**2.3.2  BeanDefinition的载入和解析**

对IoC容器来说，BeanDefinition的载入过程相当于把我们定义的BeanDefinition在IoC容器中转化成一个Spring内部表示的数据结构的过程。IoC容器对Bean的管理和依赖注入功能的实现，是通过对其持有的BeanDefinition进行各种相关的操作来完成的。这些BeanDefinition数据在IoC容器里通过一个HashMap来保持和维护，当然这只是一种比较简单的维护方式，如果你觉得需要提高IoC容器的性能和容量，完全可以自己做一些扩展。我们从DefaultListableBeanFactory来入手看看IoC容器是怎样完成BeanDefinition载入的。这个DefaultListableBeanFactory已经是我们非常熟悉的基本IoC容器，在前面已经碰到过多次，相信大家对它一定不会感到陌生。为了了解这一点，我们先回到IoC容器的初始化入口，也就是到refresh()方法去看一看。这个方法的最初是在FileSystemXmlApplicationContext的构造函数中被调用的，它的调用意味着容器的初始化或数据更新，这些初始化和更新的数据当然就是BeanDefinition，如代码清单2-7所示。

代码清单2-7  启动BeanDefinition的载入

1. **public** FileSystemXmlApplicationContext(String[]
2. configLocations, **boolean** refresh,
3. ApplicationContext parent)
4. **throws** BeansException {
5. **super**(parent);
6. setConfigLocations(configLocations);
7. //这里调用容器的refresh，是载入BeanDefinition的入口。
8. **if** (refresh) {
9. refresh();
10. }
11. }

对于容器的启动来说，refresh是一个很重要的方法，我们看看它的实现。在AbstractApplicationContext类（它是FileSystemXmlApplicationContext的基类）中找到这个方法，它详细地描述了整个ApplicationContext的初始化过程，比如BeanFactory的更新，messagesource和postprocessor的注册，等等。这里看起来更像是对ApplicationContext进行初始化的模板或执行提纲，这个执行过程为IoC容器Bean的生命周期管理提供了条件。这个IoC容器的refresh过程如代码清单2-8所示。

代码清单2-8  对IoC容器的refresh的实现

1. **public** **void** refresh() **throws** BeansException,
2. IllegalStateException {
3. **synchronized** (**this**.startupShutdownMonitor) {
4. // Prepare this context for refreshing.
5. prepareRefresh();
6. // Tell the subclass to refresh the internal
7. bean factory.
8. //
9. 这里是在子类中启动refreshBeanFactory()的地方。
10. ConfigurableListableBeanFactory beanFactory
11. = obtainFreshBeanFactory();
12. // Prepare the Bean Factory for use in this
13. context.
14. prepareBeanFactory(beanFactory);
15. **try** {
16. // Allows post-processing of the
17. bean factory in context subclasses.
18. postProcessBeanFactory(beanFactory);
19. // Invoke factory processors
20. registered as beans in the context.
22. invokeBeanFactoryPostProcessors(beanFactory);
23. // Register bean processors that
24. intercept bean creation.
26. registerBeanPostProcessors(beanFactory);
27. // Initialize message source for
28. **this** context.
29. initMessageSource();
30. // Initialize event multicaster for
31. **this** context.
32. initApplicationEventMulticaster();
33. // Initialize other special beans in
34. specific context subclasses.
35. onRefresh();
36. // Check for listener beans and
37. register them.
38. registerListeners();
39. // Instantiate all remaining
40. (non-lazy-init) singletons.
42. finishBeanFactoryInitialization(beanFactory);
43. // Last step: publish corresponding
44. event.
45. finishRefresh();
46. }
47. **catch** (BeansException ex) {
48. // Destroy already created
49. singletons to avoid dangling resources.
50. destroyBeans();
51. // Reset 'active' flag.
52. cancelRefresh(ex);
53. // Propagate exception to caller.
54. **throw** ex;
55. }
56. }
57. }

我们进入到AbstractRefreshableApplicationContext的refreshBeanFactory()方法中，在这个方法里创建了BeanFactory。在创建IoC容器前，如果已经有容器存在，那么需要把已有的容器销毁和关闭，保证在refresh以后使用的是新建立起来的IoC容器。这么看来，这个refresh非常像我们对容器的重启动，就像计算机的重启动那样。在建立好当前的IoC容器以后，开始了对容器的初始化过程，比如BeanDefinition的载入，具体的实现如代码清单2-9所示。

代码清单2-9  AbstractRefreshableApplicationContext的refreshBeanFactory方法

1. **protected** **final** **void** refreshBeanFactory() **throws** BeansException {
2. **if** (hasBeanFactory()) {
3. destroyBeans();
4. closeBeanFactory();
5. }
6. **try** {
7. //创建IoC容器，这里使用的是DefaultListableBeanFactory。
8. DefaultListableBeanFactory beanFactory =
9. createBeanFactory();
10. beanFactory.setSerializationId(getId());
11. customizeBeanFactory(beanFactory);
12. //启动对BeanDefintion的载入。
13. loadBeanDefinitions(beanFactory);
14. **synchronized** (**this**.beanFactoryMonitor) {
15. **this**.beanFactory = beanFactory;
16. }
17. }
18. **catch** (IOException ex) {
19. **throw** **new** ApplicationContextException
20. ("I/O error parsing XML document
22. **for** " + getDisplayName(), ex);
23. }
24. }

这里调用的loadBeanDefinitions实际上是一个抽象方法，那么实际的载入过程是在哪里发生的呢？我们看看前面提到的loadBeanDefinitios在AbstractRefreshableApplicationContext的子类AbstractXmlApplicationContext中的实现，在这个loadBeanDefinitions中，初始化了读取器XmlBeanDefinitionReader，然后再把这个读取器在IoC容器中设置好（过程和编程式使用XmlBeanFactory是类似的），最后是启动读入器来完成BeanDefinition在IoC容器中的载入，如代码清单2-10所示。

代码清单2-10  AbstractXmlApplicationContext中的loadBeanDefinitions

1. **public** **abstract** **class** AbstractXmlApplicationContext **extends**
2. AbstractRefreshableConfigApplicationContext {
3. **public** AbstractXmlApplicationContext() {
4. }
5. **public**
6. AbstractXmlApplicationContext(ApplicationContext parent) {
7. **super**(parent);
8. }
9. //这里是实现loadBeanDefinitions的地方。
10. **protected** **void**
11. loadBeanDefinitions(DefaultListableBeanFactory beanFactory)
12. **throws** IOException {
13. // Create a new XmlBeanDefinitionReader for
14. the given BeanFactory.
15. /\*\*
17. \*创建XmlBeanDefinitionReader，并通过回调设置到
18. BeanFactory中去，
20. \*创建BeanFactory的过程可以参考上文对编程式使用IoC容器的相关
21. 分析，这里和前面一样，
22. \*使用的也是 DefaultListableBeanFactory。
23. \*/
24. XmlBeanDefinitionReader beanDefinitionReader
25. = **new** XmlBeanDefinitionReader
26. (beanFactory);
27. /\*\*
28. \*Configure the bean definition reader with
29. this context's
30. \*resource loading environment.
31. \*/
32. /\*\*
34. \*这里设置XmlBeanDefinitionReader，为XmlBeanDefinitionReader
36. \*配置ResourceLoader，因为DefaultResourceLoader是父类，所以th
37. is可以直接被使用。
38. \*/
40. beanDefinitionReader.setResourceLoader(**this**);
41. beanDefinitionReader.setEntityResolver(**new**
42. ResourceEntityResolver(**this**));
43. /\*\*
44. \*Allow a subclass to provide custom
45. initialization of the reader,
46. \*then proceed with actually loading the bean
47. definitions.
48. \*/
49. // 这是启动Bean定义信息载入的过程。
51. initBeanDefinitionReader(beanDefinitionReader);
52. loadBeanDefinitions(beanDefinitionReader);
53. }
54. **protected** **void**
55. initBeanDefinitionReader(XmlBeanDefinitionReader
56. beanDefinitionReader) {
57. }