svip.iocoder.cn /Spring/IoC-parse-BeanDefinitions-for-constructor-arg-and-property-and-qualifier/

# 【死磕 Spring】—— IoC 之解析 标签：constructor-arg、property、qualifier

---

**本文主要基于 Spring 5.0.6.RELEASE**

摘要: 原创出处 http://cmsblogs.com/?p=2754 「小明哥」，谢谢！

作为「小明哥」的忠实读者，「老艿艿」略作修改，记录在理解过程中，参考的资料。

---

上篇博客 《【死磕 Spring】—— IoC 之解析 标签：meta、lookup-method、replace-method》 分析了 `meta`、`lookup-method`、`replace-method` 三个子元素。这篇博客，我们来分析 `constructor-arg`、`property`、`<qualifier>` 三个子元素。

# 1. constructor-arg 子元素

## 1.1 示例

举个小栗子，代码如下：

```
public class StudentService {

    private String name;

    private Integer age;

    private BookService bookService;

    StudentService(String name, Integer age, BookService bookService){

        this.name = name;

        this.age = age;

        this.bookService = bookService;

    }

}

<bean id="bookService" class="org.springframework.core.service.BookService"/>

<bean id="studentService"
class="org.springframework.core.service.StudentService">
```

```
    <constructor-arg index="0" value="chenssy"/>
    <constructor-arg name="age" value="100"/>
    <constructor-arg name="bookService" ref="bookService"/>
</bean>
```

StudentService 定义一个构造函数，配置文件中使用 `constructor-arg` 元素对其配置，该元素可以实现对 StudentService 自动寻找对应的构造函数，并在初始化的时候将值当做参数进行设置。

## 1.2 parseConstructorArgElements

`#parseConstructorArgElements(Element beanEle, BeanDefinition bd)` 方法，完成 `constructor-arg` 子元素的解析。代码如下：

```
// BeanDefinitionParserDelegate.java

public void parseConstructorArgElements(Element beanEle, BeanDefinition bd) {
        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)) { // 标签名为 constructor-arg
                        parseConstructorArgElement((Element) node, bd);
                }
        }
}
```

遍历所有子元素，如果为 `constructor-arg` 标签，则调用 `#parseConstructorArgElement(Element ele, BeanDefinition bd)` 方法，进行解析。

## 1.3 parseConstructorArgElement

```
// BeanDefinitionParserDelegate.java

public void parseConstructorArgElement(Element ele, BeanDefinition bd) {
    // 提取 index、type、name 属性值
    String indexAttr = ele.getAttribute(INDEX_ATTRIBUTE);
    String typeAttr = ele.getAttribute(TYPE_ATTRIBUTE);
    String nameAttr = ele.getAttribute(NAME_ATTRIBUTE);
    if (StringUtils.hasLength(indexAttr)) {
        try {
            // 如果有 index
            int index = Integer.parseInt(indexAttr);
            if (index < 0) {
                error("'index' cannot be lower than 0", ele);
            } else {
                try {
                    // <1>
                    this.parseState.push(new ConstructorArgumentEntry(index));
                    // <2> 解析 ele 对应属性元素
                    Object value = parsePropertyValue(ele, bd, null);
                    // <3> 根据解析的属性元素构造一个 ValueHolder 对象
                    ConstructorArgumentValues.ValueHolder valueHolder = new
ConstructorArgumentValues.ValueHolder(value);
                    if (StringUtils.hasLength(typeAttr)) {
```

```
                        valueHolder.setType(typeAttr);
                    }
                    if (StringUtils.hasLength(nameAttr)) {
                        valueHolder.setName(nameAttr);
                    }
                    valueHolder.setSource(extractSource(ele));
                    // 不允许重复指定相同参数
                    if
(bd.getConstructorArgumentValues().hasIndexedArgumentValue(index)) {
                        error("Ambiguous constructor-arg entries for index " +
index, ele);
                    } else {
                        // <4> 加入到 indexedArgumentValues 中

bd.getConstructorArgumentValues().addIndexedArgumentValue(index, valueHolder);
                    }
                } finally {
                    this.parseState.pop();
                }
            }
        } catch (NumberFormatException ex) {
            error("Attribute 'index' of tag 'constructor-arg' must be an
integer", ele);
        }
    } else {
        try {
            this.parseState.push(new ConstructorArgumentEntry());
            // 解析 ele 对应属性元素
            Object value = parsePropertyValue(ele, bd, null);
            // 根据解析的属性元素构造一个 ValueHolder 对象
            ConstructorArgumentValues.ValueHolder valueHolder = new
ConstructorArgumentValues.ValueHolder(value);
            if (StringUtils.hasLength(typeAttr)) {
                valueHolder.setType(typeAttr);
            }
            if (StringUtils.hasLength(nameAttr)) {
                valueHolder.setName(nameAttr);
            }
            valueHolder.setSource(extractSource(ele));
            // 加入到 indexedArgumentValues 中

bd.getConstructorArgumentValues().addGenericArgumentValue(valueHolder);
        } finally {
            this.parseState.pop();
        }
    }
}
```

首先获取 index、type、name 三个属性值，然后根据是否存在 index 来区分，执行后续逻辑。其实两者逻辑都**差不多**，总共分为如下几个步骤（以有 index 为例）：

1. 在 <1> 处，构造 ConstructorArgumentEntry 对象并将其加入到 ParseState 队列中。ConstructorArgumentEntry 表示构造函数的参数。
2. 在 <2> 处，调用 #parsePropertyValue(Element ele, BeanDefinition bd, String propertyName) 方法，解析 constructor-arg 子元素，返回结果值。详细解析，见「1.4 parsePropertyValue」 。

3. 在 <3> 处，根据解析的结果值，构造ConstructorArgumentValues.ValueHolder 实例对象，并将 `type`、`name` 设置到 ValueHolder 中
4. 在 <4> 处，最后，将 ValueHolder 实例对象添加到 `indexedArgumentValues` 集合中。

无 `index` 的处理逻辑差不多，只有几点不同：

- 构造 ConstructorArgumentEntry 对象时是调用**无参**构造函数
- 最后是将 ValueHolder 实例添加到 `genericArgumentValues` 集合中。

# 1.4 parsePropertyValue

调用 `#parsePropertyValue(Element ele, BeanDefinition bd, String propertyName)` 方法，解析 `constructor-arg` 子元素，返回结果值。代码如下：

```
/**
 * Get the value of a property element. May be a list etc.
 * Also used for constructor arguments, "propertyName" being null in this case.
 */
@Nullable
public Object parsePropertyValue(Element ele, BeanDefinition bd, @Nullable String
propertyName) {
    String elementName = (propertyName != null ?
            "<property> element for property '" + propertyName + "'" :
            "<constructor-arg> element");

    // <1> 查找子节点中，是否有 ref、value、list 等元素
    // 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);
        // meta 、description 不处理
        if (node instanceof Element && !nodeNameEquals(node, DESCRIPTION_ELEMENT)
&&
                !nodeNameEquals(node, META_ELEMENT)) {
            // Child element is what we're looking for.
            if (subElement != null) {
                error(elementName + " must not contain more than one sub-
element", ele);
            } else {
                subElement = (Element) node;
            }
        }
    }

    // <1> 是否有 ref 属性
    boolean hasRefAttribute = ele.hasAttribute(REF_ATTRIBUTE);
    // <1> 是否有 value 属性
    boolean hasValueAttribute = ele.hasAttribute(VALUE_ATTRIBUTE);
    // <1> 多个元素存在，报错，存在冲突。
    if ((hasRefAttribute && hasValueAttribute) || // 1. ref 和 value 都存在
            ((hasRefAttribute || hasValueAttribute) && subElement != null)) { //
2. ref he value 存在一，并且 subElement 存在
        error(elementName +
                " is only allowed to contain either 'ref' attribute OR 'value'
```

```
attribute OR sub-element", ele);
    }

    // <2> 将 ref 属性值，构造为 RuntimeBeanReference 实例对象
    if (hasRefAttribute) {
        String refName = ele.getAttribute(REF_ATTRIBUTE);
        if (!StringUtils.hasText(refName)) {
            error(elementName + " contains empty 'ref' attribute", ele);
        }
        RuntimeBeanReference ref = new RuntimeBeanReference(refName);
        ref.setSource(extractSource(ele));
        return ref;
    // <3> 将 value 属性值，构造为 TypedStringValue 实例对象
    } else if (hasValueAttribute) {
        TypedStringValue valueHolder = new
TypedStringValue(ele.getAttribute(VALUE_ATTRIBUTE));
        valueHolder.setSource(extractSource(ele));
        return valueHolder;
    // <4> 解析子元素
    } else if (subElement != null) {
        return parsePropertySubElement(subElement, bd);
    } else {
        // Neither child element nor "ref" or "value" attribute found.
        error(elementName + " must specify a ref or value", ele);
        return null;
    }
}
```

1. 在 <1> 处，提取 `constructor-arg` 的子元素、`ref` 属性值和 `value` 属性值，对其进行判断。以下两种情况是不允许存在的：
    1. `ref` 和 `value` 属性同时存在 。
    2. 存在 `ref` 或者 `value` 且又有子元素。
2. 在 <2> 处，若存在 `ref` 属性，则获取其值并将其封装进 `org.springframework.beans.factory.config.RuntimeBeanReference` 实例对象中。
3. 在 <3> 处，若存在 `value` 属性，则获取其值并将其封装进 `org.springframework.beans.factory.config.TypedStringValue` 实例对象中。
4. 在 <4> 处，如果子元素不为空，则调用 `#parsePropertySubElement(Element ele, BeanDefinition bd)` 方法，对子元素进一步解析。详细解析，见 「1.5 parsePropertySubElement」 中。

## 1.5 parsePropertySubElement

对于 `constructor-arg` 子元素的嵌套子元素，需要调用 `#parsePropertySubElement(Element ele, BeanDefinition bd)` 方法，进一步处理。

```
/**
 * Parse a value, ref or collection sub-element of a property or
 * constructor-arg element.
 * @param ele subelement of property element; we don't know which yet
 * @param defaultValueType the default type (class name) for any
 * {@code <value>} tag that might be created
 */
```

```
@Nullable
public Object parsePropertySubElement(Element ele, @Nullable BeanDefinition bd,
@Nullable String defaultValueType) {
        if (!isDefaultNamespace(ele)) {
                return parseNestedCustomElement(ele, bd);
        } else if (nodeNameEquals(ele, BEAN_ELEMENT)) { // bean 标签
                BeanDefinitionHolder nestedBd = parseBeanDefinitionElement(ele,
bd);
                if (nestedBd != null) {
                        nestedBd = decorateBeanDefinitionIfRequired(ele,
nestedBd, bd);
                }
                return nestedBd;
        } else if (nodeNameEquals(ele, REF_ELEMENT)) { // ref 标签
                // A generic reference to any name of any bean.
                String refName = ele.getAttribute(BEAN_REF_ATTRIBUTE);
                boolean toParent = false;
                if (!StringUtils.hasLength(refName)) {
                        // A reference to the id of another bean in a parent
context.
                        refName = ele.getAttribute(PARENT_REF_ATTRIBUTE);
                        toParent = true;
                        if (!StringUtils.hasLength(refName)) {
                                error("'bean' or 'parent' is required for <ref>
element", ele);
                                return null;
                        }
                }
                if (!StringUtils.hasText(refName)) {
                        error("<ref> element contains empty target attribute",
ele);
                        return null;
                }
                RuntimeBeanReference ref = new RuntimeBeanReference(refName,
toParent);
                ref.setSource(extractSource(ele));
                return ref;
        } else if (nodeNameEquals(ele, IDREF_ELEMENT)) { // idref 标签
                return parseIdRefElement(ele);
        } else if (nodeNameEquals(ele, VALUE_ELEMENT)) { // value 标签
                return parseValueElement(ele, defaultValueType);
        } else if (nodeNameEquals(ele, NULL_ELEMENT)) { // null 标签
                // It's a distinguished null value. Let's wrap it in a
TypedStringValue
                // object in order to preserve the source location.
                TypedStringValue nullHolder = new TypedStringValue(null);
                nullHolder.setSource(extractSource(ele));
                return nullHolder;
        } else if (nodeNameEquals(ele, ARRAY_ELEMENT)) { // array 标签
                return parseArrayElement(ele, bd);
        } else if (nodeNameEquals(ele, LIST_ELEMENT)) { // list 标签
                return parseListElement(ele, bd);
        } else if (nodeNameEquals(ele, SET_ELEMENT)) { // set 标签
                return parseSetElement(ele, bd);
        } else if (nodeNameEquals(ele, MAP_ELEMENT)) { // map 标签
                return parseMapElement(ele, bd);
        } else if (nodeNameEquals(ele, PROPS_ELEMENT)) { // props 标签
                return parsePropsElement(ele);
        } else { // 未知标签
                error("Unknown property sub-element: [" + ele.getNodeName() +
```

```
"]", ele);
                return null;
        }
}
```

上面对各个子类进行分类处理，详细情况，如果各位有兴趣，可以移步源码进行深一步的探究。本文，暂时不做深入分析。

# 2. property 子元素

## 2.1 示例

我们一般使用如下方式，来使用 `property` 子元素。

```
<bean id="studentService"
class="org.springframework.core.service.StudentService">
    <property name="name" value="chenssy"/>
    <property name="age" value="18"/>
</bean>
```

## 2.2 parsePropertyElements

对于 property 子元素的解析，Spring 调用 `parsePropertyElements(Element beanEle, BeanDefinition bd)` 方法。代码如下：

```
/**
 * Parse property sub-elements of the given bean element.
 */
public void parsePropertyElements(Element beanEle, BeanDefinition bd) {
        NodeList nl = beanEle.getChildNodes();
        for (int i = 0; i < nl.getLength(); i++) {
                Node node = nl.item(i);
                if (isCandidateElement(node) && nodeNameEquals(node,
PROPERTY_ELEMENT)) { // property 标签
                        parsePropertyElement((Element) node, bd);
                }
        }
}
```

和 `constructor-arg` 子元素差不多，同样是“提取”( 遍历 )所有的 `property` 的子元素，然后调用 `#parsePropertyElement((Element ele, BeanDefinition b)` 进行解析。

## 2.3 parsePropertyElement

```
/**
 * Parse a property element.
 */
public void parsePropertyElement(Element ele, BeanDefinition bd) {
    // 获取 name 属性
    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 {
            // 如果存在相同的 name ，报错
            if (bd.getPropertyValues().contains(propertyName)) {
                error("Multiple 'property' definitions for property '" + propertyName
    + "'", ele);
                return;
            }
            // 解析属性值
            Object val = parsePropertyValue(ele, bd, propertyName);
            // 创建 PropertyValue 对象
            PropertyValue pv = new PropertyValue(propertyName, val);
            parseMetaElements(ele, pv);
            pv.setSource(extractSource(ele));
            // 添加到 PropertyValue 集合中
            bd.getPropertyValues().addPropertyValue(pv);
        } finally {
            this.parseState.pop();
        }
    }
```

与解析 `constructor-arg` 子元素步骤差不多：

- 调用 `#parsePropertyElement((Element ele, BeanDefinition b)` 方法， 解析子元素属性值。
- 然后，根据该值构造 PropertyValue 实例对象。
- 最后，将 PropertyValue 添加到 BeanDefinition 中的 MutablePropertyValues 中。

# 3. qualifier 子元素

> 老艿艿：小明哥貌似忘记写 `<qualifier>` 标签的解析，不过实际情况下也用的比较少，所以老艿艿，在这里稍微简单写下。
>
> 不感兴趣的胖友，可以直接跳过这块内容。

## 3.1 示例

见 《Spring 注解实现Bean依赖注入之 @Qualifier》 文章。

## 3.2 parseQualifierElement

`#parseQualifierElement(Element ele, AbstractBeanDefinition bd)` 方法，完成 `qualifier` 子元素的解析。代码如下：

```
    /**
     * Parse a qualifier element.
```

```
     */
    public void parseQualifierElement(Element ele, AbstractBeanDefinition bd) {
        // 解析 type 属性
        String typeName = ele.getAttribute(TYPE_ATTRIBUTE);
        if (!StringUtils.hasLength(typeName)) { // 必须有 type
            error("Tag 'qualifier' must have a 'type' attribute", ele);
            return;
        }
        this.parseState.push(new QualifierEntry(typeName));
        try {
            // 创建 AutowireCandidateQualifier 对象
            AutowireCandidateQualifier qualifier = new
    AutowireCandidateQualifier(typeName);
            qualifier.setSource(extractSource(ele));
            // 解析 value 属性，并设置到 AutowireCandidateQualifier 中
            String value = ele.getAttribute(VALUE_ATTRIBUTE);
            if (StringUtils.hasLength(value)) {
                qualifier.setAttribute(AutowireCandidateQualifier.VALUE_KEY, value);
            }
            // 遍历子节点
            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)) { // attribute 标签
                    Element attributeEle = (Element) node;
                    String attributeName = attributeEle.getAttribute(KEY_ATTRIBUTE);
    // attribute 标签的 key 属性
                    String attributeValue =
    attributeEle.getAttribute(VALUE_ATTRIBUTE); // attribute 标签的 value 属性
                    if (StringUtils.hasLength(attributeName) &&
    StringUtils.hasLength(attributeValue)) {
                        // 创建 BeanMetadataAttribute 对象
                        BeanMetadataAttribute attribute = new
    BeanMetadataAttribute(attributeName, attributeValue);
                        attribute.setSource(extractSource(attributeEle));
                        // 添加到 attributes 中
                        qualifier.addMetadataAttribute(attribute);
                    } else {
                        error("Qualifier 'attribute' tag must have a 'name' and
    'value'", attributeEle);
                        return;
                    }
                }
            }
            // 添加到 qualifiers 中
            bd.addQualifier(qualifier);
        } finally {
            this.parseState.pop();
        }
    }
```

代码比较简单，胖友自己研究下哈。