**2.5.4  autowiring的实现原理**

在前面对IoC容器实现原理的分析中，一直是通过BeanDefinition的属性值和构造函数以显式的方式实现 Bean 的依赖关系管理的。在 Spring中，相对这种显式的依赖管理方式，IoC 容器还提供了自动依赖装配的方式，为应用使用容器提供更大的方便。在自动装配中，不需要对 Bean属性做显式的依赖关系声明，只需要配置好 autowire（自动依赖装配）属性，IoC 容器会根据这个属性的配置，使用反射自动地查找属性的类型或名字，然后基于属性的类型或名字来自动匹配IoC容器中的Bean，从而自动地完成依赖注入。

这是一个很有诱惑力的功能特性，使用它可以完成依赖关系管理的自动化，但是使用时一定要注意，计算机只是在自动地执行，它是不会思考的。使用这个特性的优点是能够减少用户配置Bean 的工作量，但如果使用不当，也会为应用带来不可预见的后果，就像我们使用其他方式的自动化一样。所以，使用起来需要多一些小心和谨慎。

从autowiring使用上可以知道，这个autowiring属性是在对Bean属性进行依赖注入时起作用的。对 Bean 属性依赖注入的实现原理，我们在前面已经做过分析。回顾那部分的内容，我们不难发现，对 autowirng 属性的处理，从而完成对 Bean 的属性的自动依赖装配。节选AbstractAutowireCapableBeanFactory的populateBean方法中与autowiring实现相关的部分，我们可以清楚地看到这个特性在容器中实现的入口。也就是说，对属性 autowire 的处理是populateBean处理过程的一个部分。在populateBean的实现中，在处理一般的Bean之前，先对 autowiring 属性进行处理。如果当前的 Bean 配置了 autowire\_by\_name 和autowire\_by\_type 属性，那么调用相应的 autowireByName 方法和 autowireByType 方法。这两个方法很巧妙地应用了 IoC 容器的特性。例如，对于 autowire\_by\_name，它首先通过反射机制从当前 Bean 中得到需要注入的属性名，然后使用这个属性名向容器去申请与之同名的Bean，这样实际又触发了另一个Bean的生成和依赖注入的过程。实现过程如代码清单2-35所示。

代码清单2-35  populateBean对autowire的处理

1. //开始进行依赖注入过程，先处理autowire的注入。 if
2. (mbd.getResolvedAutowireMode() ==
3. RootBeanDefinition.AUTOWIRE\_BY\_NAME ||
4. mbd.getResolvedAutowireMode() ==
5. RootBeanDefinition.AUTOWIRE\_BY\_TYPE) {
6. MutablePropertyValues newPvs = **new**
7. MutablePropertyValues(pvs);  // Add property values based on
8. autowire by name **if** applicable.  //
9. 这里是对autowire注入的处理，根据bean的名字或者type进行autowi
10. re的过程。  **if** (mbd.getResolvedAutowireMode() ==
11. RootBeanDefinition.AUTOWIRE\_BY\_NAME) {
12. autowireByName(beanName, mbd, bw, newPvs);  }  // Add
13. property values based on autowire by type **if** applicable.  **if**
14. (mbd.getResolvedAutowireMode() ==
15. RootBeanDefinition.AUTOWIRE\_BY\_TYPE) {
16. autowireByType(beanName, mbd, bw, newPvs);  }  pvs = newPvs;
17. }

在做了一些简单的对 autowiring 类型的逻辑判断以后，通过调用 autowireByName 和autowireByType 来完成自动依赖装配。以 autowireByName 为例来看看容器的自动依赖装配功能是怎样实现的；对 autowireByName 来说，它首先需要得到当前 Bean 的属性名，这些属性名已经在BeanWrapper和BeanDefinition中封装好了，然后就是对着一系列属性名进行匹配的过程。在这个匹配的过程中，因为已经有了属性的名字，那就可以直接使用这个属性名字作为Bean名字向容器getBean，这个getBean会触发当前Bean的依赖Bean的依赖注入的过程；从而得到属性对应的依赖Bean。在这个getBean完成后，把这个依赖Bean注入到当前Bean的属性中去；这样就完成了这个依赖属性名自动完成依赖注入的过程。对 autowireByType 的实现，和 autowireByName 的实现过程是非常类似的，感兴趣的读者可以自己进行分析。这些autowiring的实现如代码清单2-36所示。

代码清单2-36  autowire\_by\_name的实现

1. **protected** **void** autowireByName(  String beanName,
2. AbstractBeanDefinition mbd, BeanWrapper bw, MutableProperty
3. Values pvs) {  String[] propertyNames =
4. unsatisfiedNonSimpleProperties(mbd, bw);  **for** (String
5. propertyName : propertyNames) {   **if**
6. (containsBean(propertyName)) {    /\*\*
7. \*使用取得的当前Bean的属性名作为Bean的名字，向IoC容器getBean
8. ，    \*然后把从容器得到的Bean设置到当前Bean的属性中去。
9. \*/    Object bean = getBean(propertyName);
10. pvs.addPropertyValue(propertyName, bean);
11. registerDependentBean(propertyName, beanName);    **if**
12. (logger.isDebugEnabled()) {     logger.debug(       "Added
13. autowiring by name from bean name '" + beanName      + "'
14. via property '" + propertyName +       "' to bean named '" +
15. propertyName + "'");    }  }   **else** {    **if**
16. (logger.isTraceEnabled()) {     logger.trace("Not autowiring
17. property '" + propertyName + "' of    bean '" + beanName +
18. "' by name: no matching bean found");    }   }  } }