在 cglibSubclassingInstantiationStrategy 中我们可以看到具体的实例化过程和cglib 的使用方法，这里就不对 cglib 的使用进行过多的阐述了。如果读者有兴趣，可以去阅读cglib 的使用文档，不过这里的 Spring 代码可以为使用 cglib 提供很好的参考。这里的Enhancer 类，已经是 cglib 的类了，通过这个 Enhancer 来完成 Java 对象的生成，使用的是Enhancer的create方法。如代码清单2-26所示。

代码清单2-26  使用cglib的Enhancer生成Java对象

1. **public** Object instantiate(Constructor ctor, Object[] args) {
2. Enhancer enhancer = **new** Enhancer();
3. enhancer.setSuperclass(**this**.beanDefinition.getBeanClass());
4. enhancer.setCallbackFilter(**new** CallbackFilterImpl());
5. enhancer.setCallbacks(**new** Callback[] {    NoOp.INSTANCE,
6. **new** LookupOverrideMethodInterceptor(),    **new**
7. ReplaceOverrideMethodInterceptor()  });
8. //使用cglib的create生成实例化的bean对象。  return (ctor ==
9. **null**) ?     enhancer.create() :
10. enhancer.create(ctor.getParameterTypes(), args); }

在实例化 Bean 对象生成的基础上，我们看看 Spring 是怎样对这些对象进行处理的，也就是Bean 对象生成以后，怎样把这些 Bean 对象的依赖关系设置好，完成整个依赖注入过程。这里涉及对各种 Bean 对象的属性的处理过程（即依赖关系处理的过程），这些依赖关系处理的依据就是已经解析得到的 BeanDefinition。详细地了解这个过程，需要回到前面的 populateBean 方法，这个方法在AbstractAutowireCapableBeanFactory中的实现如代码清单2-27所示。

代码清单2-27  populateBean的实现

1. **protected** **void** populateBean(String beanName,
2. AbstractBeanDefinition mbd, BeanWrapper bw) {  /\*\*
3. \*这里取得在BeanDefinition中设置的property值，这些property来
4. 自对  \*BeanDefinition的解析。  \*/
5. //具体的解析过程可以参看载入和解析BeanDefinition的分析。
6. PropertyValues pvs = mbd.getPropertyValues();  **if** (bw ==
7. **null**) {   **if** (!pvs.isEmpty()) {    **throw** **new**
8. BeanCreationException(      mbd.getResourceDescription(),
9. beanName, "Cannot apply property     values to **null**
10. instance");   }   **else** {    // Skip property population
11. phase **for** **null** instance.    **return**;   }  }  /\*\*  \* Give any
12. InstantiationAwareBeanPostProcessors the opportunity to
13. modify the  \* state of the bean before properties are set.
14. This can be used, for example,  \* to support styles of field
15. injection.  \*/  **boolean** continueWithPropertyPopulation =
16. **true**;  **if** (!mbd.isSynthetic() &&
17. hasInstantiationAwareBeanPostProcessors()) {   **for**
18. (BeanPostProcessor bp : getBeanPostProcessors()) {    **if** (bp
19. **instanceof** InstantiationAwareBeanPostProcessor) {
20. InstantiationAwareBeanPostProcessor ibp =
21. (InstantiationAware      BeanPostProcessor) bp; **if**
22. (!ibp.postProcessAfterInstantiation(bw.getWrappedInstance(),
23. beanName)) {      continueWithPropertyPopulation = **false**;
24. **break**;     }    }   }  }  **if**
25. (!continueWithPropertyPopulation) {   **return**;  }
26. //开始进行依赖注入过程，先处理autowire的注入。  if
27. (mbd.getResolvedAutowireMode() ==
28. RootBeanDefinition.AUTOWIRE\_BY\_NAME ||
29. mbd.getResolvedAutowireMode() ==
30. RootBeanDefinition.AUTOWIRE\_BY\_TYPE) {
31. MutablePropertyValues newPvs = **new**
32. MutablePropertyValues(pvs);   // Add property values based
33. on autowire by name **if** applicable. //
34. 这里是对autowire注入的处理，根据bean的名字或者。type进行auto
35. wire的过程。   **if** (mbd.getResolvedAutowireMode() ==
36. RootBeanDefinition.AUTOWIRE\_BY\_NAME) {
37. autowireByName(beanName, mbd, bw, newPvs);   }   // Add
38. property values based on autowire by type **if** applicable.
39. **if** (mbd.getResolvedAutowireMode() ==
40. RootBeanDefinition.AUTOWIRE\_BY\_TYPE) {
41. autowireByType(beanName, mbd, bw, newPvs);   }   pvs =
42. newPvs;  }  **boolean** hasInstAwareBpps =
43. hasInstantiationAwareBeanPostProcessors();  **boolean**
44. needsDepCheck = (mbd.getDependencyCheck() !=
45. RootBeanDefinition.   DEPENDENCY\_CHECK\_NONE);  **if**
46. (hasInstAwareBpps || needsDepCheck) {   PropertyDescriptor[]
47. filteredPds = filterPropertyDescriptorsForDependency
48. Check(bw);   **if** (hasInstAwareBpps) {    **for**
49. (BeanPostProcessor bp : getBeanPostProcessors()) {     **if**
50. (bp **instanceof** InstantiationAwareBeanPostProcessor) {
51. InstantiationAwareBeanPostProcessor ibp =
52. (InstantiationAware       BeanPostProcessor) bp;      pvs =
53. ibp.postProcessPropertyValues(pvs, filteredPds,
54. bw.getWrappedInstance(), beanName);      **if** (pvs == **null**) {
55. **return**;      }     }    } }   **if** (needsDepCheck) {
56. checkDependencies(beanName, mbd, filteredPds, pvs);   }  }
57. //对属性进行注入。  applyPropertyValues(beanName, mbd, bw,
58. pvs); }
59. //我们到applyPropertyValues中去看看具体的对属性进行解析然后
60. 注入的过程： **protected** **void** applyPropertyValues(String
61. beanName, BeanDefinition mbd, BeanWrapper bw, PropertyValues
62. pvs) {  **if** (pvs == **null** || pvs.isEmpty()) {   **return**;  }
63. MutablePropertyValues mpvs = **null**;  List<PropertyValue>
64. original;  **if** (pvs **instanceof** MutablePropertyValues) {
65. mpvs = (MutablePropertyValues) pvs;   **if**
66. (mpvs.isConverted()) {    // Shortcut: use the pre-converted
67. values as-is.    **try** {     bw.setPropertyValues(mpvs);
68. **return**;    }    **catch** (BeansException ex) {     **throw** **new**
69. BeanCreationException(       mbd.getResourceDescription(),
70. beanName, "Error setting      property values", ex);    }
71. }   original = mpvs.getPropertyValueLi }  **else** {   original
72. = Arrays.asList(pvs.getPropertyValues());  }  TypeConverter
73. converter = getCustomTypeConverter();  **if** (converter ==
74. **null**) {   converter = bw;  }  /\*\*
75. \*注意这个BeanDefinitionValueResolver对BeanDefinition的解析
76. \*是在这个valueResolver中完成的。  \*/
77. BeanDefinitionValueResolver valueResolver = **new**
78. BeanDefinitionValueResolver (**this**, beanName, mbd,
79. converter);  // Create a deep copy, resolving any references
80. **for** values.  //
81. 这里为解析值创建一个拷贝，拷贝的数据将会被注入到bean中。
82. List<PropertyValue> deepCopy = **new**
83. ArrayList<PropertyValue>(original.size());  **boolean**
84. resolveNecessary = **false**;  **for** (PropertyValue pv : original)
85. {   **if** (pv.isConverted()) {    deepCopy.add(pv);   }   **else**
86. {    String propertyName = pv.getName();    Object
87. originalValue = pv.getValue();    Object resolvedValue =
88. valueResolver.resolveValueIfNecessary(pv,
89. originalValue);    Object convertedValue = resolvedValue;
90. **boolean** convertible = bw.isWritableProperty(propertyName) &&
91. !PropertyAccessorUtils.isNestedOrIndexedProperty(propertyNam
92. e);    **if** (convertible) {      convertedValue =
93. convertForProperty(resolvedValue, propertyName,         bw,
94. converter);    }    /\*\*    \* Possibly store converted value
95. in merged bean definition,    \* in order to avoid
96. re-conversion for every created bean instance.    \*/    **if**
97. (resolvedValue == originalValue) {     **if** (convertible) {
98. pv.setConvertedValue(convertedValue);     }
99. deepCopy.add(pv);    }    **else** **if** (originalValue **instanceof**
100. TypedStringValue && convertible &&      !(convertedValue
101. **instanceof** Collection ||
102. ObjectUtils.isArray(convertedValue))) {
103. pv.setConvertedValue(convertedValue);      deepCopy.add(pv);
104. }    **else** {     resolveNecessary = **true**;
105. deepCopy.add(**new** PropertyValue(pv, convertedValue));    }
106. }  }  **if** (mpvs != **null** && !resolveNecessary) {
107. mpvs.setConverted();  }  //
108. 这里是依赖注入发生的地方，会在BeanWrapperImpl中完成。  **try** {
109. bw.setPropertyValues(**new** MutablePropertyValues(deepCopy));
110. }  **catch** (BeansException ex) {   **throw** **new**
111. BeanCreationException(     mbd.getResourceDescription(),
112. beanName, "Error setting property    values", ex);  } }

这里通过使用 BeanDefinitionResolver 来对 BeanDefinition 进行解析，然后注入到property 中。下面到 BeanDefinitionValueResolver 中去看一下解析过程的实现，我们举对 Bean reference 进行 Resolve 的例子，如图 2-9 所示，可以看到整个 Resolve 的过程，具体地对Bean reference进行Resolve的过程如代码清单2-28所示。

|  |
| --- |
| <http://images.51cto.com/files/uploadimg/20091222/091814558.jpg> |
|  |