# 【死磕 Spring】—— IOC 之解析 bean 标签：constructor-arg、property 子元素 (http://cmsblogs.com/?p=2754)

2018-09-26 分类：死磕 Spring (http://cmsblogs.com/?cat=206) 阅读(6623) 评论(2)

> 原文出自：http://cmsblogs.com (http://cmsblogs.com)

上篇博客(【死磕 Spring】—— IOC 之解析 bean 标签：meta、lookup-method、replace-method (http://cmsblogs.com/?p=2736))分析了 meta 、 lookup-method、replace-method 三个子元素,这篇博客分析 constructor-arg 、property、qualifier 三个子元素。

## constructor-arg 子元素

举个小栗子：

属性值被封装到BeanDefinition的属性propertyValues内部

```
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。`parseConstructorArgElements()` 方法完成 constructor-arg 子元素的解析。

```
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)) {
            parseConstructorArgElement((Element) node, bd);
        }
    }
}
```

遍历所有子元素，如果为 constructor-arg 则调用 `parseConstructorArgElement()` 进行解析。

```
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);

    // 如果有index
    if (StringUtils.hasLength(indexAttr)) {
        try {
            int index = Integer.parseInt(indexAttr);
            if (index < 0) {
                error("'index' cannot be lower than 0", ele);
            }
            else {
                try {
                    // 构造一个 ConstructorArgumentEntry 并将其加入到 ParseState 中
                    this.parseState.push(new ConstructorArgumentEntry(index));

                    // 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));

                    // 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 {
                        // 加入到 indexedArgumentValues 中国
                         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());
                    Object value = parsePropertyValue(ele, bd, null);
                    ConstructorArgumentValues.ValueHolder valueHolder = new ConstructorArgumentValues.ValueHo
lder(value);
                    if (StringUtils.hasLength(typeAttr)) {
                        valueHolder.setType(typeAttr);
                    }
                    if (StringUtils.hasLength(nameAttr)) {
                        valueHolder.setName(nameAttr);
                    }
                    valueHolder.setSource(extractSource(ele));
                    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。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. 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. 调用 `parsePropertyValue()` 解析 constructor-arg 子元素，返回结果值
3. 根据解析的结果值构造 `ConstructorArgumentValues.ValueHolder` 实例对象
4. 将 type、name 封装到 `ConstructorArgumentValues.ValueHolder` 中，然后将 ValueHolder 实例对象添加到 indexedArgumentValues 中。即构造器参数值被封装到BeanDefinition的 indexedArgumentValues 属性中

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 中。 `parsePropertyValue()` 对子元素进一步解析。

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

    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;
            }
        }
    }

    // 解析 ref 元素
    boolean hasRefAttribute = ele.hasAttribute(REF_ATTRIBUTE);
    // 解析 value 元素
    boolean hasValueAttribute = ele.hasAttribute(VALUE_ATTRIBUTE);

    // constructor-arg 子元素有两种情况不存在
    // 1. 即存在 ref 又存在 value
    // 2. 存在 ref 或者 value，但是又有子元素
    if ((hasRefAttribute && hasValueAttribute) ||
            ((hasRefAttribute || hasValueAttribute) && subElement != null)) {
        error(elementName +
                " is only allowed to contain either 'ref' attribute OR 'value' attribute OR sub-eleme
nt", 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);
        }
        // 将 ref 属性值构造为 RuntimeBeanReference 实例对象
        RuntimeBeanReference ref = new RuntimeBeanReference(refName);
        ref.setSource(extractSource(ele));
        return ref;
    }
    else if (hasValueAttribute) {
        // 解析 value 属性值，构造 TypedStringValue 实例对象
        TypedStringValue valueHolder = new TypedStringValue(ele.getAttribute(VALUE_ATTRIBUTE));
```

```
            valueHolder.setSource(extractSource(ele));
            return valueHolder;
        }
        else if (subElement != null) {
            // 解析子元素
            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. 提取 constructor-arg 子元素的 ref 和 value 的属性值，对其进行判断，以下两种情况是不允许存在的
   - ref 和 value 属性同时存在
   - 存在 ref 或者 value 且又有子元素
2. 若存在 ref 属性，则获取其值并将其封装进 RuntimeBeanReference 实例对象中
3. 若存在 value 属性，则获取其值并将其封装进 TypedStringValue 实例对象中
4. 如果子元素不为空，则调用 `parsePropertySubElement()` 进行子元素进一步处理

对于 constructor-arg 子元素的嵌套子元素，需要调用 `parsePropertySubElement()` 进一步处理。

```
    public Object parsePropertySubElement(Element ele, @Nullable BeanDefinition bd) {
        return parsePropertySubElement(ele, bd, null);
    }

    public Object parsePropertySubElement(Element ele, @Nullable BeanDefinition bd, @Nullable String defa
ultValueType) {
        if (!isDefaultNamespace(ele)) {
            return parseNestedCustomElement(ele, bd);
        }
        else if (nodeNameEquals(ele, BEAN_ELEMENT)) {
            BeanDefinitionHolder nestedBd = parseBeanDefinitionElement(ele, bd);
            if (nestedBd != null) {
                nestedBd = decorateBeanDefinitionIfRequired(ele, nestedBd, bd);
            }
            return nestedBd;
        }
        else if (nodeNameEquals(ele, REF_ELEMENT)) {
            // 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)) {
            return parseIdRefElement(ele);
        }
        else if (nodeNameEquals(ele, VALUE_ELEMENT)) {
            return parseValueElement(ele, defaultValueType);
        }
        else if (nodeNameEquals(ele, NULL_ELEMENT)) {
            // 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)) {
```

```
            return parseArrayElement(ele, bd);
        }
        else if (nodeNameEquals(ele, LIST_ELEMENT)) {
            return parseListElement(ele, bd);
        }
        else if (nodeNameEquals(ele, SET_ELEMENT)) {
            return parseSetElement(ele, bd);
        }
        else if (nodeNameEquals(ele, MAP_ELEMENT)) {
            return parseMapElement(ele, bd);
        }
        else if (nodeNameEquals(ele, PROPS_ELEMENT)) {
            return parsePropsElement(ele);
        }
        else {
            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。

## property 子元素

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>
```

对于 property 子元素的解析，Spring 调用 `parsePropertyElements()` 。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);
        if (isCandidateElement(node) && nodeNameEquals(node, PROPERTY_ELEMENT)) {
            parsePropertyElement((Element) node, bd);
        }
    }
}
```

和 constructor-arg 子元素差不多，同样是提取所有的 property 的子元素，然后调用 `parsePropertyElement()` 进行分析。

```
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
        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);
        // 根据解析的属性值构造 PropertyValue 实例对象
        PropertyValue pv = new PropertyValue(propertyName, val);
        parseMetaElements(ele, pv);
        pv.setSource(extractSource(ele));
        // 添加到 MutablePropertyValues 中
        bd.getPropertyValues().addPropertyValue(pv);
    }
    finally {
        this.parseState.pop();
    }
}
```

与解析 constructor-arg 子元素步骤差不多。调用 `parsePropertyValue()` 解析子元素属性值，然后根据该值构造 PropertyValue 实例对象并将其添加到 BeanDefinition 中的 MutablePropertyValues 中。

---

- 【死磕 Spring】----- IOC 之解析 bean 标签：开启解析进程 (http://cmsblogs.com/?p=2731)
- 【死磕 Spring】----- IOC 之解析 bean 标签：BeanDefinition (http://cmsblogs.com/?p=2734)
- 【死磕 Spring】—— IOC 之解析 bean 标签：meta、lookup-method、replace-method (http://cmsblogs.com/?p=2736)



标签： Spring 源码解析 (http://cmsblogs.com/?tag=spring-%e6%ba%90%e7%a0%81%e8%a7%a3%e6%9e%90)

死磕Spring (http://cmsblogs.com/?tag=%e6%ad%bb%e7%a3%95spring)

**chenssy (http://cmsblogs.com/?author=1)**