代码清单2-30

1. **private** **void** setPropertyValue(PropertyTokenHolder tokens,
2. PropertyValue pv) **throws** BeansException {  String
3. propertyName = tokens.canonicalName;  String actualName =
4. tokens.actualName;  **if** (tokens.keys != **null**) {    // Apply
5. indexes and map keys: fetch value **for** all keys but the last
6. one.   PropertyTokenHolder getterTokens = **new**
7. PropertyTokenHolder();   getterTokens.canonicalName =
8. tokens.canonicalName;   getterTokens.actualName =
9. tokens.actualName;   getterTokens.keys = **new**
10. String[tokens.keys.length - 1];
11. System.arraycopy(tokens.keys, 0, getterTokens.keys, 0,
12. tokens.keys. length - 1);   Object propValue;
13. //getPropertyValue取得bean中注入对象的引用，比如Array、List
14. 、Map、Set等。   **try** {    propValue =
15. getPropertyValue(getterTokens);   }   **catch**
16. (NotReadablePropertyException ex) {    **throw** **new**
17. NotWritablePropertyException(getRootClass(), **this**.nested
18. Path + propertyName,      "Cannot access indexed value in
19. property referenced " +      "in indexed property path '" +
20. propertyName + "'", ex);   }   // Set value for last key.
21. String key = tokens.keys[tokens.keys.length - 1];   **if**
22. (propValue == **null**) {    **throw** **new**
23. NullValueInNestedPathException(getRootClass(), **this**.nested
24. Path + propertyName,      "Cannot access indexed value in
25. property referenced " +      "in indexed property path '" +
26. propertyName + "': returned null");   }
27. //这里对Array进行注入。   else if
28. (propValue.getClass().isArray()) {    Class requiredType =
29. propValue.getClass().getComponentType();    **int** arrayIndex =
30. Integer.parseInt(key);    Object oldValue = **null**;    **try** {
31. **if** (isExtractOldValueForEditor()) {      oldValue =
32. Array.get(propValue, arrayIndex);     }     Object
33. convertedValue =    **this**.typeConverterDelegate.
34. convertIfNecessary(  propertyName, oldValue, pv.getValue(),
35. requiredType);     Array.set(propValue,
36. Integer.parseInt(key), convertedValue);    }    **catch**
37. (IllegalArgumentException ex) {     PropertyChangeEvent pce
38. =       **new**
39. PropertyChangeEvent(**this**.rootObject,**this**.nestedPath      +
40. propertyName, oldValue, pv.getValue());     **throw** **new**
41. TypeMismatchException(pce, requiredType, ex);    }    **catch**
42. (IllegalStateException ex) {     PropertyChangeEvent pce =
43. **new** PropertyChangeEvent(**this**.rootObject,**this**.nestedPath
44. + propertyName, oldValue, pv.getValue());     **throw** **new**
45. ConversionNotSupportedException(pce, requiredType, ex);    }
46. **catch** (IndexOutOfBoundsException ex) {     **throw** **new**
47. InvalidPropertyException(getRootClass(),**this**.nestedPath +
48. propertyName,"Invalid array index in property path '" +
49. propertyName + "'", ex);    }   } //这里对List进行注入。
50. **else** **if** (propValue **instanceof** List) {    PropertyDescriptor
51. pd = getCachedIntrospectionResults().getPropertyDescriptor
52. (actualName);    Class requiredType =
53. GenericCollectionTypeResolver. getCollectionReturnType(
54. pd.getReadMethod(), tokens.keys.length);    List list =
55. (List) propValue;    **int** index = Integer.parseInt(key);
56. Object oldValue = **null**;   **if** (isExtractOldValueForEditor()
57. && index < list.size()) {     oldValue = list.get(index);
58. }   **try** {     Object convertedValue =
59. **this**.typeConverterDelegate. convertIf Necessary(
60. propertyName, oldValue, pv.getValue(), requiredType);    **if**
61. (index < list.size()) {      list.set(index,
62. convertedValue);    }    **else** **if** (index >= list.size()) {
63. **for** (**int** i = list.size(); i < index; i++) {      **try** {
64. list.add(**null**);      }     **catch** (NullPointerException ex) {
65. **throw** **new** InvalidPropertyException(getRootClass(),
66. **this**.nestedPath + propertyName,         "Cannot set element
67. with index " + index +       " in List of size " +
68. list.size() + ", accessed using property       path '" +
69. propertyName +         "': List does not support filling up
70. gaps       with **null** elements");      }    }
71. list.add(convertedValue);    }    }   **catch**
72. (IllegalArgumentException ex) {  PropertyChangeEvent pce =
73. **new** PropertyChangeEvent(**this**.rootObject,**this**.nestedPath
74. + propertyName, oldValue, pv.getValue());     **throw** **new**
75. TypeMismatchException(pce, requiredType, ex);    }   }
76. //里对Map进行注入。   else if (propValue instanceof Map) {
77. PropertyDescriptor pd =
78. getCachedIntrospectionResults().getPropertyDescriptor(actual
79. Name);    Class mapKeyType =
80. GenericCollectionTypeResolver.getMapKeyReturnType(
81. pd.getReadMethod(), tokens.keys.length);    Class
82. mapValueType =
83. GenericCollectionTypeResolver.getMapValueReturnType(
84. pd.getReadMethod(), tokens.keys.length);    Map map = (Map)
85. propValue;    Object convertedMapKey;    Object
86. convertedMapValue;    **try** {     /\*\*     \* IMPORTANT: Do not
87. pass full property name in here - property editors     \*
88. must not kick in for map keys but rather only for map
89. values.     \*/     convertedMapKey =
90. **this**.typeConverterDelegate.convertIfNecessary      (key,
91. mapKeyType);    }    **catch** (IllegalArgumentException ex) {
92. PropertyChangeEvent pce =       **new**
93. PropertyChangeEvent(**this**.rootObject, **this**.nestedPath +
94. propertyName, **null**, pv.getValue()); **throw** **new**
95. TypeMismatchException(pce, mapKeyType, ex);    }    Object
96. oldValue = **null**;    **if** (isExtractOldValueForEditor()) {
97. oldValue = map.get(convertedMapKey);    }    **try** {     /\*\*
98. \* Pass full property name and old value in here, since we
99. want full     \* conversion ability for map values.     \*/
100. convertedMapValue =
101. **this**.typeConverterDelegate.convertIfNecessary(
102. propertyName, oldValue, pv.getValue(), mapValueType, **null**,
103. **new** MethodParameter(pd.getReadMethod(), -1,       tokens.
104. keys.length + 1));    }    **catch** (IllegalArgumentException
105. ex) {     PropertyChangeEvent pce =      **new**
106. PropertyChangeEvent(**this**.rootObject, **this**.nested Path +
107. propertyName, oldValue, pv.getValue());     **throw** **new**
108. TypeMismatchException(pce, mapValueType, ex);    }
109. map.put(convertedMapKey, convertedMapValue);   }   **else** {
110. **throw** **new** InvalidPropertyException(getRootClass(),
111. **this**.nestedPath +   propertyName,      "Property referenced
112. in indexed property path '" +propertyName +      "' is
113. neither an array nor a List nor a Map; returned value was ["
114. + pv.getValue() + "]"); }  }//这里对非集合类的域进行注入。
115. **else** {   PropertyDescriptor pd = pv.resolvedDescriptor;   **if**
116. (pd == **null**||
117. !pd.getWriteMethod().getDeclaringClass().isInstance
118. (**this**.object)) {    pd =
119. getCachedIntrospectionResults().getPropertyDescriptor(actual
120. Name);    **if** (pd == **null** || pd.getWriteMethod() == **null**) {
121. PropertyMatches matches
122. =PropertyMatches.forProperty(propertyName,
123. getRootClass());     **throw** **new** NotWritablePropertyException(
124. getRootClass(), **this**.nestedPath + propertyName,
125. matches.buildErrorMessage(), matches.getPossibleMatches());
126. }    pv.getOriginalPropertyValue().resolvedDescriptor = pd;
127. }   Object oldValue = **null**;   **try** {    Object originalValue
128. = pv.getValue();    Object valueToApply = originalValue;
129. **if** (!Boolean.FALSE.equals(pv.conversionNecessary)) {     **if**
130. (pv.isConverted()) {      valueToApply =
131. pv.getConvertedValue();     }     **else** {      **if**
132. (isExtractOldValueForEditor() && pd.getReadMethod() !=**null**)
133. {       Method readMethod = pd.getReadMethod();       **if**
134. (!Modifier.isPublic(readMethod.getDeclaringClass().
135. getModifiers())) { readMethod.setAccessible(**true**);       }
136. **try** {        oldValue = readMethod.invoke(**this**.object);
137. }       **catch** (Exception ex) {        **if**
138. (logger.isDebugEnabled()) {         logger.debug("Could not
139. read previous value of        property '" +
140. **this**.nestedPath + propertyName + "'", ex);        }       }
141. }      valueToApply =
142. **this**.typeConverterDelegate.convertIfNecessary
143. (oldValue,originalValue, pd);     }
144. pv.getOriginalPropertyValue().conversionNecessary =
145. (valueToApply ! = originalValue);    }
146. //这里取得注入属性的set方法，通过反射机制，把对象注入进去。
147. Method writeMethod = pd.getWriteMethod();    **if**
148. (!Modifier.isPublic(writeMethod.getDeclaringClass().getModif
149. iers())) {     writeMethod.setAccessible(**true**);    }
150. writeMethod.invoke(**this**.object, valueToApply);   }   **catch**
151. (InvocationTargetException ex) {    PropertyChangeEvent
152. propertyChangeEvent =      **new**
153. PropertyChangeEvent(**this**.rootObject, **this**.nestedPath +
154. propertyName, oldValue, pv.getValue());    **if**
155. (ex.getTargetException() **instanceof** ClassCastException) {
156. pd.getPropertyType(),ex.getTargetException());    }    **else**
157. {     **throw** **new**
158. MethodInvocationException(propertyChangeEvent,
159. ex.getTargetException());    }   }   **catch**
160. (IllegalArgumentException ex) {    PropertyChangeEvent pce =
161. **new** PropertyChangeEvent(**this**.rootObject, **this**.nestedPath +
162. propertyName, oldValue, pv.getValue());    **throw** **new**
163. TypeMismatchException(pce, pd.getPropertyType(), ex);   }
164. **catch** (IllegalStateException ex) {    PropertyChangeEvent
165. pce =      **new** PropertyChangeEvent(**this**.rootObject,
166. **this**.nestedPath +     propertyName, oldValue,
167. pv.getValue());    **throw** **new**
168. ConversionNotSupportedException(pce, pd.getPropertyType(),
169. ex);   }   **catch** (IllegalAccessException ex) {
170. PropertyChangeEvent pce =      **new**
171. PropertyChangeEvent(**this**.rootObject, **this**.nestedPath +
172. propertyName, oldValue, pv.getValue());    **throw** **new**
173. MethodInvocationException(pce, ex);   }  }}

这样就完成了对各种bean属性的依赖注入过程。笔者以前探寻过Spring 2.0的源代码，在这里发现 Spring 3.0 的源代码已经有了很大的改进，整个过程更为清晰了，特别是关于依赖注入的这一部分。如果读者有兴趣，可以分别研究并比较一下 Spring 2.0 和 Spring 3.0 对这部分的实现，这样可以更清晰地看到 Spring 源代码的演进过程，也可以看到 Spring 团队对代码进行重构的思路。

在 Bean 的创建和对象依赖注入的过程中，需要依据 BeanDefinition 中的信息来递归地完成依赖注入。从上面可以看到几个递归过程，这些递归都是以 getBean 为入口的。一个是在上下文体系中查找需要的 Bean 和创建 Bean 的递归调用；另一个递归在依赖注入时，通过递归调用容器的 getBean 方法，得到当前 Bean 的依赖 Bean，同时也触发对依赖 Bean 的创建和注入。在对 Bean 的属性进行依赖注入时，解析的过程也是一个递归的过程。这样，根据依赖关系，一层一层地完成 Bean 的创建和注入，直到最后完成当前 Bean 的创建，有了这个顶层 Bean 的创建和对它的属性依赖注入的完成，也意味着和当前 Bean 相关的整个依赖链的注入完成。

在 Bean 建立和依赖注入完成以后，在 IoC 容器中建立起一系列靠依赖关系联系起来的Bean，这个Bean 已经不是简单的 Java 对象了。这个 Bean 系列建立完成以后，通过 IoC 容器的相关接口方法，就可以非常方便地让上层应用使用了。回到前面的关于水桶的例子，到这里，我们不但找到了水源，而且成功地把水装到了水桶中，同时对水桶里的水完成了一系列的处理，比如消毒、煮沸……尽管还是水，但经过一系列的处理以后，这些水已经是开水了，已经可以直接饮用了！