



Spring源码分析之IoC容器篇

目录

1. Spring

1.1 Spring源码解析1 IOC容器的初始化	3
1.2 Spring源码解析 2 IOC容器的初始化	13
1.3 Spring源码解析3 IOC容器的初始化	30
1.4 Spring源码解析 依赖注入	33
1.5 Spring源码解析 lazy-init属性和预实例化	56
1.6 Spring源码解析 BeanPostProcessor的实现	58

1.1 Spring源码解析1 IOC容器的初始化

发表时间: 2010-10-01

参考《Spring技术内幕》一书：

IoC容器的基本接口是由BeanFactory来定义的，也就是说BeanFactory定义了IoC容器的最基本的形式，并且提供了 IoC容器所应该遵守的最基本的服务契约。BeanFactory只是一个接口类，并没有给出容器的具体实现。

DefaultListableBeanFactory,XmlBeanFactory,ApplicationContext,FileSystemXmlBeanFactory,ClassPathXmlBeanFactory都实现了BeanFactory接口并且扩展了IoC容器的功能。

首先介绍BeanFactory:

```
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
    String FACTORY_BEAN_PREFIX = "&";

    //这里根据bean的名字，在IOC容器中得到bean实例，这个IOC容器就是一个大的抽象工厂。
    Object getBean(String name) throws BeansException;

    //这里根据bean的名字和Class类型来得到bean实例，和上面的方法不同在于它会抛出异常：如果根据名字取得的
    Object getBean(String name, Class requiredType) throws BeansException;

    //这里提供对bean的检索，看看是否在IOC容器有这个名字的bean
    boolean containsBean(String name);

    //这里根据bean名字得到bean实例，并同时判断这个bean是不是单件
    boolean isSingleton(String name) 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);  
  
}
```

用户使用容器时，可以使用转义字符'&'来得到FactoryBean本身，用来区分通过容器来获取FactoryBean产生的对象还是获取FactoryBean本身。在Spring中所有的Bean都是由BeanFactory来管理的，而对于FactoryBean，它是一个能产生或者修饰对象生成的工厂Bean。BeanFactory和FactoryBean:BeanFactory它指的是IoC容器的编程抽象，而FactoryBean指的是一个抽象工厂，对它的调用返回的是工厂产生的对象，而不是它本身。

我们先通过编程实现IoC容器：

```
public class UserBeanFatory {  
    public static void main(String[] args) {  
        //创建一个BeanFactory,这里使用DefaultListableBeanFactory，包含IoC容器的重要功能  
        DefaultListableBeanFactory factory=new DefaultListableBeanFactory();  
        /*  
        * 创建一个载入BeanDefinition的读取器，这里使用XmlBeanDefinitionReader来载入XML文件  
        * BeanDefinition，使用一个回调配置给BeanFactory  
        */  
        XmlBeanDefinitionReader reader=new XmlBeanDefinitionReader(factory);  
        /*  
        * 创建Ioc配置文件的抽象资源，这个抽象资源中包含了BeanDefinition的定义信息  
        */  
        ClassPathResource res=new ClassPathResource("applicationContext-beans.xml");  
        /*  
        * 从定义好的资源位置读入配置信息，具体的解析过程是由XmlBeanDefinitionReader  
        * 来完成的。完成整个的载入与注册Bean定义之后，需要的IoC容器就建立起来了  
        */  
    }  
}
```

```
    */
    reader.loadBeanDefinitions(res);

    User user=(User)factory.getBean("user");
    System.out.println(user.getUsername()+":"+user.getPassword());
    //等价于
    XmlBeanFactory xmlfactory=new XmlBeanFactory(new ClassPathResource("applicationContext-beans.xml"));
    User xmluser=(User)factory.getBean("user");
    System.out.println(xmluser.getUsername()+":"+xmluser.getPassword());
    ApplicationContext ac=new FileSystemXmlApplicationContext("D:/java/kcsj/SourceCode/beans.xml");
    ac.getBean("user");
}
}
```

由上面我们可以想到IoC 容器初始化分为三个步骤：

- 1 BeanDefinition的Resource定位
- 2 BeanDefinition的载入和解析
- 3 BeanDefinition的注册

我们先看BeanDefinition的Resource定位：

下面以FileSystemXmlApplicationContext为例，通过分析这个ApplicationContext的实现来看看它是怎样完成Resource的定位的。

```
ApplicationContext ac=new FileSystemXmlApplicationContext("D:/java/kcsj/SourceCode/XmpBeanFactory/src/applicationContext-beans.xml");
```

我们首先看看FileSystemXmlApplicationContext的源码：


```
        invokeBeanFactoryPostProcessors(beanFactory);

        // Register bean processors that intercept bean creation.
        registerBeanPostProcessors(beanFactory);

        // Initialize message source for this context.
        initMessageSource();

        // Initialize event multicaster for this context.
        initApplicationEventMulticaster();

        // Initialize other special beans in specific context subclasses
        onRefresh();

        // Check for listener beans and register them.
        registerListeners();

        // Instantiate all remaining (non-lazy-init) singletons.
        finishBeanFactoryInitialization(beanFactory);

        // Last step: publish corresponding event.
        finishRefresh();
    }

    catch (BeansException ex) {
        // Destroy already created singletons to avoid dangling resource
        destroyBeans();

        // Reset 'active' flag.
        cancelRefresh(ex);

        // Propagate exception to caller.
        throw ex;
    }
}
```

它包含了IoC容器的整个初始化的过程，包括：BeanFactory 的更新，初始化messagesource，配置和注册后置处理器，注册监听器和事件触发器，还有进行预实例化(non-lazy-init)的处理等等。它把资源的定位交给了obtainFreshBeanFactory方法：

```
protected ConfigurableListableBeanFactory obtainFreshBeanFactory() {
    refreshBeanFactory();
    ConfigurableListableBeanFactory beanFactory = getBeanFactory();
    if (logger.isDebugEnabled()) {
        logger.debug("Bean factory for " + getDisplayName() + ": " + beanFactory);
    }
    return beanFactory;
}
```

然后又通过调用抽象方法refreshBeanFactory,它的实现在AbstractRefreshableApplicaitonContext中：

```
protected final void refreshBeanFactory() throws BeansException {
    if (hasBeanFactory()) {
        destroyBeans();
        closeBeanFactory();
    }
    try {
        DefaultListableBeanFactory beanFactory = createBeanFactory();
        beanFactory.setSerializationId(getId());
        customizeBeanFactory(beanFactory);
        loadBeanDefinitions(beanFactory);
        synchronized (this.beanFactoryMonitor) {
            this.beanFactory = beanFactory;
        }
    }
    catch (IOException ex) {
        throw new ApplicationContextException("I/O error parsing bean definition");
    }
}
```

这里先判断是否已经建立了BeanFactory,如果建立则销毁并关闭BeanFactory,然后创建BeanFactory,这里创建的是DefaultListableBeanFactory,然后调用loadBeanDefinitions载入BeanDefinition的配置信息。接着我们去看loadBeanDefinitions方法的具体执行过程:

```
protected void loadBeanDefinitions(DefaultListableBeanFactory beanFactory) throws BeansExcepti
    // Create a new XmlBeanDefinitionReader for the given BeanFactory.
    XmlBeanDefinitionReader beanDefinitionReader = new XmlBeanDefinitionReader(beanFacto

    // 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);
}
```

这里它先创建一个BeanDefinition的Xml读取器,并且回调配置给BeanFactory,如同我们前面通过编程实现IoC容器的初始化,然后再转到loadBeanDefinitions(beanDefintionReader)中:

```
protected void loadBeanDefinitions(XmlBeanDefinitionReader reader) throws BeansException, IOExc
    Resource[] configResources = getConfigResources();
    if (configResources != null) {
        reader.loadBeanDefinitions(configResources);
    }
    String[] configLocations = getConfigLocations();
    if (configLocations != null) {
        reader.loadBeanDefinitions(configLocations);
    }
}
```

它首先获得BeanDefinition的配置文件的资源,判断是否存在,如果存在在加载,然后获取配置文件的路径,判断是否存在,如果存在则加载。一种是从资源中加载,另一种是从给定的路径中加载。由于我们在没有显式

的定义资源，我们只是给定了一个配置文件的路径，所以它会从路径加载。也就是调用 `reader.loadBeanDefinitions(configLocations)`方法。

```
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;
}
```

然后继续调用 `loadBeanDefinitions(location)`方法：

```
public int loadBeanDefinitions(String location) throws BeanDefinitionStoreException {
    return loadBeanDefinitions(location, null);
}
```

再转到：`loadBeanDefinitions(String location, Set<Resource> actualResources)`方法中

```
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 + "] because no ResourceLoader is configured");
    }

    if (resourceLoader instanceof ResourcePatternResolver) {
        // Resource pattern matching available.
        try {
            Resource[] resources = ((ResourcePatternResolver) resourceLoader).getResources(location);
            int loadCount = loadBeanDefinitions(resources);
            if (actualResources != null) {
                for (Resource resource : resources) {
                    actualResources.add(resource);
                }
            }
        } catch (IOException ex) {
            throw new BeanDefinitionStoreException("Could not load bean definitions from location [" + location + "]: " + ex.getMessage(), ex);
        }
    } else {
        // Resource pattern matching not available.
        Resource resource = resourceLoader.getResource(location);
        int loadCount = loadBeanDefinitions(resource);
        if (actualResources != null) {
            actualResources.add(resource);
        }
    }
    return loadCount;
}
```

```

        }
        if (logger.isDebugEnabled()) {
            logger.debug("Loaded " + loadCount + " bean definitions
        }
        return loadCount;
    }
    catch (IOException ex) {
        throw new BeanDefinitionStoreException(
            "Could not resolve bean definition resource pat
    }
}
else {
    // Can only load single resources by absolute URL.
    Resource resource = resourceLoader.getResource(location);
    int loadCount = loadBeanDefinitions(resource);
    if (actualResources != null) {
        actualResources.add(resource);
    }
    if (logger.isDebugEnabled()) {
        logger.debug("Loaded " + loadCount + " bean definitions from lc
    }
    return loadCount;
}
}
}

```

Resource resource = resourceLoader.getResource(location);就是用来定位BeanDefinition的资源，它会交给DefaultResourceLoader的getResource()方法处理：

```

public Resource getResource(String location) {
    Assert.notNull(location, "Location must not be null");
    if (location.startsWith(CLASSPATH_URL_PREFIX)) {
        return new ClassPathResource(location.substring(CLASSPATH_URL_PREFIX.le
    }
    else {
        try {
            // Try to parse the location as a URL...
            URL url = new URL(location);

```

```
        return new UrlResource(url);
    }
    catch (MalformedURLException ex) {
        // No URL -> resolve as resource path.
        return getResourceByPath(location);
    }
}
}
```

最后它又调用`getResourceByPath(location)`,它被`FileSystemXmlApplicationContext`中`getResourceByPath()`覆盖了,具体源码如下:

```
protected Resource getResourceByPath(String path) {
    if (path != null && path.startsWith("/")) {
        path = path.substring(1);
    }
    return new FileSystemResource(path);
}
```

到这里为止, IoC容器的初始化的第一步骤已经完成了,总结可得`BeanDefinition`的`Resource`的定位是通过`DefaultResourceLoader`来`getResource()`方法来定位的,在`getResource()`中又调用`getResourceByPath()`,它被不同的`BeanFactory`覆盖。

1.2 Spring源码解析 2 IOC容器的初始化

发表时间: 2010-10-02

前面我们分析了：IoC容器的第一个步骤BeanDefinition的Resource定位，接下来我们分析BeanDefinition的载入和解析。我们先总的描述一下的BeanDefinition的载入和解析：

BeanDefinition的载入过程包括两部分，首先是通过调用XML的解析器得到Document对象，但这些Document对象并没有按照Spring的Bean规则进行解析。

按照Spring的Bean规则进行解析过程是在DocumentReader中实现的。这里使用默认设置好的DefaultBeanDefinitionDocumentReader。然后再完成对BeanDefinition的处理，处理结果由BeanDefinitionHolder对象来持有。BeanDefinitionHolder是BeanDefinition的封装类，它封装了BeanDefinition，Bean的名字和别名。得到这个BeanDefinitionHolder实际上就得到了BeanDefinition，BeanDefinitionHolder是通过BeanDefinitionParserDelegate对XML元素的信息按照Spring的Bean规则解析得到的。

我们在AbstractBeanDefinitionReader中的loadBeanDefinition方法中得到了BeanDefintion的资源：

```
public int loadBeanDefinitions(String location, Set<Resource> actualResources) throws BeanDefin
    ResourceLoader resourceLoader = getResourceLoader();
    if (resourceLoader == null) {
        throw new BeanDefinitionStoreException(
            "Cannot import bean definitions from location [" + loca
    }

    if (resourceLoader instanceof ResourcePatternResolver) {
        // Resource pattern matching available.
        try {
            Resource[] resources = ((ResourcePatternResolver) resourceLoader
            int loadCount = loadBeanDefinitions(resources);
            if (actualResources != null) {
                for (Resource resource : resources) {
                    actualResources.add(resource);
                }
            }
        }
        if (logger.isDebugEnabled()) {
            logger.debug("Loaded " + loadCount + " bean definitions
```

```
        }
        return loadCount;
    }
    catch (IOException ex) {
        throw new BeanDefinitionStoreException(
            "Could not resolve bean definition resource pat
        }
    }
else {
    // Can only load single resources by absolute URL.
    Resource resource = resourceLoader.getResource(location);
    int loadCount = loadBeanDefinitions(resource);
    if (actualResources != null) {
        actualResources.add(resource);
    }
    if (logger.isDebugEnabled()) {
        logger.debug("Loaded " + loadCount + " bean definitions from lc
    }
    return loadCount;
}
}
```

接着调用loadBeanDefinitions(resource)载入资源，它是一个接口方法，它的实现在XmlBeanDefinitionReader的

loadBeanDefinitions(EncodedResource encodedResource)方法中：

```
public int loadBeanDefinitions(EncodedResource encodedResource) throws BeanDefiniti
    Assert.notNull(encodedResource, "EncodedResource must not be null")
    if (logger.isInfoEnabled()) {
        logger.info("Loading XML bean definitions from " + encodedR
    }

    Set<EncodedResource> currentResources = this.resourcesCurrentlyBein
    if (currentResources == null) {
```

```
        currentResources = new HashSet<EncodedResource>(4);
        this.resourcesCurrentlyBeingLoaded.set(currentResources);
    }
    if (!currentResources.add(encodedResource)) {
        throw new BeanDefinitionStoreException(
            "Detected cyclic loading of " + encodedResource.getName() + " from " +
            this.getResourceUri());
    }
    try {
        InputStream inputStream = encodedResource.getResource().getInputStream();
        try {
            InputSource inputSource = new InputSource(inputStream);
            if (encodedResource.getEncoding() != null) {
                inputSource.setEncoding(encodedResource.getEncoding());
            }
            return doLoadBeanDefinitions(inputSource, encodedResource.getResourceUri());
        }
        finally {
            inputStream.close();
        }
    }
    catch (IOException ex) {
        throw new BeanDefinitionStoreException(
            "IOException parsing XML document from " +
            this.getResourceUri(), ex);
    }
    finally {
        currentResources.remove(encodedResource);
        if (currentResources.isEmpty()) {
            this.resourcesCurrentlyBeingLoaded.set(null);
        }
    }
}
```

它包含了对输入流设置字符编码，然后转到doLoadBeanDefinitions(inputSource, encodedResource.getResource())中：

```
protected int doLoadBeanDefinitions(InputSource inputSource, Resource resource)
    throws BeanDefinitionStoreException {
    try {
        int validationMode = getValidationModeForResource(resource)
        Document doc = this.documentLoader.loadDocument(
            inputSource, getEntityResolver(), this.errorHandler);
        return registerBeanDefinitions(doc, resource);
    }
    catch (BeanDefinitionStoreException ex) {
        throw ex;
    }
    catch (SAXParseException ex) {
        throw new XmlBeanDefinitionStoreException(resource.getDescription(),
            "Line " + ex.getLineNumber() + " in XML document from " + resource);
    }
    catch (SAXException ex) {
        throw new XmlBeanDefinitionStoreException(resource.getDescription(),
            "XML document from " + resource + " is invalid");
    }
    catch (ParserConfigurationException ex) {
        throw new BeanDefinitionStoreException(resource.getDescription(),
            "Parser configuration exception parsing XML from " + resource);
    }
    catch (IOException ex) {
        throw new BeanDefinitionStoreException(resource.getDescription(),
            "IOException parsing XML document from " + resource);
    }
    catch (Throwable ex) {
        throw new BeanDefinitionStoreException(resource.getDescription(),
            "Unexpected exception parsing XML document from " + resource);
    }
}
```

```
    }  
}
```

这里首先取得XML文件的Document的对象，这个解析过程交给了documentLoader完成。接着就是转到registerBeanDefinitions(doc, resource)方法中：

```
public int registerBeanDefinitions(Document doc, Resource resource) throws BeanDefi  
    // Read document based on new BeanDefinitionDocumentReader SPI.  
    BeanDefinitionDocumentReader documentReader = createBeanDefinitionD  
    int countBefore = getRegistry().getBeanDefinitionCount();  
    documentReader.registerBeanDefinitions(doc, createReaderContext(res  
    return getRegistry().getBeanDefinitionCount() - countBefore;  
}
```

统计注册的BeanDefinition的个数。创建一个BeanDefinitionDocumentReader来解析Xml文件形式的BeanDefinition.

```
public void registerBeanDefinitions(Document doc, XmlReaderContext readerContext) {  
    this.readerContext = readerContext;  
  
    logger.debug("Loading bean definitions");  
    Element root = doc.getDocumentElement();
```

```
        BeanDefinitionParserDelegate delegate = createHelper(readerContext,  
  
        preProcessXml(root);  
        parseBeanDefinitions(root, delegate);  
        postProcessXml(root);  
    }
```

具体的操作委托给BeanDefinitionParserDelegate：

```
protected void parseBeanDefinitions(Element root, BeanDefinitionParserDelegate dele  
    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);  
                }  
                else {  
                    delegate.parseCustomElement(ele);  
                }  
            }  
        }  
    }  
    else {  
        delegate.parseCustomElement(root);  
    }  
}
```

然后再到parseDefaultElement(ele, delegate)包含了具体的解析过程：

```
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);
    }
}
```

判断节点是否是import节点，alias节点，bean节点，对于不同的节点分别调用不同的函数进行解析。我们重点看解析bean节点的具体步骤：

```
protected void processBeanDefinition(Element ele, BeanDefinitionParserDelegate delegate) {
    BeanDefinitionHolder bdHolder = delegate.parseBeanDefinitionElement(ele);
    if (bdHolder != null) {
        bdHolder = delegate.decorateBeanDefinitionIfRequired(ele, bdHolder);
        try {
            // Register the final decorated instance.
            BeanDefinitionReaderUtils.registerBeanDefinition(bdHolder, this);
        }
    }
}
```

```
    }  
    catch (BeanDefinitionStoreException ex) {  
        getReaderContext().error("Failed to register bean c  
            bdHolder.getBeanName() + "'", ele,  
    }  
    // Send registration event.  
    getReaderContext().fireComponentRegistered(new BeanComponen  
    }  
}
```

这里他创建了一个BeanDefinitionHolder 用来封装BeanDefinition。详细源码如下：

```
public class BeanDefinitionHolder implements BeanMetadataElement {  
  
    private final BeanDefinition beanDefinition;  
  
    private final String beanName;  
  
    private final String[] aliases;  
        .....  
}
```

然后注册BeanDefinition,注册具体的步骤我们放在下一次讲，注册完成后发送消息。

具体的Spring BeanDefinition的解析式在BeanDefinitionParserDelegate中完成的。这个类中包含了Spring Bean 定义规则的处理。其中包含了id,name,aliase等属性元素以及BeanDefinition，把这些属性元素读出来，设置到BeanDefinitionHolder中。

```
public BeanDefinitionHolder parseBeanDefinitionElement(Element ele, BeanDefinition  
    String id = ele.getAttribute(ID_ATTRIBUTE);  
    String nameAttr = ele.getAttribute(NAME_ATTRIBUTE);  
  
    List<String> aliases = new ArrayList<String>();  
    if (StringUtils.hasLength(nameAttr)) {
```

```
String[] nameArr = StringUtils.tokenizeToStringArray(nameAt
aliases.addAll(Arrays.asList(nameArr));
}

String beanName = id;
if (!StringUtils.hasText(beanName) && !aliases.isEmpty()) {
    beanName = aliases.remove(0);
    if (logger.isDebugEnabled()) {
        logger.debug("No XML 'id' specified - using '" + be
            "' as bean name and " + aliases + "
    }
}

if (containingBean == null) {
    checkNameUniqueness(beanName, aliases, ele);
}

AbstractBeanDefinition beanDefinition = parseBeanDefinitionElement(
if (beanDefinition != null) {
    if (!StringUtils.hasText(beanName)) {
        try {
            if (containingBean != null) {
                beanName = BeanDefinitionReaderUtil
                    beanDefinition, thi
            }
            else {
                beanName = this.readerContext.gener
                // Register an alias for the plain
                // if the generator returned the cl
                // This is expected for Spring 1.2/
                String beanClassName = beanDefiniti
                if (beanClassName != null &&
                    beanName.startsWith
                    !this.readerContext
                aliases.add(beanClassName);
            }
        }
    }
}
```

```

        }
        if (logger.isDebugEnabled()) {
            logger.debug("Neither XML 'id' nor
                "using generated be
        }
    }
    catch (Exception ex) {
        error(ex.getMessage(), ele);
        return null;
    }
}
String[] aliasesArray = StringUtils.toStringArray(aliases);
return new BeanDefinitionHolder(beanDefinition, beanName, a
}

return null;
}

```

AbstractBeanDefinition beanDefinition = parseBeanDefinitionElement(ele, beanName, containingBean);我们创建了一个AbstractBeanDefinition 对象，它实现了BeanDefinition接口。

```

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,
parseBeanDefinitionAttributes(ele, beanName, containingBean
bd.setDescription(DomUtils.getChildElementValueByTagName(ele
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
}
catch (Throwable ex) {
    error("Unexpected failure during bean definition parsing",
}
finally {
    this.parseState.pop();
}

return null;
```

```
}
```

首先读取<bean>中设置的class名字，然后获取parent属性，创建一个AbstractBeanDefinition然后将class和parent设置到BeanDefinition中，然后解析bean元素的各种节点(包括description,meta,lookup-method,constructor-arg,replaced-method,qualifier,property).我们主要看一下如何解析property节点的：

```
public void parsePropertyElements(Element beanEle, BeanDefinition bd) {
    NodeList nl = beanEle.getChildNodes();
    for (int i = 0; i < nl.getLength(); i++) {
        Node node = nl.item(i);
        if (isCandidateElement(node) && nodeNameEquals(node, PROPER
            parsePropertyElement((Element) node, bd);
        }
    }
}
```

```
public void parsePropertyElement(Element ele, BeanDefinition bd) {
    String propertyName = ele.getAttribute(NAME_ATTRIBUTE);
    if (!StringUtils.hasLength(propertyName)) {
        error("Tag 'property' must have a 'name' attribute", ele);
        return;
    }
    this.parseState.push(new PropertyEntry(propertyName));
    try {
        if (bd.getPropertyValues().contains(propertyName)) {
            error("Multiple 'property' definitions for property
            return;
        }
        Object val = parsePropertyValue(ele, bd, propertyName);
        PropertyValue pv = new PropertyValue(propertyName, val);
        parseMetaElements(ele, pv);
    }
}
```

```
        pv.setSource(extractSource(ele));
        bd.getPropertyValues().addPropertyValue(pv);
    }
    finally {
        this.parseState.pop();
    }
}
```

上面是解析property值的地方，返回的对象对应Bean定义的property属性设置的解析的结果，这个解析的结果会被封装到PropertyValue中，然后设置到BeanDefinitionHolder中。具体的解析过程如下：

```
public Object parsePropertyValue(Element ele, BeanDefinition bd, String propertyName,
    String elementName = (propertyName != null) ?
        "<property> element for property '" +
        propertyName + "' : " +
        "<constructor-arg> element";

    // Should only have one child element: ref, value, list, etc.
    NodeList nl = ele.getChildNodes();
    Element subElement = null;
    for (int i = 0; i < nl.getLength(); i++) {
        Node node = nl.item(i);
        if (node instanceof Element && !nodeNameEquals(node, DESCRIPTION_ELEMENT) &&
            !nodeNameEquals(node, META_ELEMENT)) {
            // Child element is what we're looking for.
            if (subElement != null) {
                error(elementName + " must not contain more than one child element");
            }
            else {
                subElement = (Element) node;
            }
        }
    }

    boolean hasRefAttribute = ele.hasAttribute(REF_ATTRIBUTE);
    boolean hasValueAttribute = ele.hasAttribute(VALUE_ATTRIBUTE);
    if ((hasRefAttribute && hasValueAttribute) ||
```

```

        ((hasRefAttribute || hasValueAttribute) && subElement
error(elementName +
        " is only allowed to contain either 'ref' a
    }

    if (hasRefAttribute) {
        String refName = ele.getAttribute(REF_ATTRIBUTE);
        if (!StringUtils.hasText(refName)) {
            error(elementName + " contains empty 'ref' attribut
        }
        RuntimeBeanReference ref = new RuntimeBeanReference(refName
        ref.setSource(extractSource(ele));
        return ref;
    }
    else if (hasValueAttribute) {
        TypedStringValue valueHolder = new TypedStringValue(ele.get
        valueHolder.setSource(extractSource(ele));
        return valueHolder;
    }
    else if (subElement != null) {
        return parsePropertySubElement(subElement, bd);
    }
    else {
        // Neither child element nor "ref" or "value" attribute fou
        error(elementName + " must specify a ref or value", ele);
        return null;
    }
}
}
}

```

这里主要判断property的属性的值是ref还是value,不允许同时是value和ref,如果是ref则创建一个ref的数据对象RuntimeBeanReference ref = new RuntimeBeanReference(refName)这个数据对象封装了ref的信息。如果是value则创建一个TypeStringValue对象TypedStringValue valueHolder = new TypedStringValue(ele.getAttribute(VALUE_ATTRIBUTE))这个数据对象封装了value 的信息。

如果还有子元素,则触发了对子元素的解析parsePropertySubElement(subElement, bd)

```
public Object parsePropertySubElement(Element ele, BeanDefinition bd, String defaultValueType)
    if (!isDefaultNamespace(ele)) {
        return parseNestedCustomElement(ele, bd);
    }
    else if (nodeNameEquals(ele, BEAN_ELEMENT)) {
        BeanDefinitionHolder nestedBd = parseBeanDefinitionElement(ele, bd);
        if (nestedBd != null) {
            nestedBd = decorateBeanDefinitionIfRequired(ele, nestedBd, bd);
        }
        return nestedBd;
    }
    else if (nodeNameEquals(ele, REF_ELEMENT)) {
        // A generic reference to any name of any bean.
        String refName = ele.getAttribute(BEAN_REF_ATTRIBUTE);
        boolean toParent = false;
        if (!StringUtils.hasLength(refName)) {
            // A reference to the id of another bean in the same XML file.
            refName = ele.getAttribute(LOCAL_REF_ATTRIBUTE);
            if (!StringUtils.hasLength(refName)) {
                // A reference to the id of another bean in a parent context
                refName = ele.getAttribute(PARENT_REF_ATTRIBUTE);
                toParent = true;
                if (!StringUtils.hasLength(refName)) {
                    error("'bean', 'local' or 'parent' is required", ele);
                    return null;
                }
            }
        }
        if (!StringUtils.hasText(refName)) {
            error("<ref> element contains empty target attribute", ele);
            return null;
        }
        RuntimeBeanReference ref = new RuntimeBeanReference(refName, toParent);
        ref.setSource(extractSource(ele));
        return ref;
    }
    else if (nodeNameEquals(ele, IDREF_ELEMENT)) {
```

```
        return parseIdRefElement(ele);
    }
    else if (nodeNameEquals(ele, VALUE_ELEMENT)) {
        return parseValueElement(ele, defaultValueType);
    }
    else if (nodeNameEquals(ele, NULL_ELEMENT)) {
        // It's a distinguished null value. Let's wrap it in a TypedStringValue
        // object in order to preserve the source location.
        TypedStringValue nullHolder = new TypedStringValue(null);
        nullHolder.setSource(extractSource(ele));
        return nullHolder;
    }
    else if (nodeNameEquals(ele, ARRAY_ELEMENT)) {
        return parseArrayElement(ele, bd);
    }
    else if (nodeNameEquals(ele, LIST_ELEMENT)) {
        return parseListElement(ele, bd);
    }
    else if (nodeNameEquals(ele, SET_ELEMENT)) {
        return parseSetElement(ele, bd);
    }
    else if (nodeNameEquals(ele, MAP_ELEMENT)) {
        return parseMapElement(ele, bd);
    }
    else if (nodeNameEquals(ele, PROPS_ELEMENT)) {
        return parsePropsElement(ele);
    }
    else {
        error("Unknown property sub-element: [" + ele.getNodeName() + "]", ele);
        return null;
    }
}
```

这里分别有各种属性值对象的解析过程。

经过这样逐层的解析，定义BeanDefinition的Xml配置文件被加载到了IoC容器中，并且在容器中建立起相应的数据映射，这些数据结构是以AbstractBeanDefinition为入口点的，你可以让IoC容器执行各种操作。IoC容器中存在只是一些静态的配置信息。要发挥容器的作用还需完成数据项容器的注册。

1.3 Spring源码解析3 IOC容器的初始化

发表时间: 2010-10-02

上一次我们了解BeanDefinition的在载入和解析，现在我们来看一下BeanDefinition的注册过程：

当BeanDefinition在IoC容器中载入和解析完成后，用户定义的BeanDefinition信息已经在IoC容器内建立起了自己的数据结构以及相应的数据表示，在DefaultListableBeanFactory中通过一个HashMap来持有载入的BeanDefinition。

DefaultBeanDefinitionDocumentReader

```
private final Map<String, BeanDefinition> beanDefinitionMap = new ConcurrentHashMap<String, BeanDefinition>();  
<!--EndFragment-->
```

```
protected void processBeanDefinition(Element ele, BeanDefinitionParserDelegate delegate) {  
    /*按照Spring的Bean规则解析得到的了BeanDefinition的封装类BeanDefinitionHolder后,调用BeanDefinitionReaderUtil  
    BeanDefinitionHolder bdHolder = delegate.parseBeanDefinitionElement(ele);  
    if (bdHolder != null) {  
        bdHolder = delegate.decorateBeanDefinitionIfRequired(ele, bdHolder);  
        try {  
            // Register the final decorated instance.  
            BeanDefinitionReaderUtils.registerBeanDefinition(bdHolder, getReaderContext());  
        }  
        catch (BeanDefinitionStoreException ex) {  
            getReaderContext().error("Failed to register bean definition with name '" +  
                bdHolder.getBeanName() + "'", ele, ex);  
        }  
        // Send registration event.  
        getReaderContext().fireComponentRegistered(new BeanComponentDefinition(bdHolder));  
    }  
}
```

BeanDefinitionReaderUtils :

```
<!--EndFragment-->
```

```
public static void registerBeanDefinition(  
    BeanDefinitionHolder definitionHolder, BeanDefinitionRegistry registry)
```

```
        throws BeanDefinitionStoreException {

        // Register bean definition under primary name.
        String beanName = definitionHolder.getBeanName();
//真正的BeanDefinition的注册在这里,它调用DefaultListableBeanFactory
//下的registerBeanDefinition完成了注册
        registry.registerBeanDefinition(beanName, definitionHolder.getBeanDefinition());

        // Register aliases for bean name, if any.
        String[] aliases = definitionHolder.getAliases();
        if (aliases != null) {
            for (String alias : aliases) {
                registry.registerAlias(beanName, alias);
            }
        }
    }
}
```

DefaultListableBeanFactory:

```
public void registerBeanDefinition(String beanName, BeanDefinition beanDefinition)
        throws BeanDefinitionStoreException {

    Assert.hasText(beanName, "Bean name must not be empty");
    Assert.notNull(beanDefinition, "BeanDefinition must not be null");

    if (beanDefinition instanceof AbstractBeanDefinition) {
        try {
            ((AbstractBeanDefinition) beanDefinition).validate();
        }
        catch (BeanDefinitionValidationException ex) {
            throw new BeanDefinitionStoreException(beanDefinition.getResource()
                + "Validation of bean definition failed", ex);
        }
    }

    synchronized (this.beanDefinitionMap) {
```

```
Object oldBeanDefinition = this.beanDefinitionMap.get(beanName);
if (oldBeanDefinition != null) {
    if (!this.allowBeanDefinitionOverriding) {
        throw new BeanDefinitionStoreException(beanDefinition.getDisplayName(),
            "Cannot register bean definition [" + beanDefinition.getDisplayName() +
            "]: There is already [" + oldBeanDefinition.getDisplayName() +
            "] defined for the same name [" + beanName + "].");
    }
    else {
        if (this.logger.isInfoEnabled()) {
            this.logger.info("Overriding bean definition for bean '" + beanName +
                "': replacing [" + oldBeanDefinition.getDisplayName() +
                "] with [" + beanDefinition.getDisplayName() + "].");
        }
    }
}
else {
    this.beanDefinitionNames.add(beanName);
    this.frozenBeanDefinitionNames = null;
}
this.beanDefinitionMap.put(beanName, beanDefinition);

resetBeanDefinition(beanName);
}
}
```

这是正常的注册过程，把Bean的名字存入到BeanDefinitionNames的同时，将beanName作为Map的key,把beanDefinition作为value存入到IoC容器持有的beanDefinitionMap中去。完成了BeanDefinition的注册，就完成了IoC容器的初始化过程。现在我们可以使用这些BeanDefinition了。

<!--EndFragment-->

1.4 Spring源码解析 依赖注入

发表时间: 2010-10-02

当IoC容器的初始化完毕后，我们就要接触IoC容器的核心功能：依赖注入

在基本的IoC容器接口BeanFactory中，有一个getBean的接口方法的定义，这个接口的实现就是触发依赖注入发生的地方。我们从DefaultListableBeanFactory的基类AbstractBeanFactory入手了解getBean()的实现：

```
public Object getBean(String name) throws BeansException {  
    return doGetBean(name, null, null, false);  
}
```

这里的getBean接口方法最终通过调用doGetBean来实现的，也就是触发依赖注入发生的地方。

```
protected <T> T doGetBean(  
    final String name, final Class<T> requiredType, final Object[] args, boolean  
    throws BeansException {  
  
    final String beanName = transformedBeanName(name);  
    Object bean;  
  
    // Eagerly check singleton cache for manually registered singletons.  
    Object sharedInstance = getSingleton(beanName);  
    if (sharedInstance != null && args == null) {  
        if (logger.isDebugEnabled()) {  
            if (isSingletonCurrentlyInCreation(beanName)) {  
                logger.debug("Returning eagerly cached instance of singleton  
                    '' that is not fully initialized yet -  
            }  
        }  
        else {  
            logger.debug("Returning cached instance of singleton be
```

```
        }
    }
    bean = getObjectForBeanInstance(sharedInstance, name, beanName, null);
}

else {
    // Fail if we're already creating this bean instance:
    // We're assumably within a circular reference.
    if (isPrototypeCurrentlyInCreation(beanName)) {
        throw new BeanCurrentlyInCreationException(beanName);
    }

    // Check if bean definition exists in this factory.
    BeanFactory parentBeanFactory = getParentBeanFactory();
    if (parentBeanFactory != null && !containsBeanDefinition(beanName)) {
        // Not found -> check parent.
        String nameToLookup = originalBeanName(name);
        if (args != null) {
            // Delegation to parent with explicit args.
            return (T) parentBeanFactory.getBean(nameToLookup, args);
        }
        else {
            // No args -> delegate to standard getBean method.
            return parentBeanFactory.getBean(nameToLookup, required);
        }
    }

    if (!typeCheckOnly) {
        markBeanAsCreated(beanName);
    }

    final RootBeanDefinition mbd = getMergedLocalBeanDefinition(beanName);
    checkMergedBeanDefinition(mbd, beanName, args);

    // Guarantee initialization of beans that the current bean depends on.
    String[] dependsOn = mbd.getDependsOn();
    if (dependsOn != null) {
```

```
        for (String dependsOnBean : dependsOn) {
            getBean(dependsOnBean);
            registerDependentBean(dependsOnBean, beanName);
        }
    }

    // Create bean instance.
    if (mbd.isSingleton()) {
        sharedInstance = getSingleton(beanName, new ObjectFactory() {
            public Object getObject() throws BeansException {
                try {
                    return createBean(beanName, mbd, args);
                }
                catch (BeansException ex) {
                    // Explicitly remove instance from singleton cache, so
                    // eagerly by the creation process, to avoid returning
                    // null from getSingleton(beanName);
                    destroySingleton(beanName);
                    throw ex;
                }
            }
        });
        bean = getObjectForBeanInstance(sharedInstance, name, beanName, mbd);
    }

    else if (mbd.isPrototype()) {
        // It's a prototype -> create a new instance.
        Object prototypeInstance = null;
        try {
            beforePrototypeCreation(beanName);
            prototypeInstance = createBean(beanName, mbd, args);
        }
        finally {
            afterPrototypeCreation(beanName);
        }
        bean = getObjectForBeanInstance(prototypeInstance, name, beanName, mbd);
    }
}
```

```
else {
    String scopeName = mbd.getScope();
    final Scope scope = this.scopes.get(scopeName);
    if (scope == null) {
        throw new IllegalStateException("No Scope registered for " + scopeName);
    }
    try {
        Object scopedInstance = scope.get(beanName, new ObjectFactory() {
            public Object getObject() throws BeansException {
                beforePrototypeCreation(beanName);
                try {
                    return createBean(beanName, mbd, args);
                }
                finally {
                    afterPrototypeCreation(beanName);
                }
            }
        });
        bean = getObjectForBeanInstance(scopedInstance, name, beanType);
    }
    catch (IllegalStateException ex) {
        throw new BeanCreationException(beanName,
            "Scope '" + scopeName + "' is not active for the current thread: " +
            "consider defining a scoped proxy for this bean if you intend to inject it",
            ex);
    }
}

// Check if required type matches the type of the actual bean instance.
if (requiredType != null && bean != null && !requiredType.isAssignableFrom(bean.getClass())) {
    throw new BeanNotOfRequiredTypeException(name, requiredType, bean.getClass());
}

return (T) bean;
}
```

首先从缓存中取，处理已经被创建过的单件模式的bean，对这种bean的请求不需要重复地创建，接着 `bean = getObjectForBeanInstance(sharedInstance, name, beanName, null)` 主要完成的是判断它是一个 `FactoryBean` 还是普通的 `Bean` 对象，如果是普通的 `Bean` 对象则直接返回，如果是 `FactoryBean`，则创建一个新的对象 `Bean` 返回，具体详细的步骤就不讲了，接着检查 `IoC` 容器里的 `BeanDefinition`，看是否取得我们需要的 `Bean`。如果在当前工厂中取不到，则到双亲 `BeanFactory` 中去取，如果当前的双亲工厂娶不到，那就顺着双亲 `BeanFactory` 链一直向上寻找。 `final RootBeanDefinition mbd = getMergedLocalBeanDefinition(beanName)` 根据 `beanName` 获取到 `BeanDefinition`。
`String[] dependsOn = mbd.getDependsOn()` 取得当前bean的所有依赖 `Bean`，这样会触发 `getBean` 的递归调用，直至渠道一个没有任何依赖的 `bean` 为止。

接着就是创建 `Bean` 的实例，首先它判断的 `Bean` 的生命周期，然后才调用 `createBean` 创建 `Bean` 的实例。然后对创建出来的 `Bean` 进行类型检查，这里的 `bean` 已经是包含依赖关系的 `Bean`。

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

    if (logger.isDebugEnabled()) {
        logger.debug("Creating instance of bean '" + beanName + "'");
    }
    // Make sure bean class is actually resolved at this point.
    resolveBeanClass(mbd, beanName);

    // Prepare method overrides.
    try {
        mbd.prepareMethodOverrides();
    }
    catch (BeanDefinitionValidationException ex) {
        throw new BeanDefinitionStoreException(mbd.getResourceDescription(),
            beanName, "Validation of method overrides failed", ex);
    }

    try {
        // Give BeanPostProcessors a chance to return a proxy instead of the target
        Object bean = resolveBeforeInstantiation(beanName, mbd);
        if (bean != null) {
            return bean;
        }
    }
}
```

```
catch (Throwable ex) {
    throw new BeanCreationException(mbd.getResourceDescription(), beanName,
        "BeanPostProcessor before instantiation of bean failed"
    )
}

Object beanInstance = doCreateBean(beanName, mbd, args);
if (logger.isDebugEnabled()) {
    logger.debug("Finished creating instance of bean '" + beanName + "'");
}
return beanInstance;
}
```

```
resolveBeanClass(mbd, beanName);
```

首先判断需要创建的bean是否可以被实例化，这个类是否可以通过类装载机来载入。

```
Object bean = resolveBeforeInstantiation(beanName, mbd);
```

接着如果bean配置了PostProcessor，那么这里返回的是一个代理

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

这里是创建bean的调用。

```
protected Object doCreateBean(final String beanName, final RootBeanDefinition mbd, final Object
    // 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.getWrappedInstan
    Class beanType = (instanceWrapper != null ? instanceWrapper.getWrappedClass() :

    // Allow post-processors to modify the merged bean definition.
    synchronized (mbd.postProcessingLock) {
```

```
        if (!mbd.postProcessed) {
            applyMergedBeanDefinitionPostProcessors(mbd, beanType, beanName);
            mbd.postProcessed = true;
        }
    }

    // Eagerly cache singletons to be able to resolve circular references
    // even when triggered by lifecycle interfaces like BeanFactoryAware.
    boolean earlySingletonExposure = (mbd.isSingleton() && this.allowCircularReferences
        && !mbd.isSingletonCurrentlyInCreation(beanName));

    if (earlySingletonExposure) {
        if (logger.isDebugEnabled()) {
            logger.debug("Eagerly caching bean '" + beanName +
                "' to allow for resolving potential circular references");
        }
        addSingletonFactory(beanName, new ObjectFactory() {
            public Object getObject() throws BeansException {
                return getEarlyBeanReference(beanName, mbd, bean);
            }
        });
    }

    // Initialize the bean instance.
    Object exposedObject = bean;
    try {
        populateBean(beanName, mbd, instanceWrapper);
        if (exposedObject != null) {
            exposedObject = initializeBean(beanName, exposedObject, mbd);
        }
    }
    catch (Throwable ex) {
        if (ex instanceof BeanCreationException && beanName.equals(((BeanCreationException) ex).getBeanName()))
            throw (BeanCreationException) ex;
        else {
            throw new BeanCreationException(mbd.getResourceDescription(), beanName, ex);
        }
    }
}
```

```
}

if (earlySingletonExposure) {
    Object earlySingletonReference = getSingleton(beanName, false);
    if (earlySingletonReference != null) {
        if (exposedObject == bean) {
            exposedObject = earlySingletonReference;
        }
        else if (!this.allowRawInjectionDespiteWrapping && hasDependent
            String[] dependentBeans = getDependentBeans(beanName);
            Set<String> actualDependentBeans = new LinkedHashSet<St
            for (String dependentBean : dependentBeans) {
                if (!removeSingletonIfCreatedForTypeCheckOnly(c
                    actualDependentBeans.add(dependentBean)
                }
            }
            if (!actualDependentBeans.isEmpty()) {
                throw new BeanCurrentlyInCreationException(bean
                    "Bean with name '" + beanName +
                    StringUtils.collectionToCommaDe
                    "]" in its raw version as part o
                    "wrapped. This means that said
                    "bean. This is often the result
                    "'getBeanNamesOfType' with the
            }
        }
    }
}

// Register bean as disposable.
try {
    registerDisposableBeanIfNecessary(beanName, bean, mbd);
}
catch (BeanDefinitionValidationException ex) {
    throw new BeanCreationException(mbd.getResourceDescription(), beanName,
}
```

```
        return exposedObject;
    }
}
```

BeanWrapper是用来持有创建出来的bean对象的。如果是singleton，先把缓存中的同名bean清除。如果bean首次创建，那么缓存中没有bean的实例，即进入到创建bean。

```
instanceWrapper = createBeanInstance(beanName, mbd, args);
```

```
protected BeanWrapper createBeanInstance(String beanName, RootBeanDefinition mbd, Object[] args) {
    // Make sure bean class is actually resolved at this point.
    Class beanClass = resolveBeanClass(mbd, beanName);

    if (beanClass != null && !Modifier.isPublic(beanClass.getModifiers()) && !mbd.isAllowNonPublicAccess())
        throw new BeanCreationException(mbd.getResourceDescription(), beanName,
            "Bean class isn't public, and non-public access not allowed");

    if (mbd.getFactoryMethodName() != null) {
        return instantiateUsingFactoryMethod(beanName, mbd, args);
    }

    // Shortcut when re-creating the same bean...
    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);
    }
}

// Need to determine the constructor...
Constructor[] ctors = determineConstructorsFromBeanPostProcessors(beanClass, beanName);
if (ctors != null ||
    mbd.getResolvedAutowireMode() == RootBeanDefinition.AUTOWIRE_CONSTRUCTOR ||
    mbd.hasConstructorArgumentValues() || !ObjectUtils.isEmpty(args))
    return autowireConstructor(beanName, mbd, ctors, args);

// No special handling: simply use no-arg constructor.
return instantiateBean(beanName, mbd);
}

```

这里包含使用工厂方法对Bean进行实例化；使用构造函数对Bean进行实例化；使用默认构造函数对Bean进行实例化，我们接下来看看最常见的实例化过程instantiateBean：

```

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() {
                public Object run() {
                    return getInstantiationStrategy().instantiate(beanName, mbd, 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,
    }
}
```

这里我们使用默认的实例化策略对Bean进行实例化，默认的实例化策略是

CglibSubclassingInstantiationStrategy，也就是用cglib来对bean进行实例化。

<!--EndFragment-->

首先取得在BeanFactory中设置的property的值，这些值来自对BeanDefinition的解析。接着开始进行依赖注入的过程，先处理autowire的注入接着对属性进行注入。接下来我们看看具体对属性进行解析和注入的过程。

```
protected void applyPropertyValues(String beanName, BeanDefinition mbd, BeanWrapper bw, PropertyValues pvs) {
    if (pvs == null || pvs.isEmpty()) {
        return;
    }

    MutablePropertyValues mpvs = null;
    List<PropertyValue> original;

    if (System.getSecurityManager() != null) {
        if (bw instanceof BeanWrapperImpl) {
            ((BeanWrapperImpl) bw).setSecurityContext(getAccessControlContext());
        }
    }

    if (pvs instanceof MutablePropertyValues) {
        mpvs = (MutablePropertyValues) pvs;
        if (mpvs.isConverted()) {
            // Shortcut: use the pre-converted values as-is.
            try {
                bw.setPropertyValues(mpvs);
                return;
            }
        }
    }
}
```

```
        catch (BeansException ex) {
            throw new BeanCreationException(
                mbd.getResourceDescription(), beanName,
            )
        }
        original = mpvs.getPropertyValueList();
    }
    else {
        original = Arrays.asList(pvs.getPropertyValues());
    }

    TypeConverter converter = getCustomTypeConverter();
    if (converter == null) {
        converter = bw;
    }
    BeanDefinitionValueResolver valueResolver = new BeanDefinitionValueResolver(this)

    // Create a deep copy, resolving any references for values.
    List<PropertyValue> deepCopy = new ArrayList<PropertyValue>(original.size());
    boolean resolveNecessary = false;
    for (PropertyValue pv : original) {
        if (pv.isConverted()) {
            deepCopy.add(pv);
        }
        else {
            String propertyName = pv.getName();
            Object originalValue = pv.getValue();
            Object resolvedValue = valueResolver.resolveValueIfNecessary(pv, originalValue);
            Object convertedValue = resolvedValue;
            boolean convertible = bw.isWritableProperty(propertyName) &&
                !PropertyAccessorUtils.isNestedOrIndexedProperty(propertyName);
            if (convertible) {
                convertedValue = convertForProperty(resolvedValue, propertyName, converter);
            }
            // Possibly store converted value in merged bean definition,
            // in order to avoid re-conversion for every created bean instance
            if (resolvedValue == originalValue) {
                if (convertible) {
                    convertedValue = originalValue;
                }
            }
            else {
                deepCopy.add(new PropertyValue(pv.getName(), convertedValue));
            }
        }
    }
    return deepCopy;
```

```

        if (convertible) {
            pv.setConvertedValue(convertedValue);
        }
        deepCopy.add(pv);
    }
    else if (convertible && originalValue instanceof TypedStringValue
            !((TypedStringValue) originalValue).isDynamic()
            !(convertedValue instanceof Collection || Object
        pv.setConvertedValue(convertedValue);
        deepCopy.add(pv);
    }
    else {
        resolveNecessary = true;
        deepCopy.add(new PropertyValue(pv, convertedValue));
    }
}

}

if (mpvs != null && !resolveNecessary) {
    mpvs.setConverted();
}

// Set our (possibly massaged) deep copy.
try {
    bw.setPropertyValues(new MutablePropertyValues(deepCopy));
}
catch (BeansException ex) {
    throw new BeanCreationException(
        mbd.getResourceDescription(), beanName, "Error setting
}
}
}

```

首先为解析值创建一个拷贝，拷贝的数据将会被注入到Bean中，最后设置到BeanWrapper中。

```
public Object resolveValueIfNecessary(Object argName, Object value) {
    // We must check each value to see whether it requires a runtime reference
    // to another bean to be resolved.
    if (value instanceof RuntimeBeanReference) {
        RuntimeBeanReference ref = (RuntimeBeanReference) value;
        return resolveReference(argName, ref);
    }
    else if (value instanceof RuntimeBeanNameReference) {
        String refName = ((RuntimeBeanNameReference) value).getBeanName();
        refName = String.valueOf(evaluate(refName));
        if (!this.beanFactory.containsBean(refName)) {
            throw new BeanDefinitionStoreException(
                "Invalid bean name '" + refName + "' in bean re
            )
        }
        return refName;
    }
    else if (value instanceof BeanDefinitionHolder) {
        // Resolve BeanDefinitionHolder: contains BeanDefinition with name and
        BeanDefinitionHolder bdHolder = (BeanDefinitionHolder) value;
        return resolveInnerBean(argName, bdHolder.getBeanName(), bdHolder.getBe
    }
    else if (value instanceof BeanDefinition) {
        // Resolve plain BeanDefinition, without contained name: use dummy name
        BeanDefinition bd = (BeanDefinition) value;
        return resolveInnerBean(argName, "(inner bean)", bd);
    }
    else if (value instanceof ManagedArray) {
        // May need to resolve contained runtime references.
        ManagedArray array = (ManagedArray) value;
        Class elementType = array.resolvedElementType;
        if (elementType == null) {
            String elementTypeName = array.getElementTypeName();
            if (StringUtils.hasText(elementTypeName)) {
                try {
                    elementType = ClassUtils.forName(elementTypeNam
                    array.resolvedElementType = elementType;
                }
            }
        }
    }
}
```

```
        catch (Throwable ex) {
            // Improve the message by showing the context.
            throw new BeanCreationException(
                this.beanDefinition.getResourceLocation(),
                "Error resolving array type for " + argName, ex);
        }
    }
    else {
        elementType = Object.class;
    }
}
return resolveManagedArray(argName, (List<?>) value, elementType);
}
else if (value instanceof ManagedList) {
    // May need to resolve contained runtime references.
    return resolveManagedList(argName, (List<?>) value);
}
else if (value instanceof ManagedSet) {
    // May need to resolve contained runtime references.
    return resolveManagedSet(argName, (Set<?>) value);
}
else if (value instanceof ManagedMap) {
    // May need to resolve contained runtime references.
    return resolveManagedMap(argName, (Map<?, ?>) value);
}
else if (value instanceof ManagedProperties) {
    Properties original = (Properties) value;
    Properties copy = new Properties();
    for (Map.Entry propEntry : original.entrySet()) {
        Object propKey = propEntry.getKey();
        Object propValue = propEntry.getValue();
        if (propKey instanceof TypedStringValue) {
            propKey = evaluate((TypedStringValue) propKey);
        }
        if (propValue instanceof TypedStringValue) {
            propValue = evaluate((TypedStringValue) propValue);
        }
    }
}
```

```

        copy.put(propKey, propValue);
    }
    return copy;
}
else if (value instanceof TypedStringValue) {
    // Convert value to target type here.
    TypedStringValue typedStringValue = (TypedStringValue) value;
    Object valueObject = evaluate(typedStringValue);
    try {
        Class<?> resolvedTargetType = resolveTargetType(typedStringValue.getType());
        if (resolvedTargetType != null) {
            return this.typeConverter.convertIfNecessary(valueObject, resolvedTargetType);
        }
        else {
            return valueObject;
        }
    }
    catch (Throwable ex) {
        // Improve the message by showing the context.
        throw new BeanCreationException(
            this.beanDefinition.getResourceDescription(), "Invalid property value",
            "Error converting typed String value for " + annotatedElement.getName(), ex);
    }
}
else {
    return evaluate(value);
}
}

```

这个函数里定义了对各种类型属性的解析而真正的注入是在BeanWrapperImpl类中的setProperty方法中：

```

private void setPropertyValue(PropertyTokenHolder tokens, PropertyValue pv) throws BeansException {
    String propertyName = tokens.canonicalName;

```

```
String actualName = tokens.actualName;

if (tokens.keys != null) {
    // Apply indexes and map keys: fetch value for all keys but the last one
    PropertyTokenHolder getterTokens = new PropertyTokenHolder();
    getterTokens.canonicalName = tokens.canonicalName;
    getterTokens.actualName = tokens.actualName;
    getterTokens.keys = new String[tokens.keys.length - 1];
    System.arraycopy(tokens.keys, 0, getterTokens.keys, 0, tokens.keys.length - 1);
    Object propValue;
    try {
        propValue = getPropertyValue(getterTokens);
    }
    catch (NotReadablePropertyException ex) {
        throw new NotWritablePropertyException(getRootClass(), this.name,
            "Cannot access indexed value in property reference '" + canonicalName + "' " +
            "in indexed property path '" + propertyName + "'");
    }
    // Set value for last key.
    String key = tokens.keys[tokens.keys.length - 1];
    if (propValue == null) {
        throw new NullValueInNestedPathException(getRootClass(), this.name,
            "Cannot access indexed value in property reference '" + canonicalName + "' " +
            "in indexed property path '" + propertyName + "'");
    }
    else if (propValue.getClass().isArray()) {
        PropertyDescriptor pd = getCachedIntrospectionResults().getPropertyDescriptor(pd.getName());
        Class requiredType = propValue.getClass().getComponentType();
        int arrayIndex = Integer.parseInt(key);
        Object oldValue = null;
        try {
            if (isExtractOldValueForEditor()) {
                oldValue = Array.get(propValue, arrayIndex);
            }
            Object convertedValue = convertIfNecessary(propertyName, propValue,
                new PropertyTypeDescriptor(pd, new MethodMetadata(pd.getWriteMethod())));
            Array.set(propValue, arrayIndex, convertedValue);
        }
    }
}
```

```
    }
    catch (IndexOutOfBoundsException ex) {
        throw new InvalidPropertyException(getRootClass(), this
            "Invalid array index in property path '
    }
}
else if (propValue instanceof List) {
    PropertyDescriptor pd = getCachedIntrospectionResults().getProp
    Class requiredType = GenericCollectionTypeResolver.getCollectio
        pd.getReadMethod(), tokens.keys.length);
    List list = (List) propValue;
    int index = Integer.parseInt(key);
    Object oldValue = null;
    if (isExtractOldValueForEditor() && index < list.size()) {
        oldValue = list.get(index);
    }
    Object convertedValue = convertIfNecessary(propertyName, oldVal
        new PropertyTypeDescriptor(pd, new MethodParamete
    if (index < list.size()) {
        list.set(index, convertedValue);
    }
    else if (index >= list.size()) {
        for (int i = list.size(); i < index; i++) {
            try {
                list.add(null);
            }
            catch (NullPointerException ex) {
                throw new InvalidPropertyException(getR
                    "Cannot set element with index " +
                    list.size() + ", access
                    "' : List does not support
            }
        }
        list.add(convertedValue);
    }
}
else if (propValue instanceof Map) {
```

```

    PropertyDescriptor pd = getCachedIntrospectionResults().getProp
    Class mapKeyType = GenericCollectionTypeResolver.getMapKeyReturn
        pd.getReadMethod(), tokens.keys.length);
    Class mapValueType = GenericCollectionTypeResolver.getMapValueF
        pd.getReadMethod(), tokens.keys.length);
    Map map = (Map) propValue;
    // IMPORTANT: Do not pass full property name in here - property
    // must not kick in for map keys but rather only for map values
    Object convertedMapKey = convertIfNecessary(null, null, key, ma
        new PropertyTypeDescriptor(pd, new MethodParameter
    Object oldValue = null;
    if (isExtractOldValueForEditor()) {
        oldValue = map.get(convertedMapKey);
    }
    // Pass full property name and old value in here, since we want
    // conversion ability for map values.
    Object convertedMapValue = convertIfNecessary(
        propertyName, oldValue, pv.getValue(), mapValue
        new TypeDescriptor(new MethodParameter(pd.getRe
    map.put(convertedMapKey, convertedMapValue);
}
else {
    throw new InvalidPropertyException(getRootClass(), this.nestedP
        "Property referenced in indexed property path '"
        "' is neither an array nor a List nor a Map; re
}
}

else {
    PropertyDescriptor pd = pv.resolvedDescriptor;
    if (pd == null || !pd.getWriteMethod().getDeclaringClass().isInstance(t
        pd = getCachedIntrospectionResults().getPropertyDescriptor(actu
        if (pd == null || pd.getWriteMethod() == null) {
            if (pv.isOptional()) {
                logger.debug("Ignoring optional value for propert
                    "' - property not found on bean
            }
            return;
        }
    }
}

```

```

        }
        else {
            PropertyMatches matches = PropertyMatches.forPr
            throw new NotWritablePropertyException(
                getRootClass(), this.nestedPath
                matches.buildErrorMessage(), ma
        }
    }
    pv.getOriginalPropertyValue().resolvedDescriptor = pd;
}

Object oldValue = null;
try {
    Object originalValue = pv.getValue();
    Object valueToApply = originalValue;
    if (!Boolean.FALSE.equals(pv.conversionNecessary)) {
        if (pv.isConverted()) {
            valueToApply = pv.getConvertedValue();
        }
        else {
            if (isExtractOldValueForEditor() && pd.getReadM
                final Method readMethod = pd.getReadMet
                if (!Modifier.isPublic(readMethod.getDe
                    !readMethod.isAccessib]
                if (System.getSecurityManager()
                    AccessController.doPriv
                        public Object r
                            readMet
                                return
                                    }
                                });
                            }
                        }
                    else {
                        readMethod.setAccessib]
                    }
                }
            }
        }
    }
}
try {

```



```
        else {
            writeMethod.setAccessible(true);
        }
    }
    final Object value = valueToApply;
    if (System.getSecurityManager() != null) {
        try {
            AccessController.doPrivileged(new PrivilegedExec
                public Object run() throws Exception {
                    writeMethod.invoke(object, value);
                    return null;
                }
            }, acc);
        }
        catch (PrivilegedActionException ex) {
            throw ex.getException();
        }
    }
    else {
        writeMethod.invoke(this.object, value);
    }
}
catch (TypeMismatchException ex) {
    throw ex;
}
catch (InvocationTargetException ex) {
    PropertyChangeEvent propertyChangeEvent =
        new PropertyChangeEvent(this.rootObject, this.r
    if (ex.getTargetException() instanceof ClassCastException) {
        throw new TypeMismatchException(propertyChangeEvent, pc
    }
    else {
        throw new MethodInvocationException(propertyChangeEvent
    }
}
catch (Exception ex) {
    PropertyChangeEvent pce =
```

```
        new PropertyChangeEvent(this.rootObject, this.r  
        throw new MethodInvocationException(pce, ex);  
    }  
}  
}
```

到这里依赖注入已经完成了。

1.5 Spring源码解析 lazy-init属性和预实例化

发表时间: 2010-10-02

默认情况下会在容器启动时初始化bean, 但是我们可以指定Bean节点的 lazy-init="true" 来延迟初始化bean, 这时候, 只有第一次获取bean才会初始化bean。在IoC容器的初始化过程中, 主要的工作是对BeanDefinition的资源定位, 载入, 解析和注册。此时依赖注入并没有发生, 依赖注入发生在应用第一次向容器所要Bean时。对于容器的初始化有另外一种情况, 就是用户可以通过设置Bean的lazy-init属性来控制预实例化的过程, 这个预实例化在容器初始化时就完成了依赖注入。

在refresh中调用的finishBeanFactoryInitialization(beanFactory)方法中封装了对lazy-init属性的处理, 而实际的处理过程是在DefaultListableBeanFactory中

```
public void preInstantiateSingletons() throws BeansException {
    if (this.logger.isInfoEnabled()) {
        this.logger.info("Pre-instantiating singletons in " + this);
    }

    synchronized (this.beanDefinitionMap) {
        for (String beanName : this.beanDefinitionNames) {
            RootBeanDefinition bd = getMergedLocalBeanDefinition(beanName);
            if (!bd.isAbstract() && bd.isSingleton() && !bd.isLazyInit()) {
                if (isFactoryBean(beanName)) {
                    final FactoryBean factory = (FactoryBean) getBean(beanName);
                    boolean isEagerInit;
                    if (System.getSecurityManager() != null && factory instanceof AccessController) {
                        isEagerInit = AccessController.doPrivileged(new PrivilegedAction() {
                            public Boolean run() {
                                return ((SmartFactoryBean) factory).isEagerInit();
                            }
                        }, getAccessControlContext());
                    } else {
                        isEagerInit = factory instanceof SmartFactoryBean;
                    }
                } else {
                    isEagerInit = factory instanceof SmartFactoryBean;
                }
            }
        }
    }
}
```

```
        if (isEagerInit) {
            getBean(beanName);
        }
    }
    else {
        getBean(beanName);
    }
}
}
}
```

注意这里调用了`getBean(beanName)`,这个`getBean()`和上面分析的触发依赖注入的过程是一样的,只是发生的地方不同;如果不设置`lazy-init`属性,采用默认的`lazy-init=default`,即`lazy-init=false`,那么这个依赖注入是发生在容器初始化结束前.如果我们想对所有bean都应用延迟依赖注入,可以在根节点`beans`设置`default-lazy-init="true"`。

1.6 Spring源码解析 BeanPostProcessor的实现

发表时间: 2010-10-02

BeanPostProcessor是使用IoC容器时经常使用会遇到的一个特性，这个Bean的后置处理器是一个监听器，它可以监听容器触发的事件。把它向IoC容器注册以后，使得容器中管理的Bean具备接收IoC 容器事件回调的能力。具体的后置处理器需要实现接口BeanPostProcessor，然后设置到XML的Bean的配置文件中。这个BeanPostProcessor是一个接口类，它有两个接口方法，一个是Object postProcessBeforeInitialization(Object bean, String beanName) throws BeansException;，为在Bean的初始化前提供回调入口。另一个：Object postProcessAfterInitialization(Object bean, String beanName) throws BeansException;，为在Bean的初始化以后提供回调入口，这两个回调的触发都是和容器管理Bean的生命周期相关。

AbstractAutowireCapableBeanFactory的doCreateBean方法中

```
// Initialize the bean instance.
    Object exposedObject = bean;
    try {
        populateBean(beanName, mbd, instanceWrapper);
        if (exposedObject != null) {
            exposedObject = initializeBean(beanName, exposedObject, mbd);
        }
    }
    catch (Throwable ex) {
        if (ex instanceof BeanCreationException && beanName.equals(((BeanCreationException) ex).getBeanName()))
            throw (BeanCreationException) ex;
        else {
            throw new BeanCreationException(mbd.getResourceDescription(), beanName, ex);
        }
    }
}
```

在对Bean的生成和依赖注入完成后，开始对bean进行初始化，这个初始化过程包含了对后置处理器的调用

```
protected Object initializeBean(final String beanName, final Object bean, RootBeanDefinition mbd) {
    if (System.getSecurityManager() != null) {
        AccessController.doPrivileged(new PrivilegedAction<Object>() {
            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, mbd);
    }

    try {
        invokeInitMethods(beanName, wrappedBean, mbd);
    }
    catch (Throwable ex) {
        throw new BeanCreationException(
            beanName, (mbd != null ? mbd.getResourceDescription() : null),
            "Invocation of init method failed", ex);
    }

    if (mbd == null || !mbd.isSynthetic()) {
        wrappedBean = applyBeanPostProcessorsAfterInitialization(wrappedBean,
            beanName, mbd);
    }

    return wrappedBean;
}
```

具体的初始化过程也是依赖注入的一个重要部分。在initializeBean方法里，需要Bean的名字，完成依赖注入以后的bean对象，以及这个Bean对应的Beandefinition。在这些输入的帮助下完成Bean的初始化的工作，包括为类型是BeanNameAware的Bean设置Bean的名字，为类型BeanClassLoaderAware的Bean设置类装载器，为类型BeanFactoryAware的Bean设置其自身所在的IoC容器以供回调使用。这些实现在：

```
private void invokeAwareMethods(final String beanName, final Object bean) {
    if (bean instanceof BeanNameAware) {
        ((BeanNameAware) bean).setBeanName(beanName);
    }
    if (bean instanceof BeanClassLoaderAware) {
        ((BeanClassLoaderAware) bean).setBeanClassLoader(getBeanClassLoader());
    }
    if (bean instanceof BeanFactoryAware) {
        ((BeanFactoryAware) bean).setBeanFactory(AbstractAutowireCapableBeanFac
    }
}
```

中，接着是对后置处理器BeanPostProcessors的postProcessBeforeInitialization的回调方法的调用：

```
if (mbd == null || !mbd.isSynthetic()) {
    wrappedBean = applyBeanPostProcessorsBeforeInitialization(wrappedBean, beanName);
}
```

具体的实现过程见applyBeanPostProcessorsBeforeInitialization：

```
public Object applyBeanPostProcessorsBeforeInitialization(Object existingBean, String beanName)
    throws BeansException {

    Object result = existingBean;
    for (BeanPostProcessor beanProcessor : getBeanPostProcessors()) {
        result = beanProcessor.postProcessBeforeInitialization(result, beanName);
        if (result == null) {
            return result;
        }
    }
    return result;
}
```

调用Bean的初始化方法吗，这个初始化方法是在BeanDefinition中通过定义init-method属性指定的，同时如果Bean实现了InitializingBean接口，那么Bean的afterPropertiesSet的实现也会被调用。

```
try {
    invokeInitMethods(beanName, wrappedBean, mbd);
}
```

```

    }
    catch (Throwable ex) {
        throw new BeanCreationException(
            (mbd != null ? mbd.getResourceDescription() : null),
            beanName, "Invocation of init method failed", ex);
    }

```

具体的实现步骤如下：

```

protected void invokeInitMethods(String beanName, final Object bean, RootBeanDefinition mbd)
    throws Throwable {

    boolean isInitializingBean = (bean instanceof InitializingBean);
    if (isInitializingBean && (mbd == null || !mbd.isExternallyManagedInitMethod("afterPropertiesSet"))) {
        if (logger.isDebugEnabled()) {
            logger.debug("Invoking afterPropertiesSet() on bean with name '" + beanName + "'");
        }
        if (System.getSecurityManager() != null) {
            try {
                AccessController.doPrivileged(new PrivilegedExceptionAction<Object>() {
                    public Object run() throws Exception {
                        ((InitializingBean) bean).afterPropertiesSet();
                        return null;
                    }
                }, getAccessControlContext());
            }
            catch (PrivilegedActionException pae) {
                throw pae.getException();
            }
        }
        else {
            ((InitializingBean) bean).afterPropertiesSet();
        }
    }

    if (mbd != null) {
        String initMethodName = mbd.getInitMethodName();
    }

```

```
        if (initMethodName != null && !(isInitializingBean && "afterPropertiesSet"
            !mbd.isExternallyManagedInitMethod(initMethodName)) {
            invokeCustomInitMethod(beanName, bean, mbd);
        }
    }
}
```

然后就是对后置处理器BeanPostProcessor的postProcessAfterInitialization的回调方法调用

```
wrappedBean = applyBeanPostProcessorsAfterInitialization(wrappedBean, beanName);
```

具体的实现步骤如下：

```
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;
}
```

由上面的源码分析可知：BeanPostProcessors的postProcessBeforeInitialization先于Bean的属性init-method中配置的方法或者是Bean继承的InitializingBean接口后实现afterPropertiesSet方法执行。接着继续执行BeanPostProcessors的postProcessAfterInitialization方法。