**2.5  容器其他相关特性的实现**

在前面的IoC 原理分析中，我们对IoC 容器的主要功能进行了分析，比如BeanDefinition的载入和解析，依赖注入的实现，等等。为了更全面地理解 IoC 容器的特性，下面对容器的一些其他相关特性的实现原理也进行简要的分析，这些特性都是我们在使用容器时会经常遇到的。这些特性其实很多，这里只选择了几个例子供读者参考。在了解了 IoC 容器的整体运行原理以后，对这些特性的分析不再是一件困难的事情。如果读者对其他 IoC 容器的功能特性感兴趣，也可以按照相同的思路进行分析。

**2.5.1  lazy-init属性和预实例化**

在 IoC 容器的初始化过程中，主要的工作是对 BeanDefinition 的定位、载入、解析和注册。此时依赖注入并没有发生，依赖注入发生在应用第一次向容器索要 Bean 时。向容器索要Bean是通过getBean的调用来完成的，这个getBean是容器提供Bean服务的最基本的接口。对于容器的初始化也有一种例外情况，就是用户可以通过设置 Bean 的 lazy-init 属性来控制预实例化的过程。这个预实例化在初始化容器时完成 Bean 的依赖注入，毫无疑问，这种容器的使用方式会对容器初始化的性能有一些影响，但却能够提高应用第一次取得 Bean 的性能。因为应用在第一次取得Bean时，依赖注入已经结束了，应用可以取到现成的Bean。

我们回头看看在上下文的初始化过程中，也就是 refresh 中的代码实现，可以看到预实例化是整个refresh初始化IoC容器的一个步骤。在AbstractApplicationContext中看看这个refresh 方法的实现，这个初始化的过程在前面分析 IoC 容器初始化时已经分析过，只不过是从载入和注册BeanDefinition的角度进行分析的。

下面，我们将从 lazy-init 属性配置实现的角度进行分析。对这个属性的处理也是容器refresh 的一部分，在 finishBeanFactoryInitialization 的方法中，封装了对 lazy-init 属性的处理，实际的处理是在 DefaultListableBeanFactory 这个基本容器的preInstantiateSingletons 方法完成的。这个方法对单件 Bean 完成预实例化，这个预实例化的完成巧妙地委托给容器来实现。如果需要预实例化，那么就直接在这里采用 getBean 去触发依赖注入，与正常依赖注入的触发相比，只有触发的时间和场合不同。在这里，依赖注入发生在容器 refresh 的过程中，也就是发生在 IoC 容器初始化的过程中，而不像一般的依赖注入是发生在IoC 容器初始化完成以后，第一次向容器 getBean 时。具体的实现脉络清晰而简洁，如代码清单2-31所示。

代码清单2-31  refresh中的预实例化

1. **public** **void** refresh() **throws** BeansException,
2. IllegalStateException {
3. **synchronized** (**this**.startupShutdownMonitor) {   // Prepare
4. **this** context **for**
5. refreshing.   prepareRefresh();   // Tell the subclass to
6. refresh the internal
7. bean factory.   ConfigurableListableBeanFactory beanFactory
8. =
9. obtainFreshBeanFactory();   // Prepare the bean factory for
10. use in **this** context.
11. prepareBeanFactory(beanFactory);   **try** {    // Allows
12. post-processing of the
13. bean factory in context subclasses.
14. postProcessBeanFactory(beanFactory);
15. // Invoke factory processors registered as beans in the
16. context.
17. invokeBeanFactoryPostProcessors(beanFactory);    // Register
18. bean processors
19. that intercept bean creation.
20. registerBeanPostProcessors(beanFactory);    //
21. Initialize message source **for** **this** context.
22. initMessageSource();    //
23. Initialize event multicaster **for** **this** context.
24. initApplicationEventMulticaster();    // Initialize other
25. special beans in
26. specific context subclasses.    onRefresh();    // Check for
27. listener beans and
28. register them.    registerListeners();    // Instantiate all
29. remaining
30. (non-lazy-init) singletons.    //
31. 这里是对lazy-init属性进行处理的地方。
32. finishBeanFactoryInitialization(beanFactory);    // Last
33. step: publish
34. corresponding event.    finishRefresh();}   **catch**
35. (BeansException ex) {     //
36. Destroy already created singletons to avoid dangling
37. resources.
38. destroyBeans();    // Reset 'active' flag.
39. cancelRefresh(ex);    // Propagate
40. exception to caller.    **throw** ex;   } } }
41. //我们到finishBeanFactoryInitialization中去看一下具体的处理
42. 过程： **protected** **void**
43. finishBeanFactoryInitialization(ConfigurableListableBeanFact
44. ory beanFactory) {
45. // Stop using the temporary ClassLoader for type matching.
46. beanFactory.setTempClassLoader(**null**);  // Allow for caching
47. all bean definition
48. metadata, not expecting further changes.
49. beanFactory.freezeConfiguration();
50. // Instantiate all remaining (non-lazy-init) singletons.
51. /\*\*  \*
52. 这里调用的是BeanFactory的 preInstantiateSingletons，这个方法
53. \*是由DefaultListableBeanFactory实现的。  \*/
54. beanFactory.preInstantiateSingletons(); }
55. //在DefaultListableBeanFactory中的preInstantiateSingletons是
56. 这样的： **public** **void**
57. preInstantiateSingletons() **throws** BeansException {  **if**
58. (**this**.logger.isInfoEnabled()) {
59. **this**.logger.info("Pre-instantiating singletons
60. in " + **this**);  }
61. //在这里就开始去getBean了，也就是去触发bean的依赖注入。  /\*\*
62. \*这个getBean和在上面分析的触发依赖注入的过程是一样的，只是发
63. 生的地方不同。
64. \*如果不设置lazy-init属性，那么这个依赖注入是发生在容器初始化
65. 结束以后。第一次
66. \*向容器getBean时，如果设置了lazy-init属性，那么依赖注入发生
67. 在容器初始化的过程中
68. ，  \*会对
69. beanDefinitionMap中所有的bean进行依赖注入，这样在初始化过程
70. 结束以后，
71. \*向容器getBean得到的就是已经准备好的bean，不需要进行依赖注入
72. 。\*/  **synchronized**
73. (**this**.beanDefinitionMap) {   **for** (String beanName :
74. **this**.beanDefinitionNames) {
75. RootBeanDefinition bd =
76. getMergedLocalBeanDefinition(beanName);    **if**
77. (!bd.isAbstract() && bd.isSingleton() && !bd.isLazyInit()) {
78. **if**
79. (isFactoryBean(beanName)) {      FactoryBean factory =
80. (FactoryBean)getBean(FACTORY\_BEAN\_     PREFIX + beanName);
81. **if** (factory
82. **instanceof** SmartFactoryBean &&((SmartFactoryBean)
83. factory).isEagerInit()) {
84. getBean(beanName);      }     }     **else** {
85. getBean(beanName);     }    }
86. }  }}

根据上面的分析得知，我们可以通过lazy-init属性来对整个IoC容器的初始化和依赖注入过程做一些简单的控制。这些控制是可以由容器的使用者来决定的，具体来说，可以通过在BeanDefinition 中设置 lazy-init 属性来进行控制。这为我们使用容器提供了一定的灵活性，如果了解了这些控制原理，可以帮助我们更好地利用这些特性，希望这些小技巧能够对读者更好地使用容器提供帮助。