**2.5.3  BeanPostProcessor的实现**

BeanPostProcessor 是使用 IoC 容器时经常会遇到的一个特性，这个 Bean 的后置处理器是一个监听器，它可以监听容器触发的事件。把它向 IoC 容器注册以后，使得容器中管理的 Bean具备接收 IoC 容器事件回调的能力。BeanPostProcessor 的使用非常简单，只需要设计一个具体的后置处理器来实现。同时，这个具体的后置处理器需要实现接口类 BeanPostProcessor，然后设置到 XML 的 Bean 配置文件中。这个 BeanPostProcessor 是一个接口类，它有两个接口方法，一个是 postProcessBeforeInitialization，为在 Bean 的初始化前提供回调入口；一个是postProcessAfterInitialization，为在Bean的初始化以后提供回调入口，这两个回调的触发都是和容器管理 Bean 的生命周期相关的。这两个回调方法的参数都是一样的，分别是 Bean 的实例化对象和 Bean 的名字，为具体的处理提供基本的回调输入，如代码清单 2-33所示。

代码清单2-33  BeanPostProcessor接口定义

1. **public** **interface** BeanPostProcessor { /\*\*  \* Apply this
2. BeanPostProcessor to the given new bean instance
3. <i>before</i> any   \* bean initialization callbacks (like
4. InitializingBean's  \* <code>afterPropertiesSet</code> or a
5. custom init-method).The bean will already be  \* populated
6. with property values. The returned bean instance may be a
7. wrapper  \* around the original.  \* / Object
8. postProcessBeforeInitialization(Object bean, String
9. beanName) throws BeansException; /\*\*  \* Apply this
10. BeanPostProcessor to the given new bean instance
11. <i>after</i> any bean  \* initialization callbacks (like
12. InitializingBean's <code>  \* afterPropertiesSet</code> or a
13. custom init-method). The bean will already be   \* populated
14. with property values. The returned bean instance may be a
15. wrapper  \* around the original.  \* <p>In case of a
16. FactoryBean, this callback will be invoked for both the
17. FactoryBean  \* instance and the objects created by the
18. FactoryBean (as of Spring 2.0). The  \* post-processor can
19. decide whether to apply to either the FactoryBean or  \*
20. created objects or both through corresponding <code>bean
21. instanceof FactoryBean  \* </code> checks.<p>This callback
22. will also be invoked after a short-circuiting triggered by a
23. \* {@link
24. InstantiationAwareBeanPostProcessor#postProcessBeforeInstant
25. iation} method,  \* in contrast to all other
26. BeanPostProcessor callbacks.  \*/  Object
27. postProcessAfterInitialization(Object bean, String beanName)
28. **throws** BeansException; }

对于这些接口是在什么地方与 IoC 结合在一起的，可以看一下这个方法的调用关系，如图 2-11所示。

|  |
| --- |
| <http://images.51cto.com/files/uploadimg/20091222/094252722.jpg> |
|  |

postProcessBeforeInitialization 是在 populateBean 完成之后被调用的。我们从BeanPostProcessor中的一个回调接口入手，对另一个回调接口postProcessAfter- Initialization 方法的调用，实际上也是在同一个地方封装完成的，这个地方就是populateBean 方法中的 initializeBean 调用。关于这一点，读者会在接下来的分析中看得很清楚。在前面对 IoC 的依赖注入进行分析时，对这个 populateBean 有过分析，这个方法实际上完成了 Bean 的依赖注入，在容器中建立 Bean 的依赖关系，是容器功能实现的一个很重要的部分。节选 doCreateBean 中的代码就可以看到 postProcessBeforeInitialization 调用和populateBean调用的关系，如下所示。

1. Object exposedObject = bean; **try** {  populateBean(beanName,
2. mbd, instanceWrapper);  /\*\*
3. \*在对Bean的生成和依赖注入完成以后，开始对Bean进行初始化，这
4. 个初始化过程
5. \*包含了对后置处理器的postProcessBeforeInitialization回调。
6. \*/  exposedObject = initializeBean(beanName, exposedObject,
7. mbd); }

具体的初始化过程也是 IoC 容器完成依赖注入的一个重要部分。在 initializeBean 方法里，需要使用 Bean 的名字，完成依赖注入以后的 Bean 对象，以及这个 Bean 对应的BeanDefinition。在这些输入的帮助下，完成 Bean 的初始化工作，这些工作包括为类型是BeanNameAware的Bean设置Bean的名字，为类型是BeanClassLoaderAware的Bean设置类装载器，为类型是BeanFactoryAware的Bean设置其自身所在的IoC容器以供回调使用。当然，还有对 postProcessBeforeInitialization/postProcessAfterInitialization 的回调和初始化属性init-method 的处理等。经过这一系列的初始化处理之后，得到的结果就是我们可以正常使用的由IoC容器托管的Bean了。具体的实现过程如代码清单2-34所示。

代码清单2-34  IoC容器对Bean的初始化

1. /\*\*  \* Initialize the given bean instance, applying factory
2. callbacks  \* as well as init methods and bean post
3. processors.  \* <p>Called from {@link #createBean} for
4. traditionally defined beans,  \* and from {@link
5. #initializeBean} for existing bean instances.  \*/ **protected**
6. Object initializeBean(String beanName, Object bean,
7. RootBeanDefinition mbd) {  **if** (bean **instanceof**
8. BeanNameAware) {   ((BeanNameAware)
9. bean).setBeanName(beanName);  }  **if** (bean **instanceof**
10. BeanClassLoaderAware) {   ((BeanClassLoaderAware)
11. bean).setBeanClassLoader(getBeanClassLoader());  }  **if** (bean
12. **instanceof** BeanFactoryAware) {   ((BeanFactoryAware)
13. bean).setBeanFactory(**this**);  }  /\*\*
14. \*这里是对后置处理器BeanPostProcessors的postProcessBeforeInit
15. ialization  \*的回调方法调用。  \*/  Object wrappedBean =
16. bean;  **if** (mbd == **null** || !mbd.isSynthetic()) {
17. wrappedBean =
18. applyBeanPostProcessorsBeforeInitialization(wrappedBean,
19. beanName);  }  /\*\*
20. \*调用Bean的初始化方法，这个初始化方法是在BeanDefinition中通
21. 过定义init-method属性指定的。
22. \*同时，如果Bean实现了InitializingBean接口，那么Bean的afterPr
23. opertiesSet实现也会被调用。  \*/  **try** {
24. invokeInitMethods(beanName, wrappedBean, mbd);  }  **catch**
25. (Throwable ex) {   **throw** **new** BeanCreationException(     (mbd
26. != **null** ? mbd.getResourceDescription() : **null**),
27. beanName, "Invocation of init method failed", ex);  }  /\*\*
28. \*这里是对后置处理器BeanPostProcessors的postProcessAfterIniti
29. alization  \*的回调方法调用。  \*/  **if** (mbd == **null** ||
30. !mbd.isSynthetic()) {   wrappedBean =
31. applyBeanPostProcessorsAfterInitialization(wrappedBean,
32. beanName);  }  **return** wrappedBean; } /\*\*
33. \*对设置好的BeanPostProcessors的postProcessBeforeInitializati
34. on回调进行依次调用。 \*/ **public** Object
35. applyBeanPostProcessorsBeforeInitialization(Object
36. existingBean, String beanName)   **throws** BeansException {
37. Object result = existingBean;  **for** (BeanPostProcessor
38. beanProcessor : getBeanPostProcessors())  result =
39. beanProcessor.postProcessBeforeInitialization(result,
40. beanName); } **return** result;  \*\*
41. 对设置好的BeanPostProcessors的postProcessAfterInitialization
42. 回调进行依次调用。 / ublic Object
43. applyBeanPostProcessorsAfterInitialization(Object
44. existingBean, String beanName)  **throws** BeansException {
45. Object result = existingBean; **for** (BeanPostProcessor
46. beanProcessor : getBeanPostProcessors()) {   result =
47. beanProcessor.postProcessAfterInitialization(result,
48. beanName); } **return** result;

从以上的代码实现中可以看到，这两个Bean后置处理器定义的接口方法，围绕着Bean定义的 init-method 方法调用，与 IoC 容器对 Bean 的管理有机地结合起来了。对于这个特性的理解，离不开对 IoC 容器基本实现原理的理解。了解了 Bean 后置处理器的实现原理后，就能更灵活地使用它。这些 IoC 容器的附加特性还有很多，它们代表了容器的一些特色和高级的使用技巧，掌握这些特性对应用开发有很大的帮助。下面就分析一下容器的 autowiring 特性是怎样实现的。