具体的Spring BeanDefinition的解析是在BeanDefinitionParserDelegate中完成的。这个类里包含了各种Spring Bean定义规则的处理，感兴趣的读者可以仔细研究。比如我们最熟悉的对Bean元素的处理是怎样完成的，也就是在XML定义文件中出现的<bean></bean>这个最常见的元素信息是怎样处理的。在这里，我们会看到那些熟悉的BeanDefinition定义的处理，比如id、name、aliase等属性元素。把这些元素的值从XML文件相应的元素的属性中读取出来以后，会被设置到生成的BeanDefinitionHolder中去。这些属性的解析还是比较简单的。对于其他元素配置的解析，比如各种Bean的属性配置，通过一个较为复杂的解析过程，这个过程是由parseBeanDefinitionElement来完成的。解析完成以后，会把解析结果放到BeanDefinition对象中并设置到BeanDefinitionHolder中去，如代码清单2-16所示。

代码清单2-16  BeanDefinitionParserDelegate对bean元素定义的处理

1. **public** BeanDefinitionHolder
2. parseBeanDefinitionElement(Element ele, BeanDefinition
3. containingBean) {
5. //这里取得在<bean>元素中定义的id、name和aliase属性的值。
6. String id = ele.getAttribute(ID\_ATTRIBUTE);
7. String nameAttr =
8. ele.getAttribute(NAME\_ATTRIBUTE);
9. List<String> aliases = **new**
10. ArrayList<String>();
11. **if** (StringUtils.hasLength(nameAttr)) {
12. String[] nameArr =
13. StringUtils.tokenizeToStringArray(nameAttr, BEAN\_NAME\_
14. DELIMITERS);
16. aliases.addAll(Arrays.asList(nameArr));
17. }
18. String beanName = id;
19. **if** (!StringUtils.hasText(beanName) &&
20. !aliases.isEmpty()) {
21. beanName = aliases.remove(0);
22. **if** (logger.isDebugEnabled()) {
23. logger.debug("No XML 'id'
24. specified - using '" + beanName +
25. "' as bean
26. name and " + aliases + " as aliases");
27. }
28. }
29. **if** (containingBean == **null**) {
30. checkNameUniqueness(beanName,
31. aliases, ele);
32. }
33. //这个方法会引发对bean元素的详细解析。
34. AbstractBeanDefinition beanDefinition =
35. parseBeanDefinitionElement(ele,
36. beanName,containingBean);
37. **if** (beanDefinition != **null**) {
38. **if** (!StringUtils.hasText(beanName))
39. {
40. **try** {
41. **if** (containingBean
42. != **null**) {
43. beanName =
44. BeanDefinitionReaderUtils.generateBeanName(
46. beanDefinition, **this**.readerContext.getRegistry(),
48. **true**);
49. }
50. **else** {
51. beanName =
53. **this**.readerContext.generateBeanName(beanDefinition);
54. /\*\*
55. \*Register an
56. alias for the plain bean class name, if still
57. \*possible,if
58. the generator returned the class name plus a
59. \*suffix.This
60. is expected for Spring 1.2/2.0 backwards
62. \*compatibility.
63. \*/
64. String
65. beanClassName = beanDefinition.getBeanClassName();
66. **if**
67. (beanClassName != **null** &&
69. beanName.startsWith(beanClassName) &&
71. beanName.length() > beanClassName.length() &&
72. !**this**.readerContext.getRegistry().isBeanNameInUse(be
73. anClassName)) {
75. aliases.add(beanClassName);
76. }
77. }
78. **if**
79. (logger.isDebugEnabled()) {
81. logger.debug("Neither XML 'id' nor 'name' specified - " +
83. "using generated bean name [" + beanName + "]");
84. }
85. }
86. **catch** (Exception ex) {
88. error(ex.getMessage(), ele);
89. **return** **null**;
90. }
91. }
92. String[] aliasesArray =
93. StringUtils.toStringArray(aliases);
94. **return** **new**
95. BeanDefinitionHolder(beanDefinition, beanName,
96. aliasesArray);
97. }
98. **return** **null**;
99. }

我们看到了对Bean元素进行解析的过程，也就是BeanDefinition依据XML的<bean>定义被创建的过程。这个BeanDefinition可以看成是<bean>定义的抽象，如图2-7所示。这个数据对象里封装的数据大多都是与<bean>定义相关的，也有很多就是我们在定义Bean时看到的那些Spring标记，比如我们熟悉的init-method、destroy-method、factory-method，等等，这个BeanDefinition数据类型是非常重要的，它封装了很多基本数据。有了这些基本数据，IoC容器才能对Bean配置进行处理，才能实现相应的容器特性。

|  |
| --- |
| <http://images.51cto.com/files/uploadimg/20091221/161848673.jpg> |
|  |

看起来很熟悉吧，beanClass、description、lazyInit这些属性都是在配置Bean时经常碰到的，原来都跑到这里来了。这个BeanDefinition是IoC容器体系中非常重要的核心数据结构。通过解析以后，这些数据已经做好在IoC容器里大显身手的准备了。对BeanDefinition的元素的处理如代码清单2-17所示，在这个过程中可以看到对Bean定义的相关处理，比如对元素attribute值的处理，对元素属性值的处理，对构造函数设置的处理，等等。

代码清单2-17  对BeanDefinition定义元素的处理

1. **public** AbstractBeanDefinition
2. parseBeanDefinitionElement(
3. Element ele, String beanName,
4. BeanDefinition containingBean) {
5. **this**.parseState.push(**new**
6. BeanEntry(beanName));
7. /\*\*
9. \*这里只读取定义的<bean>中设置的class名字，然后载入到BeanDefi
10. nition中去，
12. \*只是做个记录，并不涉及对象的实例化过程，对象的实例化实际上
13. 是在依赖注入时完成的。
14. \*/
15. String className = **null**;
16. **if** (ele.hasAttribute(CLASS\_ATTRIBUTE)) {
17. className =
18. ele.getAttribute(CLASS\_ATTRIBUTE).trim();
19. }
20. **try** {
21. String parent = **null**;
22. **if**
23. (ele.hasAttribute(PARENT\_ATTRIBUTE)) {
24. parent =
25. ele.getAttribute(PARENT\_ATTRIBUTE);
26. }
28. //这里生成需要的BeanDefinition对象，为Bean定义信息的载入做准
29. 备。
30. AbstractBeanDefinition bd =
31. createBeanDefinition(className, parent);
33. //这里对当前的Bean元素进行属性解析，并设置description的信息
34. 。
35. parseBeanDefinitionAttributes(ele,
36. beanName, containingBean, bd);
38. bd.setDescription(DomUtils.getChildElementValueByTagName(ele
39. ,
40. DESCRIPTION\_ELEMENT));
42. //从名字可以清楚地看到，这里是对各种<bean>元素的信息进行解析
43. 的地方。
44. parseMetaElements(ele, bd);
45. parseLookupOverrideSubElements(ele,
46. bd.getMethodOverrides());
47. parseReplacedMethodSubElements(ele,
48. bd.getMethodOverrides());
49. //解析<bean>的构造函数设置。
50. parseConstructorArgElements(ele,
51. bd);
52. //解析<bean>的property设置。
53. parsePropertyElements(ele, bd);
54. parseQualifierElements(ele, bd);
56. bd.setResource(**this**.readerContext.getResource());
57. bd.setSource(extractSource(ele));
58. **return** bd;
59. }
60. /\*\*
61. \*下面这些异常是我们在配置bean出现问题时经常可以看到的，原来
62. 是在这里抛出的，这些检查是在
63. \*createBeanDefinition时进行的，会检查bean的class设置是否正确
64. ，比如这个类是不是能找到。
65. \*/
66. **catch** (ClassNotFoundException ex) {
67. error("Bean class [" + className +
68. "] not found", ele, ex);
69. }
70. **catch** (NoClassDefFoundError err) {
71. error("Class that bean class [" +
72. className + "] depends on not found",
73. ele, err);
74. }
75. **catch** (Throwable ex) {
76. error("Unexpected failure during
77. bean definition parsing", ele, ex);
78. }
79. **finally** {
80. **this**.parseState.pop();
81. }
82. **return** **null**;
83. }