**2.4  IoC容器的依赖注入**

假设当前IoC容器已经载入了用户定义的Bean信息，并开始分析依赖注入的原理。首先，注意到依赖注入的过程是用户第一次向IoC容器索要Bean时触发的，当然也有例外，也就是我们可以在BeanDefinition信息中通过控制lazy-init属性来让容器完成对Bean的预实例化。这个预实例化实际上也是一个完成依赖注入的过程，但它在初始化的过程中完成，稍后我们会详细分析这个预实例化的处理。当用户向IoC容器索要Bean时，如果读者还有印象，那么一定还记得在基本的IoC容器接口BeanFactory中，有一个getBean的接口定义，这个接口的实现就是触发依赖注入发生的地方。为了进一步了解这个依赖注入过程的实现，我们从DefaultListableBeanFactory的基类AbstractBeanFactory入手去看看getBean的实现，如代码清单2-22所示。

代码清单2-22  getBean触发的依赖注入

1. //----------------------------------------------------------
2. ----------
3. // Implementation of BeanFactory interface.
4. // 这里是对
5. BeanFactory接口的实现，比如getBean接口方法。
6. //
7. 这些getBean接口方法最终是通过调用doGetBean来实现的。
8. //--------------------------------------------------
9. ------------------
10. **public** Object getBean(String name) **throws**
11. BeansException {
12. **return** doGetBean(name, **null**, **null**, **false**);
13. }
14. **public**  T getBean(String name, Class
15. requiredType) **throws** BeansException {
16. **return** doGetBean(name, requiredType, **null**,
17. **false**);
18. }
19. **public** Object getBean(String name, Object... args)
20. **throws** BeansException {
21. **return** doGetBean(name, **null**, args, **false**);
22. }
23. **public**  T getBean(String name, Class
24. requiredType, Object[] args) **throws**
25. BeansException {
26. **return** doGetBean(name, requiredType, args,
27. **false**);
28. }
29. //这里是实际去取bean的地方，也是触发依赖注入发生的地
30. 方。
31. **protected**  T doGetBean(
32. **final** String name, **final** Class
33. requiredType, **final** Object[] args,
34. **boolean** typeCheckOnly)
35. **throws** BeansException {
36. **final** String beanName =
37. transformedBeanName(name);
38. Object bean;
39. // Eagerly check singleton cache for
40. manually registered singletons.
42. //先从缓存中去取,处理已经被创建过的单件模式的bean，对这种bea
43. n的请求不需要重复地创建。
44. Object sharedInstance =
45. getSingleton(beanName);
46. **if** (sharedInstance != **null** && args == **null**)
47. {
48. **if** (logger.isDebugEnabled()) {
49. **if**
50. (isSingletonCurrentlyInCreation(beanName)) {
52. logger.debug("Returning eagerly cached instance of singleton
53. bean '" + beanName +
54. "' that is not fully
55. initialized yet - a consequence of a
56. circular reference");
57. }
58. **else** {
60. logger.debug("Returning cached instance of singleton bean '"
61. +
62. beanName + "'");
63. }
64. }
65. /\*\*
67. \*这里的getObjectForBeanInstance完成的是FactoryBean的相关处理
68. ，
70. \*以取得FactoryBean的生产结果，我们在前面介绍过BeanFactory和F
71. actoryBean的区别，
72. \*这个过程我们在下面还会详细地分析。
73. \*/
74. bean = getObjectForBeanInstance(sharedInstance,
75. name, beanName, **null**);
76. }
77. **else** {
78. // Fail if we're already creating
79. **this** bean instance:
80. // We're assumably within a circular
81. reference.
82. **if**
83. (isPrototypeCurrentlyInCreation(beanName)) {
84. **throw** **new**
85. BeanCurrentlyInCreationException(beanName);
86. }
87. // Check if bean definition exists in this
88. factory.
89. /\*\*
91. \*这里对IoC容器里的BeanDefintion是否存在进行检查，检查是否能
92. 在当前的BeanFactory中
94. \*取到我们需要的bean。如果在当前的工厂中取不到，则到双亲BeanF
95. actory中去取；如果当前的
97. \*双亲工厂取不到，那就顺着双亲BeanFactory链一直向上查找。
98. \*/
99. BeanFactory parentBeanFactory =
100. getParentBeanFactory();
101. **if** (parentBeanFactory != **null** &&
102. !containsBeanDefinition(beanName)) {
103. // Not found -> check
104. parent.
105. String nameToLookup =
106. originalBeanName(name);
107. **if** (args != **null**) {
108. // Delegation to
109. parent with explicit args.
110. **return** (T)
111. parentBeanFactory.getBean(nameToLookup, args);
112. }
113. **else** {
114. // No args ->
115. delegate to standard getBean method.
116. **return**
117. parentBeanFactory.getBean(nameToLookup, requiredType);
118. }
119. }
120. **if** (!typeCheckOnly) {
121. markBeanAsCreated(beanName);
122. }
124. //这里根据bean的名字取得BeanDefinition。
125. **final** RootBeanDefinition mbd =
126. getMergedLocalBeanDefinition(beanName);
127. checkMergedBeanDefinition(mbd,
128. beanName, args);
129. // Guarantee initialization of beans
130. that the current bean depends on.
131. /\*\*
133. \*取当前bean的所有依赖bean，这样会触发getBean的递归调用，直至
134. 取到一个
135. \*没有任何依赖的bean为止。
136. \*/
137. String[] dependsOn =
138. mbd.getDependsOn();
139. **if** (dependsOn != **null**) {
140. **for** (String dependsOnBean :
141. dependsOn) {
143. getBean(dependsOnBean);
145. registerDependentBean(dependsOnBean, beanName);
146. }
147. }
148. // Create bean instance.
149. /\*\*
150. \*这里创建Singleton
151. bean的实例，通过调用createBean方法，这里有一个
153. \*回调函数getObject，会在getSingleton中去调用ObjectFactory的c
154. reateBean
155. \*/
156. //
157. 下面会进入到createBean中去进行详细分析。
158. **if** (mbd.isSingleton()) {
159. sharedInstance =
160. getSingleton(beanName, **new** ObjectFactory() {
161. **public** Object
162. getObject() **throws** BeansException {
163. **try** {
165. **return** createBean(beanName, mbd, args);
166. }
167. **catch**
168. (BeansException ex) {
169. /\*\*
171. \*Explicitly remove instance from singleton cache: It
173. \*might have been put there eagerly by the creation
175. \*process, to allow for circular reference resolution.
177. \*Also remove any beans that received a temporary
179. \*reference to the bean.
180. /\*
182. destroySingleton(beanName);
184. throw ex;
185. }
186. }
187. });
188. bean =
189. getObjectForBeanInstance(sharedInstance, name, beanName,
190. mbd);
191. }
192. //这里是创建prototype bean的地方。
193. else if (mbd.isPrototype()) {
194. // It's a prototype ->
195. create a new instance.
196. Object prototypeInstance =
197. null;
198. try {
200. beforePrototypeCreation(beanName);
201. prototypeInstance =
202. createBean(beanName, mbd, args);
203. }
204. finally {
206. afterPrototypeCreation(beanName);
207. }
208. bean =
209. getObjectForBeanInstance(prototypeInstance, name, beanName,
210. mbd);
211. }
212. else {
213. String scopeName =
214. mbd.getScope();
215. final Scope scope =
216. this.scopes.get(scopeName);
217. if (scope == null) {
218. throw new
219. IllegalStateException("No Scope registered for scope
220. '" + scopeName + "'");
221. }
222. try {
223. Object
224. scopedInstance = scope.get(beanName, new ObjectFactory() {
225. public
226. Object getObject() throws BeansException {
228. beforePrototypeCreation(beanName);
229. try
230. {
232. return createBean(beanName, mbd, args);
233. }
235. finally {
237. afterPrototypeCreation(beanName);
238. }
239. }
240. });
241. bean =
242. getObjectForBeanInstance(scopedInstance, name, beanName,
243. mbd);
244. }
245. catch (IllegalStateException
246. ex) {
247. throw new
248. BeanCreationException(beanName,
250. "Scope '" + scopeName + "' is not active for the
251. current thread; "
252. +"consider defining a scoped Proxy
253. for this bean if you
254. intend to refer to it from a
255. singleton",ex);
256. }
257. }
258. }
259. // Check if required type matches the type
260. of the actual bean instance.
261. /\*\*
263. \*这里对创建出来的bean进行类型检查，如果没有问题，就返回这个
264. 新创建出来的bean，
265. \*这个bean已经是包含了依赖关系的bean。
266. \*/
267. **if** (requiredType != **null** && bean != **null** &&
268. !requiredType.isAssignableFrom
269. (bean.getClass())) {
270. **throw** **new**
271. BeanNotOfRequiredTypeException(name, requiredType, bean.
272. getClass());
273. }
274. **return** (T) bean;
275. }

这个就是依赖注入的入口，在这里触发了依赖注入，而依赖注入的发生是在容器中的BeanDefinition数据已经建立好的前提下进行的。"程序=数据+算法"，很经典的一句话，前面的BeanDefinition就是数据，下面看看这些数据是怎样为依赖注入服务的。虽然依赖注入的过程不涉及复杂的算法问题，但这个过程也不简单，因为我们都知道，对于IoC容器的使用，Spring提供了许多参数的配置，每一个参数配置实际上代表了一个IoC容器的实现特性，这些特性的实现很多都需要在依赖注入的过程中或者对Bean进行生命周期管理的过程中来完成。尽管我们可以用最简单的方式来描述IoC容器，说它就是一个hashMap而已。是的，我们可以这样说，但只能说这个hashMap是容器的最基本的数据结构，而不是IoC容器的全部。Spring IoC容器的价值体现在一系列相关的产品特性上，这些产品特性以依赖反转模式的实现为核心，为用户更好地使用依赖反转提供便利，从而实现了一个完整的IoC容器产品。这些产品特性的实现并不是一个简单的过程，它提供了一个成熟的IoC容器产品来供用户使用。所以，尽管它没有什么独特的算法，但却可以看成是一个成功的软件工程产品，有许多值得我们学习的地方。