关于这个依赖注入的详细过程我们会在下面进行分析。接着看 createBean 中的实现，在这个过程中，Bean对象会依据BeanDefinition定义的要求生成出来。在AbstractAutowire CapableBeanFactory中实现了这个createBean，createBean不但生成了需要的Bean，还对 Bean 初始化进行了处理，比如实现了在 BeanDefinition 中的 init-method 属性定义，Bean后置处理器的实现，等等。具体的过程如代码清单2-23所示。

代码清单2-23  AbstractAutowireCapableBeanFactory中的createBean

1. **protected** Object createBean(**final** String beanName, **final**
2. RootBeanDefinition  mbd, **final** Object[] args)   **throws**
3. BeanCreationException {  AccessControlContext acc =
4. AccessController.getContext();  **return**
5. AccessController.doPrivileged(**new** PrivilegedAction<Object>()
6. {   **public** Object run() {    **if** (logger.isDebugEnabled()) {
7. logger.debug("Creating instance of bean '" + beanName +
8. "'");    }    // Make sure bean class is actually resolved
9. at **this** point.    //
10. 这里判断需要创建的bean的是否可以实例化，这个类是否可以通过类
11. 装载器来载入。    resolveBeanClass(mbd, beanName);     //
12. Prepare method overrides.    **try** {
13. mbd.prepareMethodOverrides();    }    **catch**
14. (BeanDefinitionValidationException ex) {     **throw** **new**
15. BeanDefinitionStoreException(mbd.getResourceDescription(),
16. beanName, "Validation of method overrides failed", ex);    }
17. **try** {     /\*\*     \* Give BeanPostProcessors a chance to
18. return a proxy     \* instead of thetarget bean instance.
19. \*/     //
20. 如果bean配置了PostProcessor，那么这里返回的是一个proxy。
21. Object bean = resolveBeforeInstantiation(beaif (bean !=
22. **null**) {       **return** bean;     }    }    **catch** (Throwable
23. ex) {     **throw** **new**
24. BeanCreationException(mbd.getResourceDescription(),
25. beanName,       "BeanPostProcessor before instantiation of
26. bean failed", ex);    }    //这里是创建bean的调用。
27. Object beanInstance = doCreateBean(beanName, mbd, args);
28. **if** (logger.isDebugEnabled()) {     logger.debug("Finished
29. creating instance of bean '" + beanName + "'");    }
30. **return** beanInstance;   }  }, acc); }
31. //我们接着到doCreateBean中去看看bean是怎样生成的：
32. **protected** Object doCreateBean(**final** String beanName, **final**
33. RootBean Definitionmbd, **final** Object[] args) {  //
34. Instantiate the bean.  //
35. 这个BeanWrapper是用来持有创建出来的bean对象的。  BeanWrapper
36. instanceWrapper = **null**;  //
37. 如果是singleton，先把缓存中的同名bean清除。  **if**
38. (mbd.isSingleton()) {   instanceWrapper =
39. **this**.factoryBeanInstanceCache.remove(beanName);  }
40. //这里是创建bean的地方，由createBeanInstance来完成。  if
41. (instanceWrapper == **null**) {   instanceWrapper =
42. createBeanInstance(beanName, mbd, args);  }  **final** Object
43. bean = (instanceWrapper != **null** ? instanceWrapper.get
44. Wrapped Instance() : **null**);  Class beanType =
45. (instanceWrapper != **null** ? instanceWrapper.getWrapped
46. Class() : **null**);   // Allow post-processors to modify the
47. merged bean definition.  **synchronized**
48. (mbd.postProcessingLock) {   **if** (!mbd.postProcessed) {
49. applyMergedBeanDefinitionPostProcessors(mbd, beanType,
50. beanName);    mbd.postProcessed = **true**;   }  }  /\*\*  \*
51. Eagerly cache singletons to be able to resolve circular
52. references  \* even when triggered by lifecycle interfaces
53. like BeanFactoryAware.  \*/  **boolean** earlySingletonExposure =
54. (mbd.isSingleton() && **this**.allowCircular References &&
55. isSingletonCurrentlyInCreation(beanName)); **if**
56. (earlySingletonExposure) {   **if** (logger.isDebugEnabled()) {
57. logger.debug("Eagerly caching bean '" + beanName +      "'
58. to allow **for** resolving potential circular references");   }
59. addSingletonFactory(beanName, **new** ObjectFactory() {
60. **public** Object getObject() **throws** BeansException {     **return**
61. getEarlyBeanReference(beanName, mbd, bean);    }   });  }
62. // Initialize the bean instance.  /\*\*  \*
63. 这里是对bean的初始化，依赖注入往往在这里发生，这个exposedObj
64. ect在初始化处理完以后会  \* 返回作为依赖注入完成后的bean。
65. \*/  Object exposedObject = bean;  **try** {
66. populateBean(beanName, mbd, instanceWrapper);
67. exposedObject = initializeBean(beanName, exposedObject,
68. mbd);  }  **catch** (Throwable ex) {   **if** (ex **instanceof**
69. BeanCreationException && beanName.equals(((BeanCreation
70. Exception) ex).getBeanName())) {    **throw**
71. (BeanCreationException) ex;   }   **else** { **throw** **new**
72. BeanCreationException(mbd.getResourceDescription(),beanName,
73. "Initialization of bean failed", ex);   }  }  **if**
74. (earlySingletonExposure) {   Object earlySingletonReference
75. = getSingleton(beanName, **false**);   **if**
76. (earlySingletonReference != **null**) {    **if** (exposedObject ==
77. bean) {     exposedObject = earlySingletonReference;    }
78. **else** **if** (!**this**.allowRawInjectionDespiteWrapping &&
79. hasDependentBean (beanName)) {     String[] dependentBeans =
80. getDependentBeans(beanName);     Set<String>
81. actualDependentBeans = **new** LinkedHashSet<String>
82. (dependentBeans.length);     **for** (String dependentBean :
83. dependentBeans) {      **if**
84. (!removeSingletonIfCreatedForTypeCheckOnly(dependentBean)) {
85. actualDependentBeans.add(dependentBean);      }     }     **if**
86. (!actualDependentBeans.isEmpty()) {      **throw** **new**
87. BeanCurrentlyInCreationException(beanName,        "Bean with
88. name '" + beanName + "' has been injected      into other
89. beans [" +
90. StringUtils.collectionToCommaDelimitedString
91. (actualDependentBeans) + "] in its raw version as part
92. of a circular reference, but has eventually been " +
93. "wrapped. This means that said other beans **do** not       use
94. the **final** version of the " +       "bean. This is often the
95. result of over-eager type       matching -
96. "'getBeanNamesOfType' with the 'allowEagerInit'       flag
97. turned off, **for** example.");     }    }   }  }  // Register
98. bean as disposable.  **try** {
99. registerDisposableBeanIfNecessary(beanName, bean, mbd);  }
100. **catch** (BeanDefinitionValidationException ex) {   **throw** **new**
101. BeanCreationException(mbd.getResourceDescription(),
102. beanName,  "Invalid destruction signature", ex);  }  **return**
103. exposedObject; }

这里我们看到与依赖注入关系特别密切的方法有 createBeanInstance 和 populateBean，下面分别到这两个方法里看看发生了什么。在 createBeanInstance 中生成了 Bean 所包含的Java 对象，这个对象的生成有很多种不同的方式，可以通过工厂方法生成，也可以通过容器的autowire特性生成，这些生成方式都是由相关的BeanDefinition来指定的。如代码清单2-24 所示，可以看到不同生成方式对应的实现。

代码清单2-24  Bean包含的Java对象的生成

1. **protected** BeanWrapper createBeanInstance(String beanName,
2. RootBeanDefinition mbd, Object[] args) {  // Make sure bean
3. **class** is actually resolved at **this** point.  //
4. 确认需要创建的bean实例的类可以实例化。  Class beanClass =
5. resolveBeanClass(mbd, beanName);
6. //这里使用工厂方法对bean进行实例化。  if
7. (mbd.getFactoryMethodName() != **null**)  {   **return**
8. instantiateUsingFactoryMethod(beanName, mbd, args);  }  //
9. Shortcut when re-creating the same bean...  **if**
10. (mbd.resolvedConstructorOrFactoryMethod != **null**) {   **if**
11. (mbd.constructorArgumentsResolved) {    **return**
12. autowireConstructor(beanName, mbd, **null**, args);   }   **else** {
13. **return** instantiateBean(beanName, mbd);   }  }  // Need to
14. determine the constructor...  // 使用构造函数进行实例化。
15. Constructor[] ctors =
16. determineConstructorsFromBeanPostProcessors(beanClass,
17. beanName);  **if** (ctors != **null** ||
18. mbd.getResolvedAutowireMode() ==
19. RootBeanDefinition.AUTOWIRE\_   CONSTRUCTOR ||
20. mbd.hasConstructorArgumentValues() ||
21. !ObjectUtils.isEmpty(args))  {   **return**
22. autowireConstructor(beanName, mbd, ctors, args);  }  // No
23. special handling: simply use no-arg constructor.  //
24. 使用默认的构造函数对bean进行实例化。  **return**
25. instantiateBean(beanName, mbd); }
26. //我们看看最常见的实例化过程instantiateBean：protected
27. BeanWrapper instantiateBean(String beanName,
28. RootBeanDefinition mbd) { /\*\*
29. \*使用默认的实例化的策略对bean进行实例化，默认的实例化策略是C
30. glibSubclassingInstantiation
31. \*Strategy，也就是用cglib来对bean进行实例化。 \*/
32. //我们接着回去看看CglibSubclassingInstantiationStrategy的实
33. 现。  **try** {    Object beanInstance =
34. getInstantiationStrategy().instantiate(mbd, beanName, **this**);
35. BeanWrapper bw = **new** BeanWrapperImpl(beanInstance);
36. initBeanWrapper(bw);   **return** bw;  }  **catch** (Throwable ex) {
37. **throw** **new**
38. BeanCreationException(mbd.getResourceDescription(),
39. beanName,  "Instantiation of bean failed", ex);  } }

在这里用到了cglib 对Bean 进行实例化，cglib 是一个常用的字节码生成器的类库，它提供了一系列的 API 来提供 Java 的字节码生成和转换的功能。在 Spring AOP 中也是使用了 cglib来对Java的字节码进行了增强。我们看看在IoC容器里是怎样使用cglib来生成Bean对象的，到类SimpleInstantiationStrategy中去看一下，这个Strategy是Spring用来生成Bean对象的默认类。它提供了两种实例化Java对象的方法，一种是通过BeanUtils，它使用了JDK的反射功能，一种是通过cglib来生成的，如代码清单2-25所示。

代码清单2-25  使用SimpleInstantiationStrategy生成Java对象

1. **public** **class** SimpleInstantiationStrategy **implements**
2. InstantiationStrategy { **public** Object instantiate(
3. RootBeanDefinition beanDefinition, String beanName,
4. BeanFactory owner) {  // Don't override the class with CGLIB
5. **if** no overrides.  **if**
6. (beanDefinition.getMethodOverrides().isEmpty()) {
7. //这里取得指定的构造器或者生成对象的工厂方法来对bean进行实例
8. 化。   Constructor constructorToUse =
9. (Constructor)beanDefinition. resolvedConstructor
10. OrFactoryMethod;   **if** (constructorToUse == **null**) {    Class
11. clazz = beanDefinition.getBeanClass();    **if**
12. (clazz.isInterface()) {     **throw** **new**
13. BeanInstantiationException(clazz, "Specified **class** is    an
14. **interface**");    }    **try** {     constructorToUse =
15. clazz.getDeclaredConstructor((Class[]) **null**);
16. beanDefinition.resolvedConstructorOrFactoryMethod
17. =constructorToUse;    }    **catch** (Exception ex) {     **throw**
18. **new** BeanInstantiationException(clazz, "No **default**
19. constructor found", ex);    }   }  /\*\*
20. \*通过BeanUtils进行实例化，这个BeanUtils的实例化通过Construct
21. or来实例化bean,
22. \*在BeanUtils中可以看到具体的调用ctor.newInstance(args)。  \*/
23. **return** BeanUtils.instantiateClass(constructorToUse, **null**);
24. }  **else** {    // Must generate CGLIB
25. subc//使用CGLIB来实例化对象。   return
26. instantiateWithMethodInjection(beanDefinition, beanName,
27. owner);  } }