinitions  **public** **int** loadBeanDefinitions(EncodedResource encodedResource) **throws** BeanDefinitionStoreException {  Assert.*notNull*(encodedResource, "EncodedResource must not be null");  **if** (logger.isInfoEnabled()) {  logger.info("Loading XML bean definitions from " + encodedResource.getResource());  }  Set<EncodedResource> currentResources = **this**.resourcesCurrentlyBeingLoaded.get();  **if** (currentResources == **null**) {  currentResources = **new** HashSet<EncodedResource>(4);  **this**.resourcesCurrentlyBeingLoaded.set(currentResources);  }  **if** (!currentResources.add(encodedResource)) {  **throw** **new** BeanDefinitionStoreException(  "Detected cyclic loading of " + encodedResource + " - check your import definitions!");  }  **try** {  InputStream inputStream = encodedResource.getResource().getInputStream();//获取IO  **try** {  InputSource inputSource = **new** InputSource(inputStream);  **if** (encodedResource.getEncoding() != **null**) {  inputSource.setEncoding(encodedResource.getEncoding());  }  **return** doLoadBeanDefinitions(inputSource, encodedResource.getResource());//这个方法从流中读取  }  **finally** {  inputStream.close();  }  }  **catch** (IOException ex) {  **throw** **new** BeanDefinitionStoreException(  "IOException parsing XML document from " + encodedResource.getResource(), ex);  }  **finally** {  currentResources.remove(encodedResource);  **if** (currentResources.isEmpty()) {  **this**.resourcesCurrentlyBeingLoaded.remove();  }  }  }  进入到doLoadBeanDefinitions Resource IO封装  **protected** **int** doLoadBeanDefinitions(InputSource inputSource, Resource resource)  **throws** BeanDefinitionStoreException {  **try** {  **int** validationMode = getValidationModeForResource(resource);  Document doc = **this**.documentLoader.loadDocument(  inputSource, getEntityResolver(), **this**.errorHandler, validationMode, isNamespaceAware());  **return** registerBeanDefinitions(doc, resource); //解析XML  }  **catch** (BeanDefinitionStoreException ex) {  **throw** ex;  }  **catch** (SAXParseException ex) {  **throw** **new** XmlBeanDefinitionStoreException(resource.getDescription(),  "Line " + ex.getLineNumber() + " in XML document from " + resource + " is invalid", ex);  }  **catch** (SAXException ex) {  **throw** **new** XmlBeanDefinitionStoreException(resource.getDescription(),  "XML document from " + resource + " is invalid", ex);  }  **catch** (ParserConfigurationException ex) {  **throw** **new** BeanDefinitionStoreException(resource.getDescription(),  "Parser configuration exception parsing XML from " + resource, ex);  }  **catch** (IOException ex) {  **throw** **new** BeanDefinitionStoreException(resource.getDescription(),  "IOException parsing XML document from " + resource, ex);  }  **catch** (Throwable ex) {  **throw** **new** BeanDefinitionStoreException(resource.getDescription(),  "Unexpected exception parsing XML document from " + resource, ex);  }  }  进入到registerBeanDefinitions  /\*\*  \* Register the bean definitions contained in the given DOM document.  \* Called by <code>loadBeanDefinitions</code>.  \* <p>Creates a new instance of the parser class and invokes  \* <code>registerBeanDefinitions</code> on it.  \* **@param** doc the DOM document  \* **@param** resource the resource descriptor (for context information)  \* **@return** the number of bean definitions found  \* **@throws** BeanDefinitionStoreException in case of parsing errors  \* **@see** #loadBeanDefinitions  \* **@see** #setDocumentReaderClass  \* **@see** BeanDefinitionDocumentReader#registerBeanDefinitions  \*/  **public** **int** registerBeanDefinitions(Document doc, Resource resource) **throws** BeanDefinitionStoreException {  // Read document based on new BeanDefinitionDocumentReader SPI.  BeanDefinitionDocumentReader documentReader = createBeanDefinitionDocumentReader();  **int** countBefore = getRegistry().getBeanDefinitionCount();  documentReader.registerBeanDefinitions(doc, createReaderContext(resource));  **return** getRegistry().getBeanDefinitionCount() - countBefore;  }  documentReader.registerBeanDefinitionsXML解析  /\*\*  \* Parses bean definitions according to the "spring-beans" DTD.  \* <p>Opens a DOM Document; then initializes the default settings  \* specified at <code>&lt;beans&gt;</code> level; then parses  \* the contained bean definitions.  \*/  **public** **void** registerBeanDefinitions(Document doc, XmlReaderContext readerContext) {  **this**.readerContext = readerContext;  logger.debug("Loading bean definitions");  Element root = doc.getDocumentElement();  BeanDefinitionParserDelegate delegate = createHelper(readerContext, root);  preProcessXml(root);  parseBeanDefinitions(root, delegate);  postProcessXml(root);  }  **-----遍历节点**  **protected** **void** parseBeanDefinitions(Element root, BeanDefinitionParserDelegate delegate) {  **if** (delegate.isDefaultNamespace(root)) {  NodeList nl = root.getChildNodes();  **for** (**int** i = 0; i < nl.getLength(); i++) {  Node node = nl.item(i);  **if** (node **instanceof** Element) {  Element ele = (Element) node;  **if** (delegate.isDefaultNamespace(ele)) {  parseDefaultElement(ele, delegate); //默认解析  }  **else** {  delegate.parseCustomElement(ele);  }  }  }  }  **else** {  delegate.parseCustomElement(root);  }  }  ---判断解析类  **private** **void** parseDefaultElement(Element ele, BeanDefinitionParserDelegate delegate) {  **if** (delegate.nodeNameEquals(ele, ***IMPORT\_ELEMENT***)) {  importBeanDefinitionResource(ele);//import类型  }  **else** **if** (delegate.nodeNameEquals(ele, ***ALIAS\_ELEMENT***)) {  processAliasRegistration(ele);//别名方式  }  **else** **if** (delegate.nodeNameEquals(ele, ***BEAN\_ELEMENT***)) {  processBeanDefinition(ele, delegate);//bean解析方式  }  } |

## Bean的解析方式

进入到 AbstractBeanDefinition beanDefinition = parseBeanDefinitionElement(ele, beanName, containingBean); 使用反射初始化类

|  |
| --- |
| **public** AbstractBeanDefinition parseBeanDefinitionElement(  Element ele, String beanName, BeanDefinition containingBean) {  **this**.parseState.push(**new** BeanEntry(beanName));  String className = **null**;  **if** (ele.hasAttribute(***CLASS\_ATTRIBUTE***)) {  className = ele.getAttribute(***CLASS\_ATTRIBUTE***).trim();  }  **try** {  String parent = **null**;  **if** (ele.hasAttribute(***PARENT\_ATTRIBUTE***)) {  parent = ele.getAttribute(***PARENT\_ATTRIBUTE***);  }  AbstractBeanDefinition bd = createBeanDefinition(className, parent);  parseBeanDefinitionAttributes(ele, beanName, containingBean, bd);  bd.setDescription(DomUtils.*getChildElementValueByTagName*(ele, ***DESCRIPTION\_ELEMENT***));  parseMetaElements(ele, bd);  parseLookupOverrideSubElements(ele, bd.getMethodOverrides());  parseReplacedMethodSubElements(ele, bd.getMethodOverrides());  parseConstructorArgElements(ele, bd);  parsePropertyElements(ele, bd);  parseQualifierElements(ele, bd);  bd.setResource(**this**.readerContext.getResource());  bd.setSource(extractSource(ele));  **return** bd;  }  **catch** (ClassNotFoundException ex) {  error("Bean class [" + className + "] not found", ele, ex);  }  **catch** (NoClassDefFoundError err) {  error("Class that bean class [" + className + "] depends on not found", ele, err);  }  **catch** (Throwable ex) {  error("Unexpected failure during bean definition parsing", ele, ex);  }  **finally** {  **this**.parseState.pop();  }  **return** **null**;  }  进入到AbstractBeanDefinition bd = createBeanDefinition(className, parent);  **protected** AbstractBeanDefinition createBeanDefinition(String className, String parentName)  **throws** ClassNotFoundException {  **return** BeanDefinitionReaderUtils.*createBeanDefinition*(  parentName, className, **this**.readerContext.getBeanClassLoader());  }  进入到BeanDefinitionReaderUtils.*createBeanDefinition*  **public** **static** AbstractBeanDefinition createBeanDefinition(  String parentName, String className, ClassLoader classLoader) **throws** ClassNotFoundException {  GenericBeanDefinition bd = **new** GenericBeanDefinition();  bd.setParentName(parentName);  **if** (className != **null**) {  **if** (classLoader != **null**) {  bd.setBeanClass(ClassUtils.*forName*(className, classLoader));//使用java反射机制初始化  }  **else** {  bd.setBeanClassName(className);  }  }  **return** bd;  } |

# Bean生命周期分析

1.Spring对Bean进行实例化（相当于程序中的new Xx()）

2.Spring将值和Bean的引用注入进Bean对应的属性中

3.如果Bean实现了BeanNameAware接口，Spring将Bean的ID传递给setBeanName()方法   
（实现BeanNameAware清主要是为了通过Bean的引用来获得Bean的ID，一般业务中是很少有用到Bean的ID的）

4.如果Bean实现了BeanFactoryAware接口，Spring将调用setBeanDactory(BeanFactory bf)方法并把BeanFactory容器实例作为参数传入。   
（实现BeanFactoryAware 主要目的是为了获取Spring容器，如Bean通过Spring容器发布事件等）

5.如果Bean实现了ApplicationContextAwaer接口，Spring容器将调用setApplicationContext()方法，把bean所在的应用上下文的引用传入.   
(作用与BeanFactory类似都是为了获取Spring容器，不同的是Spring容器在调用setApplicationContext方法时会把它自己作为setApplicationContext 的参数传入，而Spring容器在调用setBeanDactory前需要程序员自己指定（注入）setBeanDactory里的参数BeanFactory )

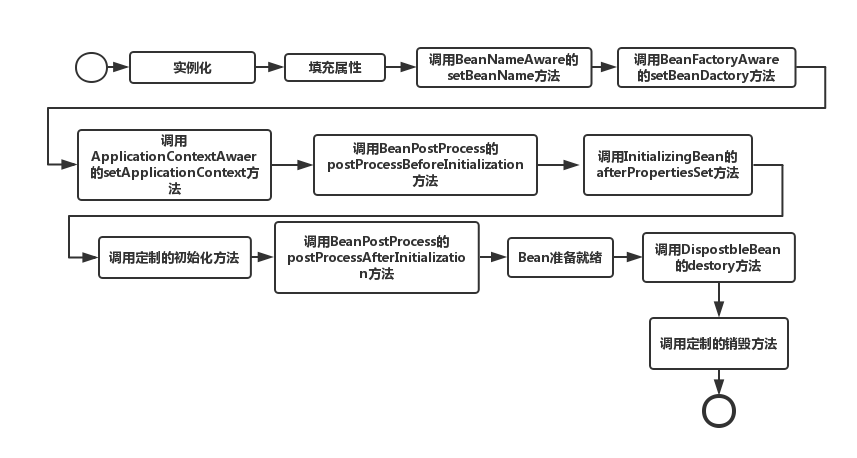
6.如果Bean实现了BeanPostProcess接口，Spring将调用它们的postProcessBeforeInitialization（预初始化）方法   
（作用是在Bean实例创建成功后对进行增强处理，如对Bean进行修改，增加某个功能）

7.如果Bean实现了InitializingBean接口，Spring将调用它们的afterPropertiesSet方法，作用与在配置文件中对Bean使用init-method声明初始化的作用一样，都是在Bean的全部属性设置成功后执行的初始化方法。

8.如果Bean实现了BeanPostProcess接口，Spring将调用它们的postProcessAfterInitialization（后初始化）方法（作用与6的一样，只不过6是在Bean初始化前执行的，而这个是在Bean初始化后执行的，时机不同 )

9.此时，bean已经准备就绪，可以被程序使用了，Bean将一直驻留在应用上下文中给应用使用，直到应用上下文被销毁

10.如果Bean实现了DispostbleBean接口，Spring将调用它的destory方法，作用与在配置文件中对Bean使用destory-method属性的作用一样，都是在Bean实例销毁前执行的方法。



|  |
| --- |
| @Component  **public** **class** UserEntity **implements** BeanNameAware, BeanFactoryAware, ApplicationContextAware, InitializingBean,DisposableBean {  **private** String userName;  **private** Integer age = **null**;  **public** UserEntity() {  System.***out***.println("无惨构造函数.....");  }  **public** UserEntity(String userName, Integer age) {  System.***out***.println("我是有参构造函数 userName:" + userName + ",age:" + age);  **this**.userName = userName;  **this**.age = age;  }  **public** String getUserName() {  **return** userName;  }  **public** **void** setUserName(String userName) {  **this**.userName = userName;  }  **public** Integer getAge() {  **return** age;  }  **public** **void** setAge(Integer age) {  **this**.age = age;  }  @Override  **public** String toString() {  **return** "UserEntity [userName=" + userName + ", age=" + age + "]";  }  **public** **void** setBeanName(String name) {  System.***out***.println("BeanName:" + name);  }  **public** **void** setBeanFactory(BeanFactory beanFactory) **throws** BeansException {  System.***out***.println("setBeanFactory");  }  **public** **void** setApplicationContext(ApplicationContext applicationContext) **throws** BeansException {  System.***out***.println("setApplicationContext");  }  **public** Object postProcessBeforeInitialization(Object bean, String beanName) **throws** BeansException {  System.***out***.println("postProcessBeforeInitialization bean初始化之前" + beanName);  **return** bean;  }  **public** Object postProcessAfterInitialization(Object bean, String beanName) **throws** BeansException {  System.***out***.println("postProcessAfterInitialization bean初始化之后" + beanName);  **return** bean;  }  **public** **void** init(){  System.***out***.println("init()");  }  **public** **void** afterPropertiesSet() **throws** Exception {  System.***out***.println("afterPropertiesSet");  }  **public** **void** destroy() **throws** Exception {  System.***out***.println("destroy 销毁bean");    }  } |
| ClassPathXmlApplicationContext app = **new** ClassPathXmlApplicationContext("spring003.xml");  UserEntity user = (UserEntity) app.getBean("userEntity");  app.destroy(); |

# SpringAop源码分析

## AOP简介

### 概念

切面（Aspect） ：官方的抽象定义为“一个关注点的模块化，这个关注点可能会横切多个对象”。  
连接点（Joinpoint） ：程序执行过程中的某一行为。  
通知（Advice） ：“切面”对于某个“连接点”所产生的动作。  
切入点（Pointcut） ：匹配连接点的断言，在AOP中通知和一个切入点表达式关联。  
目标对象（Target Object） ：被一个或者多个切面所通知的对象。  
AOP代理（AOP Proxy） 在Spring AOP中有两种代理方式，JDK动态代理和CGLIB代理。  
  
**通知（Advice）类型**  
前置通知（Before advice） ：在某连接点（JoinPoint）之前执行的通知，但这个通知不能阻止连接点前的执行。ApplicationContext中在<aop:aspect>里面使用<aop:before>元素进行声明。  
后通知（After advice） ：当某连接点退出的时候执行的通知（不论是正常返回还是异常退出）。ApplicationContext中在<aop:aspect>里面使用<aop:after>元素进行声明。  
返回后通知（After return advice） ：在某连接点正常完成后执行的通知，不包括抛出异常的情况。ApplicationContext中在<aop:aspect>里面使用<after-returning>元素进行声明。  
环绕通知（Around advice） ：包围一个连接点的通知，类似Web中Servlet规范中的Filter的doFilter方法。可以在方法的调用前后完成自定义的行为，也可以选择不执行。ApplicationContext中在<aop:aspect>里面使用<aop:around>元素进行声明。  
抛出异常后通知（After throwing advice） ： 在方法抛出异常退出时执行的通知。 ApplicationContext中在<aop:aspect>里面使用<aop:after-throwing>元素进行声明。

切入点表达式 ：如execution(\* com.spring.service.\*.\*(..))

### 特点

1、降低模块之间的耦合度

2、使系统容易扩展

3、更好的代码复用。

## 流程说明

1）AOP标签的定义解析刘彻骨肯定是从NamespaceHandlerSupport的实现类开始解析的，这个实现类就是AopNamespaceHandler。至于为什么会是从NamespaceHandlerSupport的实现类开始解析的，这个的话我想读者可以去在回去看看Spring自定义标签的解析流程，里面说的比较详细。

2）要启用AOP，我们一般会在Spring里面配置<aop:aspectj-autoproxy/>  ，所以在配置文件中在遇到aspectj-autoproxy标签的时候我们会采用AspectJAutoProxyBeanDefinitionParser解析器

3）进入AspectJAutoProxyBeanDefinitionParser解析器后，调用AspectJAutoProxyBeanDefinitionParser已覆盖BeanDefinitionParser的parser方法，然后parser方法把请求转交给了AopNamespaceUtils的registerAspectJAnnotationAutoProxyCreatorIfNecessary去处理

4）进入AopNamespaceUtils的registerAspectJAnnotationAutoProxyCreatorIfNecessary方法后，先调用AopConfigUtils的registerAspectJAnnotationAutoProxyCreatorIfNecessary方法，里面在转发调用给registerOrEscalateApcAsRequired，注册或者升级AnnotationAwareAspectJAutoProxyCreator类。对于AOP的实现，基本是靠AnnotationAwareAspectJAutoProxyCreator去完成的，它可以根据@point注解定义的切点来代理相匹配的bean。

5）AopConfigUtils的registerAspectJAnnotationAutoProxyCreatorIfNecessary方法处理完成之后，接下来会调用useClassProxyingIfNecessary() 处理proxy-target-class以及expose-proxy属性。如果将proxy-target-class设置为true的话，那么会强制使用CGLIB代理，否则使用jdk动态代理，expose-proxy属性是为了解决有时候目标对象内部的自我调用无法实现切面增强。

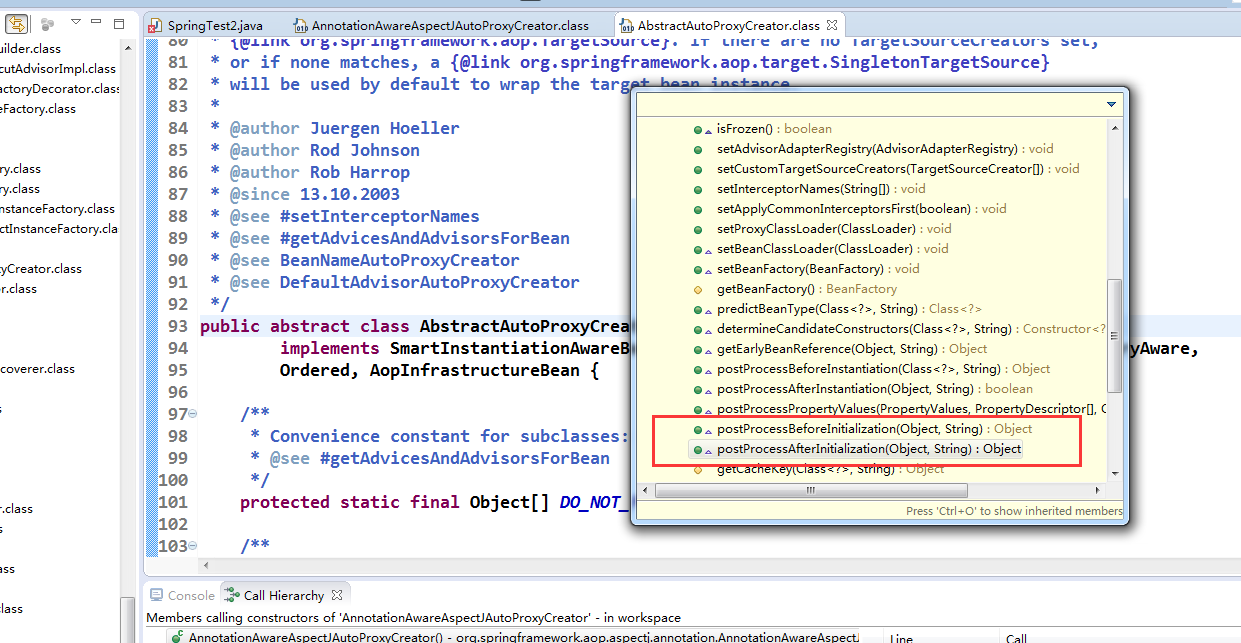
6）最后的调用registerComponentIfNecessary 方法，注册组建并且通知便于监听器做进一步处理。

## 创建AOP代理

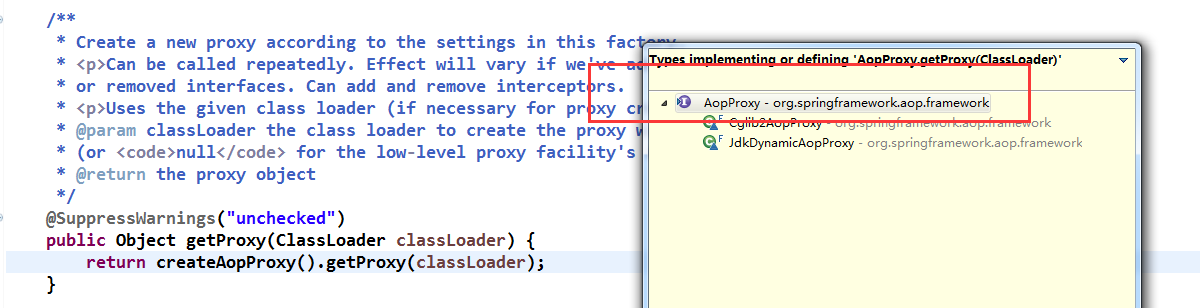
上面说到AOP的核心逻辑是在AnnotationAwareAspectJAutoProxyCreator类里面实现，那么我们先来看看这个类的层次关系

## 流程说明

1. spring 容器启动，每个bean的实例化之前都会先经过AbstractAutoProxyCreator类的postProcessAfterInitialization（）这个方法，然后接下来是调用wrapIfNecessary方法。



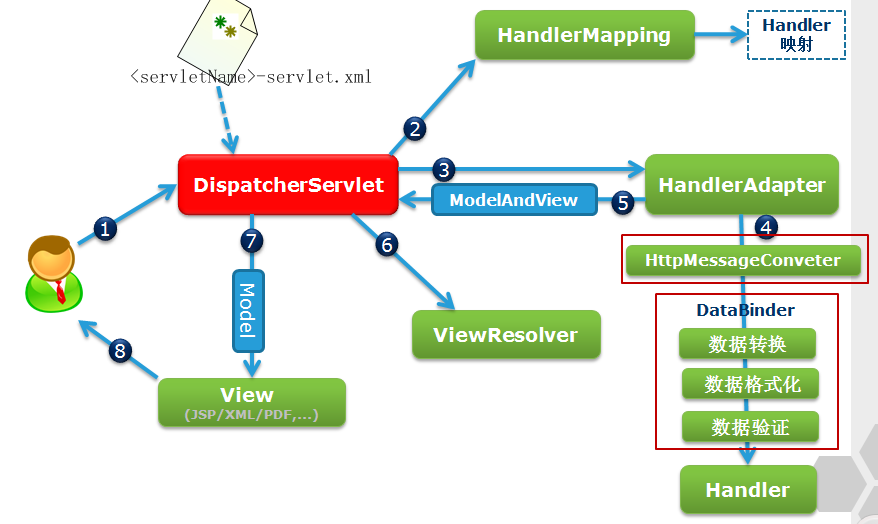
|  |
| --- |
| **public** Object postProcessAfterInitialization(Object bean, String beanName) **throws** BeansException {  **if** (bean != **null**) {  Object cacheKey = getCacheKey(bean.getClass(), beanName);  **if** (!**this**.earlyProxyReferences.contains(cacheKey)) {  **return** wrapIfNecessary(bean, beanName, cacheKey);  }  }  **return** bean;  }  **protected** Object wrapIfNecessary(Object bean, String beanName, Object cacheKey) {  **if** (**this**.targetSourcedBeans.contains(beanName)) {  **return** bean;  }  **if** (**this**.nonAdvisedBeans.contains(cacheKey)) {  **return** bean;  }  **if** (isInfrastructureClass(bean.getClass()) || shouldSkip(bean.getClass(), beanName)) {  **this**.nonAdvisedBeans.add(cacheKey);  **return** bean;  }  // Create proxy if we have advice.  Object[] specificInterceptors = getAdvicesAndAdvisorsForBean(bean.getClass(), beanName, **null**);  **if** (specificInterceptors != ***DO\_NOT\_PROXY***) {  **this**.advisedBeans.add(cacheKey);  Object proxy = createProxy(bean.getClass(), beanName, specificInterceptors, **new** SingletonTargetSource(bean));  **this**.proxyTypes.put(cacheKey, proxy.getClass());  **return** proxy;  }  **this**.nonAdvisedBeans.add(cacheKey);  **return** bean;  }  创建代理对象  **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** (!shouldProxyTargetClass(beanClass, beanName)) {  // Must allow for introductions; can't just set interfaces to  // the target's interfaces only.  Class<?>[] targetInterfaces = ClassUtils.*getAllInterfacesForClass*(beanClass, **this**.proxyClassLoader);  **for** (Class<?> targetInterface : targetInterfaces) {  proxyFactory.addInterface(targetInterface);  }  }  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);  } |



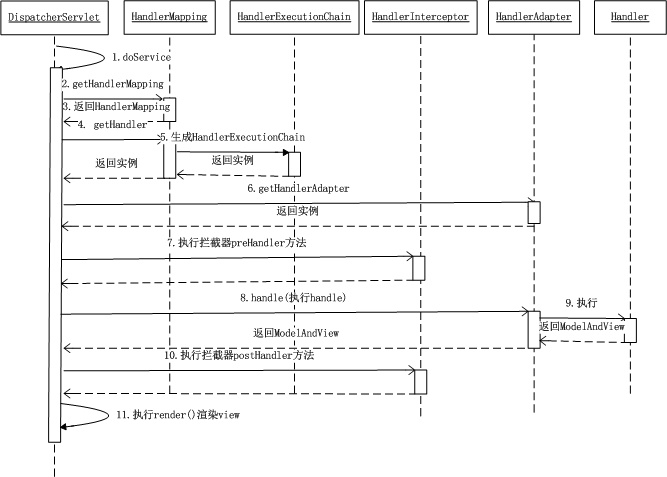
# SpringMVC执行流程

Spring MVC工作流程图

图一



图二



**Spring工作流程描述**

      1. 用户向服务器发送请求，请求被Spring 前端控制Servelt DispatcherServlet捕获；

      2. DispatcherServlet对请求URL进行解析，得到请求资源标识符（URI）。然后根据该URI，调用HandlerMapping获得该Handler配置的所有相关的对象（包括Handler对象以及Handler对象对应的拦截器），最后以HandlerExecutionChain对象的形式返回；

      3. DispatcherServlet 根据获得的Handler，选择一个合适的HandlerAdapter。（**附注**：如果成功获得HandlerAdapter后，此时将开始执行拦截器的preHandler(...)方法）

       4.  提取Request中的模型数据，填充Handler入参，开始执行Handler（Controller)。 在填充Handler的入参过程中，根据你的配置，Spring将帮你做一些额外的工作：

      HttpMessageConveter： 将请求消息（如Json、xml等数据）转换成一个对象，将对象转换为指定的响应信息

      数据转换：对请求消息进行数据转换。如String转换成Integer、Double等

      数据根式化：对请求消息进行数据格式化。 如将字符串转换成格式化数字或格式化日期等

      数据验证： 验证数据的有效性（长度、格式等），验证结果存储到BindingResult或Error中

      5.  Handler执行完成后，向DispatcherServlet 返回一个ModelAndView对象；

      6.  根据返回的ModelAndView，选择一个适合的ViewResolver（必须是已经注册到Spring容器中的ViewResolver)返回给DispatcherServlet ；

      7. ViewResolver 结合Model和View，来渲染视图

      8. 将渲染结果返回给客户端。

Spring工作流程描述

    为什么Spring只使用一个Servlet(DispatcherServlet)来处理所有请求？

     详细见J2EE设计模式-前端控制模式

    Spring为什么要结合使用HandlerMapping以及HandlerAdapter来处理Handler?

    符合面向对象中的单一职责原则，代码架构清晰，便于维护，最重要的是代码可复用性高。如HandlerAdapter可能会被用于处理多种Handler。