芋道源码 —— 知识星球

我是一段不羁的公告！
记得给艿艿这 3 个项目加油，添加一个 STAR 噢。
https://github.com/YunaiV/SpringBoot-Labs
https://github.com/YunaiV/onemall
https://github.com/YunaiV/ruoyi-vue-pro
2019-03-04
Spring

【死磕 Spring】—— IoC 之解析 <bean> 标签：开启解析进程

本文主要基于 Spring 5.0.6.RELEASE

摘要: 原创出处 http://cmsblogs.com/?p=2731 「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。

---

import 标签解析完毕了，我们一起来看看 Spring 中最复杂也是最重要的标签 bean 标签的解析过程。

1. processBeanDefinition

在方法 #parseDefaultElement(...) 方法中，如果遇到标签为 bean 时，则调用 #processBeanDefinition(Element ele, BeanDefinitionParserDelegate delegate) 方法，进行 bean 标签的解析。代码如下：

```
// DefaultBeanDefinitionDocumentReader.java

/**
 * Process the given bean element, parsing the bean definition
 * and registering it with the registry.
 */
protected void processBeanDefinition(Element ele, BeanDefinitionParserDelegate delegate) {
    // 进行 bean 元素解析。
    // <1> 如果解析成功，则返回 BeanDefinitionHolder 对象。而 BeanDefinitionHolder 为 name 和 alias 的 BeanDefinition
    // 如果解析失败，则返回 null 。
    BeanDefinitionHolder bdHolder = delegate.parseBeanDefinitionElement(ele);
    if (bdHolder != null) {
        // <2> 进行自定义标签处理
        bdHolder = delegate.decorateBeanDefinitionIfRequired(ele, bdHolder);
        try {
            // <3> 进行 BeanDefinition 的注册
            // Register the final decorated instance.
            BeanDefinitionReaderUtils.registerBeanDefinition(bdHolder, getReaderContext().getRegistry());
        } catch (BeanDefinitionStoreException ex) {
```

```
                getReaderContext().error("Failed to register bean definition with name '" +
                        bdHolder.getBeanName() + "'", ele, ex);
            }
            // <4> 发出响应事件，通知相关的监听器，已完成该 Bean 标签的解析。
            // Send registration event.
            getReaderContext().fireComponentRegistered(new BeanComponentDefinition(bdHolder));
        }
    }
```

整个过程分为四个步骤：

1. 调用 `BeanDefinitionParserDelegate#parseBeanDefinitionElement(Element ele, BeanDefinitionParserDelegate delegate)` 方法，进行元素解析。
   如果解析失败，则返回 `null`，错误由 ProblemReporter 处理。
   如果解析成功，则返回 BeanDefinitionHolder 实例 `bdHolder` 。BeanDefinitionHolder 为持有 `name` 和 `alias` 的 BeanDefinition。
   详细解析，见 「2. parseBeanDefinitionElement」 。
2. 若实例 `bdHolder` 不为空，则调用 `BeanDefinitionParserDelegate#decorateBeanDefinitionIfRequired(Element ele, BeanDefinitionHolder bdHolder)` 方法，进行自定义标签处理。
3. 解析完成后，则调用 `BeanDefinitionReaderUtils#registerBeanDefinition(BeanDefinitionHolder definitionHolder, BeanDefinitionRegistry registry)` 方法，对 `bdHolder` 进行 BeanDefinition 的注册。
4. 发出响应事件，通知相关的监听器，完成 Bean 标签解析。

2. parseBeanDefinitionElement

`BeanDefinitionParserDelegate#parseBeanDefinitionElement(Element ele, BeanDefinitionParserDelegate delegate)` 方法，进行 `<bean>` 元素解析。代码如下：

```
// BeanDefinitionParserDelegate.java

/**
 * Parses the supplied {@code <bean>} element. May return {@code null}
 * if there were errors during parse. Errors are reported to the
 * {@link org.springframework.beans.factory.parsing.ProblemReporter}.
 */
@Nullable
public BeanDefinitionHolder parseBeanDefinitionElement(Element ele) {
    return parseBeanDefinitionElement(ele, null);
}

/**
 * Parses the supplied {@code <bean>} element. May return {@code null}
 * if there were errors during parse. Errors are reported to the
 * {@link org.springframework.beans.factory.parsing.ProblemReporter}.
 *
 * @param containingBean TODO 芋艿，需要进一步确认
 */
@Nullable
public BeanDefinitionHolder parseBeanDefinitionElement(Element ele, @Nullable BeanDefinition containingBean) {
    // <1> 解析 id 和 name 属性
    String id = ele.getAttribute(ID_ATTRIBUTE);
    String nameAttr = ele.getAttribute(NAME_ATTRIBUTE);
```

```
// <1> 计算别名集合
List<String> aliases = new ArrayList<>();
if (StringUtils.hasLength(nameAttr)) {
    String[] nameArr = StringUtils.tokenizeToStringArray(nameAttr, MULTI_VALUE_ATTRIBUTE_DELIMITERS);
    aliases.addAll(Arrays.asList(nameArr));
}

// <3.1> beanName ，优先，使用 id
String beanName = id;
// <3.2> beanName ，其次，使用 aliases 的第一个
if (!StringUtils.hasText(beanName) && !aliases.isEmpty()) {
    beanName = aliases.remove(0); // 移除出别名集合
    if (logger.isTraceEnabled()) {
        logger.trace("No XML 'id' specified - using '" + beanName +
                "' as bean name and " + aliases + " as aliases");
    }
}

// <2> 检查 beanName 的唯一性
if (containingBean == null) {
    checkNameUniqueness(beanName, aliases, ele);
}

// <4> 解析属性，构造 AbstractBeanDefinition 对象
AbstractBeanDefinition beanDefinition = parseBeanDefinitionElement(ele, beanName, containingBean);
if (beanDefinition != null) {
    // <3.3> beanName ，再次，使用 beanName 生成规则
    if (!StringUtils.hasText(beanName)) {
        try {
            if (containingBean != null) {
                // <3.3> 生成唯一的 beanName
                beanName = BeanDefinitionReaderUtils.generateBeanName(
                        beanDefinition, this.readerContext.getRegistry(), true);
            } else {
                // <3.3> 生成唯一的 beanName
                beanName = this.readerContext.generateBeanName(beanDefinition);
                // TODO 芋艿，需要进一步确认
                // Register an alias for the plain bean class name, if still possible,
                // if the generator returned the class name plus a suffix.
                // This is expected for Spring 1.2/2.0 backwards compatibility.
                String beanClassName = beanDefinition.getBeanClassName();
                if (beanClassName != null &&
                        beanName.startsWith(beanClassName) && beanName.length() > beanClassName.length() &&
                        !this.readerContext.getRegistry().isBeanNameInUse(beanClassName)) {
                    aliases.add(beanClassName);
                }
            }
            if (logger.isTraceEnabled()) {
                logger.trace("Neither XML 'id' nor 'name' specified - " +
                        "using generated bean name [" + beanName + "]");
            }
        } catch (Exception ex) {
            error(ex.getMessage(), ele);
            return null;
        }
    }
    // <5> 创建 BeanDefinitionHolder 对象
    String[] aliasesArray = StringUtils.toStringArray(aliases);
    return new BeanDefinitionHolder(beanDefinition, beanName, aliasesArray);
}
```

```
    return null;
}
```

这个方法还没有对 bean 标签进行解析，只是在解析动作之前做了一些功能架构，主要的工作有：

<1> 处，解析 id、name 属性，确定 aliases 集合

<2> 处，检测 beanName 是否唯一。代码如下：

```
/**
 * 已使用 Bean 名字的集合
 *
 * Stores all used bean names so we can enforce uniqueness on a per
 * beans-element basis. Duplicate bean ids/names may not exist within the
 * same level of beans element nesting, but may be duplicated across levels.
 */
private final Set<String> usedNames = new HashSet<>();

/**
 * Validate that the specified bean name and aliases have not been used already
 * within the current level of beans element nesting.
 */
protected void checkNameUniqueness(String beanName, List<String> aliases, Element beanElement) {
 // 寻找是否 beanName 已经使用
    String foundName = null;
 if (StringUtils.hasText(beanName) && this.usedNames.contains(beanName)) {
        foundName = beanName;
    }
 if (foundName == null) {
        foundName = CollectionUtils.findFirstMatch(this.usedNames, aliases);
    }
 // 若已使用，使用 problemReporter 提示错误
 if (foundName != null) {
        error("Bean name '" + foundName + "' is already used in this <beans> element", beanElement);
    }

 // 添加到 usedNames 集合
 this.usedNames.add(beanName);
 this.usedNames.addAll(aliases);
}
```

这里有必要说下 beanName 的命名规则：

- <3.1> 处，如果 id 不为空，则 beanName = id 。
- <3.2> 处，如果 id 为空，但是 aliases 不空，则 beanName 为 aliases 的第一个元素
- <3.3> 处，如果两者都为空，则根据默认规则来设置 beanName 。因为默认规则不是本文的重点，所以暂时省略。感兴趣的胖友，自己研究下哈。

<4> 处，调用 #parseBeanDefinitionElement(Element ele, String beanName, BeanDefinition containingBean) 方法，对属性进行解析并封装成 AbstractBeanDefinition 实例 beanDefinition 。详细解析，见「2.1 parseBeanDefinitionElement」 。

<5> 处，根据所获取的信息（beanName、aliases、beanDefinition）构造 BeanDefinitionHolder 实例对象并返回。其中，BeanDefinitionHolder 的简化代码如下：

```
    /**
     * BeanDefinition 对象
     */
    private final BeanDefinition beanDefinition;
    /**
     * Bean 名字
     */
    private final String beanName;
    /**
     * 别名集合
     */
    @Nullable
    private final String[] aliases;
```

TODO 芋艿，需要进一步确认，未来参考下 《Spring专题之IOC源码分析》 ，进行细化。

2.1 parseBeanDefinitionElement

#parseBeanDefinitionElement(Element ele, String beanName, BeanDefinition containingBean) 方法，对属性进行解析并封装成 AbstractBeanDefinition 实例，代码如下：

```
/**
 * Parse the bean definition itself, without regard to name or aliases. May return
 * {@code null} if problems occurred during the parsing of the bean definition.
 */
@Nullable
public AbstractBeanDefinition parseBeanDefinitionElement(
        Element ele, String beanName, @Nullable BeanDefinition containingBean) {

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

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

    try {
        // 创建用于承载属性的 AbstractBeanDefinition 实例
        AbstractBeanDefinition bd = createBeanDefinition(className, parent);

        // 解析默认 bean 的各种属性
        parseBeanDefinitionAttributes(ele, beanName, containingBean, bd);
        // 提取 description
        bd.setDescription(DomUtils.getChildElementValueByTagName(ele, DESCRIPTION_ELEMENT));

        // tips：
        // 下面的一堆是解析 <bean>.....</bean> 内部的子元素，
        // 解析出来以后的信息都放到 bd 的属性中
```

```
            // 解析元数据 <meta />
            parseMetaElements(ele, bd);
            // 解析 lookup-method 属性 <lookup-method />
            parseLookupOverrideSubElements(ele, bd.getMethodOverrides());
            // 解析 replaced-method 属性 <replaced-method />
            parseReplacedMethodSubElements(ele, bd.getMethodOverrides());

            // 解析构造函数参数 <constructor-arg />
            parseConstructorArgElements(ele, bd);
            // 解析 property 子元素 <property />
            parsePropertyElements(ele, bd);
            // 解析 qualifier 子元素 <qualifier />
            parseQualifierElements(ele, bd);

            bd.setResource(this.readerContext.getResource());
            bd.setSource(extractSource(ele));

            return bd;
        } catch (ClassNotFoundException ex) {
            error("Bean class [" + className + "] not found", ele, ex);
        } catch (NoClassDefFoundError err) {
            error("Class that bean class [" + className + "] depends on not found", ele, err);
        } catch (Throwable ex) {
            error("Unexpected failure during bean definition parsing", ele, ex);
        } finally {
            this.parseState.pop();
        }

        return null;
    }
```

到这里，bean 标签的所有属性我们都可以看到其解析的过程，也就说到这里我们已经解析一个基本可用的 BeanDefinition 。

2.2 createBeanDefinition

#createBeanDefinition(String className, String parentName) 方法，创建 AbstractBeanDefinition 对象。代码如下：

```
    /**
     * Create a bean definition for the given class name and parent name.
     * @param className the name of the bean class
     * @param parentName the name of the bean's parent bean
     * @return the newly created bean definition
     * @throws ClassNotFoundException if bean class resolution was attempted but failed
     */
    protected AbstractBeanDefinition createBeanDefinition(@Nullable String className, @Nullable String parentName)
            throws ClassNotFoundException {
        return BeanDefinitionReaderUtils.createBeanDefinition(
                parentName, className, this.readerContext.getBeanClassLoader());
    }
```

3. 小节

由于解析过程 bean 标签的属性较为漫长，篇幅较大，为了更好的观看体验，将这篇博文进行拆分。

下篇博客主要介绍 BeanDefinition ，以及解析默认 bean 标签的各种属性的过程，即 #parseBeanDefinitionAttributes(Element ele, String beanName, BeanDefinition containingBean, AbstractBeanDefinition bd) 方法。

文章目录