第5章

bean标签解析

在第 4 章中对 BeanDefinition 的注册流程进行了相关分析，在 SpringXML 开发模式下 BeanDefinition 的数据来源是 SpringXML 原始标签中的 bean 标签，该标签会定义 BeanDefinition 对象中的各个属性。本章将围绕 bean 标签进行测试环境搭建和 bean 标签解析流程的分析。在 Spring 中对于 SpringXML 文件中 bean 标签的解析过程如下。

(1) 将 Element 交给 BeanDefinitionParserDelegate 解析。

(2) 注册 BeanDefinition。

(3) 发布组件注册事件。

5.1 创建 bean 标签解析环境

在进行 bean 标签的源码分析之前，还需要做一些准备工作：搭建一个 bean 标签解析的环境。

5.1.1 编写 SpringXML 配置文件

下面先创建一个 SpringXML 配置文件，并将其命名为 bean-node.xml，向该 bean-node.xml 文件中填充下面这段代码。

```
<?xml version = "1.0" encoding = "UTF - 8"?>
< beans xmlns: xsi = "http://www.w3.org/2001/XMLSchema - instance"
       xmlns = "http://www.springframework.org/schema/beans"
       xsi: schemaLocation = " http://www. springframework. org/schema/beans http://www.
springframework.org/schema/beans/spring - beans.xsd">

   < bean id = "people" class = "com.source.hot.ioc.book.pojo.PeopleBean">
```

```
    </bean>
</beans>
```

5.1.2 编写 bean-node 对应的测试用例

创建一个名为 BeanNodeTest 的 Java 对象，代码如下。

```
class BeanNodeTest {
    @Test
    void testBean() {
        ClassPathXmlApplicationContext context
                = new ClassPathXmlApplicationContext("META-INF/bean-node.xml");

        Object people = context.getBean("people");
        context.close();
    }
}
```

通过上述操作测试用例准备完毕，下面请回忆一下 Spring 中对于 bean 标签的解析方法。Spring 中对于 bean 标签解析的方法签名是 org.springframework.beans.factory.xml.DefaultBeanDefinitionDocumentReader#processBeanDefinition。processBeanDefinition 方法的代码如下。

```
protected void processBeanDefinition(Element ele, BeanDefinitionParserDelegate delegate) {
    //创建 BeanDefinition
    BeanDefinitionHolder bdHolder = delegate.parseBeanDefinitionElement(ele);
    if (bdHolder != null) {
        //BeanDefinition 装饰
        bdHolder = delegate.decorateBeanDefinitionIfRequired(ele, bdHolder);
        try {
            //注册 BeanDefinition
            BeanDefinitionReaderUtils.registerBeanDefinition(bdHolder, getReaderContext().
getRegistry());
        }
        catch (BeanDefinitionStoreException ex) {
            getReaderContext().error("Failed to register bean definition with name '" +
                    bdHolder.getBeanName() + "'", ele, ex);
        }
        //component 注册事件触发
        getReaderContext().fireComponentRegistered(new BeanComponentDefinition(bdHolder));
    }
}
```

Spring 中通过 BeanDefinitionParserDelegate#parseBeanDefinitionElement 方法来对 Element 对象进行解析，映射到 XML 文件中 Element 对象就是一个 XML 标签，下面将进入到该方法的分析阶段。

5.2 parseBeanDefinitionElement 方法处理

5.2.1 parseBeanDefinitionElement 第一部分处理

通过前文找到了需要分析的方法，Spring 中对于 bean 标签的 id 属性和 name 属性的处理代码如下。

```
String id = ele.getAttribute(ID_ATTRIBUTE);
//获取 name
String nameAttr = ele.getAttribute(NAME_ATTRIBUTE);

//别名列表
List<String> aliases = new ArrayList<>();
//是否有 name 属性
if (StringUtils.hasLength(nameAttr)) {
   //获取名称列表,根据 ,; 进行分隔
  String[] nameArr =
StringUtils.tokenizeToStringArray(nameAttr,MULTI_VALUE_ATTRIBUTE_DELIMITERS);
  //添加所有
  aliases.addAll(Arrays.asList(nameArr));
}
```

第一部分的处理很简单，获取 bean 标签中的 id 和 name 属性，对 name 属性做分隔符切分后将切分结果作为别名。在第一部分中提到了分隔符这一信息，在这段代码中的分隔符总共有以下 3 种。

(1) 逗号。

(2) 分号。

(3) 空格。

5.2.2 parseBeanDefinitionElement 第二部分处理

接下来将进入第二部分的分析，Spring 中对于 BeanName 的处理代码如下。

```
String beanName = id;
if (!StringUtils.hasText(beanName) && !aliases.isEmpty()) {
 //别名的第一个设置为 beanName
 beanName = aliases.remove(0);
 if (logger.isTraceEnabled()) {
    logger.trace("No XML 'id' specified - using '" + beanName +
          "' as bean name and " + aliases + " as aliases");
 }
}

//BeanDefinition 为空
if (containingBean == null) {
```

```
//判断 beanName 是否被使用,Bean 别名是否被使用
checkNameUniqueness(beanName,aliases,ele);
}
```

在第二部分的处理中可以再将其分为以下两小段内容。

(1) 关于 BeanName 的推论。

(2) 关于 BeanName 是否被使用,别名是否被使用的验证。

Spring 中对于 BeanName 的推论规则如下。

(1) BeanName 可以是 bean 标签中的 id 属性。

(2) BeanName 可以是 bean 标签中的 name 属性,注意 name 属性可以存在多个,多个 name 使用分隔符分隔,当存在多个 name 属性时会取第一个值作为 BeanName。

(3) 当 bean 标签中同时出现 id 属性和 name 属性时会用 id 作为 BeanName。

在了解了 BeanName 的生成规则后来编写相关测试用例,首先要变写的是关于 id 的测试用例。

```
< bean id = "people" class = "com. source. hot. ioc. book. pojo. PeopleBean">
</bean >
```

根据前文的分析此时可以推论出 BeanName 应该是 people,调试信息如图 5.1 所示。

接下来编写关于 name 的测试用例。

```
< bean name = "peopleBean,people" class = "com. source. hot. ioc. book. pojo. PeopleBean">
</bean >
```

根据前文的分析此时可以推论出 BeanName 应该是 peopleBean,调试信息如图 5.2 所示。

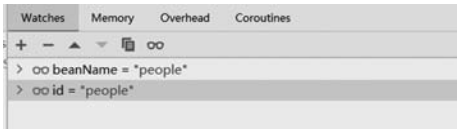

图 5.1 设置 id 时的 BeanName

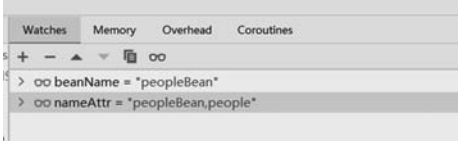

图 5.2 设置 name 时的 BeanName

最后来编写 id 和 name 同时设置的测试用例。

```
< bean id = "p1" name = "peopleBean,people"
class = "com. source. hot. ioc. book. pojo. PeopleBean">
</bean >
```

根据前文的分析此时可以推论出 BeanName 应该是 p1,调试信息如图 5.3 所示。

图 5.3 同时设置 id 和 name 时的 BeanName

关于 BeanName 的推论相关的分析到此结束，下面进入到第二部分关于 BeanName 是否被使用，别名是否被使用的验证的分析。Spring 中 checkNameUniqueness 的完整代码内容如下。

```
protected void checkNameUniqueness(String beanName,List < String > aliases,Element beanElement) {
   //当前寻找的 name
   String foundName = null;

   //是否有 BeanName
   //使用过的 name 中是否存在
   if (StringUtils.hasText(beanName) && this.usedNames.contains(beanName)) {
      foundName = beanName;
   }
   if (foundName == null) {
      //寻找匹配的第一个
      foundName = CollectionUtils.findFirstMatch(this.usedNames,aliases);
   }
   //抛出异常
   if (foundName != null) {
       error("Bean name '" + foundName + "' is already used in this < beans > element",
beanElement);
   }

   //加入使用队列
   this.usedNames.add(beanName);
   this.usedNames.addAll(aliases);
}
```

在 checkNameUniqueness 方法中需要重点关注的是 usedNames 变量，usedNames 变量是指已经使用过的名称包含 BeanName 和 Alias。整个验证过程就是判断传递的参数是否在 usedNames 容器中存在，一旦存在就会抛出异常。如果不存在则将 BeanName 和 Alias 加入 usedNames 容器等待后续使用。

至此，parseBeanDefinitionElement 第二部分的分析告一段落，下面将进入第三部分的分析。

5.2.3 parseBeanDefinitionElement 第三部分处理

Spring 中 parseBeanDefinitionElement 方法中的第三部分代码如下。

```
AbstractBeanDefinition beanDefinition = parseBeanDefinitionElement(ele,beanName,containingBean);
```

这段代码其实是一个方法的调用，真正应该阅读的代码应该是下面这一段。

```
public AbstractBeanDefinition parseBeanDefinitionElement(
  Element ele,String beanName,@Nullable BeanDefinition containingBean) {

  //第一部分
  //设置阶段 Bean 定义解析阶段
```

```
        this.parseState.push(new BeanEntry(beanName));

        String className = null;
        //是否包含属性 class
        if (ele.hasAttribute(CLASS_ATTRIBUTE)) {
            className = ele.getAttribute(CLASS_ATTRIBUTE).trim();
        }
        String parent = null;
        //是否包含属性 parent
        if (ele.hasAttribute(PARENT_ATTRIBUTE)) {
            parent = ele.getAttribute(PARENT_ATTRIBUTE);
        }

        //第二部分

        //创建 BeanDefinition
        AbstractBeanDefinition bd = createBeanDefinition(className,parent);

        //BeanDefinition 属性设置
        parseBeanDefinitionAttributes(ele,beanName,containingBean,bd);
        //设置描述
        bd.setDescription(DomUtils.getChildElementValueByTagName(ele,DESCRIPTION_ELEMENT));
        //元信息设置 meta 标签解析
        parseMetaElements(ele,bd);
        //lookup-override 标签解析
        parseLookupOverrideSubElements(ele,bd.getMethodOverrides());
        //replaced-method 标签解析
        parseReplacedMethodSubElements(ele,bd.getMethodOverrides());

        //constructor-arg 标签解析
        parseConstructorArgElements(ele,bd);
        //property 标签解析
        parsePropertyElements(ele,bd);
        //qualifier 标签解析
        parseQualifierElements(ele,bd);
        //资源设置
        bd.setResource(this.readerContext.getResource());
        //source 设置
        bd.setSource(extractSource(ele));

        return bd;

    }
```

在这个方法中 Spring 做了 11 步操作，下面将围绕这 11 步依次解析它们的处理行为。操作细节如下。

(1) 处理 className 和 parent 属性。

(2) 创建基本的 BeanDefinition 对象，具体类：AbstractBeanDefinition、GenericBeanDefinition。

(3) 读取 bean 标签的属性，为 BeanDefinition 对象进行赋值。

(4) 处理描述标签 description。

(5) 处理 meta 标签。

(6) 处理 lookup-override 标签。

(7) 处理 replaced-method 标签。

(8) 处理 constructor-arg 标签。

(9) 处理 property 标签。

(10) 处理 qualifier 标签。

(11) 设置资源对象和 source 属性。

注意：在这段方法中出现的 parseState 对象操作不属于本节的分析内容，可以简单被理解成这是一个阶段标记。

1. 处理 class name 和 parent 属性

Spring 中对于 bean 标签的 class name 和 parent 两个属性的处理代码如下。

```
String className = null;
//是否包含属性 class
if (ele.hasAttribute(CLASS_ATTRIBUTE)) {
   className = ele.getAttribute(CLASS_ATTRIBUTE).trim();
}
String parent = null;
//是否包含属性 parent
if (ele.hasAttribute(PARENT_ATTRIBUTE)) {
   parent = ele.getAttribute(PARENT_ATTRIBUTE);
}
```

在这段代码中对于 className 和 parent 的处理十分简单，直接从 Element 对象中提取，提取完成数据后为下面创建 AbstractBeanDefinition 对象提供了基础数据支持。下面来看创建 AbstractBeanDefinition 对象的细节。

2. 创建 AbstractBeanDefinition 对象

Spring 中对于创建 AbstractBeanDefinition 对象的代码如下。

```
//parseBeanDefinitionElement 方法内调用
AbstractBeanDefinition bd = createBeanDefinition(className,parent);

//createBeanDefinition 详情
protected AbstractBeanDefinition
createBeanDefinition(@Nullable String className,@Nullable String parentName)
    throws ClassNotFoundException {


    return BeanDefinitionReaderUtils.createBeanDefinition(
        parentName,className,this.readerContext.getBeanClassLoader());
}
```

在这段方法中存在一层引用方法，这段引用方法的代码如下。

```
public static AbstractBeanDefinition createBeanDefinition(
```

```
    @Nullable String parentName, @Nullable String className, @Nullable ClassLoader
classLoader) throws ClassNotFoundException {

    GenericBeanDefinition bd = new GenericBeanDefinition();
    //设置父 BeanName
    bd.setParentName(parentName);
    if (className != null) {
        if (classLoader != null) {
            //设置 class
            //内部是通过反射创建 class
            bd.setBeanClass(ClassUtils.forName(className,classLoader));
        }
        else {
            //设置 className
            bd.setBeanClassName(className);
        }
    }
    return bd;
}
```

在这段方法中，createBeanDefinition 方法是一个静态方法，其完整的方法签名是 org.springframework.beans.factory.support.BeanDefinitionReaderUtils#createBeanDefinition，在这个方法中 Spring 对其做出了如下操作。

（1）创建类型为 GenericBeanDefinition 的 BeanDefinition 对象。

（2）为创建的 GenericBeanDefinition 对象进行数据设置，设置的属性是 parentName 和 beanClass。

注意：beanClass 的数据类型存在以下两种情况。

（1）beanClass 的类型是 String。

（2）beanClass 的类型是 Class。

下面进入调试阶段，首先编写一段 SpringXML 配置：

```
<bean id = "people" class = "com.source.hot.ioc.book.pojo.PeopleBean">
</bean>
```

在完成配置文件编写后启动项目观察 createBeanDefinition 执行后的结果对象 bd(BeanDefinition：bean 定义信息)，如图 5.4 所示。

可以看到，beanClass 还是字符串类型。现在 AbstractBeanDefinition 基础对象已经准备完毕，接下来就是补充这个对象的其他属性。

3. 设置 BeanDefinition 的基本信息

接下来进行 parseBeanDefinitionAttributes 方法的解析，Spring 中 parseBeanDefinitionAttribute 的细节代码如下。

```
public AbstractBeanDefinition parseBeanDefinitionAttributes(Element ele,
String beanName,
      @Nullable BeanDefinition containingBean,AbstractBeanDefinition bd) {

   //是否存在 singleton 属性
```

图 5.4 createBeanDefinition 执行后结果

```
if (ele.hasAttribute(SINGLETON_ATTRIBUTE)) {
   error("Old 1.x 'singleton' attribute in use - upgrade to 'scope' declaration",ele);
}
//是否存在 scope 属性
else if (ele.hasAttribute(SCOPE_ATTRIBUTE)) {
   //设置 scope 属性
   bd.setScope(ele.getAttribute(SCOPE_ATTRIBUTE));
}
//Bean 定义是否为空
else if (containingBean != null) {
   //设置 BeanDefinition 中的 scope
   bd.setScope(containingBean.getScope());
}

//是否存在 abstract 属性
if (ele.hasAttribute(ABSTRACT_ATTRIBUTE)) {
   //设置 abstract 属性

   bd.setAbstract(TRUE_VALUE.equals(ele.getAttribute(ABSTRACT_ATTRIBUTE)));
}

//获取 lazy-init 属性
String lazyInit = ele.getAttribute(LAZY_INIT_ATTRIBUTE);
//是否是默认的 lazy-init 属性
if (isDefaultValue(lazyInit)) {
   //获取默认值
   lazyInit = this.defaults.getLazyInit();
```

```
    }
    //设置 lazy-init 属性
    bd.setLazyInit(TRUE_VALUE.equals(lazyInit));

    //获取注入方式
    //autowire 属性
    String autowire = ele.getAttribute(AUTOWIRE_ATTRIBUTE);
    //设置注入方式
    bd.setAutowireMode(getAutowireMode(autowire));

    //依赖的 Bean
    //depends-on 属性
    if (ele.hasAttribute(DEPENDS_ON_ATTRIBUTE)) {
        String dependsOn = ele.getAttribute(DEPENDS_ON_ATTRIBUTE);
        bd.setDependsOn(StringUtils.tokenizeToStringArray(dependsOn,MULTI_VALUE_ATTRIBUTE_
DELIMITERS));
    }

    //autowire-candidate 是否自动注入判断
    String autowireCandidate = ele.getAttribute(AUTOWIRE_CANDIDATE_ATTRIBUTE);
    if (isDefaultValue(autowireCandidate)) {
        String candidatePattern = this.defaults.getAutowireCandidates();
        if (candidatePattern != null) {
            String[] patterns =
StringUtils.commaDelimitedListToStringArray(candidatePattern);
            //* 匹配设置数据
            bd.setAutowireCandidate(PatternMatchUtils.simpleMatch(patterns,beanName));
        }
    }
    else {
        bd.setAutowireCandidate(TRUE_VALUE.equals(autowireCandidate));
    }

    //获取 primary 属性
    if (ele.hasAttribute(PRIMARY_ATTRIBUTE)) {
        bd.setPrimary(TRUE_VALUE.equals(ele.getAttribute(PRIMARY_ATTRIBUTE)));
    }

    //获取 init-method 属性
    if (ele.hasAttribute(INIT_METHOD_ATTRIBUTE)) {
        String initMethodName = ele.getAttribute(INIT_METHOD_ATTRIBUTE);
        bd.setInitMethodName(initMethodName);
    }
    //没有 init-method 的情况处理
    else if (this.defaults.getInitMethod() != null) {
        bd.setInitMethodName(this.defaults.getInitMethod());
        bd.setEnforceInitMethod(false);
    }

    //获取 destroy-method 属性
    if (ele.hasAttribute(DESTROY_METHOD_ATTRIBUTE)) {
```

```
        String destroyMethodName =
ele.getAttribute(DESTROY_METHOD_ATTRIBUTE);
        bd.setDestroyMethodName(destroyMethodName);
    }
    //没有 destroy-method 的情况处理
    else if (this.defaults.getDestroyMethod() != null) {
        bd.setDestroyMethodName(this.defaults.getDestroyMethod());
        bd.setEnforceDestroyMethod(false);
    }

    //获取 factory-method 属性
    if (ele.hasAttribute(FACTORY_METHOD_ATTRIBUTE)) {
        bd.setFactoryMethodName(ele.getAttribute(FACTORY_METHOD_ATTRIBUTE));
    }
    //获取 factory-bean 属性
    if (ele.hasAttribute(FACTORY_BEAN_ATTRIBUTE)) {
        bd.setFactoryBeanName(ele.getAttribute(FACTORY_BEAN_ATTRIBUTE));
    }

    return bd;
}
```

这段代码的处理过程与 className 和 parent 的处理过程基本类似,它们的处理思路都是从 bean 标签中提取属性对应的属性值,将属性值设置给 BeanDefinition 对象(BeanDefinition 对象是通过上一步创建出来),在这个处理过程中有一个值得关注的变量 defaults,先来看 defaults 变量的定义代码。

```
private final DocumentDefaultsDefinition defaults = new DocumentDefaultsDefinition();
```

这段代码中 defaults 指定了一个具体的数据类型 DocumentDefaultsDefinition,在 DocumentDefaultsDefinition 类型中存放了 Spring 对一些属性的默认值,DocumentDefaultsDefinition 中的定义的属性及属性默认值如表 5.1 所示。

表 5.1 DocumentDefaultsDefinition 属性及属性默认值

默认值属性名称	默 认 值
lazyInit	false
merge	false
autowire	no
autowireCandidates	null
initMethod	null
destroyMethod	null
source	null

在了解 defaults 对象中存有的数据后,来进一步观察 BeanDefinition 经过 parseBeanDefinitionAttributes 方法处理后得到的信息,如图 5.5 所示。

4. 设置 BeanDefinition 描述信息

接下来进行 BeanDefintion 描述信息设置方法的解析,首先需要编写一段 SpringXML

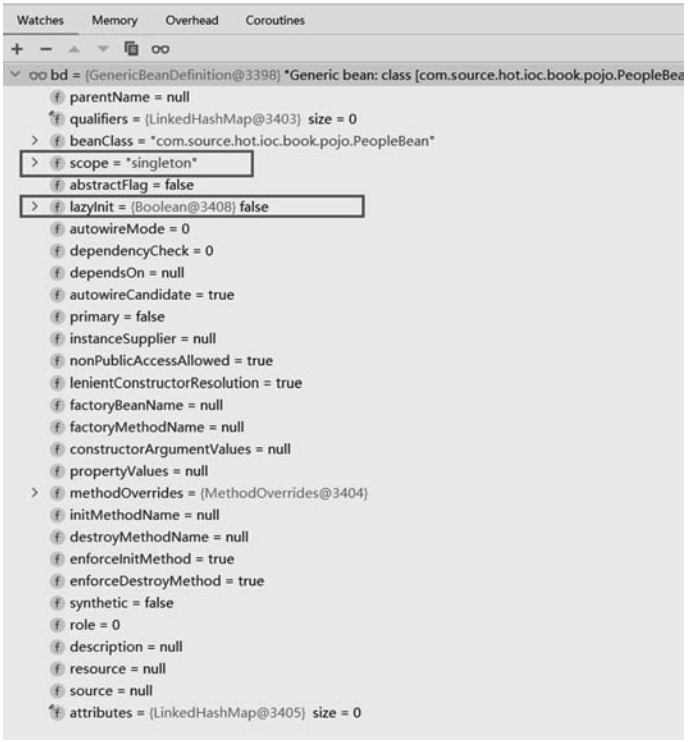

图 5.5　parseBeanDefinitionAttributes 方法执行后结果

配置，在这段配置中需要设置 bean 标签的属性值，具体代码如下。

```
<bean id = "people" class = "com.source.hot.ioc.book.pojo.PeopleBean">
        <description>this is a bean description</description>
</bean>
```

下面是设置 bean 描述信息的代码内容。

```
bd.setDescription(DomUtils.getChildElementValueByTagName(ele,DESCRIPTION_ELEMENT));
```

这段代码的处理比较明了，提取当前节点下 description 节点中的数据，将这个数据赋值给 BeanDefinition 中的 description 属性，处理结果如图 5.6 所示。

5. 设置 meta 属性

接下来进行 parseBeanDefinitionElement 方法的解析。Spring 中 parseBeanDefinitionElement 方法的详细信息如下。

```
public void parseMetaElements(Element ele,
BeanMetadataAttributeAccessor attributeAccessor) {
   //获取下级标签
   NodeList nl = ele.getChildNodes();
   //循环子标签
   for (int i = 0; i < nl.getLength(); i++) {
      Node node = nl.item(i);
      //设置数据
```

图 5.6 设置 bean 描述信息后结果

```
        //是不是 meta 标签
        if (isCandidateElement(node) && nodeNameEquals(node,META_ELEMENT)) {
            Element metaElement = (Element) node;
            //获取 key 属性
            String key = metaElement.getAttribute(KEY_ATTRIBUTE);
            //获取 value 属性
            String value = metaElement.getAttribute(VALUE_ATTRIBUTE);
            //元数据对象设置
            BeanMetadataAttribute attribute = new BeanMetadataAttribute(key,value);
            //设置 source
            attribute.setSource(extractSource(metaElement));
            //信息添加
            attributeAccessor.addMetadataAttribute(attribute);
        }
    }
}
```

下面先来编写一段测试用例再进行源代码分析,这段代码处理的是 meta 标签的数据,改进 SpringXML 配置文件。

```
<bean id = "people" class = "com.source.hot.ioc.book.pojo.PeopleBean">
        <description>this is a bean description</description>
        <meta key = "key" value = "value"/>
</bean>
```

通过这段配置文件可以知道，meta 标签中存在两个属性 key 和 value。在 parseMetaElements 方法中对 bean 标签的下级标签 meta 的处理就是提取 key 和 value 属性的属性值，在提取后创建 BeanMetadataAttribute 对象，创建对象后将其放到 BeanMetadataAttributeAccessor 集合中。在这段处理过程中出现了一个新的对象 BeanMetadataAttributeAccessor，这个对象是什么呢？回答这个问题需要从 AbstractBeanDefinition 出发，观察 AbstractBeanDefinition 的类图，如图 5.7 所示。

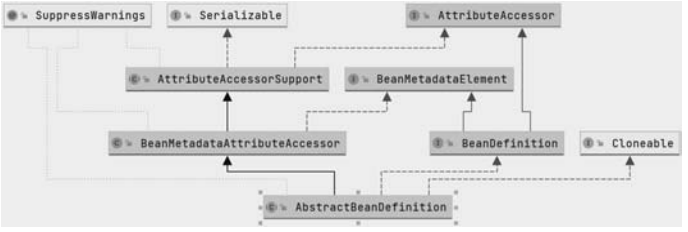

图 5.7　AbstractBeanDefinition 类图

在类图中可以直观地看到，AbstractBeanDefinition 是 BeanMetadataAttributeAccessor 的子类。对于 BeanMetadataAttributeAccessor 的设置其实就是为 AbstractBeanDefinition 添加属性。了解了这些理论知识后，经过 parseMetaElements 方法处理后 BeanDefinition 的对象信息如图 5.8 所示。

图 5.8　parseMetaElements 执行后 BeanDefinition 的信息

6. lookup-override 标签处理

在处理完 meta 标签后将处理 lookup-override 标签。前文所编写的一些测试代码在这个方法中是不足以支持断点调试的，需要对测试代码进行补充。下面编写测试用例，假设现在有一个商店在出售水果，水果可以是苹果、香蕉等，此时客户需要通过不同的商店获取各个商店所售卖的内容，下面先定义几个 Java 对象。

水果对象详细代码如下。

```
public class Fruits {
  private String name;

  public String getName() {
     return name;
  }

  public void setName(String name) {
     this.name = name;
  }
}
```

苹果对象详细代码如下。

```
public class Apple extends Fruits {
  public Apple() {
     this.setName("apple");
  }

  public void hello() {
     System.out.println("hello");
  }

}
```

商店对象详细代码如下。

```
public abstract class Shop {
  public abstract Fruits getFruits();
}
```

编写 SpringXML 配置文件，文件名称为 spring-lookup-method.xml，详细代码如下。

```
<?xml version = "1.0" encoding = "UTF - 8"?>
< beans xmlns: xsi = "http://www.w3.org/2001/XMLSchema - instance"
      xmlns = "http://www.springframework.org/schema/beans"
      xsi: schemaLocation = "http://www.springframework.org/schema/beans http://www.springframework.
org/schema/beans/spring - beans.xsd">
  < bean id = "apple" class = "com.source.hot.ioc.book.pojo.lookup.Apple">
  </bean >

  < bean id = "shop" class = "com.source.hot.ioc.book.pojo.lookup.Shop">
     < lookup - method name = "getFruits" bean = "apple"/>
```

```
    </bean>


</beans>
```

编写测试用例，详细代码如下。

```
@Test
void testLookupMethodBean() {
    ClassPathXmlApplicationContext context =
new ClassPathXmlApplicationContext("META-INF/spring-lookup-method.xml");

    Shop shop = context.getBean("shop",Shop.class);
    System.out.println(shop.getFruits().getName());
    assert context.getBean("apple").equals(shop.getFruits());
}
```

测试方法 testLookupMethodBean 的执行结果输出 apple 并且测试通过。

先来整理测试用例中的执行流程，在 SpringXML 配置文件中配置 bean 标签的 lookup-method 属性，在调用方法时会根据 lookup-method 中的 bean 属性在 Spring 容器中寻找 bean 属性值对应的 Bean 实例，当调用 lookup-method 中 name 属性值的方法时将 bean 属性对应的 Bean 实例作为返回结果返回。在测试用例中调用 shop#getFruits 方法时会在容器中找到 apple 这个 Bean 实例将其作为返回值。

简单了解使用逻辑后，现在来看 Spring 是如何处理 lookup-method 标签的。先阅读处理方法 parseLookupOverrideSubElements。

```
public void parseLookupOverrideSubElements(Element beanEle,MethodOverrides overrides) {
    //获取子标签
    NodeList nl = beanEle.getChildNodes();
    for (int i = 0; i < nl.getLength(); i++) {
        Node node = nl.item(i);
        //是否有 lookup-method 标签
        if (isCandidateElement(node) &&
nodeNameEquals(node,LOOKUP_METHOD_ELEMENT)) {
            Element ele = (Element) node;
            //获取 name 属性
            String methodName = ele.getAttribute(NAME_ATTRIBUTE);
            //获取 bean 属性
            String beanRef = ele.getAttribute(BEAN_ELEMENT);
            //创建覆盖依赖
            LookupOverride override = new LookupOverride(methodName,beanRef);
            //设置 source
            override.setSource(extractSource(ele));
            overrides.addOverride(override);
        }
    }
}
```

这段处理模式和 meta 标签的处理模式基本相同。处理方式：获取所有子节点，如果子节点是 lookup-method 就提取 name 和 bean 两个属性，在属性获取后创建对应的 Java 对

象 LookupOverride 并设置给 MethodOverrides，MethodOverrides 是 BeanDefinition 的一个成员变量。了解理论内容后再看处理结果，如图 5.9 所示。

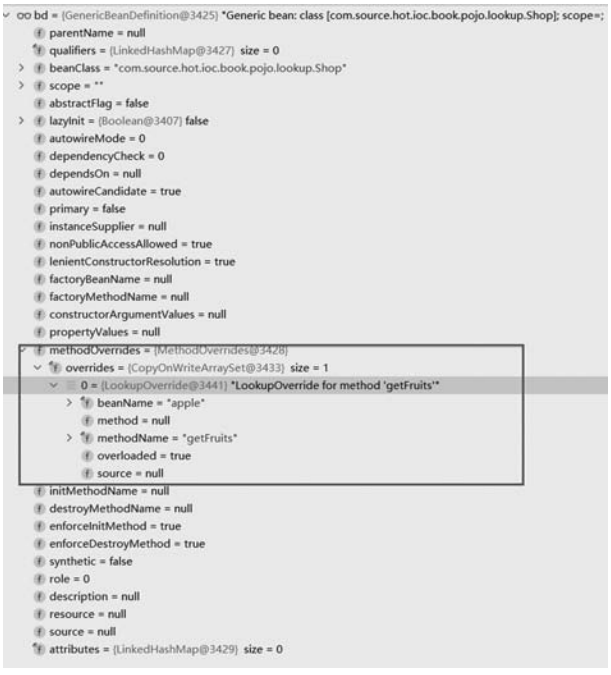

图 5.9 parseLookupOverrideSubElements 执行后 BeanDefinition 的信息

现在对于 lookup-method 标签的解析已经完成，下面思考一个问题：在执行时发生了什么？即当调用 shop.getFruits()方法时发生了什么？回答这个问题需要了解另一个知识点对象代理，本章不会对代理过程做一个完善的分析，仅做调用方法的分析。

先看 shop 对象，shop 对象并不是一个普通的 Java 对象，它是一个增强对象，shop 对象信息如图 5.10 所示。

图 5.10 shop 对象

通过图 5.10 可以确认，shop 对象中存在 LookupOverrideMethodInterceptor 对象，接下来需要找到 LookupOverrideMethodInterceptor 对象，在这个对象中存有需要分析的方法：

```
@Override
public Object intercept(Object obj,Method method,Object[] args,MethodProxy mp) throws Throwable {
    LookupOverride lo =
(LookupOverride) getBeanDefinition().getMethodOverrides().getOverride(method);
    Assert.state(lo != null,"LookupOverride not found");
```

```
        Object[] argsToUse = (args.length > 0 ? args : null);
        if (StringUtils.hasText(lo.getBeanName())) {
            return (argsToUse != null ? this.owner.getBean(lo.getBeanName(),argsToUse) :
                    this.owner.getBean(lo.getBeanName()));
        }
        else {
            return (argsToUse != null ?
    this.owner.getBean(method.getReturnType(),argsToUse) :
                    this.owner.getBean(method.getReturnType()));
        }
    }
```

这段方法的本质就是动态代理的方法增强，可以简单理解为原有方法的执行结果被 intercept 替换了，替换过程如下。

(1) 通过参数 method 在 lookupOverride 容器中找到替换的 LookupOverride 对象。

(2) 从 LookupOverride 中提取 beanName 属性，在 Spring IoC 容器(BeanFactory)中通过 beanName 或者 beanName＋构造参数列表获得 Bean 实例，将 Bean 实例作为返回结果。

了解了执行流程后下面对一些关键信息进行复盘。在这个测试用例中不存在构造参数列表数据，代码会执行 this.owner.getBean(lo.getBeanName())方法，注意 owner 就是 BeanFactory，提取实例需要依赖它。下面来看通过 method 找到 LookupOverride 对象中 LookupOverride 存储的数据内容，如图 5.11 所示。

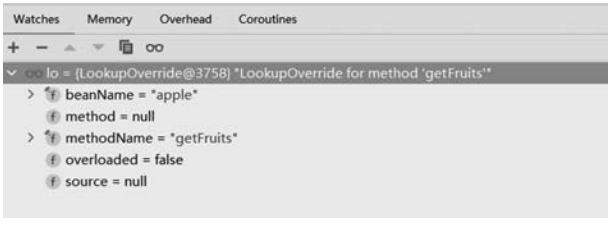

图 5.11 LookupOverride 存储的数据内容

在图 5.11 中可以看到，需要加载的 beanName 是 apple，现在具备获取 Bean 实例的工具 owner，接下来就是从 owner 中将 beanName 为 apple 的 Bean 实例提取作为返回值返回即可。

7. replaced-method 标签处理

前文对 lookup-override 做了充分的分析，在这个基础上阅读 replaced-method 方法将事半功倍，它们的底层处理方式可以说是大同小异。首先来编写 replaced-method 的测试用例。

第一步需要编写 MethodReplacer 接口实现类，实现类代码如下。

```
public class MethodReplacerApple implements MethodReplacer {
    @Override
    public Object reimplement(Object obj,Method method,Object[] args) throws Throwable {
        System.out.println("方法替换");
```

```
        return obj;
    }
}
```

编写 SpringXML 配置文件，文件名称为 spring-replaced-method.xml，文件内容如下。

```
<?xml version = "1.0" encoding = "UTF - 8"?>
< beans xmlns: xsi = "http://www.w3.org/2001/XMLSchema - instance"
       xmlns = "http://www.springframework.org/schema/beans"
       xsi: schemaLocation = "http://www.springframework.org/schema/beans http://www.springframework.
org/schema/beans/spring - beans.xsd">
    < bean id = "apple" class = "com.source.hot.ioc.book.pojo.lookup.Apple">
        < replaced - method replacer = "methodReplacerApple" name = "hello" >
            < arg - type > String </arg - type >
        </replaced - method >

    </bean >

    < bean
id = "methodReplacerApple" class = "com.source.hot.ioc.book.pojo.replacer.MethodReplacerApple">
    </bean >

</beans >
```

编写测试方法，具体代码如下。

```
@Test
void testReplacedMethod(){
     ClassPathXmlApplicationContext context =
new ClassPathXmlApplicationContext("META - INF/spring - replaced - method.xml");
     Apple apple = context.getBean("apple",Apple.class);
     apple.hello();
}
```

测试方法执行结果：控制台输出方法替换。

接下来进行 replaced-method 标签解析的讲解，先阅读处理代码。

```
public void parseReplacedMethodSubElements(Element beanEle,
MethodOverrides overrides) {
    //子节点获取
    NodeList nl = beanEle.getChildNodes();
    for (int i = 0; i < nl.getLength(); i++) {
        Node node = nl.item(i);
        //是否包含 replaced - method 标签
        if (isCandidateElement(node)
&& nodeNameEquals(node,REPLACED_METHOD_ELEMENT)) {
            Element replacedMethodEle = (Element) node;
            //获取 name 属性
            String name = replacedMethodEle.getAttribute(NAME_ATTRIBUTE);
            //获取 replacer
            String callback = replacedMethodEle.getAttribute(REPLACER_ATTRIBUTE);
```

```
            //对象组装
            ReplaceOverride replaceOverride = new ReplaceOverride(name,callback);
            //Look for arg-type match elements.
            //子节点属性
            //处理 arg-type 标签
            List<Element> argTypeEles
 = DomUtils.getChildElementsByTagName(replacedMethodEle,ARG_TYPE_ELEMENT);

            for (Element argTypeEle : argTypeEles) {
               //获取 match 数据值
               String match =
argTypeEle.getAttribute(ARG_TYPE_MATCH_ATTRIBUTE);
               //match 信息设置
               match = (StringUtils.hasText(match) ? match :
DomUtils.getTextValue(argTypeEle));
               if (StringUtils.hasText(match)) {
                  //添加类型标识
                  replaceOverride.addTypeIdentifier(match);
               }
            }
            //设置 source
            replaceOverride.setSource(extractSource(replacedMethodEle));
            //重载列表添加
            overrides.addOverride(replaceOverride);
         }
      }
   }
```

在这一段代码中对于 replaced-method 标签的处理分为以下三个步骤。

(1) 提取 replaced-method 标签中的 name 和 replacer 属性。

(2) 提取子标签 arg-type 的属性。

(3) 将提取得到的数据进行组装并赋值给 BeanDefinition 对象。

在了解执行流程后再看看 parseReplacedMethodSubElements 方法执行后 BeanDefinition 的数据信息,如图 5.12 所示。

下面来看 apple.hello 方法执行时发生了什么? 首先明确一点,apple 对象是一个代理对象,并不是一个原始的 Java 对象,数据信息如图 5.13 所示。

从图 5.13 中可以看到一个叫作 ReplaceOverrideMethodInterceptor 的类名,这个类就是真正需要分析的目标,在 ReplaceOverrideMethodInterceptor 对象中关于 intercept 方法的处理池代码如下。

```
@Override
public Object intercept(Object obj,Method method,Object[] args,MethodProxy mp) throws Throwable {
   ReplaceOverride ro =
(ReplaceOverride) getBeanDefinition().getMethodOverrides().getOverride(method);
   Assert.state(ro != null,"ReplaceOverride not found");
   MethodReplacer mr = this.owner.getBean(
ro.getMethodReplacerBeanName(),MethodReplacer.class);
```

图 5.12 parseReplacedMethodSubElements 执行后的 BeanDefinition 信息

图 5.13 replaced-method 中的 apple 对象

```
    return mr.reimplement(obj,method,args);
}
```

在这个方法中 Spring 会根据 method 进行 ReplaceOverride 的搜索,在这里 ReplaceOverride 就是通过 parseReplacedMethodSubElements 方法解析得到的对象,ro 数据信息如图 5.14 所示。

在图 5.14 中可以看到,在 SpringXML 配置中配置过的一些数据信息,Spring 会通过 methodReplacerBeanName+MethodReplace.class 在 Spring IoC 容器中找到最终的实现类并调用其方法,将方法的处理结果作为返回值返回,从而达到方法替换的作用。

8. constructor-arg 标签处理

接下来将进入 constructor-arg 标签的源码分析阶段,进入方法分析之前需要制作一些

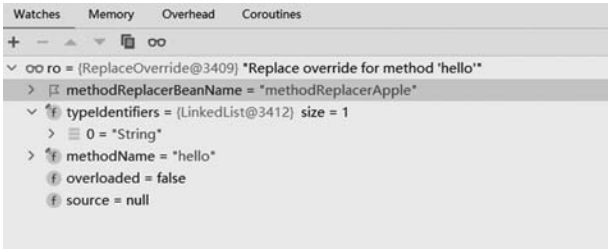

图 5.14　ro 数据信息

测试用例。

首先创建一个 Java 对象。

```
public class PeopleBean {
    private String name;

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public PeopleBean() {
    }

    public PeopleBean(String name) {
        this.name = name;
    }
}
```

在这个 Java 对象中定义了一个无参构造和有参构造，下面的分析将围绕有参构造进行。

在完成 Java 对象创建后需要编写 SpringXML 配置文件，文件名称为 spring-constructor-arg.xml，具体代码如下。

```
<?xml version = "1.0" encoding = "UTF - 8"?>
< beans xmlns = "http://www.springframework.org/schema/beans"
       xmlns: xsi = "http://www.w3.org/2001/XMLSchema - instance"
       xsi: schemaLocation = " http://www.springframework.org/schema/beans http://www.
springframework.org/schema/beans/spring - beans.xsd">

    < bean id = "people" class = "com.source.hot.ioc.book.pojo.PeopleBean">
        < constructor - arg index = "0" type = "java.lang.String" value = "zhangsan"/>
    </bean >

</beans >
```

最后来编写测试方法。

```
@Test
void testConstructArg() {
    ClassPathXmlApplicationContext context =
new ClassPathXmlApplicationContext("META-INF/spring-constructor-arg.xml");
    PeopleBean people = context.getBean("people",PeopleBean.class);
    assert people.getName().equals("zhangsan");
}
```

测试用例准备完成接下来进入源代码的分析。首先找到需要分析的方法签名 org.springframework.beans.factory.xml.BeanDefinitionParserDelegate#parseConstructorArgElements，找到方法签名后进入方法内部阅读内部的代码。

```
public void parseConstructorArgElements(Element beanEle,BeanDefinition bd) {
   //获取
   NodeList nl = beanEle.getChildNodes();
   for (int i = 0; i < nl.getLength(); i++) {
      Node node = nl.item(i);
      if (isCandidateElement(node) &&
nodeNameEquals(node,CONSTRUCTOR_ARG_ELEMENT)) {
         //解析 constructor-arg 下级标签
         parseConstructorArgElement((Element) node,bd);
      }
   }
}
```

这段方法中进行了另一个方法 parseConstructorArgElement 的引用，下面请阅读在 parseConstructorArgElement 中的第一部分代码。

```
//获取 index 属性
String indexAttr = ele.getAttribute(INDEX_ATTRIBUTE);
//获取 type 属性
String typeAttr = ele.getAttribute(TYPE_ATTRIBUTE);
//获取 name 属性
String nameAttr = ele.getAttribute(NAME_ATTRIBUTE);
```

在第一部分代码中的处理逻辑很好理解，其主要目的是获取 constructor-arg 标签中的 index、type 和 name 三个属性，当属性获取完成 Spring 会根据 index 属性是否存在做出两种不同的处理，在前文提到的测试用例中 index 属性是存在的情况，下面先进行 index 属性存在的分析。先阅读处理代码。

```
try {
   //构造参数的索引位置
   int index = Integer.parseInt(indexAttr);
   if (index < 0) {
      error("'index' cannot be lower than 0",ele);
   }
   else {
      try {
         //设置阶段,构造函数处理阶段
         this.parseState.push(new ConstructorArgumentEntry(index));
```

```
            //解析 property 标签
            Object value = parsePropertyValue(ele,bd,null);
            //创建构造函数的属性控制类
            ConstructorArgumentValues.ValueHolder valueHolder =
    new ConstructorArgumentValues.ValueHolder(value);
            if (StringUtils.hasLength(typeAttr)) {
               //类型设置
               valueHolder.setType(typeAttr);
            }
            if (StringUtils.hasLength(nameAttr)) {
               //名称设置
               valueHolder.setName(nameAttr);
            }
            //源设置
            valueHolder.setSource(extractSource(ele));
            if (bd.getConstructorArgumentValues().hasIndexedArgumentValue(index)) {
               error("Ambiguous constructor-arg entries for index " + index,ele);
            }
            else {
               //添加构造函数信息
               bd.getConstructorArgumentValues().addIndexedArgumentValue(index,valueHolder);
            }
         }
         finally {
            //移除当前阶段
            this.parseState.pop();
         }
      }
    }
    catch (NumberFormatException ex) {
       error("Attribute 'index' of tag 'constructor-arg' must be an integer",ele);
    }
```

在这段代码中需要重点关注的对象是 ConstructorArgumentValues，注意在这段方法的分析中对于标签 constructor-arg 的解析会需要进行 property 标签的解析，这一标签的分析会在 5.4.9 节中进行，本节对于 property 标签处理暂时跳过。

通过阅读源代码可以确认在标签中提取的 index、value 和 type 三个属性最终放到了 ConstructorArgumentValues.ValueHolder 对象和 ConstructorArgumentValues 中，进入调试阶段观察具体的数据存储，首先观察 type 和 value 的存储，信息如图 5.15 所示。

进一步观察 index、type 和 value 的存储，信息如图 5.16 所示。

现在对于 index 数据存在的情况相关源码分析结束，下面进入 index 不存在的情况，来看 Spring 是如何对这种情况进行处理的。首先需要编写一个 index 不填写的测试用例，然后再编写一个 SpringXML 配置文件，文件为 spring-constructor-arg.xml，具体代码如下。

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.
```

图 5.15 type 和 value 的存储信息

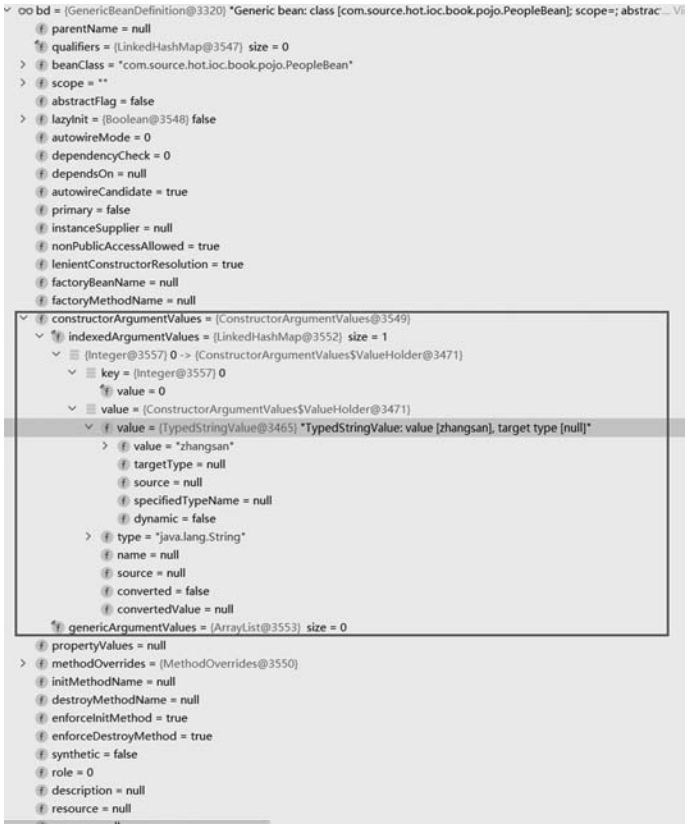

图 5.16 index、type 和 value 的存储信息

```
springframework.org/schema/beans/spring-beans.xsd">

    <bean id="people2" class="com.source.hot.ioc.book.pojo.PeopleBean">
        <constructor-arg name="name" type="java.lang.String" value="zhangsan">
        </constructor-arg>
    </bean>

</beans>
```

编写完成 SpringXML 配置文件后来编写测试方法。

```
@Test
void testConstructArgForName() {
    ClassPathXmlApplicationContext context =
new ClassPathXmlApplicationContext("META-INF/spring-constructor-arg.xml");
    PeopleBean people = context.getBean("people2",PeopleBean.class);
    assert people.getName().equals("zhangsan");
}
```

在 Spring 中这种模式称为参数名称绑定数据信息,下面进入这种模式的源代码分析,首先请阅读下面这段代码。

```
try {
   //设置阶段,构造函数处理阶段
   this.parseState.push(new ConstructorArgumentEntry());
   //解析 property 标签
   Object value = parsePropertyValue(ele,bd,null);
   //创建构造函数的属性控制类
   ConstructorArgumentValues.ValueHolder valueHolder =
new ConstructorArgumentValues.ValueHolder(value);
   if (StringUtils.hasLength(typeAttr)) {
      //类型设置
      valueHolder.setType(typeAttr);
   }
   if (StringUtils.hasLength(nameAttr)) {
      //名称设置
      valueHolder.setName(nameAttr);
   }
   //源设置
   valueHolder.setSource(extractSource(ele));
   //添加构造函数信息
   bd.getConstructorArgumentValues().addGenericArgumentValue(valueHolder);
}
finally {
   //移除当前阶段
   this.parseState.pop();
}
```

将这段处理形式和 index 存在情况下的模式进行对比,两种模式的差异是 index 的控制,其他信息处理相同。在这个方法中,关键对象是 ConstructorArgumentValues.ValueHolder,处理模式相同的情况下分析内容不多直接进入调试阶段来看经过处理后的 valueHolder 对象数据,信息如图 5.17 所示。

执行这段代码之后再进一步观察 BeanDefinition 的对象信息,如图 5.18 所示。

现在构造器的数据信息已经准备完毕,请思考一个问题:构造函数的信息都准备完毕如何使用?在 Spring 中对于这种情况的处理是交给一个方法进行处理的,方法的具体签名是 org.springframework.beans.BeanUtils#instantiateClass(java.lang.reflect.Constructor<T>,java.lang.Object…),可以根据需求对该方法进行阅读,本节仅该方法的调用链路整理出来,如图 5.19 所示。

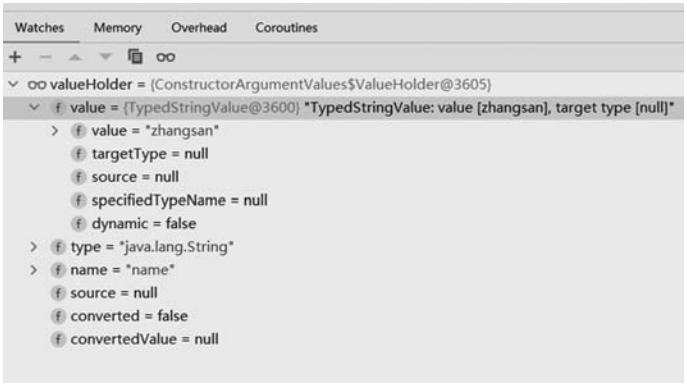

图 5.17 参数名称绑定模式下 valueHolder 对象信息

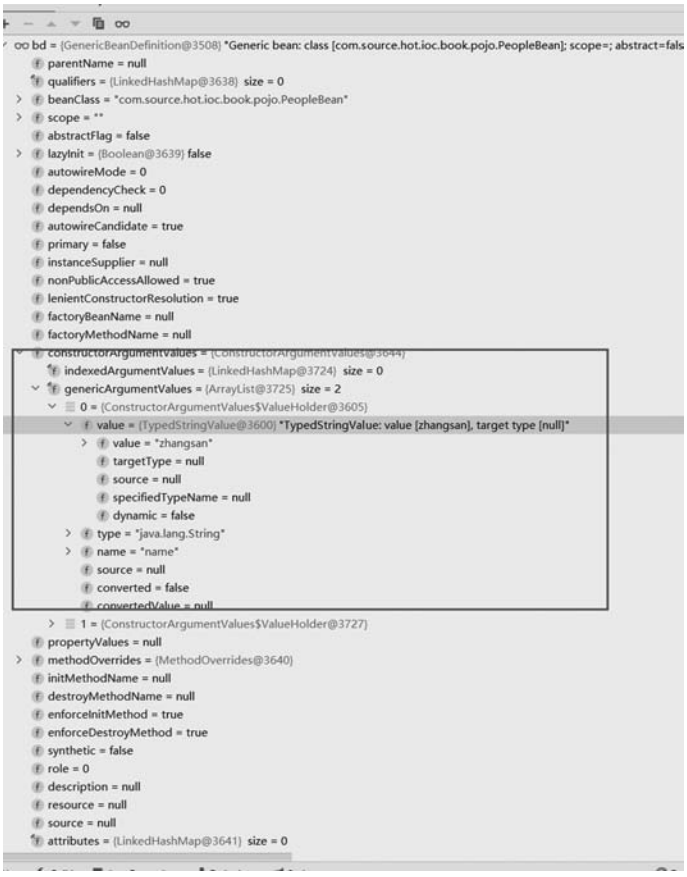

图 5.18 经过构造标签解析后 BeanDefinition 的数据信息

9. property 标签处理

接下来将进入 property 标签处理的源码分析。首先需要编写一个测试用例，第一步编写 SpringXML 配置文件，文件名为 spring-property.xml，该文件的代码内容如下。

```
<?xml version = "1.0" encoding = "UTF - 8"?>
```

图 5.19　instantiateClass 调用堆栈

```
<beans xmlns: xsi = "http://www.w3.org/2001/XMLSchema - instance"
       xmlns = "http://www.springframework.org/schema/beans"
       xsi: schemaLocation = " http://www. springframework. org/schema/beans  http://www.
springframework.org/schema/beans/spring - beans.xsd">

    <bean id = "people" class = "com.source.hot.ioc.book.pojo.PeopleBean">
        <property name = "name" value = "zhangsan"/>
    </bean>

</beans>
```

完成 SpringXML 配置文件的编写后需要进行单元测试的编写，具体单元测试代码如下。

```
@Test
void testProperty(){
    ClassPathXmlApplicationContext context =
```

```
new ClassPathXmlApplicationContext("META-INF/spring-property.xml");
    PeopleBean people = context.getBean("people",PeopleBean.class);
    assert people.getName().equals("zhangsan");
}
```

测试用例准备完毕，找到需要分析的方法 parsePropertyElements，该方法的方法签名是 org.springframework.beans.factory.xml.BeanDefinitionParserDelegate # parsePropertyElements，具体代码内容如下。

```
public void parsePropertyElements(Element beanEle,BeanDefinition bd) {
   NodeList nl = beanEle.getChildNodes();
   for (int i = 0; i < nl.getLength(); i++) {
      Node node = nl.item(i);
      //是否存在 property 标签
      if (isCandidateElement(node) && nodeNameEquals(node,PROPERTY_ELEMENT)) {
         //解析单个标签
         parsePropertyElement((Element) node,bd);
      }
   }
}
```

在这段方法中没有太多的处理细节，最终的能力提供者是 parsePropertyElement 方法，该方法是最终的分析目标，先阅读方法的内容。

```
public void parsePropertyElement(Element ele,BeanDefinition bd) {
   String propertyName = ele.getAttribute(NAME_ATTRIBUTE);
   if (!StringUtils.hasLength(propertyName)) {
      error("Tag 'property' must have a 'name' attribute",ele);
      return;
   }
   this.parseState.push(new PropertyEntry(propertyName));
   try {
      if (bd.getPropertyValues().contains(propertyName)) {
         error("Multiple 'property' definitions for property '" + propertyName + "'",ele);
         return;
      }
      //解析 property 标签
      Object val = parsePropertyValue(ele,bd,propertyName);
      //构造 PropertyValue 对象
      PropertyValue pv = new PropertyValue(propertyName,val);
      //解析元信息
      parseMetaElements(ele,pv);
      pv.setSource(extractSource(ele));
      //添加 pv 结构
      bd.getPropertyValues().addPropertyValue(pv);
   }
   finally {
      this.parseState.pop();
   }
}
```

在 parsePropertyElement 方法中总共有以下四步处理。

(1) 提取 property 标签的 name 属性值。

(2) 提取 property 标签的 value 属性值。

(3) 解析 property 标签中可能存在的 meta 标签。

(4) 将数据解析封装后设置给 BeanDefinition。

在整个处理流程中需要关注 property 标签的数据存储对象 PropertyValue。在上述四个处理步骤中第一步和第三步是一个比较简单的处理,比较复杂的内容是第二步操作,下面将对第二步操作进行详细分析。首先阅读 spring-beans.dtd 文件中对于 property 标签的定义。

```
<!ELEMENT property (
    description?,meta*,
    (bean | ref | idref | value | null | list | set | map | props)?
)>

<!ATTLIST property name CDATA #REQUIRED>

<!ATTLIST property ref CDATA #IMPLIED>

<!ATTLIST property value CDATA #IMPLIED>
```

进一步阅读 Spring 中对 property 标签的另一种定义方式,该方式的数据在 spring-beans.xsd 中存放,具体内容如下。

```
<xsd:element name="property" type="propertyType">
  <xsd:annotation>
    <xsd:documentation><![CDATA[
Bean definitions can have zero or more properties.
Property elements correspond to JavaBean setter methods exposed
by the bean classes. Spring supports primitives,references to other
beans in the same or related factories,lists,maps and properties.
    ]]></xsd:documentation>
  </xsd:annotation>
</xsd:element>


<xsd:complexType name="propertyType">
  <xsd:sequence>
      <xsd:element ref="description" minOccurs="0"/>
      <xsd:choice minOccurs="0" maxOccurs="1">
          <xsd:element ref="meta"/>
          <xsd:element ref="bean"/>
          <xsd:element ref="ref"/>
          <xsd:element ref="idref"/>
          <xsd:element ref="value"/>
          <xsd:element ref="null"/>
          <xsd:element ref="array"/>
          <xsd:element ref="list"/>
          <xsd:element ref="set"/>
```

```
            <xsd:element ref="map"/>
            <xsd:element ref="props"/>
            <xsd:any namespace="##other" processContents="strict"/>
        </xsd:choice>
    </xsd:sequence>
    <xsd:attribute name="name" type="xsd:string" use="required">
        <xsd:annotation>
            <xsd:documentation><![CDATA[
The name of the property, following JavaBean naming conventions.
    ]]></xsd:documentation>
        </xsd:annotation>
    </xsd:attribute>
    <xsd:attribute name="ref" type="xsd:string">
        <xsd:annotation>
            <xsd:documentation><![CDATA[
A short-cut alternative to a nested "<ref bean='...'/>".
    ]]></xsd:documentation>
        </xsd:annotation>
    </xsd:attribute>
    <xsd:attribute name="value" type="xsd:string">
        <xsd:annotation>
            <xsd:documentation><![CDATA[
A short-cut alternative to a nested "<value>...</value>" element.
    ]]></xsd:documentation>
        </xsd:annotation>
    </xsd:attribute>
</xsd:complexType>
```

在了解 property 的两种定义内容后下面对 parsePropertyValue 方法的处理流程进行分析，先阅读处理方法。

```
@Nullable
public Object parsePropertyValue(Element ele, BeanDefinition bd, @Nullable String propertyName) {
    String elementName = (propertyName != null ?
            "<property> element for property '" + propertyName + "'" :
            "<constructor-arg> element");

    //计算子节点
    //Should only have one child element: ref, value, list, etc.
    NodeList nl = ele.getChildNodes();
    Element subElement = null;
    for (int i = 0; i < nl.getLength(); i++) {
        Node node = nl.item(i);
        if (node instanceof Element &&
            !nodeNameEquals(node, DESCRIPTION_ELEMENT) &&
                !nodeNameEquals(node, META_ELEMENT)) {
            if (subElement != null) {
                error(elementName + " must not contain more than one sub-element", ele);
            }
            else {
                subElement = (Element) node;
```

```
            }
        }
    }

    //ref 属性是否存在
    boolean hasRefAttribute = ele.hasAttribute(REF_ATTRIBUTE);
    //value 属性是否存在
    boolean hasValueAttribute = ele.hasAttribute(VALUE_ATTRIBUTE);
    if ((hasRefAttribute && hasValueAttribute) ||
            ((hasRefAttribute || hasValueAttribute) && subElement != null)) {
        error(elementName +
                " is only allowed to contain either 'ref' attribute OR 'value' attribute OR sub-
element",ele);
    }

    if (hasRefAttribute) {
        //获取 ref 属性值
        String refName = ele.getAttribute(REF_ATTRIBUTE);
        if (!StringUtils.hasText(refName)) {
            error(elementName + " contains empty 'ref' attribute",ele);
        }
        //创建连接对象
        RuntimeBeanReference ref = new RuntimeBeanReference(refName);

        ref.setSource(extractSource(ele));
        return ref;
    }
    else if (hasValueAttribute) {
        //获取 value
        TypedStringValue valueHolder =
new TypedStringValue(ele.getAttribute(VALUE_ATTRIBUTE));
        valueHolder.setSource(extractSource(ele));
        return valueHolder;
    }
    else if (subElement != null) {
        return parsePropertySubElement(subElement,bd);
    }
    else {
        error(elementName + " must specify a ref or value",ele);
        return null;
    }
}
```

在 parsePropertyValue 方法中可以将其分为下面四个步骤进行阅读。

(1) 提取 property 下的子节点标签中非 description 和 meta 标签。

(2) 提取 property 中的 ref 属性,存在的情况下创建 RuntimeBeanReference 对象并返回。

(3) 提取 property 中的 value 属性,存在的情况下创建 TypedStringValue 对象并返回。

(4) 解析第一步中得到的子标签信息。

通过前文对 property 标签的两个定义文件中的内容可以得出 property 的子标签有 11 个,分别如下。

(1) meta。

(2) bean。

(3) ref。

(4) idref。

(5) value。

(6) null。

(7) array。

(8) list。

(9) set。

(10) map。

(11) props。

在这 11 个下级标签中,除了 meta 标签以外都是第一步中会提取得到的数据内容。第二步中获取 ref 属性值操作很简单,直接使用 getAttribute 即可获取,创建 RuntimeBeanReference 对象的过程也是一个简单的 new 操作。第三步和第二步的操作理论上相同,都是获取标签中的数据内容再进行特定对象的创建。第四步处理子标签的行为其实就是对上述标签的处理(除 meta 标签外),处理这些标签时所使用的方法是 parsePropertySubElement,下面将对这个方法做一个分析。

开始分析之前需要准备测试用例中需要的代码,首先需要的就是 JavaBean 对象,该 JavaBean 内容如下。

```
public class PeopleBean {
    private String name;

    private List < String > list;

    private Map < String,String > map;
    //省略构造函数,getter,setter
}
```

完成 JavaBean 的改造后进一步修改 SpringXML 配置文件的内容,配置文件名称为 spring-property. xml,文件内容如下。

```
< bean id = "people" class = "com. source. hot. ioc. book. pojo. PeopleBean">
    < property name = "name" value = "zhangsan"/>
    < property name = "list">
        < list value - type = "java. lang. String" merge = "default">
            < value > a </value >
            < value > b </value >
        </list >

    </property >
```

```
        <property name="map">
            <map key-type="java.lang.String" value-type="java.lang.String">
                <entry key="a" value="1"/>
            </map>
        </property>
    </bean>
```

编写完测试代码后先对 list 标签进行分析，首先找到 list 标签的处理方法 parseListElement，这里需要注意对于 list 标签、array 标签、set 标签最终都是交给 parseCollectionElements 方法来进行处理，在这个处理过程中就是将 value 标签的数据提取转换为对应的 Java 对象，下面来看经过 parseListElement 处理过后的数据内容，如图 5.20 所示。

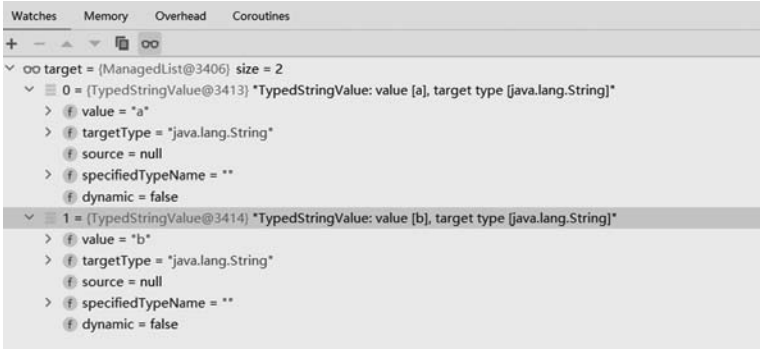

图 5.20　parseListElement 执行后的数据

完成了 list 标签的处理流程分析后接下来需要分析的就是 map 标签。在 SpringXML 配置文件中编写了 map 标签的 key-type 属性和 value-type 属性，以及子标签 entry 和 entry 标签的 key 属性和 value 属性。在当前的测试用例中并没有采用 ref 属性，仅以字符串作为直接字面量作为分析目标，处理 map 标签的方法是 parseMapElement。在处理 list 标签时采取的是字面量，最终得到的对象类型是 TypedStringValue。在 map 标签处理中用到的也是这个类型，只不过是从 list 存储转换成 map 存储。下面来看处理后的结果，如图 5.21 所示。

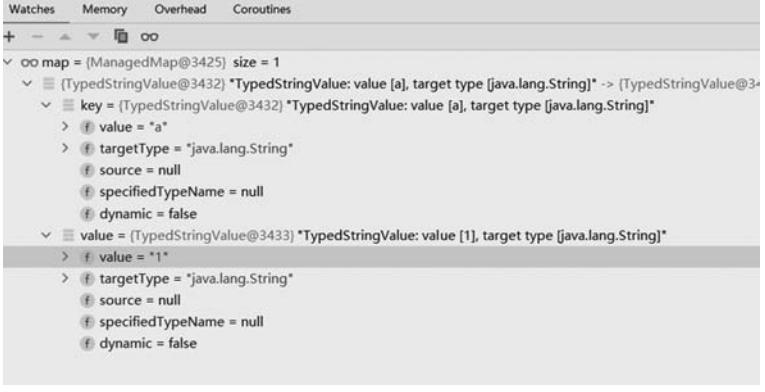

图 5.21　parseMapElement 方法执行后处理结果

现在完成了 map 标签的处理内容，剩下还有一些标签的处理，读者可以对这些具体标签的处理方法再做进一步的阅读。

10. qualifier 标签处理

接下来将进入 qualifier 标签的解析分析。首先需要编写一个测试用例，第一步创建一个 JavaBean 对象，该对象名称是 PeopleBeanTwo，详细代码如下。

```
public class PeopleBeanTwo {
    @Autowired
    @Qualifier("p1")
    private PeopleBean peopleBean;
    //省略 getter 和 setter
}
```

完成 JavaBean 编写后进一步编写 SpringXML 配置文件内容，该文件名称为 spring-qualifier.xml，详细代码如下。

```
<?xml version = "1.0" encoding = "UTF - 8"?>
< beans xmlns: xsi = "http://www.w3.org/2001/XMLSchema - instance"
        xmlns: context = "http://www.springframework.org/schema/context"
        xmlns = "http://www.springframework.org/schema/beans"
        xsi: schemaLocation = " http://www. springframework. org/schema/beans http://www.
springframework.org/schema/beans/spring - beans.xsd http://www.springframework.org/schema/
context https://www.springframework.org/schema/context/spring - context.xsd">

    < context: annotation - config/>

    < bean id = "p2" class = "com.source.hot.ioc.book.pojo.PeopleBeanTwo">

    </bean >

    < bean id = "peopleBean" class = "com.source.hot.ioc.book.pojo.PeopleBean">
        < property name = "name" value = "zhangsan"/>
        < qualifier value = "p1"/>
    </bean >
</beans >
```

最后编写单元测试。

```
@Test
void testQualifier() {
    ClassPathXmlApplicationContext context =
new ClassPathXmlApplicationContext("META - INF/spring - qualifier.xml");
    PeopleBeanTwo peopleTwo = context.getBean("p2",PeopleBeanTwo.class);
    assert peopleTwo.getPeopleBean().equals(
context.getBean("peopleBean",PeopleBean.class));
}
```

现在测试用例准备完毕，接下来找到需要分析的方法 parseQualifierElements，先阅读其中的代码内容。

```
public void parseQualifierElements(Element beanEle,AbstractBeanDefinition bd) {
    NodeList nl = beanEle.getChildNodes();
    for (int i = 0; i < nl.getLength(); i++) {
        Node node = nl.item(i);
        if (isCandidateElement(node) && nodeNameEquals(node,QUALIFIER_ELEMENT)) {
            //单个解析
            parseQualifierElement((Element) node,bd);
        }
    }
}
```

由于 Spring 是允许 qualifier 标签在 bean 标签下存在多个的，在 Spring 中对单个 qualifier 标签的处理是交给 parseQualifierElement 方法进行，下面先阅读 parseQualifierElement 的方法内容。

```
public void parseQualifierElement(Element ele,AbstractBeanDefinition bd) {
    //获取 type 属性
    String typeName = ele.getAttribute(TYPE_ATTRIBUTE);
    if (!StringUtils.hasLength(typeName)) {
        error("Tag 'qualifier' must have a 'type' attribute",ele);
        return;
    }
    //设置阶段,处理 qualifier 阶段
    this.parseState.push(new QualifierEntry(typeName));
    try {
        //自动注入对象创建
        AutowireCandidateQualifier qualifier = new AutowireCandidateQualifier(typeName);
        //设置源
        qualifier.setSource(extractSource(ele));
        //获取 value 属性
        String value = ele.getAttribute(VALUE_ATTRIBUTE);
        if (StringUtils.hasLength(value)) {
            //设置属性
            qualifier.setAttribute(AutowireCandidateQualifier.VALUE_KEY,value);
        }
        NodeList nl = ele.getChildNodes();
        for (int i = 0; i < nl.getLength(); i++) {
            Node node = nl.item(i);
            if (isCandidateElement(node) &&
nodeNameEquals(node,QUALIFIER_ATTRIBUTE_ELEMENT)) {
                Element attributeEle = (Element) node;
                //获取 key 属性
                String attributeName = attributeEle.getAttribute(KEY_ATTRIBUTE);
                //获取 value 属性
                String attributeValue = attributeEle.getAttribute(VALUE_ATTRIBUTE);
                if (StringUtils.hasLength(attributeName) &&
StringUtils.hasLength(attributeValue)) {
                    //key-value 属性映射
                    BeanMetadataAttribute attribute =
new BeanMetadataAttribute(attributeName,attributeValue);
```

```
                attribute.setSource(extractSource(attributeEle));
                //添加 qualifier 属性值
                qualifier.addMetadataAttribute(attribute);
            }
            else {
                error("Qualifier 'attribute' tag must have a
'name' and 'value'",attributeEle);
                return;
            }
        }
    }
    //添加 qualifier
    bd.addQualifier(qualifier);
  }
  finally {
    //移除阶段
    this.parseState.pop();
  }
}
```

这个解析过程就是对 qualifier 标签提取各个属性值再将其转换成 Java 对象，在 Spring 中 qualifier 标签对应的 Java 对象是 AutowireCandidateQualifier，下面来看经过解析后的数据，如图 5.22 所示。

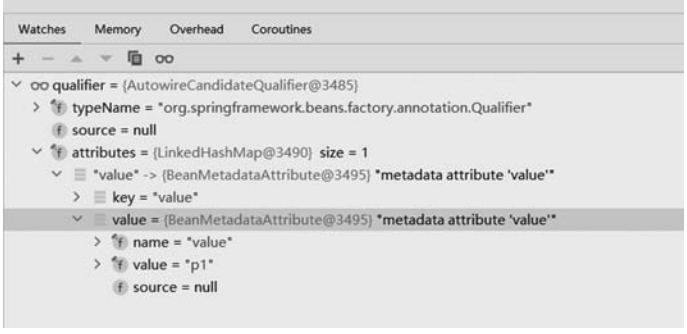

图 5.22 AutowireCandidateQualifier 数据信息

11. 设置 resource 和 source 属性

前文已经完成了 bean 标签的属性及下级标签的分析，最后还有两个属性需要设置，这两个属性分别是 resource 和 source，对于这两个属性的测试使用本章中提到的任何一个测试用例都可以进行测试。既然提到了 resource，那么什么是 resource(资源)呢？在 Java 工程中有一个 resources 文件夹，一般情况下认为该文件夹下的内容就是资源。在 Spring 中对于资源的定义可以简单地理解为 SpringXML 配置文件，不过还有其他的 resource，不只是配置文件，但是在此时需要设置的 resource 属性就是 SpringXML 配置文件，具体信息如图 5.23 所示。

在完成 resource 对象的设置后，Spring 对 source 进行了设置。在 Spring 中 source 的解析是交给 SourceExtractor 类进行处理的，下面了解一下 SourceExtractor 的类图，如图 5.24 所示。

图 5.23　resource 对象信息

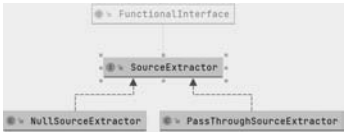

图 5.24　SourceExtractor 类图

通过类图可以发现 Spring 提供了以下两个关于 SourceExtractor 的实现。

(1) NullSourceExtractor 实现类直接将 null 作为解析结果进行返回。

(2) PassThroughSourceExtractor 实现类直接将需要解析的对象本身作为解析结果返回。

在 bean 标签解析的过程中,SourceExtractor 的具体实现类是 NullSourceExtractor,所以在设置 BeanDefinition 的 source 属性时设置的数据是 null,执行结果如图 5.25 所示。

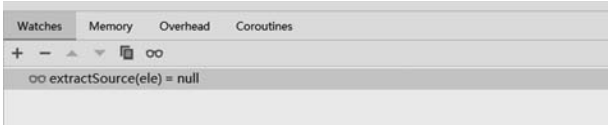

图 5.25　执行后 source 的数据内容

至此,对于 SpringXML 配置文件中关于 bean 标签的解析全部完成,现在 Spring 得到了 bean 标签对应的 BeanDefinition 对象。

5.3　BeanDefinition 装饰

在得到 BeanDefinition 对象后,Spring 对这个对象进行了装饰操作,接下来将对该处理进行分析。在 DefaultBeanDefinitionDocumentReader # processBeanDefinition 方法中可以找到 Spring 获取 BeanDefinition 对象后的操作:对象装饰(数据补充)。下面请阅读处理代码。

```
protected void processBeanDefinition(Element ele,BeanDefinitionParserDelegate delegate) {
   //创建 BeanDefinition
   BeanDefinitionHolder bdHolder = delegate.parseBeanDefinitionElement(ele);
   if (bdHolder != null) {
      //BeanDefinition 装饰
      bdHolder = delegate.decorateBeanDefinitionIfRequired(ele,bdHolder);
      try {
         //注册 BeanDefinition
         BeanDefinitionReaderUtils.registerBeanDefinition(bdHolder,getReaderContext().
getRegistry());
      }
      catch (BeanDefinitionStoreException ex) {
         getReaderContext().error("Failed to register bean definition with name '" +
               bdHolder.getBeanName() + "'",ele,ex);
      }
      //component 注册事件触发
```

```
            getReaderContext().fireComponentRegistered(new BeanComponentDefinition(bdHolder));
        }
    }
```

在这段方法中真正的处理是交给 decorateBeanDefinitionIfRequired 方法进行的，进一步追踪源代码阅读 decorateBeanDefinitionIfRequired 方法内容：

```
public BeanDefinitionHolder decorateBeanDefinitionIfRequired(
        Element ele, BeanDefinitionHolder originalDef, @Nullable BeanDefinition containingBd) {

    BeanDefinitionHolder finalDefinition = originalDef;

    NamedNodeMap attributes = ele.getAttributes();
    for (int i = 0; i < attributes.getLength(); i++) {
        Node node = attributes.item(i);
        finalDefinition = decorateIfRequired(node, finalDefinition, containingBd);
    }

    NodeList children = ele.getChildNodes();
    for (int i = 0; i < children.getLength(); i++) {
        Node node = children.item(i);
        if (node.getNodeType() == Node.ELEMENT_NODE) {
            finalDefinition = decorateIfRequired(node, finalDefinition, containingBd);
        }
    }
    return finalDefinition;
}
```

在这段方法中可以整理出下面两种情况需要对 BeanDefinition 进行装饰。

(1) 当 bean 标签存在属性时。

(2) 当 bean 标签存在下级标签时。

在 decorateBeanDefinitionIfRequired 方法中确认了需要进行装饰的原因，最终进行装饰处理的方法是 decorateIfRequired，该方法就是需要分析的重点了，请先阅读源码。

```
public BeanDefinitionHolder decorateIfRequired(
        Node node, BeanDefinitionHolder originalDef,
@Nullable BeanDefinition containingBd) {

    //命名空间 url
    String namespaceUri = getNamespaceURI(node);
    if (namespaceUri != null && !isDefaultNamespace(namespaceUri)) {
        NamespaceHandler handler
= this.readerContext.getNamespaceHandlerResolver().resolve(namespaceUri);
        if (handler != null) {
            //命名空间进行装饰
            BeanDefinitionHolder decorated =
                    handler.decorate(node, originalDef, new ParserContext(this.readerContext,
this, containingBd));
            if (decorated != null) {
                return decorated;
```

```
            }
        }
        else if (namespaceUri.startsWith("http://www.springframework.org/schema/")) {
            error("Unable to locate Spring NamespaceHandler for XML schema namespace [" +
namespaceUri + "]",node);
        }
        else {
            if (logger.isDebugEnabled()) {
                logger.debug("No Spring NamespaceHandler found
for XML schema namespace [" + namespaceUri + "]");
            }
        }
    }
    return originalDef;
}
```

在这个方法的处理过程中可以发现比较熟悉的对象是 NamespaceHandler，在第 4 章中对于 NamespaceHandler 接口做了关于自定义标签解析的相关内容，下面将延续第 4 章中的测试用例补充实现 decorate 方法，修改 UserXsdNamespaceHandler 类，具体修改后内容如下。

```
public class UserXsdNamespaceHandler extends NamespaceHandlerSupport {

    @Override
    public void init() {
        registerBeanDefinitionParser("user_xsd",new UserXsdParser());
    }

    @Override
    public BeanDefinitionHolder decorate(Node node, BeanDefinitionHolder definition,
ParserContext parserContext) {
        BeanDefinition beanDefinition = definition.getBeanDefinition();
        beanDefinition.getPropertyValues().addPropertyValue("namespace","namespace");
        return definition;
    }
}
```

修改 SpringXML 配置文件，修改后内容如下。

```
<?xml version = "1.0" encoding = "UTF - 8"?>
< beans xmlns = "http://www.springframework.org/schema/beans"
       xmlns: xsi = "http://www.w3.org/2001/XMLSchema - instance"
       xmlns: myname = "http://www.huifer.com/schema/user"
       xsi: schemaLocation = " http://www.springframework.org/schema/beans http://www.
springframework.org/schema/beans/spring - beans.xsd
   http://www.huifer.com/schema/user http://www.huifer.com/schema/user.xsd
">
    < bean id = "p1" class = "com.source.hot.ioc.book.pojo.PeopleBean">
        < myname: user_xsd id = "testUserBean" name = "huifer" idCard = "123"/>
    </bean >
</beans >
```

完成 SpringXML 配置文件编写后编写单元测试，具体单元测试代码如下。

```
@Test
void testXmlCustom() {
    ClassPathXmlApplicationContext context =
new ClassPathXmlApplicationContext("META-INF/custom-xml.xml");
    UserXsd testUserBean = context.getBean("testUserBean",UserXsd.class);
    assert testUserBean.getName().equals("huifer");
    assert testUserBean.getIdCard().equals("123");
    context.close();
}
```

在第 4 章中已经讲述过从 namespaceUri 转换成 NamespaceHandler 的过程，相信读者对下面这段代码已经有了一定的了解。

```
BeanDefinitionHolder decorated = handler.decorate(node,originalDef,new
ParserContext(this.readerContext,this,containingBd));
```

这段代码的主要目的是调用在测试用例中所编写的 UserXsdNamespaceHandler # decorate 方法，下面来看经过装饰后的 BeanDefinition 对象信息，如图 5.26 所示。

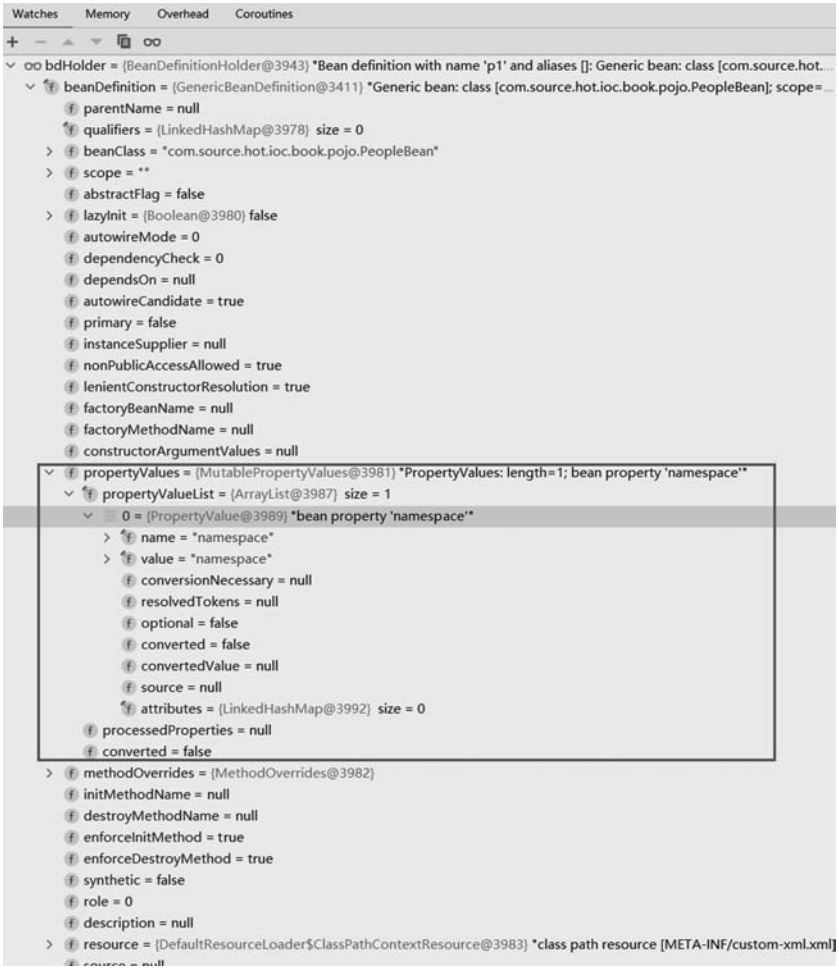

图 5.26 装饰后的 BeanDefinition

至此，对于 BeanDefinition 的装饰过程分析完成。

5.4 BeanDefinition 细节

本节将会对 BeanDefinition 的各个属性进行介绍，首先阅读 BeanDefinition 的类图，如图 5.27 所示。

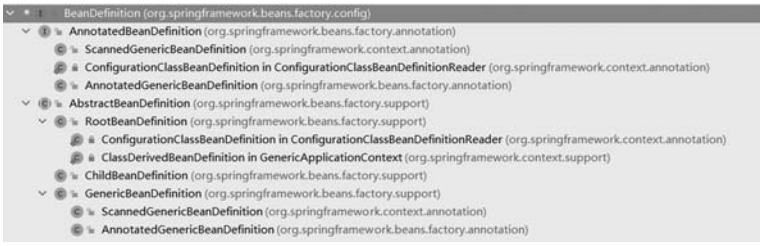

图 5.27 BeanDefinition 类图

从 BeanDefinition 的类图上可以知道所有 BeanDefinition 的根对象是 AbstractBeanDefinition，在这个对象中存有 BeanDefinition 的大量数据字段。

5.4.1 AbstractBeanDefinition 属性

下面来看 AbstractBeanDefinition 的属性，如表 5.2 所示，列举了 AbstractBeanDefinition 对象中属性对应的类型和含义。

表 5.2 AbstractBeanDefinition 对象属性表

属性名称	属性类型	属性含义
beanClass	Object	存储 Bean 类型
scope	String	作用域，默认""，常见作用域有 singleton、prototype、request、session 和 globalsession
lazyInit	Boolean	是否懒加载
abstractFlag	boolean	是否是 abstract 修饰的
autowireMode	int	自动注入方式，常见的注入方式有 no、byName、byType 和 constructor
dependencyCheck	int	依赖检查级别，常见的依赖检查级别有 DEPENDENCY_CHECK_NONE、DEPENDENCY_CHECK_OBJECTS、DEPENDENCY_CHECK_SIMPLE 和 DEPENDENCY_CHECK_ALL
dependsOn	String[]	依赖的 BeanName 列表
autowireCandidate	boolean	是否自动注入，默认值：true
primary	boolean	是否是主要的，通常在同类型多个备案的情况下使用
instanceSupplier	Supplier	Bean 实例提供器

续表

属性名称	属性类型	属性含义
nonPublicAccessAllowed	boolean	是否禁止公开访问
lenientConstructorResolution	String	
factoryBeanName	String	工厂 Bean 名称
factoryMethodName	String	工厂函数名称
constructorArgumentValues	ConstructorArgumentValues	构造函数对象，可以是 XML 中 constructor-arg 标签的解析结果，也可以是 Java 中构造函数的解析结果
propertyValues	MutablePropertyValues	属性列表
methodOverrides	MethodOverrides	重写的函数列表
initMethodName	String	Bean 初始化的函数名称
destroyMethodName	String	Bean 摧毁的函数名称
enforceInitMethod	boolean	是否强制执行 initMethodName 对应的 Java 方法
enforceDestroyMethod	boolean	是否强制执行 destroyMethodName 对应的 Java 方法
synthetic	boolean	合成标记
role	int	Spring 中的角色，一般有 ROLE_APPLICATION、ROLE_SUPPORT 和 ROLE_INFRASTRUCTURE
description	String	Bean 的描述信息
resource	Resource	资源对象

在 AbstractBeanDefinition 属性表中 beanClass 属性的类型是 Object，通常情况下会有 String 类型和 Class 类型，属性 scope 默认为单例，属性 propertyValues 含义为属性列表，它是一个 key-value 结构，用于存储开发者对 Bean 对象的属性定义，key 表示属性名称，value 表示属性值。

5.4.2 RootBeanDefinition 属性

RootBeanDefinition 属性详见表 5.3，列举了 RootBeanDefinition 对象中属性对应的类型和含义。

表 5.3 RootBeanDefinition 属性

属性名称	属性类型	属性含义
constructorArgumentLock	Object	构造阶段的锁
postProcessingLock	Object	后置处理阶段的锁
stale	boolean	是否需要重新合并定义
allowCaching	boolean	是否缓存
isFactoryMethodUnique	boolean	工厂方法是否唯一
targetType	ResolvableType	目标类型
resolvedTargetType	Class	目标类型，Bean 的类型

续表

属性名称	属性类型	属性含义
isFactoryBean	Boolean	是否是工厂 Bean
factoryMethodReturnType	ResolvableType	工厂方法返回值
factoryMethodToIntrospect	Method	
resolvedConstructorOrFactoryMethod	Executable	执行器
constructorArgumentsResolved	boolean	构造函数的参数是否需要解析
resolvedConstructorArguments	Object[]	解析过的构造参数列表
preparedConstructorArguments	Object[]	未解析的构造参数列表
postProcessed	boolean	是否需要进行后置处理
beforeInstantiationResolved	Boolean	是否需要进行前置处理
decoratedDefinition	BeanDefinitionHolder	BeanDefinition 持有者
qualifiedElement	AnnotatedElement	qualified 注解信息
externallyManagedConfigMembers	Set < Member >	外部配置的成员
externallyManagedInitMethods	Set < String >	外部的初始化方法列表
externallyManagedDestroyMethods	Set < String >	外部的摧毁方法列表

在 RootBeanDefinition 成员变量中会有一些成对出现的内容，如 constructorArgumentsResolved、resolvedConstructorArguments 和 preparedConstructorArguments，它们用于对构造参数解析状态进行控制，externallyManagedConfigMembers、externallyManagedInitMethods 和 externallyManagedDestroyMethods，它们用于对外部成员依赖进行控制。

5.4.3 ChildBeanDefinition 属性

ChildBeanDefinition 属性详见表 5.4，列举了 ChildBeanDefinition 对象中属性对应的类型和含义。

表 5.4 ChildBeanDefinition 属性

属性名称	属性类型	属性含义
parentName	String	父 BeanDefinition 的名称

ChildBeanDefinition 对象是 AbstractBeanDefinition 的子类，具备 AbstractBeanDefinition 的所有属性，但是它比 AbstractBeanDefinition 增加了 parentName 属性用于指向父 BeanDefinition 对象。

5.4.4 GenericBeanDefinition 属性

GenericBeanDefinition 属性详见表 5.5，列举了 GenericBeanDefinition 对象中属性对应的类型和含义。

表 5.5 GenericBeanDefinition 属性

属性名称	属性类型	属性含义
parentName	String	父 BeanDefinition 的名称

GenericBeanDefinition 对象是 AbstractBeanDefinition 的子类，具备 AbstractBeanDefinition 的所有属性，但是它比 AbstractBeanDefinition 增加了 parentName 属性用于指向父 BeanDefinition 对象。

5.4.5 AnnotatedGenericBeanDefinition 属性

AnnotatedGenericBeanDefinition 属性详见表 5.6，该表列举了 AnnotatedGenericBeanDefinition 对象中属性对应的类型和含义。

表 5.6 AnnotatedGenericBeanDefinition 属性

属性名称	属性类型	属性含义
metadata	AnnotationMetadata	注解元信息
factoryMethodMetadata	MethodMetadata	工厂函数的元信息

AnnotatedGenericBeanDefinition 对象继承自 GenericBeanDefinition 对象，在 GenericBeanDefinition 的基础上增加了注解元信息和工厂函数的元信息变量，这两个变量为 Spring 注解模式开发中的注解 Bean 定义提供了充分支持。

小结

在本章中主要对 parseBeanDefinitionElement 方法进行分析（完整方法签名：org.springframework.beans.factory.xml.BeanDefinitionParserDelegate # parseBeanDefinitionElement(org.w3c.dom.Element,java.lang.String,org.springframework.beans.factory.config.BeanDefinition)），下面对整个处理过程进行总结，处理过程分为以下 12 步。

(1) 处理 className 和 parent 属性。

(2) 创建基本的 BeanDefinition 对象，具体类：AbstractBeanDefinition、GenericBeanDefinition。

(3) 读取 bean 标签的属性，为 BeanDefinition 对象进行赋值。

(4) 处理描述标签 description。

(5) 处理 meta 标签。

(6) 处理 lookup-override 标签。

(7) 处理 replaced-method 标签。

(8) 处理 constructor-arg 标签。

(9) 处理 property 标签。

(10) 处理 qualifier 标签。

(11) 设置资源对象。

(12) 设置 source 属性。

上述 12 个步骤是 Spring 中对于 bean 标签的解析细节，在完成 bean 标签的解析后 Spring 会对 BeanDefinition 对象进行装饰，具体装饰行为操作如下。

(1) 读取标签所对应的 namespaceUri。

(2) 根据 namespaceUri 在 NamespaceHandler 容器中寻找对应的 NamespaceHandler。

(3) 调用 NamespaceHandler 所提供的 decorate 方法。

在完成 BeanDefinition 对象的创建及其属性赋值后需要对 BeanDefinition 对象进行注册,有关 BeanDefinition 对象的注册内容请查阅第 7 章。

最后将 bean 标签解析的入口方法的签名贴出,读者可以根据需求进行查阅。入口签名为 org. springframework. beans. factory. xml. DefaultBeanDefinitionDocumentReader # processBeanDefinition,入口方法如下。

```
protected void processBeanDefinition(Element ele,BeanDefinitionParserDelegate delegate) {
   //创建 BeanDefinition
   BeanDefinitionHolder bdHolder = delegate.parseBeanDefinitionElement(ele);
   if (bdHolder != null) {
      //BeanDefinition 装饰
      bdHolder = delegate.decorateBeanDefinitionIfRequired(ele,bdHolder);
      try {
         //注册 BeanDefinition
         BeanDefinitionReaderUtils.registerBeanDefinition(bdHolder,getReaderContext().
getRegistry());
      }
      catch (BeanDefinitionStoreException ex) {
         getReaderContext().error("Failed to register bean definition with name '" +
               bdHolder.getBeanName() + "'",ele,ex);
      }
      //component 注册事件触发
      getReaderContext().fireComponentRegistered(new BeanComponentDefinition(bdHolder));
   }
}
```