**从Dubbo启动看Dubbo和Spring的关系**

Spring启动过程写的过于冗杂，如果对于这部分不感兴趣可以直接跳到Dubbo和Spring的关系，依然能够愉快的阅读。

## Spring启动过程

第一步：

ClassPathXmlApplicationContext的构造方法：

1. public class ClassPathXmlApplicationContext extends AbstractXmlApplicationContext {
2. private Resource[] configResources;
4. *// 如果已经有 ApplicationContext 并需要配置成父子关系，那么调用这个构造方法*
5. public ClassPathXmlApplicationContext(ApplicationContext parent) {
6. super(parent);
7. }
8. ...
9. public ClassPathXmlApplicationContext(String[] configLocations, boolean refresh, ApplicationContext parent)
10. throws BeansException {
12. super(parent);
13. *// 根据提供的路径，处理成配置文件数组(以分号、逗号、空格、tab、换行符分割)*
14. setConfigLocations(configLocations);
15. if (refresh) {
16. refresh(); *// 核心方法*
17. }
18. }
19. ...
20. }

接下来，就是refresh()，将原来的ApplicationContext销毁，然后重新执行一次初始化操作。

1. @Override
2. public void refresh() throws BeansException, IllegalStateException {
3. *// 来个锁，不然 refresh() 还没结束，你又来个启动或销毁容器的操作，那不就乱套了嘛*
4. synchronized (this.startupShutdownMonitor) {
6. *// 准备工作，记录下容器的启动时间、标记“已启动”状态、处理配置文件中的占位符*
7. prepareRefresh();
9. *// 这步比较关键，这步完成后，配置文件就会解析成一个个 Bean 定义，注册到 BeanFactory 中，*
10. *// 当然，这里说的 Bean 还没有初始化，只是配置信息都提取出来了，*
11. *// 注册也只是将这些信息都保存到了注册中心(说到底核心是一个 beanName-> beanDefinition 的 map)*
12. ConfigurableListableBeanFactory beanFactory = obtainFreshBeanFactory();
14. *// 设置 BeanFactory 的类加载器，添加几个 BeanPostProcessor，手动注册几个特殊的 bean*
15. *// 这块待会会展开说*
16. prepareBeanFactory(beanFactory);
18. try {
19. *// 【这里需要知道 BeanFactoryPostProcessor 这个知识点，Bean 如果实现了此接口，*
20. *// 那么在容器初始化以后，Spring 会负责调用里面的 postProcessBeanFactory 方法。】*
22. *// 这里是提供给子类的扩展点，到这里的时候，所有的 Bean 都加载、注册完成了，但是都还没有初始化*
23. *// 具体的子类可以在这步的时候添加一些特殊的 BeanFactoryPostProcessor 的实现类或做点什么事*
24. postProcessBeanFactory(beanFactory);
25. *// 调用 BeanFactoryPostProcessor 各个实现类的 postProcessBeanFactory(factory) 方法*
26. invokeBeanFactoryPostProcessors(beanFactory);
28. *// 注册 BeanPostProcessor 的实现类，注意看和 BeanFactoryPostProcessor 的区别*
29. *// 此接口两个方法: postProcessBeforeInitialization 和 postProcessAfterInitialization*
30. *// 两个方法分别在 Bean 初始化之前和初始化之后得到执行。注意，到这里 Bean 还没初始化*
31. registerBeanPostProcessors(beanFactory);
33. *// 初始化当前 ApplicationContext 的 MessageSource，国际化这里就不展开说了，不然没完没了了*
34. initMessageSource();
36. *// 初始化当前 ApplicationContext 的事件广播器，这里也不展开了*
37. initApplicationEventMulticaster();
39. *// 从方法名就可以知道，典型的模板方法(钩子方法)，*
40. *// 具体的子类可以在这里初始化一些特殊的 Bean（在初始化 singleton beans 之前）*
41. onRefresh();
43. *// 注册事件监听器，监听器需要实现 ApplicationListener 接口。这也不是我们的重点，过*
44. registerListeners();
46. *// 重点，重点，重点*
47. *// 初始化所有的 singleton beans*
48. *//（lazy-init 的除外）*
49. finishBeanFactoryInitialization(beanFactory);
51. *// 最后，广播事件，ApplicationContext 初始化完成*
52. finishRefresh();
53. }
55. catch (BeansException ex) {
56. if (logger.isWarnEnabled()) {
57. logger.warn("Exception encountered during context initialization - " +
58. "cancelling refresh attempt: " + ex);
59. }
61. *// Destroy already created singletons to avoid dangling resources.*
62. *// 销毁已经初始化的 singleton 的 Beans，以免有些 bean 会一直占用资源*
63. destroyBeans();
65. *// Reset 'active' flag.*
66. cancelRefresh(ex);
68. *// 把异常往外抛*
69. throw ex;
70. }
72. finally {
73. *// Reset common introspection caches in Spring's core, since we*
74. *// might not ever need metadata for singleton beans anymore...*
75. resetCommonCaches();
76. }
77. }
78. }

创建Bean容器前的准备工作：

1. protected void prepareRefresh() {
2. *// 记录启动时间，*
3. *// 将 active 属性设置为 true，closed 属性设置为 false，它们都是 AtomicBoolean 类型*
4. this.startupDate = System.currentTimeMillis();
5. this.closed.set(false);
6. this.active.set(true);
8. if (logger.isInfoEnabled()) {
9. logger.info("Refreshing " + this);
10. }
12. *// Initialize any placeholder property sources in the context environment*
13. initPropertySources();
15. *// 校验 xml 配置文件*
16. getEnvironment().validateRequiredProperties();
18. this.earlyApplicationEvents = new LinkedHashSet<ApplicationEvent>();
19. }

创建Bean容器，加载并注册Bean

这里会初始化BeanFactory、加载Bean、注册Bean等等。

AbstractApplicationContext.java

1. protected ConfigurableListableBeanFactory obtainFreshBeanFactory() {
2. *//关闭就得BeanFactory，创建新的BeanFactory，加载Bean定义、注册Bean等等。*
3. this.refreshBeanFactory();
4. *//返回刚刚创建的BeanFactory*
5. ConfigurableListableBeanFactory beanFactory = this.getBeanFactory();
6. if (this.logger.isDebugEnabled()) {
7. this.logger.debug("Bean factory for " + this.getDisplayName() + ": " + beanFactory);
8. }
10. return beanFactory;
11. }

AbstractRefreshableApplicationContext.java 120

1. @Override
2. protected final void refreshBeanFactory() throws BeansException {
3. *// 如果 ApplicationContext 中已经加载过 BeanFactory 了，销毁所有 Bean，关闭 BeanFactory*
4. *// 注意，应用中 BeanFactory 本来就是可以多个的，这里可不是说应用全局是否有 BeanFactory，而是当前*
5. *// ApplicationContext 是否有 BeanFactory*
6. if (hasBeanFactory()) {
7. destroyBeans();
8. closeBeanFactory();
9. }
10. try {
11. *// 初始化一个 DefaultListableBeanFactory，为什么用这个，我们马上说。*
12. DefaultListableBeanFactory beanFactory = createBeanFactory();
13. *// 用于 BeanFactory 的序列化，我想不部分人应该都用不到*
14. beanFactory.setSerializationId(getId());
16. *// 下面这两个方法很重要，别跟丢了，具体细节之后说*
17. *// 设置 BeanFactory 的两个配置属性：是否允许 Bean 覆盖、是否允许循环引用*
18. customizeBeanFactory(beanFactory);
20. *// 加载 Bean 到 BeanFactory 中*
21. loadBeanDefinitions(beanFactory);
22. synchronized (this.beanFactoryMonitor) {
23. this.beanFactory = beanFactory;
24. }
25. }
26. catch (IOException ex) {
27. throw new ApplicationContextException("I/O error parsing bean definition source for " + getDisplayName(), ex);
28. }
29. }

customizeBeanFactory(beanFactory)

1. protected void customizeBeanFactory(DefaultListableBeanFactory beanFactory) {
2. if (this.allowBeanDefinitionOverriding != null) {
3. *// 是否允许 Bean 定义覆盖*
4. beanFactory.setAllowBeanDefinitionOverriding(this.allowBeanDefinitionOverriding);
5. }
6. if (this.allowCircularReferences != null) {
7. *// 是否允许 Bean 间的循环依赖*
8. beanFactory.setAllowCircularReferences(this.allowCircularReferences);
9. }
10. }

AbstractXmlApplicationContext.java 80 loadBeanDefinitions(beanFactory)

1. */\*\* 我们可以看到，此方法将通过一个 XmlBeanDefinitionReader 实例来加载各个 Bean。\*/*
2. @Override
3. protected void loadBeanDefinitions(DefaultListableBeanFactory beanFactory) throws BeansException, IOException {
4. *// 给这个 BeanFactory 实例化一个 XmlBeanDefinitionReader*
5. XmlBeanDefinitionReader beanDefinitionReader = new XmlBeanDefinitionReader(beanFactory);
7. *// Configure the bean definition reader with this context's*
8. *// resource loading environment.*
9. beanDefinitionReader.setEnvironment(this.getEnvironment());
10. beanDefinitionReader.setResourceLoader(this);
11. beanDefinitionReader.setEntityResolver(new ResourceEntityResolver(this));
13. *// 初始化 BeanDefinitionReader，其实这个是提供给子类覆写的，*
14. *// 我看了一下，没有类覆写这个方法，我们姑且当做不重要吧*
15. initBeanDefinitionReader(beanDefinitionReader);
16. *// 重点来了，继续往下*
17. loadBeanDefinitions(beanDefinitionReader);
18. }

现在还在这个类中，接下来用刚刚初始化的 Reader 开始来加载 xml 配置，这块代码读者可以选择性跳过，不是很重要。也就是说，下面这个代码块，读者可以很轻松地略过。

// AbstractXmlApplicationContext.java 120

1. protected void loadBeanDefinitions(XmlBeanDefinitionReader reader) throws BeansException, IOException {
2. Resource[] configResources = getConfigResources();
3. if (configResources != null) {
4. *// 往下看*
5. reader.loadBeanDefinitions(configResources);
6. }
7. String[] configLocations = getConfigLocations();
8. if (configLocations != null) {
9. reader.loadBeanDefinitions(configLocations);
10. }
11. }
13. *// 上面虽然有两个分支，不过第二个分支很快通过解析路径转换为 Resource 以后也会进到这里*
14. @Override
15. public int loadBeanDefinitions(Resource... resources) throws BeanDefinitionStoreException {
16. Assert.notNull(resources, "Resource array must not be null");
17. int counter = 0;
18. *// 注意这里是个 for 循环，也就是每个文件是一个 resource*
19. for (Resource resource : resources) {
20. *// 继续往下看*
21. counter += loadBeanDefinitions(resource);
22. }
23. return counter;
24. }
26. *// XmlBeanDefinitionReader 303*
27. @Override
28. public int loadBeanDefinitions(Resource resource) throws BeanDefinitionStoreException {
29. return loadBeanDefinitions(new EncodedResource(resource));
30. }
32. *// XmlBeanDefinitionReader 314*
33. public int loadBeanDefinitions(EncodedResource encodedResource) throws BeanDefinitionStoreException {
34. Assert.notNull(encodedResource, "EncodedResource must not be null");
35. if (logger.isInfoEnabled()) {
36. logger.info("Loading XML bean definitions from " + encodedResource.getResource());
37. }
38. *// 用一个 ThreadLocal 来存放所有的配置文件资源*
39. Set<EncodedResource> currentResources = this.resourcesCurrentlyBeingLoaded.get();
40. if (currentResources == null) {
41. currentResources = new HashSet<EncodedResource>(4);
42. this.resourcesCurrentlyBeingLoaded.set(currentResources);
43. }
44. if (!currentResources.add(encodedResource)) {
45. throw new BeanDefinitionStoreException(
46. "Detected cyclic loading of " + encodedResource + " - check your import definitions!");
47. }
48. try {
49. InputStream inputStream = encodedResource.getResource().getInputStream();
50. try {
51. InputSource inputSource = new InputSource(inputStream);
52. if (encodedResource.getEncoding() != null) {
53. inputSource.setEncoding(encodedResource.getEncoding());
54. }
55. *// 核心部分*
56. return doLoadBeanDefinitions(inputSource, encodedResource.getResource());
57. }
58. finally {
59. inputStream.close();
60. }
61. }
62. catch (IOException ex) {
63. throw new BeanDefinitionStoreException(
64. "IOException parsing XML document from " + encodedResource.getResource(), ex);
65. }
66. finally {
67. currentResources.remove(encodedResource);
68. if (currentResources.isEmpty()) {
69. this.resourcesCurrentlyBeingLoaded.remove();
70. }
71. }
72. }
74. *// 还在这个文件中，第 388 行*
75. protected int doLoadBeanDefinitions(InputSource inputSource, Resource resource)
76. throws BeanDefinitionStoreException {
77. try {
78. *// 这里就不看了*
79. Document doc = doLoadDocument(inputSource, resource);
80. *// 继续*
81. return registerBeanDefinitions(doc, resource);
82. }
83. catch (...
84. }
85. *// 还在这个文件中，第 505 行*
86. *// 返回从当前配置文件加载了多少数量的 Bean*
87. public int registerBeanDefinitions(Document doc, Resource resource) throws BeanDefinitionStoreException {
88. BeanDefinitionDocumentReader documentReader = createBeanDefinitionDocumentReader();
89. int countBefore = getRegistry().getBeanDefinitionCount();
90. documentReader.registerBeanDefinitions(doc, createReaderContext(resource));
91. return getRegistry().getBeanDefinitionCount() - countBefore;
92. }
93. *// DefaultBeanDefinitionDocumentReader 90*
94. @Override
95. public void registerBeanDefinitions(Document doc, XmlReaderContext readerContext) {
96. this.readerContext = readerContext;
97. logger.debug("Loading bean definitions");
98. Element root = doc.getDocumentElement();
99. doRegisterBeanDefinitions(root);
100. }

经过漫长的链路，一个配置文件终于转换为一颗DOM树了，注意，这里指的是其中一个配置文件，不是所有。下面从根节点开始解析：

doRegisterBeanDefinitions：

1. *// DefaultBeanDefinitionDocumentReader 116*
2. protected void doRegisterBeanDefinitions(Element root) {
3. *// 我们看名字就知道，BeanDefinitionParserDelegate 必定是一个重要的类，它负责解析 Bean 定义，*
4. *// 这里为什么要定义一个 parent? 看到后面就知道了，是递归问题，*
5. *// 因为 <beans /> 内部是可以定义 <beans /> 的，所以这个方法的 root 其实不一定就是 xml 的根节点，也可以是嵌套在里面的 <beans /> 节点，从源码分析的角度，我们当做根节点就好了*
6. BeanDefinitionParserDelegate parent = this.delegate;
7. this.delegate = createDelegate(getReaderContext(), root, parent);
9. if (this.delegate.isDefaultNamespace(root)) {
10. *// 这块说的是根节点 <beans ... profile="dev" /> 中的 profile 是否是当前环境需要的，*
11. *// 如果当前环境配置的 profile 不包含此 profile，那就直接 return 了，不对此 <beans /> 解析*
12. *// 不熟悉 profile 为何物，不熟悉怎么配置 profile 读者的请移步附录区*
13. String profileSpec = root.getAttribute(PROFILE\_ATTRIBUTE);
14. if (StringUtils.hasText(profileSpec)) {
15. String[] specifiedProfiles = StringUtils.tokenizeToStringArray(
16. profileSpec, BeanDefinitionParserDelegate.MULTI\_VALUE\_ATTRIBUTE\_DELIMITERS);
17. if (!getReaderContext().getEnvironment().acceptsProfiles(specifiedProfiles)) {
18. if (logger.isInfoEnabled()) {
19. logger.info("Skipped XML bean definition file due to specified profiles [" + profileSpec +
20. "] not matching: " + getReaderContext().getResource());
21. }
22. return;
23. }
24. }
25. }
27. preProcessXml(root); *// 钩子*
28. parseBeanDefinitions(root, this.delegate);
29. postProcessXml(root); *// 钩子*
31. this.delegate = parent;
32. }

接下来，看核心解析方法parseBeanDefinitions(roo,this.delegate):

1. *// default namespace 涉及到的就四个标签 <import />、<alias />、<bean /> 和 <beans />，*
2. *// 其他的属于 custom 的*
3. protected void parseBeanDefinitions(Element root, BeanDefinitionParserDelegate delegate) {
4. if (delegate.isDefaultNamespace(root)) {
5. NodeList nl = root.getChildNodes();
6. for (int i = 0; i < nl.getLength(); i++) {
7. Node node = nl.item(i);
8. if (node instanceof Element) {
9. Element ele = (Element) node;
10. if (delegate.isDefaultNamespace(ele)) {
11. parseDefaultElement(ele, delegate);
12. }
13. else {
14. delegate.parseCustomElement(ele);
15. }
16. }
17. }
18. }
19. else {
20. delegate.parseCustomElement(root);
21. }
22. }

从上面的代码，我们可以看到，对于每个配置来说，分别进入到 parseDefaultElement(ele, delegate); 和 delegate.parseCustomElement(ele); 这两个分支了。

parseDefaultElement(ele, delegate) 代表解析的节点是 <import />、<alias />、<bean />、<beans /> 这几个。

这里的四个标签之所以是 default 的，是因为它们是处于这个 namespace 下定义的：

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

回过神来，看看处理 default 标签的方法：

1. private void parseDefaultElement(Element ele, BeanDefinitionParserDelegate delegate) {
2. if (delegate.nodeNameEquals(ele, IMPORT\_ELEMENT)) {
3. *// 处理 <import /> 标签*
4. importBeanDefinitionResource(ele);
5. }
6. else if (delegate.nodeNameEquals(ele, ALIAS\_ELEMENT)) {
7. *// 处理 <alias /> 标签定义*
8. *// <alias name="fromName" alias="toName"/>*
9. processAliasRegistration(ele);
10. }
11. else if (delegate.nodeNameEquals(ele, BEAN\_ELEMENT)) {
12. *// 处理 <bean /> 标签定义，这也算是我们的重点吧*
13. processBeanDefinition(ele, delegate);
14. }
15. else if (delegate.nodeNameEquals(ele, NESTED\_BEANS\_ELEMENT)) {
16. *// 如果碰到的是嵌套的 <beans /> 标签，需要递归*
17. doRegisterBeanDefinitions(ele);
18. }
19. }

如果每个标签都说，那我不吐血，你们都要吐血了。我们挑我们的重点 <bean /> 标签出来说。

processBeanDefinition

下面是 processBeanDefinition 解析 <bean /> 标签：

// DefaultBeanDefinitionDocumentReader 298

1. protected void processBeanDefinition(Element ele, BeanDefinitionParserDelegate delegate) {
2. *// 将 <bean /> 节点中的信息提取出来，然后封装到一个 BeanDefinitionHolder 中，细节往下看*
3. BeanDefinitionHolder bdHolder = delegate.parseBeanDefinitionElement(ele);
5. *// 下面的几行先不要看，跳过先，跳过先，跳过先，后面会继续说的*
7. if (bdHolder != null) {
8. bdHolder = delegate.decorateBeanDefinitionIfRequired(ele, bdHolder);
9. try {
10. *// Register the final decorated instance.*
11. BeanDefinitionReaderUtils.registerBeanDefinition(bdHolder, getReaderContext().getRegistry());
12. }
13. catch (BeanDefinitionStoreException ex) {
14. getReaderContext().error("Failed to register bean definition with name '" +
15. bdHolder.getBeanName() + "'", ele, ex);
16. }
17. *// Send registration event.*
18. getReaderContext().fireComponentRegistered(new BeanComponentDefinition(bdHolder));
19. }
20. }

继续往下看怎么解析之前，我们先看下 标签中可以定义哪些属性：

| **Property** |  |
| --- | --- |
| class | 类的全限定名 |
| name | 可指定 id、name(用逗号、分号、空格分隔) |
| scope | 作用域 |
| constructor arguments | 指定构造参数 |
| properties | 设置属性的值 |
| autowiring mode | no(默认值)、byName、byType、 constructor |
| lazy-initialization mode | 是否懒加载(如果被非懒加载的bean依赖了那么其实也就不能懒加载了) |
| initialization method | bean 属性设置完成后，会调用这个方法 |
| destruction method | bean 销毁后的回调方法 |

上面表格中的内容我想大家都非常熟悉吧，如果不熟悉，那就是你不够了解 Spring 的配置了。

简单地说就是像下面这样子：

1. <bean id="exampleBean" name="name1, name2, name3" class="com.javadoop.ExampleBean"
2. scope="singleton" lazy-init="true" init-method="init" destroy-method="cleanup">
4. *<!-- 可以用下面三种形式指定构造参数 -->*
5. <constructor-arg type="int" value="7500000"/>
6. <constructor-arg name="years" value="7500000"/>
7. <constructor-arg index="0" value="7500000"/>
9. *<!-- property 的几种情况 -->*
10. <property name="beanOne">
11. <ref bean="anotherExampleBean"/>
12. </property>
13. <property name="beanTwo" ref="yetAnotherBean"/>
14. <property name="integerProperty" value="1"/>
15. </bean>
16. 复制代码

当然，除了上面举例出来的这些，还有 factory-bean、factory-method、<lockup-method />、<replaced-method />、<meta />、<qualifier /> 这几个，大家是不是熟悉呢？

有了以上这些知识以后，我们再继续往里看怎么解析 bean 元素，是怎么转换到 BeanDefinitionHolder 的。

// BeanDefinitionParserDelegate 428

1. public BeanDefinitionHolder parseBeanDefinitionElement(Element ele) {
2. return parseBeanDefinitionElement(ele, null);
3. }
5. public BeanDefinitionHolder parseBeanDefinitionElement(Element ele, BeanDefinition containingBean) {
6. String id = ele.getAttribute(ID\_ATTRIBUTE);
7. String nameAttr = ele.getAttribute(NAME\_ATTRIBUTE);
9. List<String> aliases = new ArrayList<String>();
11. *// 将 name 属性的定义按照 ”逗号、分号、空格“ 切分，形成一个别名列表数组，*
12. *// 当然，如果你不定义的话，就是空的了*
13. *// 我在附录中简单介绍了一下 id 和 name 的配置，大家可以看一眼，有个20秒就可以了*
14. if (StringUtils.hasLength(nameAttr)) {
15. String[] nameArr = StringUtils.tokenizeToStringArray(nameAttr, MULTI\_VALUE\_ATTRIBUTE\_DELIMITERS);
16. aliases.addAll(Arrays.asList(nameArr));
17. }
19. String beanName = id;
20. *// 如果没有指定id, 那么用别名列表的第一个名字作为beanName*
21. if (!StringUtils.hasText(beanName) && !aliases.isEmpty()) {
22. beanName = aliases.remove(0);
23. if (logger.isDebugEnabled()) {
24. logger.debug("No XML 'id' specified - using '" + beanName +
25. "' as bean name and " + aliases + " as aliases");
26. }
27. }
29. if (containingBean == null) {
30. checkNameUniqueness(beanName, aliases, ele);
31. }
33. *// 根据 <bean ...>...</bean> 中的配置创建 BeanDefinition，然后把配置中的信息都设置到实例中,*
34. *// 细节后面再说*
35. AbstractBeanDefinition beanDefinition = parseBeanDefinitionElement(ele, beanName, containingBean);
37. *// 到这里，整个 <bean /> 标签就算解析结束了，一个 BeanDefinition 就形成了。*
38. if (beanDefinition != null) {
39. *// 如果都没有设置 id 和 name，那么此时的 beanName 就会为 null，进入下面这块代码产生*
40. *// 如果读者不感兴趣的话，我觉得不需要关心这块代码，对本文源码分析来说，这些东西不重要*
41. if (!StringUtils.hasText(beanName)) {
42. try {
43. if (containingBean != null) {*// 按照我们的思路，这里 containingBean 是 null 的*
44. beanName = BeanDefinitionReaderUtils.generateBeanName(
45. beanDefinition, this.readerContext.getRegistry(), true);
46. }
47. else {
48. *// 如果我们不定义 id 和 name，那么我们引言里的那个例子：*
49. *// 1. beanName 为：com.javadoop.example.MessageServiceImpl#0*
50. *// 2. beanClassName 为：com.javadoop.example.MessageServiceImpl*
52. beanName = this.readerContext.generateBeanName(beanDefinition);
54. String beanClassName = beanDefinition.getBeanClassName();
55. if (beanClassName != null &&
56. beanName.startsWith(beanClassName) && beanName.length() > beanClassName.length() &&
57. !this.readerContext.getRegistry().isBeanNameInUse(beanClassName)) {
58. *// 把 beanClassName 设置为 Bean 的别名*
59. aliases.add(beanClassName);
60. }
61. }
62. if (logger.isDebugEnabled()) {
63. logger.debug("Neither XML 'id' nor 'name' specified - " +
64. "using generated bean name [" + beanName + "]");
65. }
66. }
67. catch (Exception ex) {
68. error(ex.getMessage(), ele);
69. return null;
70. }
71. }
72. String[] aliasesArray = StringUtils.toStringArray(aliases);
73. *// 返回 BeanDefinitionHolder*
74. return new BeanDefinitionHolder(beanDefinition, beanName, aliasesArray);
75. }
77. return null;
78. }
79. 复制代码

看看怎么根据配置创建 BeanDefinition：

1. public AbstractBeanDefinition parseBeanDefinitionElement(
2. Element ele, String beanName, BeanDefinition containingBean) {
4. this.parseState.push(new BeanEntry(beanName));
6. String className = null;
7. if (ele.hasAttribute(CLASS\_ATTRIBUTE)) {
8. className = ele.getAttribute(CLASS\_ATTRIBUTE).trim();
9. }
11. try {
12. String parent = null;
13. if (ele.hasAttribute(PARENT\_ATTRIBUTE)) {
14. parent = ele.getAttribute(PARENT\_ATTRIBUTE);
15. }
16. *// 创建 BeanDefinition，然后设置类信息而已，很简单，就不贴代码了*
17. AbstractBeanDefinition bd = createBeanDefinition(className, parent);
19. *// 设置 BeanDefinition 的一堆属性，这些属性定义在 AbstractBeanDefinition 中*
20. parseBeanDefinitionAttributes(ele, beanName, containingBean, bd);
21. bd.setDescription(DomUtils.getChildElementValueByTagName(ele, DESCRIPTION\_ELEMENT));
23. */\*\**
24. *\* 下面的一堆是解析 <bean>......</bean> 内部的子元素，*
25. *\* 解析出来以后的信息都放到 bd 的属性中*
26. *\*/*
28. *// 解析 <meta />*
29. parseMetaElements(ele, bd);
30. *// 解析 <lookup-method />*
31. parseLookupOverrideSubElements(ele, bd.getMethodOverrides());
32. *// 解析 <replaced-method />*
33. parseReplacedMethodSubElements(ele, bd.getMethodOverrides());
34. *// 解析 <constructor-arg />*
35. parseConstructorArgElements(ele, bd);
36. *// 解析 <property />*
37. parsePropertyElements(ele, bd);
38. *// 解析 <qualifier />*
39. parseQualifierElements(ele, bd);
41. bd.setResource(this.readerContext.getResource());
42. bd.setSource(extractSource(ele));
44. return bd;
45. }
46. catch (ClassNotFoundException ex) {
47. error("Bean class [" + className + "] not found", ele, ex);
48. }
49. catch (NoClassDefFoundError err) {
50. error("Class that bean class [" + className + "] depends on not found", ele, err);
51. }
52. catch (Throwable ex) {
53. error("Unexpected failure during bean definition parsing", ele, ex);
54. }
55. finally {
56. this.parseState.pop();
57. }
59. return null;
60. }
61. 复制代码

到这里，我们已经完成了根据 <bean /> 配置创建了一个 BeanDefinitionHolder 实例。注意，是一个。

我们回到解析 <bean /> 的入口方法:

1. protected void processBeanDefinition(Element ele, BeanDefinitionParserDelegate delegate) {
2. *// 将 <bean /> 节点转换为 BeanDefinitionHolder，就是上面说的一堆*
3. BeanDefinitionHolder bdHolder = delegate.parseBeanDefinitionElement(ele);
4. if (bdHolder != null) {
5. *// 如果有自定义属性的话，进行相应的解析，先忽略*
6. bdHolder = delegate.decorateBeanDefinitionIfRequired(ele, bdHolder);
7. try {
8. *// 我们把这步叫做 注册Bean 吧*
9. BeanDefinitionReaderUtils.registerBeanDefinition(bdHolder, getReaderContext().getRegistry());
10. }
11. catch (BeanDefinitionStoreException ex) {
12. getReaderContext().error("Failed to register bean definition with name '" +
13. bdHolder.getBeanName() + "'", ele, ex);
14. }
15. *// 注册完成后，发送事件，本文不展开说这个*
16. getReaderContext().fireComponentRegistered(new BeanComponentDefinition(bdHolder));
17. }
18. }
19. 复制代码

大家再仔细看一下这块吧，我们后面就不回来说这个了。这里已经根据一个 <bean /> 标签产生了一个 BeanDefinitionHolder 的实例，这个实例里面也就是一个 BeanDefinition 的实例和它的 beanName、aliases 这三个信息，注意，我们的关注点始终在 BeanDefinition 上：

1. public class BeanDefinitionHolder implements BeanMetadataElement {
3. private final BeanDefinition beanDefinition;
5. private final String beanName;
7. private final String[] aliases;
8. ...
9. 复制代码

然后我们准备注册这个 BeanDefinition，最后，把这个注册事件发送出去。

下面，我们开始说注册 Bean 吧。

注册 Bean

// BeanDefinitionReaderUtils 143

1. public static void registerBeanDefinition(
2. BeanDefinitionHolder definitionHolder, BeanDefinitionRegistry registry)
3. throws BeanDefinitionStoreException {
5. String beanName = definitionHolder.getBeanName();
6. *// 注册这个 Bean*
7. registry.registerBeanDefinition(beanName, definitionHolder.getBeanDefinition());
9. *// 如果还有别名的话，也要根据别名统统注册一遍，不然根据别名就找不到 Bean 了，这我们就不开心了*
10. String[] aliases = definitionHolder.getAliases();
11. if (aliases != null) {
12. for (String alias : aliases) {
13. *// alias -> beanName 保存它们的别名信息，这个很简单，用一个 map 保存一下就可以了，*
14. *// 获取的时候，会先将 alias 转换为 beanName，然后再查找*
15. registry.registerAlias(beanName, alias);
16. }
17. }
18. }
19. 复制代码

别名注册的放一边，毕竟它很简单，我们看看怎么注册 Bean。

// DefaultListableBeanFactory 793

1. @Override
2. public void registerBeanDefinition(String beanName, BeanDefinition beanDefinition)
3. throws BeanDefinitionStoreException {
5. Assert.hasText(beanName, "Bean name must not be empty");
6. Assert.notNull(beanDefinition, "BeanDefinition must not be null");
8. if (beanDefinition instanceof AbstractBeanDefinition) {
9. try {
10. ((AbstractBeanDefinition) beanDefinition).validate();
11. }
12. catch (BeanDefinitionValidationException ex) {
13. throw new BeanDefinitionStoreException(...);
14. }
15. }
17. *// old? 还记得 “允许 bean 覆盖” 这个配置吗？allowBeanDefinitionOverriding*
18. BeanDefinition oldBeanDefinition;
20. *// 之后会看到，所有的 Bean 注册后会放入这个 beanDefinitionMap 中*
21. oldBeanDefinition = this.beanDefinitionMap.get(beanName);
23. *// 处理重复名称的 Bean 定义的情况*
24. if (oldBeanDefinition != null) {
25. if (!isAllowBeanDefinitionOverriding()) {
26. *// 如果不允许覆盖的话，抛异常*
27. throw new BeanDefinitionStoreException(beanDefinition.getResourceDescription()...
28. }
29. else if (oldBeanDefinition.getRole() < beanDefinition.getRole()) {
30. *// log...用框架定义的 Bean 覆盖用户自定义的 Bean*
31. }
32. else if (!beanDefinition.equals(oldBeanDefinition)) {
33. *// log...用新的 Bean 覆盖旧的 Bean*
34. }
35. else {
36. *// log...用同等的 Bean 覆盖旧的 Bean，这里指的是 equals 方法返回 true 的 Bean*
37. }
38. *// 覆盖*
39. this.beanDefinitionMap.put(beanName, beanDefinition);
40. }
41. else {
42. *// 判断是否已经有其他的 Bean 开始初始化了.*
43. *// 注意，"注册Bean" 这个动作结束，Bean 依然还没有初始化，我们后面会有大篇幅说初始化过程，*
44. *// 在 Spring 容器启动的最后，会 预初始化 所有的 singleton beans*
45. if (hasBeanCreationStarted()) {
46. *// Cannot modify startup-time collection elements anymore (for stable iteration)*
47. synchronized (this.beanDefinitionMap) {
48. this.beanDefinitionMap.put(beanName, beanDefinition);
49. List<String> updatedDefinitions = new ArrayList<String>(this.beanDefinitionNames.size() + 1);
50. updatedDefinitions.addAll(this.beanDefinitionNames);
51. updatedDefinitions.add(beanName);
52. this.beanDefinitionNames = updatedDefinitions;
53. if (this.manualSingletonNames.contains(beanName)) {
54. Set<String> updatedSingletons = new LinkedHashSet<String>(this.manualSingletonNames);
55. updatedSingletons.remove(beanName);
56. this.manualSingletonNames = updatedSingletons;
57. }
58. }
59. }
60. else {
61. *// 最正常的应该是进到这里。*
63. *// 将 BeanDefinition 放到这个 map 中，这个 map 保存了所有的 BeanDefinition*
64. this.beanDefinitionMap.put(beanName, beanDefinition);
65. *// 这是个 ArrayList，所以会按照 bean 配置的顺序保存每一个注册的 Bean 的名字*
66. this.beanDefinitionNames.add(beanName);
67. *// 这是个 LinkedHashSet，代表的是手动注册的 singleton bean，*
68. *// 注意这里是 remove 方法，到这里的 Bean 当然不是手动注册的*
69. *// 手动指的是通过调用以下方法注册的 bean ：*
70. *// registerSingleton(String beanName, Object singletonObject)*
71. *// 这不是重点，解释只是为了不让大家疑惑。Spring 会在后面"手动"注册一些 Bean，如 "environment"、"systemProperties" 等 bean*
72. this.manualSingletonNames.remove(beanName);
73. }
74. *// 这个不重要，在预初始化的时候会用到，不必管它。*
75. this.frozenBeanDefinitionNames = null;
76. }
78. if (oldBeanDefinition != null || containsSingleton(beanName)) {
79. resetBeanDefinition(beanName);
80. }
81. }
82. 复制代码

Bean 容器实例化完成后

说到这里，我们回到 refresh() 方法，我重新贴了一遍代码，看看我们说到哪了。是的，我们才说完 obtainFreshBeanFactory() 方法。

考虑到篇幅，这里开始大幅缩减掉没必要详细介绍的部分，大家直接看下面的代码中的注释就好了。

1. @Override
2. public void refresh() throws BeansException, IllegalStateException {
3. *// 来个锁，不然 refresh() 还没结束，你又来个启动或销毁容器的操作，那不就乱套了嘛*
4. synchronized (this.startupShutdownMonitor) {
6. *// 准备工作，记录下容器的启动时间、标记“已启动”状态、处理配置文件中的占位符*
7. prepareRefresh();
9. *// 这步比较关键，这步完成后，配置文件就会解析成一个个 Bean 定义，注册到 BeanFactory 中，*
10. *// 当然，这里说的 Bean 还没有初始化，只是配置信息都提取出来了，*
11. *// 注册也只是将这些信息都保存到了注册中心(说到底核心是一个 beanName-> beanDefinition 的 map)*
12. ConfigurableListableBeanFactory beanFactory = obtainFreshBeanFactory();
14. *// 设置 BeanFactory 的类加载器，添加几个 BeanPostProcessor，手动注册几个特殊的 bean*
15. *// 这块待会会展开说*
16. prepareBeanFactory(beanFactory);
18. try {
19. *// 【这里需要知道 BeanFactoryPostProcessor 这个知识点，Bean 如果实现了此接口，*
20. *// 那么在容器初始化以后，Spring 会负责调用里面的 postProcessBeanFactory 方法。】*
22. *// 这里是提供给子类的扩展点，到这里的时候，所有的 Bean 都加载、注册完成了，但是都还没有初始化*
23. *// 具体的子类可以在这步的时候添加一些特殊的 BeanFactoryPostProcessor 的实现类或做点什么事*
24. postProcessBeanFactory(beanFactory);
25. *// 调用 BeanFactoryPostProcessor 各个实现类的 postProcessBeanFactory(factory) 方法*
26. invokeBeanFactoryPostProcessors(beanFactory);
28. *// 注册 BeanPostProcessor 的实现类，注意看和 BeanFactoryPostProcessor 的区别*
29. *// 此接口两个方法: postProcessBeforeInitialization 和 postProcessAfterInitialization*
30. *// 两个方法分别在 Bean 初始化之前和初始化之后得到执行。注意，到这里 Bean 还没初始化*
31. registerBeanPostProcessors(beanFactory);
33. *// 初始化当前 ApplicationContext 的 MessageSource，国际化这里就不展开说了，不然没完没了了*
34. initMessageSource();
36. *// 初始化当前 ApplicationContext 的事件广播器，这里也不展开了*
37. initApplicationEventMulticaster();
39. *// 从方法名就可以知道，典型的模板方法(钩子方法)，*
40. *// 具体的子类可以在这里初始化一些特殊的 Bean（在初始化 singleton beans 之前）*
41. onRefresh();
43. *// 注册事件监听器，监听器需要实现 ApplicationListener 接口。这也不是我们的重点，过*
44. registerListeners();
46. *// 重点，重点，重点*
47. *// 初始化所有的 singleton beans*
48. *//（lazy-init 的除外）*
49. finishBeanFactoryInitialization(beanFactory);
51. *// 最后，广播事件，ApplicationContext 初始化完成*
52. finishRefresh();
53. }
55. catch (BeansException ex) {
56. if (logger.isWarnEnabled()) {
57. logger.warn("Exception encountered during context initialization - " +
58. "cancelling refresh attempt: " + ex);
59. }
61. *// Destroy already created singletons to avoid dangling resources.*
62. *// 销毁已经初始化的 singleton 的 Beans，以免有些 bean 会一直占用资源*
63. destroyBeans();
65. *// Reset 'active' flag.*
66. cancelRefresh(ex);
68. *// 把异常往外抛*
69. throw ex;
70. }
72. finally {
73. *// Reset common introspection caches in Spring's core, since we*
74. *// might not ever need metadata for singleton beans anymore...*
75. resetCommonCaches();
76. }
77. }
78. }
79. 复制代码

准备 Bean 容器: prepareBeanFactory

这里简单介绍下 prepareBeanFactory(factory) 方法：

1. */\*\**
2. *\* Configure the factory's standard context characteristics,*
3. *\* such as the context's ClassLoader and post-processors.*
4. *\* @param beanFactory the BeanFactory to configure*
5. *\*/*
6. protected void prepareBeanFactory(ConfigurableListableBeanFactory beanFactory) {
7. *// 设置 BeanFactory 的类加载器，我们知道 BeanFactory 需要加载类，也就需要类加载器，*
8. *// 这里设置为当前 ApplicationContext 的类加载器*
9. beanFactory.setBeanClassLoader(getClassLoader());
10. *// 设置 BeanExpressionResolver*
11. beanFactory.setBeanExpressionResolver(new StandardBeanExpressionResolver(beanFactory.getBeanClassLoader()));
12. *//*
13. beanFactory.addPropertyEditorRegistrar(new ResourceEditorRegistrar(this, getEnvironment()));
15. *// 添加一个 BeanPostProcessor，这个 processor 比较简单，*
16. *// 实现了 Aware 接口的几个特殊的 beans 在初始化的时候，这个 processor 负责回调*
17. beanFactory.addBeanPostProcessor(new ApplicationContextAwareProcessor(this));
19. *// 下面几行的意思就是，如果某个 bean 依赖于以下几个接口的实现类，在自动装配的时候忽略它们，*
20. *// Spring 会通过其他方式来处理这些依赖。*
21. beanFactory.ignoreDependencyInterface(EnvironmentAware.class);
22. beanFactory.ignoreDependencyInterface(EmbeddedValueResolverAware.class);
23. beanFactory.ignoreDependencyInterface(ResourceLoaderAware.class);
24. beanFactory.ignoreDependencyInterface(ApplicationEventPublisherAware.class);
25. beanFactory.ignoreDependencyInterface(MessageSourceAware.class);
26. beanFactory.ignoreDependencyInterface(ApplicationContextAware.class);
28. */\*\**
29. *\* 下面几行就是为特殊的几个 bean 赋值，如果有 bean 依赖了以下几个，会注入这边相应的值，*
30. *\* 之前我们说过，"当前 ApplicationContext 持有一个 BeanFactory"，这里解释了第一行*
31. *\* ApplicationContext 继承了 ResourceLoader、ApplicationEventPublisher、MessageSource*
32. *\* 所以对于这几个，可以赋值为 this，注意 this 是一个 ApplicationContext*
33. *\* 那这里怎么没看到为 MessageSource 赋值呢？那是因为 MessageSource 被注册成为了一个普通的 bean*
34. *\*/*
35. beanFactory.registerResolvableDependency(BeanFactory.class, beanFactory);
36. beanFactory.registerResolvableDependency(ResourceLoader.class, this);
37. beanFactory.registerResolvableDependency(ApplicationEventPublisher.class, this);
38. beanFactory.registerResolvableDependency(ApplicationContext.class, this);
40. *// 这个 BeanPostProcessor 也很简单，在 bean 实例化后，如果是 ApplicationListener 的子类，*
41. *// 那么将其添加到 listener 列表中，可以理解成：注册事件监听器*
42. beanFactory.addBeanPostProcessor(new ApplicationListenerDetector(this));
44. *// Detect a LoadTimeWeaver and prepare for weaving, if found.*
45. *// 这里涉及到特殊的 bean，名为：loadTimeWeaver，这不是我们的重点，忽略它*
46. if (beanFactory.containsBean(LOAD\_TIME\_WEAVER\_BEAN\_NAME)) {
47. beanFactory.addBeanPostProcessor(new LoadTimeWeaverAwareProcessor(beanFactory));
48. *// Set a temporary ClassLoader for type matching.*
49. beanFactory.setTempClassLoader(new ContextTypeMatchClassLoader(beanFactory.getBeanClassLoader()));
50. }
52. */\*\**
53. *\* 从下面几行代码我们可以知道，Spring 往往很 "智能" 就是因为它会帮我们默认注册一些有用的 bean，*
54. *\* 我们也可以选择覆盖*
55. *\*/*
57. *// 如果没有定义 "environment" 这个 bean，那么 Spring 会 "手动" 注册一个*
58. if (!beanFactory.containsLocalBean(ENVIRONMENT\_BEAN\_NAME)) {
59. beanFactory.registerSingleton(ENVIRONMENT\_BEAN\_NAME, getEnvironment());
60. }
61. *// 如果没有定义 "systemProperties" 这个 bean，那么 Spring 会 "手动" 注册一个*
62. if (!beanFactory.containsLocalBean(SYSTEM\_PROPERTIES\_BEAN\_NAME)) {
63. beanFactory.registerSingleton(SYSTEM\_PROPERTIES\_BEAN\_NAME, getEnvironment().getSystemProperties());
64. }
65. *// 如果没有定义 "systemEnvironment" 这个 bean，那么 Spring 会 "手动" 注册一个*
66. if (!beanFactory.containsLocalBean(SYSTEM\_ENVIRONMENT\_BEAN\_NAME)) {
67. beanFactory.registerSingleton(SYSTEM\_ENVIRONMENT\_BEAN\_NAME, getEnvironment().getSystemEnvironment());
68. }
69. }
70. 复制代码

初始化所有的 singleton beans

我们的重点当然是 finishBeanFactoryInitialization(beanFactory); 这个巨头了，这里会负责初始化所有的 singleton beans。

注意，后面的描述中，我都会使用**初始化**或**预初始化**来代表这个阶段。主要是 Spring 需要在这个阶段完成所有的 singleton beans 的实例化。

我们来总结一下，到目前为止，应该说 BeanFactory 已经创建完成，并且所有的实现了 BeanFactoryPostProcessor 接口的 Bean 都已经初始化并且其中的 postProcessBeanFactory(factory) 方法已经得到执行了。所有实现了 BeanPostProcessor 接口的 Bean 也都完成了初始化。

剩下的就是初始化其他还没被初始化的 singleton beans 了，我们知道它们是单例的，如果没有设置懒加载，那么 Spring 会在接下来初始化所有的 singleton beans。

// AbstractApplicationContext.java 834

1. *// 初始化剩余的 singleton beans*
2. protected void finishBeanFactoryInitialization(ConfigurableListableBeanFactory beanFactory) {
3. *// 什么，看代码这里没有初始化 Bean 啊！*
4. *// 注意了，初始化的动作包装在 beanFactory.getBean(...) 中，这里先不说细节，先往下看吧*
5. if (beanFactory.containsBean(CONVERSION\_SERVICE\_BEAN\_NAME) &&
6. beanFactory.isTypeMatch(CONVERSION\_SERVICE\_BEAN\_NAME, ConversionService.class)) {
7. beanFactory.setConversionService(
8. beanFactory.getBean(CONVERSION\_SERVICE\_BEAN\_NAME, ConversionService.class));
9. }
11. *// Register a default embedded value resolver if no bean post-processor*
12. *// (such as a PropertyPlaceholderConfigurer bean) registered any before:*
13. *// at this point, primarily for resolution in annotation attribute values.*
14. if (!beanFactory.hasEmbeddedValueResolver()) {
15. beanFactory.addEmbeddedValueResolver(new StringValueResolver() {
16. @Override
17. public String resolveStringValue(String strVal) {
18. return getEnvironment().resolvePlaceholders(strVal);
19. }
20. });
21. }
23. *// 先初始化 LoadTimeWeaverAware 类型的 Bean*
24. *// 一般用于织入第三方模块，在 class 文件载入 JVM 的时候动态织入，这里不展开说*
25. String[] weaverAwareNames = beanFactory.getBeanNamesForType(LoadTimeWeaverAware.class, false, false);
26. for (String weaverAwareName : weaverAwareNames) {
27. getBean(weaverAwareName);
28. }
30. *// Stop using the temporary ClassLoader for type matching.*
31. beanFactory.setTempClassLoader(null);
33. *// 没什么别的目的，因为到这一步的时候，Spring 已经开始预初始化 singleton beans 了，*
34. *// 肯定不希望这个时候还出现 bean 定义解析、加载、注册。*
35. beanFactory.freezeConfiguration();
37. *// 开始初始化剩下的*
38. beanFactory.preInstantiateSingletons();
39. }
40. 复制代码

从上面最后一行往里看，我们又回到 DefaultListableBeanFactory 这个类了，这个类大家应该都不陌生了吧。

preInstantiateSingletons

// DefaultListableBeanFactory 728

1. @Override
2. public void preInstantiateSingletons() throws BeansException {
3. if (this.logger.isDebugEnabled()) {
4. this.logger.debug("Pre-instantiating singletons in " + this);
5. }
7. List<String> beanNames = new ArrayList<String>(this.beanDefinitionNames);
9. *// 触发所有的非懒加载的 singleton beans 的初始化操作*
10. for (String beanName : beanNames) {
12. *// 合并父 Bean 中的配置，注意 <bean id="" class="" parent="" /> 中的 parent，用的不多吧，*
13. *// 考虑到这可能会影响大家的理解，我在附录中解释了一下 "Bean 继承"，请移步*
14. RootBeanDefinition bd = getMergedLocalBeanDefinition(beanName);
16. *// 非抽象、非懒加载的 singletons。如果配置了 'abstract = true'，那是不需要初始化的*
17. if (!bd.isAbstract() && bd.isSingleton() && !bd.isLazyInit()) {
18. *// 处理 FactoryBean(读者如果不熟悉 FactoryBean，请移步附录区了解)*
19. if (isFactoryBean(beanName)) {
20. *// FactoryBean 的话，在 beanName 前面加上 ‘&’ 符号。再调用 getBean，getBean 方法别急*
21. final FactoryBean<?> factory = (FactoryBean<?>) getBean(FACTORY\_BEAN\_PREFIX + beanName);
22. *// 判断当前 FactoryBean 是否是 SmartFactoryBean 的实现，此处忽略，直接跳过*
23. boolean isEagerInit;
24. if (System.getSecurityManager() != null && factory instanceof SmartFactoryBean) {
25. isEagerInit = AccessController.doPrivileged(new PrivilegedAction<Boolean>() {
26. @Override
27. public Boolean run() {
28. return ((SmartFactoryBean<?>) factory).isEagerInit();
29. }
30. }, getAccessControlContext());
31. }
32. else {
33. isEagerInit = (factory instanceof SmartFactoryBean &&
34. ((SmartFactoryBean<?>) factory).isEagerInit());
35. }
36. if (isEagerInit) {
38. getBean(beanName);
39. }
40. }
41. else {
42. *// 对于普通的 Bean，只要调用 getBean(beanName) 这个方法就可以进行初始化了*
43. getBean(beanName);
44. }
45. }
46. }

49. *// 到这里说明所有的非懒加载的 singleton beans 已经完成了初始化*
50. *// 如果我们定义的 bean 是实现了 SmartInitializingSingleton 接口的，那么在这里得到回调，忽略*
51. for (String beanName : beanNames) {
52. Object singletonInstance = getSingleton(beanName);
53. if (singletonInstance instanceof SmartInitializingSingleton) {
54. final SmartInitializingSingleton smartSingleton = (SmartInitializingSingleton) singletonInstance;
55. if (System.getSecurityManager() != null) {
56. AccessController.doPrivileged(new PrivilegedAction<Object>() {
57. @Override
58. public Object run() {
59. smartSingleton.afterSingletonsInstantiated();
60. return null;
61. }
62. }, getAccessControlContext());
63. }
64. else {
65. smartSingleton.afterSingletonsInstantiated();
66. }
67. }
68. }
69. }
70. 复制代码

接下来，我们就进入到 getBean(beanName) 方法了，这个方法我们经常用来从 BeanFactory 中获取一个 Bean，而初始化的过程也封装到了这个方法里。

getBean

在继续前进之前，读者应该具备 FactoryBean 的知识，如果读者还不熟悉，请移步附录部分了解 FactoryBean。

// AbstractBeanFactory 196

1. @Override
2. public Object getBean(String name) throws BeansException {
3. return doGetBean(name, null, null, false);
4. }
6. *// 我们在剖析初始化 Bean 的过程，但是 getBean 方法我们经常是用来从容器中获取 Bean 用的，注意切换思路，*
7. *// 已经初始化过了就从容器中直接返回，否则就先初始化再返回*
8. @SuppressWarnings("unchecked")
9. protected <T> T doGetBean(
10. final String name, final Class<T> requiredType, final Object[] args, boolean typeCheckOnly)
11. throws BeansException {
12. *// 获取一个 “正统的” beanName，处理两种情况，一个是前面说的 FactoryBean(前面带 ‘&’)，*
13. *// 一个是别名问题，因为这个方法是 getBean，获取 Bean 用的，你要是传一个别名进来，是完全可以的*
14. final String beanName = transformedBeanName(name);
16. *// 注意跟着这个，这个是返回值*
17. Object bean;
19. *// 检查下是不是已经创建过了*
20. Object sharedInstance = getSingleton(beanName);
22. *// 这里说下 args 呗，虽然看上去一点不重要。前面我们一路进来的时候都是 getBean(beanName)，*
23. *// 所以 args 其实是 null 的，但是如果 args 不为空的时候，那么意味着调用方不是希望获取 Bean，而是创建 Bean*
24. if (sharedInstance != null && args == null) {
25. if (logger.isDebugEnabled()) {
26. if (isSingletonCurrentlyInCreation(beanName)) {
27. logger.debug("...");
28. }
29. else {
30. logger.debug("Returning cached instance of singleton bean '" + beanName + "'");
31. }
32. }
33. *// 下面这个方法：如果是普通 Bean 的话，直接返回 sharedInstance，*
34. *// 如果是 FactoryBean 的话，返回它创建的那个实例对象*
35. *// (FactoryBean 知识，读者若不清楚请移步附录)*
36. bean = getObjectForBeanInstance(sharedInstance, name, beanName, null);
37. }
39. else {
40. if (isPrototypeCurrentlyInCreation(beanName)) {
41. *// 当前线程已经创建过了此 beanName 的 prototype 类型的 bean，那么抛异常*
42. throw new BeanCurrentlyInCreationException(beanName);
43. }
45. *// 检查一下这个 BeanDefinition 在容器中是否存在*
46. BeanFactory parentBeanFactory = getParentBeanFactory();
47. if (parentBeanFactory != null && !containsBeanDefinition(beanName)) {
48. *// 如果当前容器不存在这个 BeanDefinition，试试父容器中有没有*
49. String nameToLookup = originalBeanName(name);
50. if (args != null) {
51. *// 返回父容器的查询结果*
52. return (T) parentBeanFactory.getBean(nameToLookup, args);
53. }
54. else {
55. *// No args -> delegate to standard getBean method.*
56. return parentBeanFactory.getBean(nameToLookup, requiredType);
57. }
58. }
60. if (!typeCheckOnly) {
61. *// typeCheckOnly 为 false，将当前 beanName 放入一个 alreadyCreated 的 Set 集合中。*
62. markBeanAsCreated(beanName);
63. }
65. */\**
66. *\* 稍稍总结一下：*
67. *\* 到这里的话，要准备创建 Bean 了，对于 singleton 的 Bean 来说，容器中还没创建过此 Bean；*
68. *\* 对于 prototype 的 Bean 来说，本来就是要创建一个新的 Bean。*
69. *\*/*
70. try {
71. final RootBeanDefinition mbd = getMergedLocalBeanDefinition(beanName);
72. checkMergedBeanDefinition(mbd, beanName, args);
74. *// 先初始化依赖的所有 Bean，这个很好理解。*
75. *// 注意，这里的依赖指的是 depends-on 中定义的依赖*
76. String[] dependsOn = mbd.getDependsOn();
77. if (dependsOn != null) {
78. for (String dep : dependsOn) {
79. *// 检查是不是有循环依赖，这里的循环依赖和我们前面说的循环依赖又不一样，这里肯定是不允许出现的，不然要乱套了，读者想一下就知道了*
80. if (isDependent(beanName, dep)) {
81. throw new BeanCreationException(mbd.getResourceDescription(), beanName,
82. "Circular depends-on relationship between '" + beanName + "' and '" + dep + "'");
83. }
84. *// 注册一下依赖关系*
85. registerDependentBean(dep, beanName);
86. *// 先初始化被依赖项*
87. getBean(dep);
88. }
89. }
91. *// 创建 singleton 的实例*
92. if (mbd.isSingleton()) {
93. sharedInstance = getSingleton(beanName, new ObjectFactory<Object>() {
94. @Override
95. public Object getObject() throws BeansException {
96. try {
97. *// 执行创建 Bean，详情后面再说*
98. return createBean(beanName, mbd, args);
99. }
100. catch (BeansException ex) {
101. destroySingleton(beanName);
102. throw ex;
103. }
104. }
105. });
106. bean = getObjectForBeanInstance(sharedInstance, name, beanName, mbd);
107. }
109. *// 创建 prototype 的实例*
110. else if (mbd.isPrototype()) {
111. *// It's a prototype -> create a new instance.*
112. Object prototypeInstance = null;
113. try {
114. beforePrototypeCreation(beanName);
115. *// 执行创建 Bean*
116. prototypeInstance = createBean(beanName, mbd, args);
117. }
118. finally {
119. afterPrototypeCreation(beanName);
120. }
121. bean = getObjectForBeanInstance(prototypeInstance, name, beanName, mbd);
122. }
124. *// 如果不是 singleton 和 prototype 的话，需要委托给相应的实现类来处理*
125. else {
126. String scopeName = mbd.getScope();
127. final Scope scope = this.scopes.get(scopeName);
128. if (scope == null) {
129. throw new IllegalStateException("No Scope registered for scope name '" + scopeName + "'");
130. }
131. try {
132. Object scopedInstance = scope.get(beanName, new ObjectFactory<Object>() {
133. @Override
134. public Object getObject() throws BeansException {
135. beforePrototypeCreation(beanName);
136. try {
137. *// 执行创建 Bean*
138. return createBean(beanName, mbd, args);
139. }
140. finally {
141. afterPrototypeCreation(beanName);
142. }
143. }
144. });
145. bean = getObjectForBeanInstance(scopedInstance, name, beanName, mbd);
146. }
147. catch (IllegalStateException ex) {
148. throw new BeanCreationException(beanName,
149. "Scope '" + scopeName + "' is not active for the current thread; consider " +
150. "defining a scoped proxy for this bean if you intend to refer to it from a singleton",
151. ex);
152. }
153. }
154. }
155. catch (BeansException ex) {
156. cleanupAfterBeanCreationFailure(beanName);
157. throw ex;
158. }
159. }
161. *// 最后，检查一下类型对不对，不对的话就抛异常，对的话就返回了*
162. if (requiredType != null && bean != null && !requiredType.isInstance(bean)) {
163. try {
164. return getTypeConverter().convertIfNecessary(bean, requiredType);
165. }
166. catch (TypeMismatchException ex) {
167. if (logger.isDebugEnabled()) {
168. logger.debug("Failed to convert bean '" + name + "' to required type '" +
169. ClassUtils.getQualifiedName(requiredType) + "'", ex);
170. }
171. throw new BeanNotOfRequiredTypeException(name, requiredType, bean.getClass());
172. }
173. }
174. return (T) bean;
175. }
176. 复制代码

大家应该也猜到了，接下来当然是分析 createBean 方法：

1. protected abstract Object createBean(String beanName, RootBeanDefinition mbd, Object[] args) throws BeanCreationException;
3. 复制代码

第三个参数 args 数组代表创建实例需要的参数，不就是给构造方法用的参数，或者是工厂 Bean 的参数嘛，不过要注意，在我们的初始化阶段，args 是 null。

这回我们要到一个新的类了 AbstractAutowireCapableBeanFactory，看类名，AutowireCapable？类名是不是也说明了点问题了。

主要是为了以下场景，采用 @Autowired 注解注入属性值：

1. public class MessageServiceImpl implements MessageService {
2. @Autowired
3. private UserService userService;
5. public String getMessage() {
6. return userService.getMessage();
7. }
8. }
9. 复制代码
10. <bean id="messageService" class="com.javadoop.example.MessageServiceImpl" />
11. 复制代码

// AbstractAutowireCapableBeanFactory 447

1. */\*\**
2. *\* Central method of this class: creates a bean instance,*
3. *\* populates the bean instance, applies post-processors, etc.*
4. *\* @see #doCreateBean*
5. *\*/*
6. @Override
7. protected Object createBean(String beanName, RootBeanDefinition mbd, Object[] args) throws BeanCreationException {
8. if (logger.isDebugEnabled()) {
9. logger.debug("Creating instance of bean '" + beanName + "'");
10. }
11. RootBeanDefinition mbdToUse = mbd;
13. *// 确保 BeanDefinition 中的 Class 被加载*
14. Class<?> resolvedClass = resolveBeanClass(mbd, beanName);
15. if (resolvedClass != null && !mbd.hasBeanClass() && mbd.getBeanClassName() != null) {
16. mbdToUse = new RootBeanDefinition(mbd);
17. mbdToUse.setBeanClass(resolvedClass);
18. }
20. *// 准备方法覆写，这里又涉及到一个概念：MethodOverrides，它来自于 bean 定义中的 <lookup-method />*
21. *// 和 <replaced-method />，如果读者感兴趣，回到 bean 解析的地方看看对这两个标签的解析。*
22. *// 我在附录中也对这两个标签的相关知识点进行了介绍，读者可以移步去看看*
23. try {
24. mbdToUse.prepareMethodOverrides();
25. }
26. catch (BeanDefinitionValidationException ex) {
27. throw new BeanDefinitionStoreException(mbdToUse.getResourceDescription(),
28. beanName, "Validation of method overrides failed", ex);
29. }
31. try {
32. *// 让 BeanPostProcessor 在这一步有机会返回代理，而不是 bean 实例，*
33. *// 要彻底了解清楚这个，需要去看 InstantiationAwareBeanPostProcessor 接口，这里就不展开说了*
34. Object bean = resolveBeforeInstantiation(beanName, mbdToUse);
35. if (bean != null) {
36. return bean;
37. }
38. }
39. catch (Throwable ex) {
40. throw new BeanCreationException(mbdToUse.getResourceDescription(), beanName,
41. "BeanPostProcessor before instantiation of bean failed", ex);
42. }
43. *// 重头戏，创建 bean*
44. Object beanInstance = doCreateBean(beanName, mbdToUse, args);
45. if (logger.isDebugEnabled()) {
46. logger.debug("Finished creating instance of bean '" + beanName + "'");
47. }
48. return beanInstance;
49. }
50. 复制代码

创建 Bean

往里看 doCreateBean 这个方法：

1. */\*\**
2. *\* Actually create the specified bean. Pre-creation processing has already happened*
3. *\* at this point, e.g. checking {@code postProcessBeforeInstantiation} callbacks.*
4. *\* <p>Differentiates between default bean instantiation, use of a*
5. *\* factory method, and autowiring a constructor.*
6. *\* @param beanName the name of the bean*
7. *\* @param mbd the merged bean definition for the bean*
8. *\* @param args explicit arguments to use for constructor or factory method invocation*
9. *\* @return a new instance of the bean*
10. *\* @throws BeanCreationException if the bean could not be created*
11. *\* @see #instantiateBean*
12. *\* @see #instantiateUsingFactoryMethod*
13. *\* @see #autowireConstructor*
14. *\*/*
15. protected Object doCreateBean(final String beanName, final RootBeanDefinition mbd, final Object[] args)
16. throws BeanCreationException {
18. *// Instantiate the bean.*
19. BeanWrapper instanceWrapper = null;
20. if (mbd.isSingleton()) {
21. instanceWrapper = this.factoryBeanInstanceCache.remove(beanName);
22. }
23. if (instanceWrapper == null) {
24. *// 说明不是 FactoryBean，这里实例化 Bean，这里非常关键，细节之后再说*
25. instanceWrapper = createBeanInstance(beanName, mbd, args);
26. }
27. *// 这个就是 Bean 里面的 我们定义的类 的实例，很多地方我描述成 "bean 实例"*
28. final Object bean = (instanceWrapper != null ? instanceWrapper.getWrappedInstance() : null);
29. *// 类型*
30. Class<?> beanType = (instanceWrapper != null ? instanceWrapper.getWrappedClass() : null);
31. mbd.resolvedTargetType = beanType;
33. *// 建议跳过吧，涉及接口：MergedBeanDefinitionPostProcessor*
34. synchronized (mbd.postProcessingLock) {
35. if (!mbd.postProcessed) {
36. try {
37. *// MergedBeanDefinitionPostProcessor，这个我真不展开说了，直接跳过吧，很少用的*
38. applyMergedBeanDefinitionPostProcessors(mbd, beanType, beanName);
39. }
40. catch (Throwable ex) {
41. throw new BeanCreationException(mbd.getResourceDescription(), beanName,
42. "Post-processing of merged bean definition failed", ex);
43. }
44. mbd.postProcessed = true;
45. }
46. }
48. *// Eagerly cache singletons to be able to resolve circular references*
49. *// even when triggered by lifecycle interfaces like BeanFactoryAware.*
50. *// 下面这块代码是为了解决循环依赖的问题，以后有时间，我再对循环依赖这个问题进行解析吧*
51. boolean earlySingletonExposure = (mbd.isSingleton() && this.allowCircularReferences &&
52. isSingletonCurrentlyInCreation(beanName));
53. if (earlySingletonExposure) {
54. if (logger.isDebugEnabled()) {
55. logger.debug("Eagerly caching bean '" + beanName +
56. "' to allow for resolving potential circular references");
57. }
58. addSingletonFactory(beanName, new ObjectFactory<Object>() {
59. @Override
60. public Object getObject() throws BeansException {
61. return getEarlyBeanReference(beanName, mbd, bean);
62. }
63. });
64. }
66. *// Initialize the bean instance.*
67. Object exposedObject = bean;
68. try {
69. *// 这一步也是非常关键的，这一步负责属性装配，因为前面的实例只是实例化了，并没有设值，这里就是设值*
70. populateBean(beanName, mbd, instanceWrapper);
71. if (exposedObject != null) {
72. *// 还记得 init-method 吗？还有 InitializingBean 接口？还有 BeanPostProcessor 接口？*
73. *// 这里就是处理 bean 初始化完成后的各种回调*
74. exposedObject = initializeBean(beanName, exposedObject, mbd);
75. }
76. }
77. catch (Throwable ex) {
78. if (ex instanceof BeanCreationException && beanName.equals(((BeanCreationException) ex).getBeanName())) {
79. throw (BeanCreationException) ex;
80. }
81. else {
82. throw new BeanCreationException(
83. mbd.getResourceDescription(), beanName, "Initialization of bean failed", ex);
84. }
85. }
87. if (earlySingletonExposure) {
88. *//*
89. Object earlySingletonReference = getSingleton(beanName, false);
90. if (earlySingletonReference != null) {
91. if (exposedObject == bean) {
92. exposedObject = earlySingletonReference;
93. }
94. else if (!this.allowRawInjectionDespiteWrapping && hasDependentBean(beanName)) {
95. String[] dependentBeans = getDependentBeans(beanName);
96. Set<String> actualDependentBeans = new LinkedHashSet<String>(dependentBeans.length);
97. for (String dependentBean : dependentBeans) {
98. if (!removeSingletonIfCreatedForTypeCheckOnly(dependentBean)) {
99. actualDependentBeans.add(dependentBean);
100. }
101. }
102. if (!actualDependentBeans.isEmpty()) {
103. throw new BeanCurrentlyInCreationException(beanName,
104. "Bean with name '" + beanName + "' has been injected into other beans [" +
105. StringUtils.collectionToCommaDelimitedString(actualDependentBeans) +
106. "] in its raw version as part of a circular reference, but has eventually been " +
107. "wrapped. This means that said other beans do not use the final version of the " +
108. "bean. This is often the result of over-eager type matching - consider using " +
109. "'getBeanNamesOfType' with the 'allowEagerInit' flag turned off, for example.");
110. }
111. }
112. }
113. }
115. *// Register bean as disposable.*
116. try {
117. registerDisposableBeanIfNecessary(beanName, bean, mbd);
118. }
119. catch (BeanDefinitionValidationException ex) {
120. throw new BeanCreationException(
121. mbd.getResourceDescription(), beanName, "Invalid destruction signature", ex);
122. }
124. return exposedObject;
125. }
126. 复制代码

到这里，我们已经分析完了 doCreateBean 方法，总的来说，我们已经说完了整个初始化流程。

接下来我们挑 doCreateBean 中的三个细节出来说说。一个是创建 Bean 实例的 createBeanInstance 方法，一个是依赖注入的 populateBean 方法，还有就是回调方法 initializeBean。

注意了，接下来的这三个方法要认真说那也是极其复杂的，很多地方我就点到为止了，感兴趣的读者可以自己往里看，最好就是碰到不懂的，自己写代码去调试它。

创建 Bean 实例

我们先看看 createBeanInstance 方法。需要说明的是，这个方法如果每个分支都分析下去，必然也是极其复杂冗长的，我们挑重点说。此方法的目的就是实例化我们指定的类。

1. protected BeanWrapper createBeanInstance(String beanName, RootBeanDefinition mbd, Object[] args) {
2. *// 确保已经加载了此 class*
3. Class<?> beanClass = resolveBeanClass(mbd, beanName);
5. *// 校验一下这个类的访问权限*
6. if (beanClass != null && !Modifier.isPublic(beanClass.getModifiers()) && !mbd.isNonPublicAccessAllowed()) {
7. throw new BeanCreationException(mbd.getResourceDescription(), beanName,
8. "Bean class isn't public, and non-public access not allowed: " + beanClass.getName());
9. }
11. if (mbd.getFactoryMethodName() != null) {
12. *// 采用工厂方法实例化，不熟悉这个概念的读者请看附录，注意，不是 FactoryBean*
13. return instantiateUsingFactoryMethod(beanName, mbd, args);
14. }
16. *// 如果不是第一次创建，比如第二次创建 prototype bean。*
17. *// 这种情况下，我们可以从第一次创建知道，采用无参构造函数，还是构造函数依赖注入 来完成实例化*
18. boolean resolved = false;
19. boolean autowireNecessary = false;
20. if (args == null) {
21. synchronized (mbd.constructorArgumentLock) {
22. if (mbd.resolvedConstructorOrFactoryMethod != null) {
23. resolved = true;
24. autowireNecessary = mbd.constructorArgumentsResolved;
25. }
26. }
27. }
28. if (resolved) {
29. if (autowireNecessary) {
30. *// 构造函数依赖注入*
31. return autowireConstructor(beanName, mbd, null, null);
32. }
33. else {
34. *// 无参构造函数*
35. return instantiateBean(beanName, mbd);
36. }
37. }
39. *// 判断是否采用有参构造函数*
40. Constructor<?>[] ctors = determineConstructorsFromBeanPostProcessors(beanClass, beanName);
41. if (ctors != null ||
42. mbd.getResolvedAutowireMode() == RootBeanDefinition.AUTOWIRE\_CONSTRUCTOR ||
43. mbd.hasConstructorArgumentValues() || !ObjectUtils.isEmpty(args)) {
44. *// 构造函数依赖注入*
45. return autowireConstructor(beanName, mbd, ctors, args);
46. }
48. *// 调用无参构造函数*
49. return instantiateBean(beanName, mbd);
50. }
51. 复制代码

挑个简单的**无参构造函数**构造实例来看看：

1. protected BeanWrapper instantiateBean(final String beanName, final RootBeanDefinition mbd) {
2. try {
3. Object beanInstance;
4. final BeanFactory parent = this;
5. if (System.getSecurityManager() != null) {
6. beanInstance = AccessController.doPrivileged(new PrivilegedAction<Object>() {
7. @Override
8. public Object run() {
10. return getInstantiationStrategy().instantiate(mbd, beanName, parent);
11. }
12. }, getAccessControlContext());
13. }
14. else {
15. *// 实例化*
16. beanInstance = getInstantiationStrategy().instantiate(mbd, beanName, parent);
17. }
18. *// 包装一下，返回*
19. BeanWrapper bw = new BeanWrapperImpl(beanInstance);
20. initBeanWrapper(bw);
21. return bw;
22. }
23. catch (Throwable ex) {
24. throw new BeanCreationException(
25. mbd.getResourceDescription(), beanName, "Instantiation of bean failed", ex);
26. }
27. }
28. 复制代码

我们可以看到，关键的地方在于：

1. beanInstance = getInstantiationStrategy().instantiate(mbd, beanName, parent);
2. 复制代码

这里会进行实际的实例化过程，我们进去看看:

// SimpleInstantiationStrategy 59

1. @Override
2. public Object instantiate(RootBeanDefinition bd, String beanName, BeanFactory owner) {
4. *// 如果不存在方法覆写，那就使用 java 反射进行实例化，否则使用 CGLIB,*
5. *// 方法覆写 请参见附录"方法注入"中对 lookup-method 和 replaced-method 的介绍*
6. if (bd.getMethodOverrides().isEmpty()) {
7. Constructor<?> constructorToUse;
8. synchronized (bd.constructorArgumentLock) {
9. constructorToUse = (Constructor<?>) bd.resolvedConstructorOrFactoryMethod;
10. if (constructorToUse == null) {
11. final Class<?> clazz = bd.getBeanClass();
12. if (clazz.isInterface()) {
13. throw new BeanInstantiationException(clazz, "Specified class is an interface");
14. }
15. try {
16. if (System.getSecurityManager() != null) {
17. constructorToUse = AccessController.doPrivileged(new PrivilegedExceptionAction<Constructor<?>>() {
18. @Override
19. public Constructor<?> run() throws Exception {
20. return clazz.getDeclaredConstructor((Class[]) null);
21. }
22. });
23. }
24. else {
25. constructorToUse = clazz.getDeclaredConstructor((Class[]) null);
26. }
27. bd.resolvedConstructorOrFactoryMethod = constructorToUse;
28. }
29. catch (Throwable ex) {
30. throw new BeanInstantiationException(clazz, "No default constructor found", ex);
31. }
32. }
33. }
34. *// 利用构造方法进行实例化*
35. return BeanUtils.instantiateClass(constructorToUse);
36. }
37. else {
38. *// 存在方法覆写，利用 CGLIB 来完成实例化，需要依赖于 CGLIB 生成子类，这里就不展开了*
39. return instantiateWithMethodInjection(bd, beanName, owner);
40. }
41. }
42. 复制代码

到这里，我们就算实例化完成了。我们开始说怎么进行属性注入。

bean 属性注入

// AbstractAutowireCapableBeanFactory 1203

1. protected void populateBean(String beanName, RootBeanDefinition mbd, BeanWrapper bw) {
2. *// bean 实例的所有属性都在这里了*
3. PropertyValues pvs = mbd.getPropertyValues();
5. if (bw == null) {
6. if (!pvs.isEmpty()) {
7. throw new BeanCreationException(
8. mbd.getResourceDescription(), beanName, "Cannot apply property values to null instance");
9. }
10. else {
11. *// Skip property population phase for null instance.*
12. return;
13. }
14. }
16. *// 到这步的时候，bean 实例化完成（通过工厂方法或构造方法），但是还没开始属性设值，*
17. *// InstantiationAwareBeanPostProcessor 的实现类可以在这里对 bean 进行状态修改，*
18. *// 我也没找到有实际的使用，所以我们暂且忽略这块吧*
19. boolean continueWithPropertyPopulation = true;
20. if (!mbd.isSynthetic() && hasInstantiationAwareBeanPostProcessors()) {
21. for (BeanPostProcessor bp : getBeanPostProcessors()) {
22. if (bp instanceof InstantiationAwareBeanPostProcessor) {
23. InstantiationAwareBeanPostProcessor ibp = (InstantiationAwareBeanPostProcessor) bp;
24. *// 如果返回 false，代表不需要进行后续的属性设值，也不需要再经过其他的 BeanPostProcessor 的处理*
25. if (!ibp.postProcessAfterInstantiation(bw.getWrappedInstance(), beanName)) {
26. continueWithPropertyPopulation = false;
27. break;
28. }
29. }
30. }
31. }
33. if (!continueWithPropertyPopulation) {
34. return;
35. }
37. if (mbd.getResolvedAutowireMode() == RootBeanDefinition.AUTOWIRE\_BY\_NAME ||
38. mbd.getResolvedAutowireMode() == RootBeanDefinition.AUTOWIRE\_BY\_TYPE) {
39. MutablePropertyValues newPvs = new MutablePropertyValues(pvs);
41. *// 通过名字找到所有属性值，如果是 bean 依赖，先初始化依赖的 bean。记录依赖关系*
42. if (mbd.getResolvedAutowireMode() == RootBeanDefinition.AUTOWIRE\_BY\_NAME) {
43. autowireByName(beanName, mbd, bw, newPvs);
44. }
46. *// 通过类型装配。复杂一些*
47. if (mbd.getResolvedAutowireMode() == RootBeanDefinition.AUTOWIRE\_BY\_TYPE) {
48. autowireByType(beanName, mbd, bw, newPvs);
49. }
51. pvs = newPvs;
52. }
54. boolean hasInstAwareBpps = hasInstantiationAwareBeanPostProcessors();
55. boolean needsDepCheck = (mbd.getDependencyCheck() != RootBeanDefinition.DEPENDENCY\_CHECK\_NONE);
57. if (hasInstAwareBpps || needsDepCheck) {
58. PropertyDescriptor[] filteredPds = filterPropertyDescriptorsForDependencyCheck(bw, mbd.allowCaching);
59. if (hasInstAwareBpps) {
60. for (BeanPostProcessor bp : getBeanPostProcessors()) {
61. if (bp instanceof InstantiationAwareBeanPostProcessor) {
62. InstantiationAwareBeanPostProcessor ibp = (InstantiationAwareBeanPostProcessor) bp;
63. *// 这里有个非常有用的 BeanPostProcessor 进到这里: AutowiredAnnotationBeanPostProcessor*
64. *// 对采用 @Autowired、@Value 注解的依赖进行设值，这里的内容也是非常丰富的，不过本文不会展开说了，感兴趣的读者请自行研究*
65. pvs = ibp.postProcessPropertyValues(pvs, filteredPds, bw.getWrappedInstance(), beanName);
66. if (pvs == null) {
67. return;
68. }
69. }
70. }
71. }
72. if (needsDepCheck) {
73. checkDependencies(beanName, mbd, filteredPds, pvs);
74. }
75. }
76. *// 设置 bean 实例的属性值*
77. applyPropertyValues(beanName, mbd, bw, pvs);
78. }
79. 复制代码

initializeBean

属性注入完成后，这一步其实就是处理各种回调了。

1. protected Object initializeBean(final String beanName, final Object bean, RootBeanDefinition mbd) {
2. if (System.getSecurityManager() != null) {
3. AccessController.doPrivileged(new PrivilegedAction<Object>() {
4. @Override
5. public Object run() {
6. invokeAwareMethods(beanName, bean);
7. return null;
8. }
9. }, getAccessControlContext());
10. }
11. else {
12. *// 如果 bean 实现了 BeanNameAware、BeanClassLoaderAware 或 BeanFactoryAware 接口，回调*
13. invokeAwareMethods(beanName, bean);
14. }
16. Object wrappedBean = bean;
17. if (mbd == null || !mbd.isSynthetic()) {
18. *// BeanPostProcessor 的 postProcessBeforeInitialization 回调*
19. wrappedBean = applyBeanPostProcessorsBeforeInitialization(wrappedBean, beanName);
20. }
22. try {
23. *// 处理 bean 中定义的 init-method，*
24. *// 或者如果 bean 实现了 InitializingBean 接口，调用 afterPropertiesSet() 方法*
25. invokeInitMethods(beanName, wrappedBean, mbd);
26. }
27. catch (Throwable ex) {
28. throw new BeanCreationException(
29. (mbd != null ? mbd.getResourceDescription() : null),
30. beanName, "Invocation of init method failed", ex);
31. }
33. if (mbd == null || !mbd.isSynthetic()) {
34. *// BeanPostProcessor 的 postProcessAfterInitialization 回调*
35. wrappedBean = applyBeanPostProcessorsAfterInitialization(wrappedBean, beanName);
36. }
37. return wrappedBean;
38. }
39. 复制代码

大家发现没有，BeanPostProcessor 的两个回调都发生在这边，只不过中间处理了 init-method，是不是和读者原来的认知有点不一样了？

## Dubbo启动和Spring的关系

首先我们来看一下官方demo中给出的Dubbo启动代码：

1. public static void main(String[] args) throws Exception {
2. ClassPathXmlApplicationContext context = new ClassPathXmlApplicationContext(new String[]{"META-INF/spring/dubbo-demo-provider.xml"});
3. context.start();
4. System.in.read(); *// press any key to exit*
5. }
6. 复制代码

再看一下配置文件：

1. <beans xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
2. xmlns:dubbo="http://dubbo.apache.org/schema/dubbo"
3. xmlns="http://www.springframework.org/schema/beans"
4. xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-4.3.xsd
5. http://dubbo.apache.org/schema/dubbo http://dubbo.apache.org/schema/dubbo/dubbo.xsd">
7. *<!-- provider's application name, used for tracing dependency relationship -->*
8. <dubbo:application name="demo-provider"/>
10. *<!-- use multicast registry center to export service -->*
11. <dubbo:registry address="zookeeper://47.107.231.174:2181"/>
13. *<!-- use dubbo protocol to export service on port 20880 -->*
14. <dubbo:protocol name="dubbo" port="20880"/>
16. *<!-- service implementation, as same as regular local bean -->*
17. <bean id="demoService" class="org.apache.dubbo.demo.provider.DemoServiceImpl"/>
19. *<!-- declare the service interface to be exported -->*
20. <dubbo:service interface="org.apache.dubbo.demo.DemoService" ref="demoService"/>
22. </beans>
23. 复制代码

可以看到Dubbo启动的代码很简单，只有三行，通过构建ClassPathXmlApplicationContext来启动Dubbo。那么Dubbo是如何在ClassPathXmlApplicationContext构建过程中实现自己的相关逻辑的呢？

在Dubbo的官方文档中是这样写的:

基于 dubbo.jar 内的 META-INF/spring.handlers 配置，Spring 在遇到 dubbo 名称空间时，会回调 DubboNamespaceHandler。

所有 dubbo 的标签，都统一用 DubboBeanDefinitionParser 进行解析，基于一对一属性映射，将 XML 标签解析为 Bean 对象。

这里提到的spring.handlers配置如下：

1. http\://dubbo.apache.org/schema/dubbo=org.apache.dubbo.config.spring.schema.DubboNamespaceHandler
2. http\://code.alibabatech.com/schema/dubbo=org.apache.dubbo.config.spring.schema.DubboNamespaceHandler
3. 复制代码

这里的两个配置对应了Dubbo从阿里开源到捐赠apache的过程。

我们来看一下Spring中是如何解析Dubbo的配置文件的：

这是在启动Dubbo过程中解析文件的调用链路

可以看到开始启动之后

执行ClassPathXmlApplicationContext的构造方法，在构造方法中调用refresh构造新的ApplicationContext对象，在这里会加载BeanDefiniton。也就是说会加载我们在Dubbo的配置文件中的相关配置。

在执行解析对应的Element对象时，会进入BeanDefinitionParserDelegate.class的parseCustomElement方法:

1. public BeanDefinition parseCustomElement(Element ele, BeanDefinition containingBd) {
2. String namespaceUri = this.getNamespaceURI(ele);*//首先获得对应的namespaceUri，这里对应的namaspaceUri是http://dubbo.apche.org/schema/dubbo*
3. NamespaceHandler handler = this.readerContext.getNamespaceHandlerResolver().resolve(namespaceUri);*//根据namespaceUri获取对应的Handler*
4. if (handler == null) {
5. this.error("Unable to locate Spring NamespaceHandler for XML schema namespace [" + namespaceUri + "]", ele);
6. return null;
7. } else {
8. return handler.parse(ele, new ParserContext(this.readerContext, this, containingBd));*//handler执行相应的解析逻辑，最后跳转到DubboBeanDefinitionParser执行parse方法。*
9. }
10. }
11. 复制代码

DefaultNamespaceHandlerResolver.class类：

1. *//*
2. *// Source code recreated from a .class file by IntelliJ IDEA*
3. *// (powered by Fernflower decompiler)*
4. *//*
6. package org.springframework.beans.factory.xml;
8. import java.io.IOException;
9. import java.util.Map;
10. import java.util.Properties;
11. import java.util.concurrent.ConcurrentHashMap;
12. import org.apache.commons.logging.Log;
13. import org.apache.commons.logging.LogFactory;
14. import org.springframework.beans.BeanUtils;
15. import org.springframework.beans.FatalBeanException;
16. import org.springframework.core.io.support.PropertiesLoaderUtils;
17. import org.springframework.util.Assert;
18. import org.springframework.util.ClassUtils;
19. import org.springframework.util.CollectionUtils;
21. public class DefaultNamespaceHandlerResolver implements NamespaceHandlerResolver {
22. public static final String DEFAULT\_HANDLER\_MAPPINGS\_LOCATION = "META-INF/spring.handlers";*//默认的handler配置文件地址*
23. protected final Log logger;
24. private final ClassLoader classLoader;
25. private final String handlerMappingsLocation;
26. private volatile Map<String, Object> handlerMappings;*//namespaceUri和解析器实例的映射关系*
28. public DefaultNamespaceHandlerResolver() {
29. this((ClassLoader)null, "META-INF/spring.handlers");
30. }
32. public DefaultNamespaceHandlerResolver(ClassLoader classLoader) {
33. this(classLoader, "META-INF/spring.handlers");
34. }
36. public DefaultNamespaceHandlerResolver(ClassLoader classLoader, String handlerMappingsLocation) {
37. this.logger = LogFactory.getLog(this.getClass());
38. Assert.notNull(handlerMappingsLocation, "Handler mappings location must not be null");
39. this.classLoader = classLoader != null ? classLoader : ClassUtils.getDefaultClassLoader();
40. this.handlerMappingsLocation = handlerMappingsLocation;
41. }
43. public NamespaceHandler resolve(String namespaceUri) {
44. Map<String, Object> handlerMappings = this.getHandlerMappings();
45. Object handlerOrClassName = handlerMappings.get(namespaceUri);
46. if (handlerOrClassName == null) {
47. return null;
48. } else if (handlerOrClassName instanceof NamespaceHandler) {
49. return (NamespaceHandler)handlerOrClassName;
50. } else {
51. String className = (String)handlerOrClassName;
53. try {
54. Class<?> handlerClass = ClassUtils.forName(className, this.classLoader);
55. if (!NamespaceHandler.class.isAssignableFrom(handlerClass)) {
56. throw new FatalBeanException("Class [" + className + "] for namespace [" + namespaceUri + "] does not implement the [" + NamespaceHandler.class.getName() + "] interface");
57. } else {
58. NamespaceHandler namespaceHandler = (NamespaceHandler)BeanUtils.instantiateClass(handlerClass);*//创建对应Handler的实例*
59. namespaceHandler.init();*//创建之后调用init方法*
60. handlerMappings.put(namespaceUri, namespaceHandler);*//添加到对应map中*
61. return namespaceHandler;
62. }
63. } catch (ClassNotFoundException var7) {
64. throw new FatalBeanException("NamespaceHandler class [" + className + "] for namespace [" + namespaceUri + "] not found", var7);
65. } catch (LinkageError var8) {
66. throw new FatalBeanException("Invalid NamespaceHandler class [" + className + "] for namespace [" + namespaceUri + "]: problem with handler class file or dependent class", var8);
67. }
68. }
69. }
70. *//当需要获得对应的namespaceUri和handler的映射关系时，使用双重检查锁来构造handlerMapping。*
71. private Map<String, Object> getHandlerMappings() {
72. if (this.handlerMappings == null) {
73. synchronized(this) {
74. if (this.handlerMappings == null) {
75. try {
76. Properties mappings = PropertiesLoaderUtils.loadAllProperties(this.handlerMappingsLocation, this.classLoader);
77. if (this.logger.isDebugEnabled()) {
78. this.logger.debug("Loaded NamespaceHandler mappings: " + mappings);
79. }
81. Map<String, Object> handlerMappings = new ConcurrentHashMap(mappings.size());
82. CollectionUtils.mergePropertiesIntoMap(mappings, handlerMappings);
83. this.handlerMappings = handlerMappings;
84. } catch (IOException var5) {
85. throw new IllegalStateException("Unable to load NamespaceHandler mappings from location [" + this.handlerMappingsLocation + "]", var5);
86. }
87. }
88. }
89. }
91. return this.handlerMappings;
92. }
94. public String toString() {
95. return "NamespaceHandlerResolver using mappings " + this.getHandlerMappings();
96. }
97. }
99. 复制代码

从上面的类中我们了解到handler和namespaceUri的对应关系默认是从META-INF/spring.handlers文件中读取，这就是为什么我们只需要在META-INF/spring.handlers配置对应的解析器，spring就能自动的使用相应的解析去解析对应的名称空间下的Element对象。

从上面可以看到创建了对应的Handler实例之后，会调用该handler的init方法。

DubboNamespaceHandler.java的init方法如下:

1. @Override
2. public void init() {
3. registerBeanDefinitionParser("application", new DubboBeanDefinitionParser(ApplicationConfig.class, true));
4. registerBeanDefinitionParser("module", new DubboBeanDefinitionParser(ModuleConfig.class, true));
5. registerBeanDefinitionParser("registry", new DubboBeanDefinitionParser(RegistryConfig.class, true));
6. registerBeanDefinitionParser("monitor", new DubboBeanDefinitionParser(MonitorConfig.class, true));
7. registerBeanDefinitionParser("provider", new DubboBeanDefinitionParser(ProviderConfig.class, true));
8. registerBeanDefinitionParser("consumer", new DubboBeanDefinitionParser(ConsumerConfig.class, true));
9. registerBeanDefinitionParser("protocol", new DubboBeanDefinitionParser(ProtocolConfig.class, true));
10. registerBeanDefinitionParser("service", new DubboBeanDefinitionParser(ServiceBean.class, true));
11. registerBeanDefinitionParser("reference", new DubboBeanDefinitionParser(ReferenceBean.class, false));
12. registerBeanDefinitionParser("annotation", new AnnotationBeanDefinitionParser());
13. }
14. 复制代码

在init方法中，对dubbo中不同的实体注册了不同的解析器，并指定不同的名称解析成不同的实体对象。如果后续还需要什么新的实体添加，只需要在这个地方添加对应的解析器就可以。

return handler.parse(ele, new ParserContext(this.readerContext, this, containingBd))：这里会执行DubboBeanDefinitionParser中的parser方法：

1. private static BeanDefinition parse(Element element, ParserContext parserContext, Class<?> beanClass, boolean required) {
2. *//创建RootBeanDefinition*
3. RootBeanDefinition beanDefinition = new RootBeanDefinition();
4. beanDefinition.setBeanClass(beanClass);
5. beanDefinition.setLazyInit(false);
6. *//解析配置对象的id。若不存在，则进行生成*
7. String id = element.getAttribute("id");
8. if ((id == null || id.length() == 0) && required) {
9. String generatedBeanName = element.getAttribute("name");
10. if (generatedBeanName == null || generatedBeanName.length() == 0) {
11. if (ProtocolConfig.class.equals(beanClass)) {
12. generatedBeanName = "dubbo";
13. } else {
14. generatedBeanName = element.getAttribute("interface");
15. }
16. }
17. if (generatedBeanName == null || generatedBeanName.length() == 0) {
18. generatedBeanName = beanClass.getName();
19. }
20. id = generatedBeanName;
21. *//若id已存在，通过自增序列，解决重复*
22. int counter = 2;
23. while (parserContext.getRegistry().containsBeanDefinition(id)) {
24. id = generatedBeanName + (counter++);
25. }
26. }
27. if (id != null && id.length() > 0) {
28. if (parserContext.getRegistry().containsBeanDefinition(id)) {
29. throw new IllegalStateException("Duplicate spring bean id " + id);
30. }
31. *//添加到Spring的注册表*
32. parserContext.getRegistry().registerBeanDefinition(id, beanDefinition);
33. *//设置Bean的'id'属性值*
34. beanDefinition.getPropertyValues().addPropertyValue("id", id);
35. }
36. *//处理<dubbo:protocol/>特殊情况*
37. if (ProtocolConfig.class.equals(beanClass)) {
38. for (String name : parserContext.getRegistry().getBeanDefinitionNames()) {
39. BeanDefinition definition = parserContext.getRegistry().getBeanDefinition(name);
40. PropertyValue property = definition.getPropertyValues().getPropertyValue("protocol");
41. if (property != null) {
42. Object value = property.getValue();
43. if (value instanceof ProtocolConfig && id.equals(((ProtocolConfig) value).getName())) {
44. definition.getPropertyValues().addPropertyValue("protocol", new RuntimeBeanReference(id));
45. }
46. }
47. }
48. *//处理<dubbo:service/>的class的属性*
49. } else if (ServiceBean.class.equals(beanClass)) {
50. String className = element.getAttribute("class");
51. if (className != null && className.length() > 0) {
52. RootBeanDefinition classDefinition = new RootBeanDefinition();
53. classDefinition.setBeanClass(ReflectUtils.forName(className));
54. classDefinition.setLazyInit(false);
55. parseProperties(element.getChildNodes(), classDefinition);
56. beanDefinition.getPropertyValues().addPropertyValue("ref", new BeanDefinitionHolder(classDefinition, id + "Impl"));
57. }
58. *//解析<dubbo:provider/>的内嵌子元素<dubbo:service/>*
59. } else if (ProviderConfig.class.equals(beanClass)) {
60. parseNested(element, parserContext, ServiceBean.class, true, "service", "provider", id, beanDefinition);
61. *//解析<dubbo:consumer/>的内嵌子元素<dubbo:reference>*
62. } else if (ConsumerConfig.class.equals(beanClass)) {
63. parseNested(element, parserContext, ReferenceBean.class, false, "reference", "consumer", id, beanDefinition);
64. }
65. Set<String> props = new HashSet<String>();
66. ManagedMap parameters = null;
67. *//循环Bean对象的setting方法，将属性赋值到Bean对象*
68. for (Method setter : beanClass.getMethods()) {
69. String name = setter.getName();
70. *//判断是否是set方法*
71. if (name.length() > 3 && name.startsWith("set")
72. && Modifier.isPublic(setter.getModifiers())
73. && setter.getParameterTypes().length == 1) {
74. Class<?> type = setter.getParameterTypes()[0];
75. *//添加'props'*
76. String property = StringUtils.camelToSplitName(name.substring(3, 4).toLowerCase() + name.substring(4), "-");
77. props.add(property);
78. Method getter = null;
79. *//getting&&public&&属性值类型统一*
80. try {
81. getter = beanClass.getMethod("get" + name.substring(3), new Class<?>[0]);
82. } catch (NoSuchMethodException e) {
83. try {
84. getter = beanClass.getMethod("is" + name.substring(3), new Class<?>[0]);
85. } catch (NoSuchMethodException e2) {
86. }
87. }
88. if (getter == null
89. || !Modifier.isPublic(getter.getModifiers())
90. || !type.equals(getter.getReturnType())) {
91. continue;
92. }
93. *//解析'<dubbo:parameters/>'*
94. if ("parameters".equals(property)) {
95. parameters = parseParameters(element.getChildNodes(), beanDefinition);
96. *//解析<dubbo:method>*
97. } else if ("methods".equals(property)) {
98. parseMethods(id, element.getChildNodes(), beanDefinition, parserContext);
99. *//解析'<dubbo:argument>'*
100. } else if ("arguments".equals(property)) {
101. parseArguments(id, element.getChildNodes(), beanDefinition, parserContext);
102. } else {
103. String value = element.getAttribute(property);
104. if (value != null) {
105. value = value.trim();
106. if (value.length() > 0) {
107. *//不想注册到注册中心的情况*
108. if ("registry".equals(property) && RegistryConfig.NO\_AVAILABLE.equalsIgnoreCase(value)) {
109. RegistryConfig registryConfig = new RegistryConfig();
110. registryConfig.setAddress(RegistryConfig.NO\_AVAILABLE);
111. beanDefinition.getPropertyValues().addPropertyValue(property, registryConfig);
112. } else if ("registry".equals(property) && value.indexOf(',') != -1) {
113. *//多注册中心的情况*
114. parseMultiRef("registries", value, beanDefinition, parserContext);
115. } else if ("provider".equals(property) && value.indexOf(',') != -1) {
116. *//多服务提供者的情况*
117. parseMultiRef("providers", value, beanDefinition, parserContext);
118. } else if ("protocol".equals(property) && value.indexOf(',') != -1) {
119. *//多协议的情况*
120. parseMultiRef("protocols", value, beanDefinition, parserContext);
121. } else {
122. Object reference;
123. *//处理属性为基本属性的情况*
124. if (isPrimitive(type)) {
125. *//兼容性处理*
126. if ("async".equals(property) && "false".equals(value)
127. || "timeout".equals(property) && "0".equals(value)
128. || "delay".equals(property) && "0".equals(value)
129. || "version".equals(property) && "0.0.0".equals(value)
130. || "stat".equals(property) && "-1".equals(value)
131. || "reliable".equals(property) && "false".equals(value)) {
132. *// backward compatibility for the default value in old version's xsd*
133. value = null;
134. }
135. reference = value;
136. *//处理在'<dubbo:provider/>'或者<dubbo:service/>上定义了'protocol'属性的兼容性*
137. } else if ("protocol".equals(property)
138. && ExtensionLoader.getExtensionLoader(Protocol.class).hasExtension(value)
139. && (!parserContext.getRegistry().containsBeanDefinition(value)*//存在该注册表的实现*
140. || !ProtocolConfig.class.getName().equals(parserContext.getRegistry().getBeanDefinition(value).getBeanClassName()))) {
141. *//目前，<dubbo:provider protocol=""/>推荐独立成<dubbo:protocol/>*
142. if ("dubbo:provider".equals(element.getTagName())) {
143. logger.warn("Recommended replace <dubbo:provider protocol=\"" + value + "\" ... /> to <dubbo:protocol name=\"" + value + "\" ... />");
144. }
145. *// backward compatibility*
146. ProtocolConfig protocol = new ProtocolConfig();
147. protocol.setName(value);
148. reference = protocol;
149. *//处理onreturn属性*
150. } else if ("onreturn".equals(property)) {
151. *//按照'.'拆分*
152. int index = value.lastIndexOf(".");
153. String returnRef = value.substring(0, index);
154. String returnMethod = value.substring(index + 1);
155. *//创建RuntimeBeanReference，指向回调对象*
156. reference = new RuntimeBeanReference(returnRef);
157. *//设置'onreturnMethod'到BeanDefinition的属性值*
158. beanDefinition.getPropertyValues().addPropertyValue("onreturnMethod", returnMethod);
159. *//处理onthrow属性*
160. } else if ("onthrow".equals(property)) {
161. *//按照'.'拆分*
162. int index = value.lastIndexOf(".");
163. String throwRef = value.substring(0, index);
164. String throwMethod = value.substring(index + 1);
165. *//创建RuntimeBeanReference指向回调对象*
166. reference = new RuntimeBeanReference(throwRef);
167. *//设置’onthrowMethod‘到BeanDefinition的属性值*
168. beanDefinition.getPropertyValues().addPropertyValue("onthrowMethod", throwMethod);
169. } else if ("oninvoke".equals(property)) {
170. int index = value.lastIndexOf(".");
171. String invokeRef = value.substring(0, index);
172. String invokeRefMethod = value.substring(index + 1);
173. reference = new RuntimeBeanReference(invokeRef);
174. beanDefinition.getPropertyValues().addPropertyValue("oninvokeMethod", invokeRefMethod);
175. } else {
176. *//指向Service的bean对象，必须是单例*
177. if ("ref".equals(property) && parserContext.getRegistry().containsBeanDefinition(value)) {
178. BeanDefinition refBean = parserContext.getRegistry().getBeanDefinition(value);
179. if (!refBean.isSingleton()) {
180. throw new IllegalStateException("The exported service ref " + value + " must be singleton! Please set the " + value + " bean scope to singleton, eg: <bean id=\"" + value + "\" scope=\"singleton\" ...>");
181. }
182. }
183. *//创建RuntimeBeanReference，指向service的Bean对象*
184. reference = new RuntimeBeanReference(value);
185. }
186. *//设置BeanDefinition的属性值*
187. beanDefinition.getPropertyValues().addPropertyValue(property, reference);
188. }
189. }
190. }
191. }
192. }
193. }
194. *//将XML元素，未在上面遍历到的属性添加到'parameters'集合中。*
195. NamedNodeMap attributes = element.getAttributes();
196. int len = attributes.getLength();
197. for (int i = 0; i < len; i++) {
198. Node node = attributes.item(i);
199. String name = node.getLocalName();
200. if (!props.contains(name)) {
201. if (parameters == null) {
202. parameters = new ManagedMap();
203. }
204. String value = node.getNodeValue();
205. parameters.put(name, new TypedStringValue(value, String.class));
206. }
207. }
208. if (parameters != null) {
209. beanDefinition.getPropertyValues().addPropertyValue("parameters", parameters);
210. }
211. return beanDefinition;
212. }
213. 复制代码

这里Dubbo的配置文件解析就告一段落。

最后就是如何实现在Dubbo启动的时候讲服务注册到注册中心上去，这里就需要用到前面解析的对象。在前面我们提到DubboBeanDefinitionParser会将service解析为ServiceBean对象。

我们先看看ServiceBean类的实现:

1. */\**
2. *\* Licensed to the Apache Software Foundation (ASF) under one or more*
3. *\* contributor license agreements. See the NOTICE file distributed with*
4. *\* this work for additional information regarding copyright ownership.*
5. *\* The ASF licenses this file to You under the Apache License, Version 2.0*
6. *\* (the "License"); you may not use this file except in compliance with*
7. *\* the License. You may obtain a copy of the License at*
8. *\**
9. *\* http://www.apache.org/licenses/LICENSE-2.0*
10. *\**
11. *\* Unless required by applicable law or agreed to in writing, software*
12. *\* distributed under the License is distributed on an "AS IS" BASIS,*
13. *\* WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.*
14. *\* See the License for the specific language governing permissions and*
15. *\* limitations under the License.*
16. *\*/*
17. package org.apache.dubbo.config.spring;
19. import org.apache.dubbo.config.ApplicationConfig;
20. import org.apache.dubbo.config.ModuleConfig;
21. import org.apache.dubbo.config.MonitorConfig;
22. import org.apache.dubbo.config.ProtocolConfig;
23. import org.apache.dubbo.config.ProviderConfig;
24. import org.apache.dubbo.config.RegistryConfig;
25. import org.apache.dubbo.config.ServiceConfig;
26. import org.apache.dubbo.config.annotation.Service;
27. import org.apache.dubbo.config.spring.extension.SpringExtensionFactory;
28. import org.springframework.aop.support.AopUtils;
29. import org.springframework.beans.factory.BeanFactoryUtils;
30. import org.springframework.beans.factory.BeanNameAware;
31. import org.springframework.beans.factory.DisposableBean;
32. import org.springframework.beans.factory.InitializingBean;
33. import org.springframework.context.ApplicationContext;
34. import org.springframework.context.ApplicationContextAware;
35. import org.springframework.context.ApplicationListener;
36. import org.springframework.context.event.ContextRefreshedEvent;
37. import org.springframework.context.support.AbstractApplicationContext;
39. import java.lang.reflect.Method;
40. import java.util.ArrayList;
41. import java.util.List;
42. import java.util.Map;
44. */\*\**
45. *\* ServiceFactoryBean*
46. *\**
47. *\* @export*
48. \*/
49. public class ServiceBean<T> extends ServiceConfig<T> implements InitializingBean, DisposableBean, ApplicationContextAware, ApplicationListener<ContextRefreshedEvent>, BeanNameAware {
51. private static final long serialVersionUID = 213195494150089726L;
53. private final transient Service service;
55. private transient ApplicationContext applicationContext;
57. private transient String beanName;
59. private transient boolean supportedApplicationListener;
61. public ServiceBean() {
62. super();
63. this.service = null;
64. }
66. public ServiceBean(Service service) {
67. super(service);
68. this.service = service;
69. }
71. @Override
72. public void setApplicationContext(ApplicationContext applicationContext) {
73. this.applicationContext = applicationContext;
74. SpringExtensionFactory.addApplicationContext(applicationContext);
75. try {
76. Method method = applicationContext.getClass().getMethod("addApplicationListener", ApplicationListener.class); *// backward compatibility to spring 2.0.1*
77. method.invoke(applicationContext, this);
78. supportedApplicationListener = true;
79. } catch (Throwable t) {
80. if (applicationContext instanceof AbstractApplicationContext) {
81. try {
82. Method method = AbstractApplicationContext.class.getDeclaredMethod("addListener", ApplicationListener.class); *// backward compatibility to spring 2.0.1*
83. if (!method.isAccessible()) {
84. method.setAccessible(true);
85. }
86. method.invoke(applicationContext, this);
87. supportedApplicationListener = true;
88. } catch (Throwable t2) {
89. }
90. }
91. }
92. }
94. @Override
95. public void setBeanName(String name) {
96. this.beanName = name;
97. }
99. */\*\**
100. *\* Gets associated {@link Service}*
101. *\**
102. *\* @return associated {@link Service}*
103. *\*/*
104. public Service getService() {
105. return service;
106. }
108. @Override
109. public void onApplicationEvent(ContextRefreshedEvent event) {
110. if (isDelay() && !isExported() && !isUnexported()) {
111. if (logger.isInfoEnabled()) {
112. logger.info("The service ready on spring started. service: " + getInterface());
113. }
114. export();
115. }
116. }
118. private boolean isDelay() {
119. Integer delay = getDelay();
120. ProviderConfig provider = getProvider();
121. if (delay == null && provider != null) {
122. delay = provider.getDelay();
123. }
124. return supportedApplicationListener && (delay == null || delay == -1);
125. }
127. @Override
128. @SuppressWarnings({"unchecked", "deprecation"})
129. public void afterPropertiesSet() throws Exception {
130. if (getProvider() == null) {
131. Map<String, ProviderConfig> providerConfigMap = applicationContext == null ? null : BeanFactoryUtils.beansOfTypeIncludingAncestors(applicationContext, ProviderConfig.class, false, false);
132. if (providerConfigMap != null && providerConfigMap.size() > 0) {
133. Map<String, ProtocolConfig> protocolConfigMap = applicationContext == null ? null : BeanFactoryUtils.beansOfTypeIncludingAncestors(applicationContext, ProtocolConfig.class, false, false);
134. if ((protocolConfigMap == null || protocolConfigMap.size() == 0)
135. && providerConfigMap.size() > 1) { *// backward compatibility*
136. List<ProviderConfig> providerConfigs = new ArrayList<ProviderConfig>();
137. for (ProviderConfig config : providerConfigMap.values()) {
138. if (config.isDefault() != null && config.isDefault()) {
139. providerConfigs.add(config);
140. }
141. }
142. if (!providerConfigs.isEmpty()) {
143. setProviders(providerConfigs);
144. }
145. } else {
146. ProviderConfig providerConfig = null;
147. for (ProviderConfig config : providerConfigMap.values()) {
148. if (config.isDefault() == null || config.isDefault()) {
149. if (providerConfig != null) {
150. throw new IllegalStateException("Duplicate provider configs: " + providerConfig + " and " + config);
151. }
152. providerConfig = config;
153. }
154. }
155. if (providerConfig != null) {
156. setProvider(providerConfig);
157. }
158. }
159. }
160. }
161. if (getApplication() == null
162. && (getProvider() == null || getProvider().getApplication() == null)) {
163. Map<String, ApplicationConfig> applicationConfigMap = applicationContext == null ? null : BeanFactoryUtils.beansOfTypeIncludingAncestors(applicationContext, ApplicationConfig.class, false, false);
164. if (applicationConfigMap != null && applicationConfigMap.size() > 0) {
165. ApplicationConfig applicationConfig = null;
166. for (ApplicationConfig config : applicationConfigMap.values()) {
167. if (config.isDefault() == null || config.isDefault()) {
168. if (applicationConfig != null) {
169. throw new IllegalStateException("Duplicate application configs: " + applicationConfig + " and " + config);
170. }
171. applicationConfig = config;
172. }
173. }
174. if (applicationConfig != null) {
175. setApplication(applicationConfig);
176. }
177. }
178. }
179. if (getModule() == null
180. && (getProvider() == null || getProvider().getModule() == null)) {
181. Map<String, ModuleConfig> moduleConfigMap = applicationContext == null ? null : BeanFactoryUtils.beansOfTypeIncludingAncestors(applicationContext, ModuleConfig.class, false, false);
182. if (moduleConfigMap != null && moduleConfigMap.size() > 0) {
183. ModuleConfig moduleConfig = null;
184. for (ModuleConfig config : moduleConfigMap.values()) {
185. if (config.isDefault() == null || config.isDefault()) {
186. if (moduleConfig != null) {
187. throw new IllegalStateException("Duplicate module configs: " + moduleConfig + " and " + config);
188. }
189. moduleConfig = config;
190. }
191. }
192. if (moduleConfig != null) {
193. setModule(moduleConfig);
194. }
195. }
196. }
197. if ((getRegistries() == null || getRegistries().isEmpty())
198. && (getProvider() == null || getProvider().getRegistries() == null || getProvider().getRegistries().isEmpty())
199. && (getApplication() == null || getApplication().getRegistries() == null || getApplication().getRegistries().isEmpty())) {
200. Map<String, RegistryConfig> registryConfigMap = applicationContext == null ? null : BeanFactoryUtils.beansOfTypeIncludingAncestors(applicationContext, RegistryConfig.class, false, false);
201. if (registryConfigMap != null && registryConfigMap.size() > 0) {
202. List<RegistryConfig> registryConfigs = new ArrayList<RegistryConfig>();
203. for (RegistryConfig config : registryConfigMap.values()) {
204. if (config.isDefault() == null || config.isDefault()) {
205. registryConfigs.add(config);
206. }
207. }
208. if (!registryConfigs.isEmpty()) {
209. super.setRegistries(registryConfigs);
210. }
211. }
212. }
213. if (getMonitor() == null
214. && (getProvider() == null || getProvider().getMonitor() == null)
215. && (getApplication() == null || getApplication().getMonitor() == null)) {
216. Map<String, MonitorConfig> monitorConfigMap = applicationContext == null ? null : BeanFactoryUtils.beansOfTypeIncludingAncestors(applicationContext, MonitorConfig.class, false, false);
217. if (monitorConfigMap != null && monitorConfigMap.size() > 0) {
218. MonitorConfig monitorConfig = null;
219. for (MonitorConfig config : monitorConfigMap.values()) {
220. if (config.isDefault() == null || config.isDefault()) {
221. if (monitorConfig != null) {
222. throw new IllegalStateException("Duplicate monitor configs: " + monitorConfig + " and " + config);
223. }
224. monitorConfig = config;
225. }
226. }
227. if (monitorConfig != null) {
228. setMonitor(monitorConfig);
229. }
230. }
231. }
232. if ((getProtocols() == null || getProtocols().isEmpty())
233. && (getProvider() == null || getProvider().getProtocols() == null || getProvider().getProtocols().isEmpty())) {
234. Map<String, ProtocolConfig> protocolConfigMap = applicationContext == null ? null : BeanFactoryUtils.beansOfTypeIncludingAncestors(applicationContext, ProtocolConfig.class, false, false);
235. if (protocolConfigMap != null && protocolConfigMap.size() > 0) {
236. List<ProtocolConfig> protocolConfigs = new ArrayList<ProtocolConfig>();
237. for (ProtocolConfig config : protocolConfigMap.values()) {
238. if (config.isDefault() == null || config.isDefault()) {
239. protocolConfigs.add(config);
240. }
241. }
242. if (!protocolConfigs.isEmpty()) {
243. super.setProtocols(protocolConfigs);
244. }
245. }
246. }
247. if (getPath() == null || getPath().length() == 0) {
248. if (beanName != null && beanName.length() > 0
249. && getInterface() != null && getInterface().length() > 0
250. && beanName.startsWith(getInterface())) {
251. setPath(beanName);
252. }
253. }
254. if (!isDelay()) {
255. export();
256. }
257. }
259. @Override
260. public void destroy() throws Exception {
261. *// This will only be called for singleton scope bean, and expected to be called by spring shutdown hook when BeanFactory/ApplicationContext destroys.*
262. *// We will guarantee dubbo related resources being released with dubbo shutdown hook.*
263. *//unexport();*
264. }
266. *// merged from dubbox*
267. @Override
268. protected Class getServiceClass(T ref) {
269. if (AopUtils.isAopProxy(ref)) {
270. return AopUtils.getTargetClass(ref);
271. }
272. return super.getServiceClass(ref);
273. }
274. }

可以看到ServiceBean继承了ServiceConfig类，实现了InitializingBean, DisposableBean, ApplicationContextAware, ApplicationListener, BeanNameAware。

InitializingBean

只包括afterPropertiesSet()方法，继承该接口的类，在初始化bean的时候会执行该方法。在构造方法之后调用。

DisposableBean

DisposableBean也是一个接口，提供了一个唯一的方法destory()。在Bean生命周期结束前调用destory()方法做一些收尾工作.

init-method destory-method

@PostConstruct 和 @PreDestroy

Constructor  >> @Autowired >> @PostConstruct

ApplicationContextAware

Spring容器会检测容器中的所有Bean，如果发现某个Bean实现了ApplicationContextAware接口，Spring容器会在创建该Bean之后，自动调用该Bean的setApplicationContextAware()方法，调用该方法时，会将容器本身作为参数传给该方法

ApplicationListener

当Spring容器初始化完成之后，会发布一个ContextRefreshedEvent事件，实现ApplicationListener接口的类，会调用\*onApplicationEvent(E event)\*方法。

BeanNameAware

public void setBeanName(String beanName) {

this.beanName = beanName;

}

获得自己的名字

BeanFactoryAware

让Bean获取配置他们的BeanFactory的引用。

在Spring注册完成之后会调用finishRefresh方法，在该方法中会调用实现了ApplicationListenner类的onApplicationEvent方法。

1. public void onApplicationEvent(ContextRefreshedEvent event) {
2. if (isDelay() && !isExported() && !isUnexported()) {
3. if (logger.isInfoEnabled()) {
4. logger.info("The service ready on spring started. service: " + getInterface());
5. }
6. export();
7. }
8. }

ServiceBean的OnApplicationEvent方法如上，可以看到在该方法中调用了export实现了服务的注册。

再看一看ReferenceBean的实现：

1. public class ReferenceBean<T> extends ReferenceConfig<T> implements FactoryBean, ApplicationContextAware, InitializingBean, DisposableBean {
3. private static final long serialVersionUID = 213195494150089726L;
5. private transient ApplicationContext applicationContext;
7. public ReferenceBean() {
8. super();
9. }
11. public ReferenceBean(Reference reference) {
12. super(reference);
13. }
15. @Override
16. public void setApplicationContext(ApplicationContext applicationContext) {
17. this.applicationContext = applicationContext;
18. SpringExtensionFactory.addApplicationContext(applicationContext);
19. }
21. @Override
22. public Object getObject() {
23. return get();
24. }
26. @Override
27. public Class<?> getObjectType() {
28. return getInterfaceClass();
29. }
31. @Override
32. @Parameter(excluded = true)
33. public boolean isSingleton() {
34. return true;
35. }
37. @Override
38. @SuppressWarnings({"unchecked"})
39. public void afterPropertiesSet() throws Exception {
40. if (getConsumer() == null) {
41. Map<String, ConsumerConfig> consumerConfigMap = applicationContext == null ? null : BeanFactoryUtils.beansOfTypeIncludingAncestors(applicationContext, ConsumerConfig.class, false, false);
42. if (consumerConfigMap != null && consumerConfigMap.size() > 0) {
43. ConsumerConfig consumerConfig = null;
44. for (ConsumerConfig config : consumerConfigMap.values()) {
45. if (config.isDefault() == null || config.isDefault()) {
46. if (consumerConfig != null) {
47. throw new IllegalStateException("Duplicate consumer configs: " + consumerConfig + " and " + config);
48. }
49. consumerConfig = config;
50. }
51. }
52. if (consumerConfig != null) {
53. setConsumer(consumerConfig);
54. }
55. }
56. }
57. if (getApplication() == null
58. && (getConsumer() == null || getConsumer().getApplication() == null)) {
59. Map<String, ApplicationConfig> applicationConfigMap = applicationContext == null ? null : BeanFactoryUtils.beansOfTypeIncludingAncestors(applicationContext, ApplicationConfig.class, false, false);
60. if (applicationConfigMap != null && applicationConfigMap.size() > 0) {
61. ApplicationConfig applicationConfig = null;
62. for (ApplicationConfig config : applicationConfigMap.values()) {
63. if (config.isDefault() == null || config.isDefault()) {
64. if (applicationConfig != null) {
65. throw new IllegalStateException("Duplicate application configs: " + applicationConfig + " and " + config);
66. }
67. applicationConfig = config;
68. }
69. }
70. if (applicationConfig != null) {
71. setApplication(applicationConfig);
72. }
73. }
74. }
75. if (getModule() == null
76. && (getConsumer() == null || getConsumer().getModule() == null)) {
77. Map<String, ModuleConfig> moduleConfigMap = applicationContext == null ? null : BeanFactoryUtils.beansOfTypeIncludingAncestors(applicationContext, ModuleConfig.class, false, false);
78. if (moduleConfigMap != null && moduleConfigMap.size() > 0) {
79. ModuleConfig moduleConfig = null;
80. for (ModuleConfig config : moduleConfigMap.values()) {
81. if (config.isDefault() == null || config.isDefault()) {
82. if (moduleConfig != null) {
83. throw new IllegalStateException("Duplicate module configs: " + moduleConfig + " and " + config);
84. }
85. moduleConfig = config;
86. }
87. }
88. if (moduleConfig != null) {
89. setModule(moduleConfig);
90. }
91. }
92. }
93. if ((getRegistries() == null || getRegistries().isEmpty())
94. && (getConsumer() == null || getConsumer().getRegistries() == null || getConsumer().getRegistries().isEmpty())
95. && (getApplication() == null || getApplication().getRegistries() == null || getApplication().getRegistries().isEmpty())) {
96. Map<String, RegistryConfig> registryConfigMap = applicationContext == null ? null : BeanFactoryUtils.beansOfTypeIncludingAncestors(applicationContext, RegistryConfig.class, false, false);
97. if (registryConfigMap != null && registryConfigMap.size() > 0) {
98. List<RegistryConfig> registryConfigs = new ArrayList<>();
99. for (RegistryConfig config : registryConfigMap.values()) {
100. if (config.isDefault() == null || config.isDefault()) {
101. registryConfigs.add(config);
102. }
103. }
104. if (!registryConfigs.isEmpty()) {
105. super.setRegistries(registryConfigs);
106. }
107. }
108. }
109. if (getMonitor() == null
110. && (getConsumer() == null || getConsumer().getMonitor() == null)
111. && (getApplication() == null || getApplication().getMonitor() == null)) {
112. Map<String, MonitorConfig> monitorConfigMap = applicationContext == null ? null : BeanFactoryUtils.beansOfTypeIncludingAncestors(applicationContext, MonitorConfig.class, false, false);
113. if (monitorConfigMap != null && monitorConfigMap.size() > 0) {
114. MonitorConfig monitorConfig = null;
115. for (MonitorConfig config : monitorConfigMap.values()) {
116. if (config.isDefault() == null || config.isDefault()) {
117. if (monitorConfig != null) {
118. throw new IllegalStateException("Duplicate monitor configs: " + monitorConfig + " and " + config);
119. }
120. monitorConfig = config;
121. }
122. }
123. if (monitorConfig != null) {
124. setMonitor(monitorConfig);
125. }
126. }
127. }
128. Boolean b = isInit();
129. if (b == null && getConsumer() != null) {
130. b = getConsumer().isInit();
131. }
132. if (b != null && b) {
133. getObject();
134. }
135. }
137. @Override
138. public void destroy() {
139. *// do nothing*
140. }
141. }

在ReferenceBean中是通过FactoryBean来实现消费服务去注册中心拉取相应的服务，建立对应的proxy的。