上面是具体生成BeanDefinition的地方。在这里，我们举一个对property进行解析的例子来完成对整个BeanDefinition载入过程的分析，还是在类BeanDefinitionParserDelegate的代码中，它对BeanDefinition中的定义一层一层地进行解析，比如从属性元素集合到具体的每一个属性元素，然后才是对具体的属性值的处理。根据解析结果，对这些属性值的处理会封装成PropertyValue对象并设置到BeanDefinition对象中去，如代码清单2-18所示。

代码清单2-18  对BeanDefinition中Property元素集合的处理

1. //这里对指定bean元素的property子元素集合进行解析。
2. **public** **void** parsePropertyElements(Element beanEle,
3. BeanDefinition bd) {
4. //遍历所有bean元素下定义的property元素。
5. NodeList nl = beanEle.getChildNodes();
6. **for** (**int** i = 0; i < nl.getLength(); i++) {
7. Node node = nl.item(i);
8. **if** (node **instanceof** Element &&
9. DomUtils.nodeNameEquals(node, PROPERTY\_
10. ELEMENT)) {
12. //在判断是property元素后对该property元素进行解析的过程。
13. parsePropertyElement((Element) node,
14. bd);
15. }
16. }
17. }
18. **public** **void** parsePropertyElement(Element ele, BeanDefinition
19. bd) {
20. //这里取得property的名字。
21. String propertyName =
22. ele.getAttribute(NAME\_ATTRIBUTE);
23. **if** (!StringUtils.hasLength(propertyName)) {
24. error("Tag 'property' must have a 'name'
25. attribute", ele);
26. **return**;
27. }
28. **this**.parseState.push(**new**
29. PropertyEntry(propertyName));
30. **try** {
31. /\*\*
33. \*如果同一个bean中已经有同名property的存在，则不进行解析，直
34. 接返回。也就是说，
36. \*如果在同一个bean中有同名的property设置，那么起作用的只是第
37. 一个。
38. \*/
39. **if**
40. (bd.getPropertyValues().contains(propertyName)) {
41. error("Multiple 'property'
42. definitions **for** property '" +
43. propertyName + "'", ele);
44. **return**;
45. }
46. /\*\*
48. \*这里是解析property值的地方，返回的对象对应对Bean定义的prope
49. rty属性
51. \*设置的解析结果，这个解析结果会封装到PropertyValue对象中，然
52. 后设置到
53. \*BeanDefinitionHolder中去。
54. \*/
55. Object val = parsePropertyValue(ele, bd,
56. propertyName);
57. PropertyValue pv = **new**
58. PropertyValue(propertyName, val);
59. parseMetaElements(ele, pv);
60. pv.setSource(extractSource(ele));
61. bd.getPropertyValues().addPropertyValue(pv);
62. }
63. **finally** {
64. **this**.parseState.pop();
65. }
66. }
67. //这里取得property元素的值，也许是一个list或其他。
68. **public** Object parsePropertyValue(Element ele, BeanDefinition
69. bd, String propertyName) {
70. String elementName = (propertyName != **null**) ?
71. "<property> element
72. **for** property '" + propertyName + "'" :
73. "<constructor-arg>
74. element";
75. // Should only have one child element: ref, value,
76. list, etc.
77. NodeList nl = ele.getChildNodes();
78. Element subElement = **null**;
79. **for** (**int** i = 0; i < nl.getLength(); i++) {
80. Node node = nl.item(i);
81. **if** (node **instanceof** Element &&
82. !DomUtils.nodeNameEquals(node, DESCRIPTION\_
83. ELEMENT) &&
85. !DomUtils.nodeNameEquals(node, META\_ELEMENT)) {
86. // Child element is what we're
87. looking **for**.
88. **if** (subElement != **null**) {
89. error(elementName + " must
90. not contain more than one sub-element",
91. ele);
92. }
93. **else** {
94. subElement = (Element) node;
95. }
96. }
97. }
98. //这里判断property的属性，是ref还是value，不允许同时
99. 是ref和value。
100. **boolean** hasRefAttribute =
101. ele.hasAttribute(REF\_ATTRIBUTE);
102. **boolean** hasValueAttribute =
103. ele.hasAttribute(VALUE\_ATTRIBUTE);
104. **if** ((hasRefAttribute && hasValueAttribute) ||
105. ((hasRefAttribute ||
106. hasValueAttribute) && subElement != **null**)) {
107. error(elementName +
108. " is only allowed to contain
109. either 'ref' attribute OR 'value'
110. attribute OR sub-element", ele);
111. }
112. //如果是ref，创建一个ref的数据对象RuntimeBeanReferen
113. ce，这个对象封装了ref的信息。
114. **if** (hasRefAttribute) {
115. String refName =
116. ele.getAttribute(REF\_ATTRIBUTE);
117. **if** (!StringUtils.hasText(refName)) {
118. error(elementName + " contains empty
119. 'ref' attribute", ele);
120. }
121. RuntimeBeanReference ref = **new**
122. RuntimeBeanReference(refName);
123. ref.setSource(extractSource(ele));
124. **return** ref;
125. }
126. //如果是value，创建一个它的数据对象TypedStringValue，这个对
127. 象封装了value的信息。
128. **else** **if** (hasValueAttribute) {
129. TypedStringValue valueHolder = **new**
130. TypedStringValue(ele.getAttribute
131. (VALUE\_ATTRIBUTE));
132. valueHolder.setSource(extractSource(ele));
133. **return** valueHolder;
134. } //如果还有子元素，触发对子元素的解析。
135. **else** **if** (subElement != **null**) {
136. **return** parsePropertySubElement(subElement,
137. bd);
138. }
139. **else** {
140. // Neither child element nor "ref" or
141. "value" attribute found.
142. error(elementName + " must specify a ref or
143. value", ele);
144. **return** **null**;
145. }
146. }

这里是对property子元素的解析过程，Array、List、Set、Map、Prop等各种元素都会在这里进行解析，生成对应的数据对象，比如ManagedList、ManagedArray、ManagedSet等。这些Managed类是Spring对具体的BeanDefinition的数据封装。具体的解析过程读者可以去看看自己感兴趣的部分，比如parseArrayElement、parseListElement、parseSetElement、parseMapElement、parsePropElement对应着不同类型的数据解析，同时这些具体的解析方法在BeanDefinitionParserDelegate类中也都能够找到。因为方法命名很清晰，所以从方法名字上就能够很快地找到。下面，以对Property的元素进行解析的过程为例，通过它的实现来说明这个具体的解析过程是怎样完成的，如代码清单2-19所示。

代码清单2-19  对属性元素进行解析

1. **public** Object parsePropertySubElement(Element ele,
2. BeanDefinition bd, String
3. defaultValueType) {
4. **if** (!isDefaultNamespace(ele.getNamespaceURI())) {
5. **return** parseNestedCustomElement(ele, bd);
6. }
7. **else** **if** (DomUtils.nodeNameEquals(ele, BEAN\_ELEMENT))
8. {
9. BeanDefinitionHolder nestedBd =
10. parseBeanDefinitionElement(ele, bd);
11. **if** (nestedBd != **null**) {
12. nestedBd =
13. decorateBeanDefinitionIfRequired(ele, nestedBd, bd);
14. }
15. **return** nestedBd;
16. }
17. **else** **if** (DomUtils.nodeNameEquals(ele, REF\_ELEMENT))
18. {
19. // A generic reference to any name of any
20. bean.
21. String refName =
22. ele.getAttribute(BEAN\_REF\_ATTRIBUTE);
23. **boolean** toParent = **false**;
24. **if** (!StringUtils.hasLength(refName)) {
25. // A reference to the id of another
26. bean in the same XML file.
27. refName =
28. ele.getAttribute(LOCAL\_REF\_ATTRIBUTE);
29. **if** (!StringUtils.hasLength(refName))
30. {
31. // A reference to the id of
32. another bean in a parent context.
33. refName =
34. ele.getAttribute(PARENT\_REF\_ATTRIBUTE);
35. toParent = **true**;
36. **if**
37. (!StringUtils.hasLength(refName)) {
38. error("'bean',
39. 'local' or 'parent' is required **for** <ref>
40. element", ele);
41. **return** **null**;
42. }
43. }
44. }
45. **if** (!StringUtils.hasText(refName)) {
46. error("<ref> element contains empty
47. target attribute", ele);
48. **return** **null**;
49. }
50. RuntimeBeanReference ref = **new**
51. RuntimeBeanReference(refName, toParent);
52. ref.setSource(extractSource(ele));
53. **return** ref;
54. }
55. **else** **if** (DomUtils.nodeNameEquals(ele,
56. IDREF\_ELEMENT)) {
57. **return** parseIdRefElement(ele);
58. }
59. **else** **if** (DomUtils.nodeNameEquals(ele,
60. VALUE\_ELEMENT)) {
61. **return** parseValueElement(ele,
62. defaultValueType);
63. }
64. **else** **if** (DomUtils.nodeNameEquals(ele, NULL\_ELEMENT))
65. {
66. // It's a distinguished null value. Let's
67. wrap it in a TypedStringValue
68. // object in order to preserve the source
69. location.
70. TypedStringValue nullHolder = **new**
71. TypedStringValue(**null**);
72. nullHolder.setSource(extractSource(ele));
73. **return** nullHolder;
74. }
75. **else** **if** (DomUtils.nodeNameEquals(ele,
76. ARRAY\_ELEMENT)) {
77. **return** parseArrayElement(ele, bd);
78. }
79. **else** **if** (DomUtils.nodeNameEquals(ele, LIST\_ELEMENT))
80. {
81. **return** parseListElement(ele, bd);
82. }
83. **else** **if** (DomUtils.nodeNameEquals(ele, SET\_ELEMENT))
84. {
85. **return** parseSetElement(ele, bd);
86. }
87. **else** **if** (DomUtils.nodeNameEquals(ele, MAP\_ELEMENT))
88. {
89. **return** parseMapElement(ele, bd);
90. }
91. **else** **if** (DomUtils.nodeNameEquals(ele,
92. PROPS\_ELEMENT)) {
93. **return** parsePropsElement(ele);
94. }
95. **else** {
96. error("Unknown property sub-element: [" +
97. ele.getNodeName() + "]", ele);
98. **return** **null**;
99. }
100. }

我们看看类似List这样的属性配置是怎样被解析的，依然在BeanDefinitionParser-Delegate中，返回的是一个List对象，这个List是Spring定义的ManagedList，作为封装List这类配置定义的数据封装，如代码清单2-20所示。

代码清单2-20  解析BeanDefinition中的List元素

1. **public** List parseListElement(Element collectionEle,
2. BeanDefinition bd) {
3. String defaultElementType =
4. collectionEle.getAttribute(VALUE\_TYPE\_ATTRIBUTE);
5. NodeList nl = collectionEle.getChildNodes();
6. ManagedList<Object> target = **new**
7. ManagedList<Object>(nl.getLength());
8. target.setSource(extractSource(collectionEle));
9. target.setElementTypeName(defaultElementType);
10. target.setMergeEnabled(parseMergeAttribute(collectio
11. nEle));
12. //具体的List元素的解析过程。
13. parseCollectionElements(nl, target, bd,
14. defaultElementType);
15. **return** target;
16. }
17. **protected** **void** parseCollectionElements(
18. NodeList elementNodes, Collection<Object>
19. target, BeanDefinition bd,
20. String defaultElementType) {
21. //遍历所有的元素节点，并判断其类型是否为Element。
22. **for** (**int** i = 0; i < elementNodes.getLength(); i++) {
23. Node node = elementNodes.item(i);
24. **if** (node **instanceof** Element &&
25. !DomUtils.nodeNameEquals(node, DESCRIPTION\_
26. ELEMENT)) {
27. /\*\*
28. \*加入到target中，target是一个ManagedList，同时触发对
29. 下一层子元素的解析过程，
30. \*这是一个递归调用。
31. \*/
33. target.add(parsePropertySubElement((Element) node, bd,
34. defaultElement
35. Type));
36. }
37. }
38. }

经过这样逐层地解析，我们在XML文件中定义的BeanDefinition就被整个给载入到了IoC容器中，并在容器中建立了数据映射。在IoC容器中建立了对应的数据结构，或者说可以看成是POJO对象在IoC容器中的映像，这些数据结构可以以AbstractBeanDefinition为入口，让IoC容器执行索引、查询和操作。简单的POJO操作背后其实并不简单，经过以上的载入过程，IoC容器大致完成了管理Bean对象的数据准备工作（或者说是初始化过程）。但是，重要的依赖注入实际上在这个时候还没有发生，现在，在IoC容器BeanDefinition中存在的还只是一些静态的配置信息。严格地来说，这时候的容器还没有完全起作用，要完全发挥容器的作用，还需完成数据向容器的注册。