接着就是loadBeanDefinitions调用的地方，首先得到BeanDefinition信息的Resource定位，然后直接调用XmlBeanDefinitionReader读取，具体的载入过程是委托给BeanDefinitionReader完成的。因为这里的BeanDefinition是通过XML文件定义的，所以这里使用XmlBeanDefinitionReader来载入BeanDefinition到容器中，如代码清单2-11所示。

代码清单2-11  XmlBeanDefinitionReader载入XmlBeanDefinitionReader

1. **protected** **void** loadBeanDefinitions
2. (XmlBeanDefinitionReader reader) **throws**
3. BeansException, IOException {
4. Resource[] configResources = getConfigResources();
5. **if** (configResources != **null**) {
6. reader.loadBeanDefinitions(configResources);
7. }
8. String[] configLocations = getConfigLocations();
9. **if** (configLocations != **null**) {
10. reader.loadBeanDefinitions(configLocations);
11. }
12. }
13. **protected** Resource[] getConfigResources() {
14. **return** **null**;
15. }
16. }

通过以上实现原理的分析，我们可以看到，在初始化FileSystmXmlApplicationContext的过程中，是通过调用IoC容器的refresh来启动整个BeanDefinition的载入过程的，这个初始化是通过定义的XmlBeanDefinitionReader来完成的。同时，我们也知道实际使用的IoC容器是DefultListableBeanFactory，具体的Resource载入在XmlBeanDefinitionReader读入BeanDefinition时实现。因为Spring可以对应不同形式的BeanDefinition。由于这里使用的是XML方式的定义，所以需要使用XmlBeanDefinitionReader。如果使用了其他的BeanDefinition方式，就需要使用其他种类的BeanDefinitionReader来完成数据的载入工作。在XmlBeanDefinitionReader的实现中可以看到，是在reader.loadBeanDefinitions中开始进行BeanDefinition的载入的，而这时XmlBeanDefinitionReader的父类AbstractBeanDefinitionReader已经为BeanDefinition的载入做好了准备，如代码清单2-12所示。

代码清单2-12  AbstractBeanDefinitionReader载入BeanDefinitions

1. **public** **int** loadBeanDefinitions(Resource[] resources) **throws**
2. BeanDefinitionStoreException {
3. //如果Resource为空，则停止BeanDefinition的载入。
4. /\*\*
5. \*然后启动载入BeanDefinition的过程,
6. 这个过程会遍历整个Resource集合所包含的
7. \*BeanDefinition信息。
8. \*/
9. Assert.notNull(resources, "Resource array must not be null");
10. **int** counter = 0;
11. **for** (**int** i = 0; i < resources.length; i++) {
12. counter += loadBeanDefinitions(resources[i]);
13. }
14. **return** counter;
15. }

这里调用的是loadBeanDefinitions(Resource res)方法，然而这个方法在AbstractBeanDefinitionReader类里是没有实现的，它是一个接口方法，具体的实现在XmlBeanDefinitionReader中。在读取器中，需要得到代表XML文件的Resource，因为这个Resource对象封装了对XML文件的IO操作，所以读取器可以在打开IO流后得到XML的文件对象。有了这个Document对象以后，就可以按照Spring的Bean定义规则来对这个XML的文档树进行解析了，这个解析是交给BeanDefinitionParserDelegate来完成的，看起来实现脉络很清楚。具体可以参考代码实现，如代码清单2-13所示。

代码清单2-13  对BeanDefinition的载入实现

1. //这里是调用的入口。
2. **public** **int** loadBeanDefinitions(Resource resource) **throws**
3. BeanDefinitionStoreException {
4. **return** loadBeanDefinitions(**new**
5. EncodedResource(resource));
6. }
7. //这里是载入XML形式的BeanDefinition的地方。
8. **public** **int** loadBeanDefinitions(EncodedResource
9. encodedResource) **throws**
10. BeanDefinitionStoreException {
11. Assert.notNull(encodedResource, "EncodedResource
12. must not be **null**");
13. **if** (logger.isInfoEnabled()) {
14. logger.info("Loading XML bean definitions
15. from " + encodedResource.
16. getResource());
17. }
18. Set<EncodedResource> currentResources =
19. **this**.resourcesCurrentlyBeingLoaded.
20. get();
21. **if** (currentResources == **null**) {
22. currentResources = **new**
23. HashSet<EncodedResource>(4);
25. **this**.resourcesCurrentlyBeingLoaded.set(currentResources);
26. }
27. **if** (!currentResources.add(encodedResource)) {
28. **throw** **new** BeanDefinitionStoreException(
29. "Detected recursive loading
30. of " + encodedResource + " - check
31. your **import** definitions!");
32. }
33. //这里得到XML文件，并得到IO的InputSource准备进行读取
34. 。
35. **try** {
36. InputStream inputStream =
37. encodedResource.getResource().getInputStream();
38. **try** {
39. InputSource inputSource = **new**
40. InputSource(inputStream);
41. **if** (encodedResource.getEncoding() !=
42. **null**) {
44. inputSource.setEncoding(encodedResource.getEncoding());
45. }
46. **return**
47. doLoadBeanDefinitions(inputSource,
48. encodedResource.getResource());
49. }
50. **finally** {
51. inputStream.close();
52. }
53. }
54. **catch** (IOException ex) {
55. **throw** **new** BeanDefinitionStoreException(
56. "IOException parsing XML
57. document from " + encodedResource.
58. getResource(), ex);
59. }
60. **finally** {
61. currentResources.remove(encodedResource);
62. **if** (currentResources.isEmpty()) {
64. **this**.resourcesCurrentlyBeingLoaded.set(**null**);
65. }
66. }
67. }
68. //具体的读取过程可以在doLoadBeanDefinitions方法中找到。
69. //这是从特定的XML文件中实际载入BeanDefinition的地方。
70. **protected** **int** doLoadBeanDefinitions(InputSource inputSource,
71. Resource resource)
72. **throws** BeanDefinitionStoreException {
73. **try** {
74. **int** validationMode =
75. getValidationModeForResource(resource);
76. /\*\*
78. \*这里取得XML文件的Document对象，这个解析过程是由
79. documentLoader完成的，
81. \*这个documentLoader是DefaultDocumentLoader,在定义documentLoa
82. der的地方创建。
83. \*/
84. Document doc =
85. **this**.documentLoader.loadDocument(
86. inputSource,
87. getEntityResolver(), **this**.errorHandler, validationMode,
88. isNamespaceAware());
89. /\*\*
91. \*这里启动的是对BeanDefinition解析的详细过程，这个解析会使用
92. 到Spring的Bean
93. \*配置规则，是我们下面需要详细关注的地方。
94. \*/
95. **return** registerBeanDefinitions(doc,
96. resource);
97. }
98. **catch** (BeanDefinitionStoreException ex) {
99. **throw** ex;
100. }
101. **catch** (SAXParseException ex) {
102. **throw** **new**
103. XmlBeanDefinitionStoreException(resource.getDescription(),
104. "Line " + ex.getLineNumber()
105. + " in XML document from " +
106. resource + " is invalid", ex);
107. }
108. **catch** (SAXException ex) {
109. **throw** **new**
110. XmlBeanDefinitionStoreException(resource.getDescription(),
111. "XML document from " +
112. resource + " is invalid", ex);
113. }
114. **catch** (ParserConfigurationException ex) {
115. **throw** **new**
116. BeanDefinitionStoreException(resource.getDescription(),
117. "Parser configuration
118. exception parsing XML from " + resource,
119. ex);
120. }
121. **catch** (IOException ex) {
122. **throw** **new**
123. BeanDefinitionStoreException(resource.getDescription(),
124. "IOException parsing XML
125. document from " + resource, ex);
126. }
127. **catch** (Throwable ex) {
128. **throw** **new**
129. BeanDefinitionStoreException(resource.getDescription(),
130. "Unexpected exception
131. parsing XML document from " + resource,
132. ex);
133. }
134. }

感兴趣的读者，可以到DefaultDocumentLoader中去看看是怎样得到Document对象的，这里就不详细分析了。我们关心的是Spring的BeanDefinion是怎样按照Spring的Bean语义要求进行解析并转化为容器内部数据结构的，这个过程是在registerBeanDefinitions (doc, resource)中完成的。具体的过程是由BeanDefinitionDocumentReader来完成的，这个registerBeanDefinition还对载入的Bean的数量进行了统计。具体的过程如代码清单2-14所示。

代码清单2-14  registerBeanDefinition的代码实现

1. **public** **int** registerBeanDefinitions(Document doc, Resource
2. resource) **throws**
3. BeanDefinitionStoreException {
4. // Read document based on new
5. BeanDefinitionDocumentReader SPI.
6. // 这里得到
7. BeanDefinitionDocumentReader来对xml的BeanDefinition进行解析
8. 。
9. BeanDefinitionDocumentReader documentReader =
10. createBeanDefinitionDocumentReader();
11. **int** countBefore =
12. getRegistry().getBeanDefinitionCount();
13. //
14. 具体的解析过程在这个registerBeanDefinitions中完成。
15. documentReader.registerBeanDefinitions(doc,
16. createReaderContext(resource));
17. **return** getRegistry().getBeanDefinitionCount() -
18. countBefore;
19. }

BeanDefinition的载入包括两部分，首先是通过调用XML的解析器得到document对象，但这些document对象并没有按照Spring的Bean规则进行解析。在完成通用的XML解析以后，才是按照Spring的Bean规则进行解析的地方，按照Spring的Bean规则进行解析的过程是在documentReader中实现的。这里使用的documentReader是默认设置好的DefaultBeanDe-finitionDocumentReader。这个DefaultBeanDefinitionDocumentReader的创建是在以下的方法里完成的，然后再完成BeanDefinition的处理，处理的结果由BeanDefinition-Holder对象来持有。这个BeanDefinitionHolder除了持有BeanDefinition对象外，还持有了其他与BeanDefinition的使用相关的信息，比如Bean的名字、别名集合等。这个Bean-DefinitionHolder的生成是通过对Document文档树的内容进行解析来完成的，可以看到这个解析过程是由BeanDefinitionParserDelegate来实现（具体在processBeanDefinition方法中实现）的，同时这个解析是与Spring对BeanDefinition的配置规则紧密相关的。具体的实现原理如代码清单2-15所示。

代码清单2-15  创建BeanDefinitionDocumentReader

1. **protected** BeanDefinitionDocumentReader
2. createBeanDefinitionDocumentReader() {
3. **return**
4. BeanDefinitionDocumentReader.**class**.cast(BeanUtils.instantiat
5. eClass
6. (**this**.documentReaderClass));
7. }
8. //这样，得到了 documentReader以后，为具体的Spring
9. Bean的解析过程准备好了数据。
10. /\*\*
11. \*这里是处理BeanDefinition的地方，具体的处理是委托给
12. BeanDefinitionParserDelegate
13. \*来完成的，ele对应于我们的Spring
14. BeanDefinition中定义的xml元素。
15. \*/
16. **protected** **void** processBeanDefinition(Element ele,
17. BeanDefinitionParserDelegate
18. delegate) {
19. /\*
20. BeanDefinitionHolder是BeanDefinition对象的封装类，封装了Bean
21. Definition，
23. \*Bean的名字和别名。用它来完成向IoC容器注册。得到这个
24. BeanDefinitionHolder实际上就意
26. \*味着获得了BeanDefinition,是通过BeanDefinitionParserDelegate
27. 对XML元素的信息按照
28. \*Spring的Bean规则进行解析得到的。
29. \*/
30. BeanDefinitionHolder bdHolder =
31. delegate.parseBeanDefinitionElement(ele);
32. **if** (bdHolder != **null**) {
33. bdHolder =
34. delegate.decorateBeanDefinitionIfRequired(ele, bdHolder);
35. **try** {
36. //
37. 这里是向IoC容器注册解析得到的BeanDefinition的地方。
39. BeanDefinitionReaderUtils.registerBeanDefinition(bdHolder,
40. getReaderContext().getRegistry());
41. }
42. **catch** (BeanDefinitionStoreException
43. ex) {
45. getReaderContext().error("Failed to register bean definition
46. with
47. name '" + bdHolder.getBeanName() + "'", ele,
48. ex);
49. }
50. //
51. 在BeanDefinition向IoC容器注册完以后，发送消息。
53. getReaderContext().fireComponentRegistered(**new**
54. BeanComponentDefinition
55. (bdHolder));
56. }
57. }