title: 精尽 Dubbo 源码分析 —— XML 配置 date: 2018-01-19 tags: categories: Dubbo permalink: Dubbo/configuration-xml

---

摘要: 原创出处 http://www.iocoder.cn/Dubbo/configuration-xml/ 「芋道源码」欢迎转载，保留摘要，谢谢！

---

1. 概述

在 Dubbo 提供的几种方式中，**XML 配置**肯定是大家最熟悉的方式。

如果胖友不熟悉，可以查看如下文档：

- 《Dubbo 用户指南 —— XML 配置》
- 《Dubbo 用户指南 —— schema 配置参考手册》

XML 配置，自定义 `<dubbo: />` 标签，基于 **Spring XML** 进行解析。如果不了解的胖友，可以查看如下文档：

- 《Spring 源码阅读 —— XML 文件默认标签的解析》
- 《Spring 源码阅读 —— XML 自定义标签的解析》

2. 定义

2.1 sprng.schemas

Dubbo 在 `dubbo-spring-config` 的 `META-INF/spring.schemas` 定义如下：

```
http\://code.alibabatech.com/schema/dubbo/dubbo.xsd=META-INF/dubbo.xsd
```

- `xmlns` 为 `http://code.alibabatech.com/schema/dubbo/dubbo.xsd`
- `xds` 为 `META-INF/dubbo.xsd`

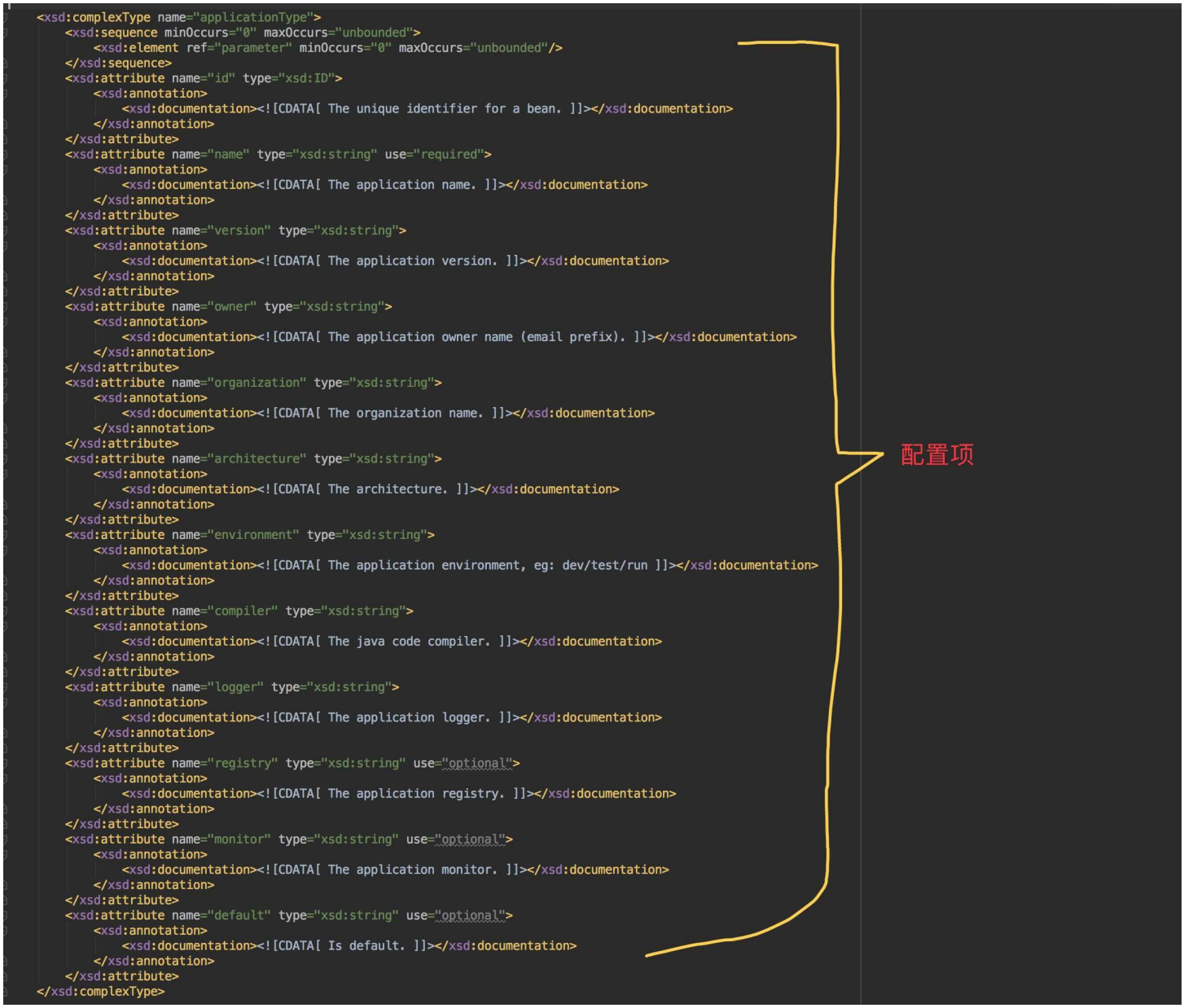
2.2 dubbo.xsd

`dubbo.xsd` 定义如下：

- `<xsd:element name="" />` ，定义了**元素的名称**。例如， `<xsd:element name="application" />` 对应 `<dubbo:application />` 。
- `<xsd:element type="" />` ，定义了**内建数据类型的名称**。例如， `<xsd:element`

`type="applicationType" />` 对应 `<xsd:complexType name="applicationType" />` 。

- `<xsd:complexType name="">` ，定义了**复杂类型**。例如 `<xsd:complexType name="applicationType" />` 如下：




```
<xsd:complexType name="applicationType">
    <xsd:sequence minOccurs="0" maxOccurs="unbounded">
        <xsd:element ref="parameter" minOccurs="0" maxOccurs="unbounded"/>
    </xsd:sequence>
    <xsd:attribute name="id" type="xsd:ID">
        <xsd:annotation>
            <xsd:documentation><![CDATA[ The unique identifier for a bean. ]]></xsd:documentation>
        </xsd:annotation>
    </xsd:attribute>
    <xsd:attribute name="name" type="xsd:string" use="required">
        <xsd:annotation>
            <xsd:documentation><![CDATA[ The application name. ]]></xsd:documentation>
        </xsd:annotation>
    </xsd:attribute>
    <xsd:attribute name="version" type="xsd:string">
        <xsd:annotation>
            <xsd:documentation><![CDATA[ The application version. ]]></xsd:documentation>
        </xsd:annotation>
    </xsd:attribute>
    <xsd:attribute name="owner" type="xsd:string">
        <xsd:annotation>
            <xsd:documentation><![CDATA[ The application owner name (email prefix). ]]></xsd:documentation>
        </xsd:annotation>
    </xsd:attribute>
    <xsd:attribute name="organization" type="xsd:string">
        <xsd:annotation>
            <xsd:documentation><![CDATA[ The organization name. ]]></xsd:documentation>
        </xsd:annotation>
    </xsd:attribute>
    <xsd:attribute name="architecture" type="xsd:string">
        <xsd:annotation>
            <xsd:documentation><![CDATA[ The architecture. ]]></xsd:documentation>
        </xsd:annotation>
    </xsd:attribute>
    <xsd:attribute name="environment" type="xsd:string">
        <xsd:annotation>
            <xsd:documentation><![CDATA[ The application environment, eg: dev/test/run ]]></xsd:documentation>
        </xsd:annotation>
    </xsd:attribute>
    <xsd:attribute name="compiler" type="xsd:string">
        <xsd:annotation>
            <xsd:documentation><![CDATA[ The java code compiler. ]]></xsd:documentation>
        </xsd:annotation>
    </xsd:attribute>
    <xsd:attribute name="logger" type="xsd:string">
        <xsd:annotation>
            <xsd:documentation><![CDATA[ The application logger. ]]></xsd:documentation>
        </xsd:annotation>
    </xsd:attribute>
    <xsd:attribute name="registry" type="xsd:string" use="optional">
        <xsd:annotation>
            <xsd:documentation><![CDATA[ The application registry. ]]></xsd:documentation>
        </xsd:annotation>
    </xsd:attribute>
    <xsd:attribute name="monitor" type="xsd:string" use="optional">
        <xsd:annotation>
            <xsd:documentation><![CDATA[ The application monitor. ]]></xsd:documentation>
        </xsd:annotation>
    </xsd:attribute>
    <xsd:attribute name="default" type="xsd:string" use="optional">
        <xsd:annotation>
            <xsd:documentation><![CDATA[ Is default. ]]></xsd:documentation>
        </xsd:annotation>
    </xsd:attribute>
</xsd:complexType>
```

2.3 spring.handlers

`spring.handlers` 定义如下：

```
http\://code.alibabatech.com/schema/dubbo=com.alibaba.dubbo.config.spring.schema.DubboNamespaceHandler
```

- 定义了 Dubbo 的 XML Namespace 的处理器 DubboNamespaceHandler 。

2.4 DubboNamespaceHandler

`com.alibaba.dubbo.config.spring.schema.DubboNamespaceHandler` ，实现 `org.springframework.beans.factory.xml.NamespaceHandlerSupport` **抽象类**，Dubbo 的 XML Namespace 的处理器。

在 `#init()` 方法，定义了每个 `<xsd:element />` 对应的 `org.springframework.beans.factory.xml.BeanDefinitionParser` ，代码如下：

```
@Override
public void init() {
    registerBeanDefinitionParser("application", new DubboBeanDefinitionParser(Appli
cationConfig.class, true));
    registerBeanDefinitionParser("module", new DubboBeanDefinitionParser(ModuleConf
ig.class, true));
    registerBeanDefinitionParser("registry", new DubboBeanDefinitionParser(Registry
Config.class, true));
    registerBeanDefinitionParser("monitor", new DubboBeanDefinitionParser(MonitorCo
nfig.class, true));
    registerBeanDefinitionParser("provider", new DubboBeanDefinitionParser(Provider
Config.class, true));
    registerBeanDefinitionParser("consumer", new DubboBeanDefinitionParser(Consumer
Config.class, true));
    registerBeanDefinitionParser("protocol", new DubboBeanDefinitionParser(Protocol
Config.class, true));
    registerBeanDefinitionParser("service", new DubboBeanDefinitionParser(ServiceBe
an.class, true));
    registerBeanDefinitionParser("reference", new DubboBeanDefinitionParser(Referen
ceBean.class, false));

    registerBeanDefinitionParser("annotation", new AnnotationBeanDefinitionParser()
); // 废弃
}
```

- 细心的胖友，会看到 `service` 标签使用的是 ServiceBean ，而不是 ServiceConfig ， `reference` 表示用的是 ReferenceBean ，因为无论是 ServiceConfig 还是 ReferenceBean ，在解析完具体配置后，需要调用它们对应的方法进行**初始化**。🙂 考虑到篇幅，我们在后续文章分享。这篇我们重点放在**解析**。

3. 解析

`com.alibaba.dubbo.config.spring.schema.DubboBeanDefinitionParser`，实现 `org.springframework.beans.factory.xml.BeanDefinitionParser` **接口**，Dubbo Bean 定义解析器。

3.1 构造方法

构造方法，代码如下：

```
/**
 * Bean 对象的类
 */
private final Class<?> beanClass;
/**
 * 是否需要 Bean 的 `id` 属性
 */
private final boolean required;

public DubboBeanDefinitionParser(Class<?> beanClass, boolean required) {
    this.beanClass = beanClass;
    this.required = required;
}
```

- `beanClass`，Bean 对象的**类**。
- `required`，是否需要在 Bean 对象的编号(`id`) 不存在时，**自动生成编号**。无需被其他应用引用的配置对象，无需自动生成编号。例如有 `<dubbo:reference />` 。

3.2 解析方法【主流程】

`#parse(Element, ParserContext)` 方法，解析 XML 元素。代码如下：

```
public BeanDefinition parse(Element element, ParserContext parserContext) {
    return parse(element, parserContext, beanClass, required);
}
```

- 调用 `#parse(Element, ParserContext, beanClass, required)` 方法，**真正**解析 XML 元素。该方法十分冗长，近 200+ 行。另外算上内部会调用其他方法，整体 500+ 行左右。🙂 下面，我们来将方法切换理解。

> 友情提示：建议胖友，能够边看边调试。

3.2.1 创建 RootBeanDefinition

`#parse(Element, ParserContext, beanClass, required)` 的第 78 至 80 行，代码如下：

```
RootBeanDefinition beanDefinition = new RootBeanDefinition();
beanDefinition.setBeanClass(beanClass);
beanDefinition.setLazyInit(false);
```

- 默认 `lazyInit = false` 。

> FROM 《Dubbo 用户指南 —— XML 配置》
> 引用缺省是延迟初始化的，只有引用被注入到其它 Bean，或被 getBean() 获取，才会初始化。如果需要饥饿加载，即没有人引用也立即生成动态代理，可以配置：`<dubbo:reference ... init="true" />`

3.2.2 处理 Bean 的编号

`#parse(Element, ParserContext, beanClass, required)` 的第 81 至 111 行，代码如下：

```
 81: // 解析配置对象的 id 。若不存在，则进行生成。
 82: String id = element.getAttribute("id");
 83: if ((id == null || id.length() == 0) && required) {
 84:     // 生成 id 。不同的配置对象，会存在不同。
 85:     String generatedBeanName = element.getAttribute("name");
 86:     if (generatedBeanName == null || generatedBeanName.length() == 0) {
 87:         if (ProtocolConfig.class.equals(beanClass)) {
 88:             generatedBeanName = "dubbo";
 89:         } else {
 90:             generatedBeanName = element.getAttribute("interface");
 91:         }
 92:     }
 93:     if (generatedBeanName == null || generatedBeanName.length() == 0) {
 94:         generatedBeanName = beanClass.getName();
 95:     }
 96:     id = generatedBeanName;
 97:     // 若 id 已存在，通过自增序列，解决重复。
 98:     int counter = 2;
 99:     while (parserContext.getRegistry().containsBeanDefinition(id)) {
100:         id = generatedBeanName + (counter++);
101:     }
102: }
103: if (id != null && id.length() > 0) {
104:     if (parserContext.getRegistry().containsBeanDefinition(id)) {
105:         throw new IllegalStateException("Duplicate spring bean id " + id);
```

```
106:     }
107:     // 添加到 Spring 的注册表
108:     parserContext.getRegistry().registerBeanDefinition(id, beanDefinition);
109:     // 设置 Bean 的 `id` 属性值
110:     beanDefinition.getPropertyValues().addPropertyValue("id", id);
111: }
```

- 第 82 至 102 行：解析 Bean 的 `id` 。若不存在，则自动进行生成。
  - 第 85 至 96 行：生成 `id` 。规则为 `name` > 特殊规则 > `className` 。
  - 第 97 至 101 行：若 `id` 在 Spring 注册表已经存在，通过添加**自增序列**作为后缀，避免冲突。
- 第 108 行：添加 Bean 到 Spring 注册表。
- 第 110 行：设置 Bean 的 `id` 。

3.2.3 处理 `<dubbo:protocol/>` 特殊情况

`#parse(Element, ParserContext, beanClass, required)` 的第 113 至 128 行，代码如下：

```
113: // 处理 `<dubbo:protocol` /> 的特殊情况
114: if (ProtocolConfig.class.equals(beanClass)) {
115:     // 需要满足第 220 至 233 行。
116:     // 例如：【顺序要这样】
117:     // <dubbo:service interface="com.alibaba.dubbo.demo.DemoService" protocol=
"dubbo" ref="demoService"/>
118:     // <dubbo:protocol id="dubbo" name="dubbo" port="20880"/>
119:     for (String name : parserContext.getRegistry().getBeanDefinitionNames()) {
120:         BeanDefinition definition = parserContext.getRegistry().getBeanDefinit
ion(name);
121:         PropertyValue property = definition.getPropertyValues().getPropertyVal
ue("protocol");
122:         if (property != null) {
123:             Object value = property.getValue();
124:             if (value instanceof ProtocolConfig && id.equals(((ProtocolConfig)
 value).getName())) {
125:                 definition.getPropertyValues().addPropertyValue("protocol", ne
w RuntimeBeanReference(id));
126:             }
127:         }
128:     }
```

在 「3.2.7 循环 Bean 对象的 setting 方法，将属性赋值到 Bean 对象」 统一解析。

3.2.4 处理 `<dubbo:service />` 的 `class` 属性

`#parse(Element, ParserContext, beanClass, required)` 的第 129 至 142 行，代码如下：

```
113: // 处理 `<dubbo:service />` 的属性 `class`
114: } else if (ServiceBean.class.equals(beanClass)) {
115:     // 处理 `class` 属性。例如 <dubbo:service id="sa" interface="com.alibaba.dubbo.demo.DemoService" class="com.alibaba.dubbo.demo.provider.DemoServiceImpl" >
116:     String className = element.getAttribute("class");
117:     if (className != null && className.length() > 0) {
118:         // 创建 Service 的 RootBeanDefinition 对象。相当于内嵌了 <bean class="com.alibaba.dubbo.demo.provider.DemoServiceImpl" />
119:         RootBeanDefinition classDefinition = new RootBeanDefinition();
120:         classDefinition.setBeanClass(ReflectUtils.forName(className));
121:         classDefinition.setLazyInit(false);
122:         // 解析 Service Bean 对象的属性
123:         parseProperties(element.getChildNodes(), classDefinition);
124:         // 设置 `<dubbo:service ref="" />` 属性
125:         beanDefinition.getPropertyValues().addPropertyValue("ref", new BeanDefinitionHolder(classDefinition, id + "Impl"));
126:     }
```

- 处理 `<dubbo:service class="" />` 场景下的处理，**大多数**情况下我们不这么使用，包括官方文档也没提供这种方式的说明。当配置 `class` 属时，会自动创建 Service Bean 对象，而无需再配置 `ref` 属性，指向 Service Bean 对象。示例如下：

  ```
  <bean id="demoDAO" class="com.alibaba.dubbo.demo.provider.DemoDAO" />

  <dubbo:service id="sa" interface="com.alibaba.dubbo.demo.DemoService"  class="com.alibaba.dubbo.demo.provider.DemoServiceImpl">
      <property name="demoDAO" ref="demoDAO" />
  </dubbo:service>
  ```

  - 通过这种方式，可以使用 `<property />` 标签，设置 Service Bean 的属性。

- 第 118 至 121 行：创建 Service 的 Bean 对象。
- 第 123 行：调用 `#parseProperties(NodeList, RootBeanDefinition)` 方法，解析 Service Bean 对象的属性们。详细解析见 「3.3.1 parseProperties」 方法。

3.2.5 解析 `<dubbo:provider />` 的内嵌子元素 `<dubbo:service />`

`#parse(Element, ParserContext, beanClass, required)` 的第 143 至 145 行，代码如下：

```
// 解析 `<dubbo:provider />` 的内嵌子元素 `<dubbo:service />`
} else if (ProviderConfig.class.equals(beanClass)) {
    parseNested(element, parserContext, ServiceBean.class, true, "service", "provid
er", id, beanDefinition);
```

- 调用 `#parseNested(...)` 方法，解析内嵌的指向的子 XML 元素。详细解析见 「3.3.2 parseNested」 方法。

3.2.6 解析 `<dubbo:consumer />` 的内嵌子元素 `<dubbo:reference />`

`#parse(Element, ParserContext, beanClass, required)` 的第 146 至 149 行，代码如下：

```
// 解析 `<dubbo:consumer />` 的内嵌子元素 `<dubbo:reference />`
} else if (ConsumerConfig.class.equals(beanClass)) {
    parseNested(element, parserContext, ReferenceBean.class, false, "reference", "c
onsumer", id, beanDefinition);
}
```

- 调用 `#parseNested(...)` 方法，解析内嵌的指向的子 XML 元素。

3.2.7 循环 Bean 对象的 setting 方法，将属性赋值到 Bean 对象

`#parse(Element, ParserContext, beanClass, required)` 的第 150 至 273 行，代码如下：

```
150: Set<String> props = new HashSet<String>(); // 已解析的属性集合
151: ManagedMap parameters = null; //
152: // 循环 Bean 对象的 setting 方法，将属性添加到 Bean 对象的属性赋值
153: for (Method setter : beanClass.getMethods()) {
154:     String name = setter.getName();
155:     if (name.length() > 3 && name.startsWith("set")
156:             && Modifier.isPublic(setter.getModifiers())
157:             && setter.getParameterTypes().length == 1) { // setting && public
&& 唯一参数
158:         Class<?> type = setter.getParameterTypes()[0];
159:         // 添加 `props`
160:         String property = StringUtils.camelToSplitName(name.substring(3, 4).to
LowerCase() + name.substring(4), "-");
161:         props.add(property);
162:         // getting && public && 属性值类型统一
```

```
163:         Method getter = null;
164:         try {
165:             getter = beanClass.getMethod("get" + name.substring(3), new Class<
?>[0]);
166:         } catch (NoSuchMethodException e) {
167:             try {
168:                 getter = beanClass.getMethod("is" + name.substring(3), new Cla
ss<?>[0]);
169:             } catch (NoSuchMethodException e2) {
170:             }
171:         }
172:         if (getter == null
173:                 || !Modifier.isPublic(getter.getModifiers())
174:                 || !type.equals(getter.getReturnType())) {
175:             continue;
176:         }
177:         // 解析 `<dubbo:parameters />`
178:         if ("parameters".equals(property)) {
179:             parameters = parseParameters(element.getChildNodes(), beanDefiniti
on);
180:         // 解析 `<dubbo:method />`
181:         } else if ("methods".equals(property)) {
182:             parseMethods(id, element.getChildNodes(), beanDefinition, parserCo
ntext);
183:         // 解析 `<dubbo:argument />`
184:         } else if ("arguments".equals(property)) {
185:             parseArguments(id, element.getChildNodes(), beanDefinition, parser
Context);
186:         } else {
187:             String value = element.getAttribute(property);
188:             if (value != null) {
189:                 value = value.trim();
190:                 if (value.length() > 0) {
191:                     // 不想注册到注册中心的情况，即 `registry=N/A` 。
192:                     if ("registry".equals(property) && RegistryConfig.NO_AVAIL
ABLE.equalsIgnoreCase(value)) {
193:                         RegistryConfig registryConfig = new RegistryConfig();
194:                         registryConfig.setAddress(RegistryConfig.NO_AVAILABLE)
;
195:                         beanDefinition.getPropertyValues().addPropertyValue(pr
operty, registryConfig);
196:                     // 多注册中心的情况
197:                     } else if ("registry".equals(property) && value.indexOf(',
') != -1) {
198:                         parseMultiRef("registries", value, beanDefinition, par
serContext);
199:                     // 多服务提供者的情况
200:                     } else if ("provider".equals(property) && value.indexOf(',
```

```
') != -1) {
                        parseMultiRef("providers", value, beanDefinition, parserContext);
                    // 多协议的情况
                    } else if ("protocol".equals(property) && value.indexOf(',') != -1) {
                        parseMultiRef("protocols", value, beanDefinition, parserContext);
                    } else {
                        Object reference;
                        // 处理属性类型为基本属性的情况
                        if (isPrimitive(type)) {
                            // 兼容性处理
                            if ("async".equals(property) && "false".equals(value)
                                    || "timeout".equals(property) && "0".equals(value)
                                    || "delay".equals(property) && "0".equals(value)
                                    || "version".equals(property) && "0.0.0".equals(value)
                                    || "stat".equals(property) && "-1".equals(value)
                                    || "reliable".equals(property) && "false".equals(value)) {
                                // backward compatibility for the default value in old version's xsd
                                value = null;
                            }
                            reference = value;
                        // 处理在 `<dubbo:provider />` 或者 `<dubbo:service />` 上定义了 `protocol` 属性的 兼容性。
                        } else if ("protocol".equals(property)
                                && ExtensionLoader.getExtensionLoader(Protocol.class).hasExtension(value) // 存在该注册协议的实现
                                && (!parserContext.getRegistry().containsBeanDefinition(value) // Spring 注册表中不存在该 `<dubbo:provider />` 的定义
                                    || !ProtocolConfig.class.getName().equals(parserContext.getRegistry().getBeanDefinition(value).getBeanClassName())) // Spring 注册表中存在该编号，但是类型不为 ProtocolConfig 。
                                ) {
                            // 目前，`<dubbo:provider protocol="" />` 推荐独立成 `<dubbo:protocol />`
                            if ("dubbo:provider".equals(element.getTagName())) {
                                logger.warn("Recommended replace <dubbo:provider protocol=\"" + value + "\" ... /> to <dubbo:protocol name=\"" + value + "\" ... />");
```

```
229:                                }
230:                                // backward compatibility
231:                                ProtocolConfig protocol = new ProtocolConfig();
232:                                protocol.setName(value);
233:                                reference = protocol;
234:                            // 处理 `onreturn` 属性
235:                            } else if ("onreturn".equals(property)) {
236:                                // 按照 `.` 拆分
237:                                int index = value.lastIndexOf(".");
238:                                String returnRef = value.substring(0, index);
239:                                String returnMethod = value.substring(index + 1);
240:                                // 创建 RuntimeBeanReference ，指向回调的对象
241:                                reference = new RuntimeBeanReference(returnRef);
242:                                // 设置 `onreturnMethod` 到 BeanDefinition 的属性值
243:                                beanDefinition.getPropertyValues().addPropertyValu
e("onreturnMethod", returnMethod);
244:                                // 处理 `onthrow` 属性
245:                            } else if ("onthrow".equals(property)) {
246:                                // 按照 `.` 拆分
247:                                int index = value.lastIndexOf(".");
248:                                String throwRef = value.substring(0, index);
249:                                String throwMethod = value.substring(index + 1);
250:                                // 创建 RuntimeBeanReference ，指向回调的对象
251:                                reference = new RuntimeBeanReference(throwRef);
252:                                // 设置 `onthrowMethod` 到 BeanDefinition 的属性值
253:                                beanDefinition.getPropertyValues().addPropertyValu
e("onthrowMethod", throwMethod);
254:                            // 通用解析
255:                            } else {
256:                                // 指向的 Service 的 Bean 对象，必须是单例
257:                                if ("ref".equals(property) && parserContext.getReg
istry().containsBeanDefinition(value)) {
258:                                    BeanDefinition refBean = parserContext.getRegi
stry().getBeanDefinition(value);
259:                                    if (!refBean.isSingleton()) {
260:                                        throw new IllegalStateException("The expor
ted service ref " + value + " must be singleton! Please set the " + value + " bean
scope to singleton, eg: <bean id=\"" + value + "\" scope=\"singleton\" ...>");
261:                                    }
262:                                }
263:                                // 创建 RuntimeBeanReference ，指向 Service 的 Bean
 对象
264:                                reference = new RuntimeBeanReference(value);
265:                            }
266:                            // 设置 BeanDefinition 的属性值
267:                            beanDefinition.getPropertyValues().addPropertyValue(pr
operty, reference);
268:                        }
```

```
269:                 }
270:             }
271:         }
272:     }
273: }
```

- 第 150 行：已解析的属性集合 `props` 。该属性在 「3.2.8 将 XML 元素未遍历到的属性，添加到 `parameters` 集合中」 会使用到。
- 第 151 行：解析的参数集合 `parameters` 。
- 第 153 行：循环 Bean 对象的 setting 方法，将属性添加到 Bean 对象的属性赋值。
- 第 154 至 157 行：判断方法符合 setting && `public` && 唯一参数的条件。
- 第 159 至 161 行：添加**属性名**到 `props` 。
- 第 162 至 176 行：判断方法符合 getting && `public` && 返回类型与 setting 参数类型**一致**的条件。
- 第 177 至 179 行：调用 `#parseParameters(id, nodeList, beanDefinition, parserContext)` 方法，解析 `<dubbo:argument />` 标签。详细解析见 「3.3.6 parseParameters」 方法。
- 第 180 至 182 行：调用 `#parseMethods(id, nodeList, beanDefinition, parserContext)` 方法，解析 `<dubbo:method />` 标签。详细解析见 「3.3.4 parseMethods」 方法。
- 第 183 至 185 行：调用 `#parseArguments(id, nodeList, beanDefinition, parserContext)` 方法，解析 `<dubbo:argument />` 标签。详细解析见 「3.3.5 parseArguments」 方法。
- 第 191 至 195 行：处理 `"registry"` 属性，不想注册到注册中心的情况，即 `registry=N/A` 。
- 第 196 至 204 行：调用 `#parseMultiRef(property, beanDefinition, parserContext)` 方法，处理多注册中心、多服务提供者、多协议的情况下。详细解析见 「3.3. parseMultiRef」 方法。
- 第 207 至 219 行：处理属性类型为**基础属性**(`#isPrimitive(Class<?>)`)的情况。
- 第 220 至 233 行：这块比较**复杂**。`"protocol"` 属性，在多个标签中会使用，例如 `<dubbo:service />` 或者 `<dubbo:provider />` 等等。根据 《Dubbo 文档 —— schema 配置参考手册》 的说明，`"protocol"` 代表的是指向的 `<dubbo:protocol />` 的编号(`id`)。
  - 【此处是猜测】但是，早期的版本，`"protocol"` 属性，代表的是**协议名**。这就麻烦了，和现有的逻辑有冲突啊！
  - 那怎么解决呢？优先以指向的 `<dubbo:protocol />` 的**编号**(`id`) 为准备，否则认为是**协议名**。
  - 那实际在解析 Bean 对象时，带有 `"protocol"` 属性的标签，无法保证一定在 `<dubbo:protocol />` **之后**解析。那咋整呢？
    - 如果带有 `"protocol"` 属性的标签**先**解析，先【第 221 至 223 行】**直接**创建 ProtocolConfig 对象并设置到 `"protocol"` 属性，再【第 119 至 127 行】在 `<dubbo:protocol />` 解析后，进行**覆盖**。这样，如果不存在 `<dubbo:protocol />` 的情况，最多不进行覆盖呢。

    - 如果带有 `"protocol"` 属性的标签**后**解析，无需走上述流程，走【第 257 至 264 行】即可。
    - 🙂 如果这段无法理解，可以给我留言，有点绕。
- 第 234 至 243 行 || 第 244 至 253 行：处理 `"onreturn"` 和 `"onthrow"` 属性。该属性用于《Dubbo 用户指南 —— 事件通知》功能。
- 第 254 至 264 行：剩余情况，通用解析，创建 RuntimeBeanReference 对象。例如，`<dubbo:service />` 的 `"ref"` 属性。
- 第 267 行：设置 Bean 的属性值。**该属性值来自第 206 至 265 行代码的逻辑。**

3.2.8 将 XML 元素未遍历到的属性，添加到 `parameters` 集合中

`#parse(Element, ParserContext, beanClass, required)` 的第 274 至 290 行，代码如下：

```
274: // 将 XML 元素，未在上面遍历到的属性，添加到 `parameters` 集合中。目前测试下来，不存在这样的情况。
275: NamedNodeMap attributes = element.getAttributes();
276: int len = attributes.getLength();
277: for (int i = 0; i < len; i++) {
278:     Node node = attributes.item(i);
279:     String name = node.getLocalName();
280:     if (!props.contains(name)) {
281:         if (parameters == null) {
282:             parameters = new ManagedMap();
283:         }
284:         String value = node.getNodeValue();
285:         parameters.put(name, new TypedStringValue(value, String.class));
286:     }
287: }
288: if (parameters != null) {
289:     beanDefinition.getPropertyValues().addPropertyValue("parameters", parameters);
290: }
```

- 第 275 至 287 行：将 XML 元素，**未在上面遍历到的属性**，添加到 `parameters` 集合中。目前测试下来，不存在这样的情况。
- 第 288 至 290 行：设置 Bean 的 `parameters` 。

3.3 解析方法【辅流程】

3.3.1 parseProperties

`#parseProperties(NodeList, RootBeanDefinition)` 方法，解析 Service Bean 对象的属性们。代码如下：

```
  1: /**
  2:  * 解析 <dubbo:service class="" /> 情况下，内涵的 `<property />` 的赋值。
  3:  *
  4:  * @param nodeList 子元素数组
  5:  * @param beanDefinition Bean 定义对象
  6:  */
  7: private static void parseProperties(NodeList nodeList, RootBeanDefinition bean
Definition) {
  8:     if (nodeList != null && nodeList.getLength() > 0) {
  9:         for (int i = 0; i < nodeList.getLength(); i++) {
 10:             Node node = nodeList.item(i);
 11:             if (node instanceof Element) {
 12:                 if ("property".equals(node.getNodeName())
 13:                         || "property".equals(node.getLocalName())) {
 14:                     String name = ((Element) node).getAttribute("name");
 15:                     if (name != null && name.length() > 0) {
 16:                         String value = ((Element) node).getAttribute("value");
 17:                         String ref = ((Element) node).getAttribute("ref");
 18:                         // value
 19:                         if (value != null && value.length() > 0) {
 20:                             beanDefinition.getPropertyValues().addPropertyValu
e(name, value);
 21:                         // ref
 22:                         } else if (ref != null && ref.length() > 0) {
 23:                             beanDefinition.getPropertyValues().addPropertyValu
e(name, new RuntimeBeanReference(ref));
 24:                         } else {
 25:                             throw new UnsupportedOperationException("Unsupport
ed <property name=\"" + name + "\"> sub tag, Only supported <property name=\"" + na
me + "\" ref=\"...\" /> or <property name=\"" + name + "\" value=\"...\" />");
 26:                         }
 27:                     }
 28:                 }
 29:             }
 30:         }
 31:     }
 32: }
```

- 第 11 至 13 行：只解析 `<property />` 标签。
- 第 18 至 20 行：优先使用 `"value"` 属性。

- 第 21 至 23 行：其次使用 `"ref"` 属性。
- 第 24 至 27 行：属性补全，抛出异常。

3.3.2 parseNested

`#parseNested(...)` 方法，解析内嵌的指向的子 XML 元素。代码如下：

```
  1: /**
  2:  * 解析内嵌的指向的子 XML 元素
  3:  *
  4:  * @param element 父 XML 元素
  5:  * @param parserContext Spring 解析上下文
  6:  * @param beanClass 内嵌解析子元素的 Bean 的类
  7:  * @param required 是否需要 Bean 的 `id` 属性
  8:  * @param tag 标签
  9:  * @param property 父 Bean 对象在子元素中的属性名
 10:  * @param ref 指向
 11:  * @param beanDefinition 父 Bean 定义对象
 12:  */
 13: private static void parseNested(Element element, ParserContext parserContext,
Class<?> beanClass, boolean required, String tag,
 14:                                 String property, String ref, BeanDefinition be
anDefinition) {
 15:     NodeList nodeList = element.getChildNodes();
 16:     if (nodeList != null && nodeList.getLength() > 0) {
 17:         boolean first = true;
 18:         for (int i = 0; i < nodeList.getLength(); i++) {
 19:             Node node = nodeList.item(i);
 20:             if (node instanceof Element) {
 21:                 if (tag.equals(node.getNodeName())
 22:                         || tag.equals(node.getLocalName())) { // 这三行，判断是
否为指定要解析的子元素
 23:                     // 【TODO 8008】 芋艿，default 是干锤子的
 24:                     if (first) {
 25:                         first = false;
 26:                         String isDefault = element.getAttribute("default");
 27:                         if (isDefault == null || isDefault.length() == 0) {
 28:                             beanDefinition.getPropertyValues().addPropertyValu
e("default", "false");
 29:                         }
 30:                     }
 31:                     // 解析子元素，创建 BeanDefinition 对象
 32:                     BeanDefinition subDefinition = parse((Element) node, parse
rContext, beanClass, required);
 33:                     // 设置子 BeanDefinition ，指向父 BeanDefinition 。
 34:                     if (subDefinition != null && ref != null && ref.length() >
```

```
 0) {
 35:                         subDefinition.getPropertyValues().addPropertyValue(pro
perty, new RuntimeBeanReference(ref));
 36:                     }
 37:                 }
 38:             }
 39:         }
 40:     }
 41: }
```

- 第 20 至 22 行：**只解析指定标签**。目前有**内嵌**的 `<dubbo:service />` 和 `<dubbo:reference />` 标签。
- 第 32 行：解析指定标签，创建子 Bean 对象。
- 第 33 至 36 行：设置创建的**子** Bean 对象，指向**父** Bean 对象。

3.3.3 parseMultiRef

`#parseMultiRef(property, beanDefinition, parserContext)` 方法，解析多指向的情况，例如多注册中心，多协议等等。代码如下：

```
  1: /**
  2:  * 解析多指向的情况，例如多注册中心，多协议等等。
  3:  *
  4:  * @param property 属性
  5:  * @param value 值
  6:  * @param beanDefinition Bean 定义对象
  7:  * @param parserContext Spring 解析上下文
  8:  */
  9: @SuppressWarnings("unchecked")
 10: private static void parseMultiRef(String property, String value, RootBeanDefin
ition beanDefinition,
 11:                                   ParserContext parserContext) {
 12:     String[] values = value.split("\\s*[,]+\\s*");
 13:     ManagedList list = null;
 14:     for (int i = 0; i < values.length; i++) {
 15:         String v = values[i];
 16:         if (v != null && v.length() > 0) {
 17:             if (list == null) {
 18:                 list = new ManagedList();
 19:             }
 20:             list.add(new RuntimeBeanReference(v));
 21:         }
 22:     }
 23:     beanDefinition.getPropertyValues().addPropertyValue(property, list);
 24: }
```

- 第 12 至 22 行：以 `.` 拆分值字符串，创建 RuntimeBeanReference 数组。
- 第 23 行：设置 Bean 对象的指定属性值。

3.3.4 parseMethods

`#parseMethods(id, nodeList, beanDefinition, parserContext)` 方法，解析 `<dubbo:method />` 标签。代码如下：

```
  1: /**
  2:  * 解析 `<dubbo:method />`
  3:  *
  4:  * @param id Bean 的 `id` 属性。
  5:  * @param nodeList 子元素节点数组
  6:  * @param beanDefinition Bean 定义对象
  7:  * @param parserContext 解析上下文
  8:  */
  9: @SuppressWarnings("unchecked")
 10: private static void parseMethods(String id, NodeList nodeList, RootBeanDefinit
ion beanDefinition,
 11:                                  ParserContext parserContext) {
 12:     if (nodeList != null && nodeList.getLength() > 0) {
 13:         ManagedList methods = null; // 解析的方法数组
 14:         for (int i = 0; i < nodeList.getLength(); i++) {
 15:             Node node = nodeList.item(i);
 16:             if (node instanceof Element) {
 17:                 Element element = (Element) node;
 18:                 if ("method".equals(node.getNodeName())
 19:                         || "method".equals(node.getLocalName())) { // 这三行,
判断值解析 `<dubbo:method />`
 20:                     // 方法名不能为空
 21:                     String methodName = element.getAttribute("name");
 22:                     if (methodName == null || methodName.length() == 0) {
 23:                         throw new IllegalStateException("<dubbo:method> name a
ttribute == null");
 24:                     }
 25:                     if (methods == null) {
 26:                         methods = new ManagedList();
 27:                     }
 28:                     // 解析 `<dubbo:method />`, 创建 BeanDefinition 对象
 29:                     BeanDefinition methodBeanDefinition = parse(((Element) nod
e), parserContext, MethodConfig.class, false);
 30:                     // 添加到 `methods` 中
 31:                     String name = id + "." + methodName;
 32:                     BeanDefinitionHolder methodBeanDefinitionHolder = new Bean
DefinitionHolder(methodBeanDefinition, name);
```

```
 33:                     methods.add(methodBeanDefinitionHolder);
 34:                 }
 35:             }
 36:         }
 37:         if (methods != null) {
 38:             beanDefinition.getPropertyValues().addPropertyValue("methods", met
hods);
 39:         }
 40:     }
 41: }
```

- 第 13 行：解析的方法数组 `methods` 。
- 第 16 至 19 行：**只**解析 `<dubbo:method>` 标签。
- 第 29 行：调用 `#parse(Element, ParserContext)` 方法，**主流程**，解析 `<dubbo:method>` 标签，创建子 Bean 对象。
- 第 33 至 36 行：设置创建的**子** Bean 对象，指向**父** Bean 对象。
- 第 30 至 33 行：添加子 Bean 对象到 `methods` 中。
- 第 37 至 39 行：设置 Bean 的 `methods` 。

3.3.5 parseArguments

`#parseArguments(id, nodeList, beanDefinition, parserContext)` 方法，解析 `<dubbo:argument />` 标签。

和 「3.3.4 parseMethods」 基本一致，🙂 胖友点击方法，直接查看代码。

3.3.6 parseParameters

`#parseParameters(id, nodeList, beanDefinition, parserContext)` 方法，解析 `<dubbo:argument />` 标签。代码如下：

```
  1: /**
  2:  * 解析 `<dubbo:parameter />`
  3:  *
  4:  * @param nodeList 子元素节点数组
  5:  * @param beanDefinition Bean 定义对象
  6:  * @return 参数集合
  7:  */
  8: @SuppressWarnings("unchecked")
  9: private static ManagedMap parseParameters(NodeList nodeList, RootBeanDefinitio
n beanDefinition) {
 10:     if (nodeList != null && nodeList.getLength() > 0) {
 11:         ManagedMap parameters = null;
```

```
 12:         for (int i = 0; i < nodeList.getLength(); i++) {
 13:             Node node = nodeList.item(i);
 14:             if (node instanceof Element) {
 15:                 if ("parameter".equals(node.getNodeName())
 16:                         || "parameter".equals(node.getLocalName())) { // 这三行
, 只解析子元素中的 `<dubbo:parameter />`
 17:                     if (parameters == null) {
 18:                         parameters = new ManagedMap();
 19:                     }
 20:                     // 添加到参数集合
 21:                     String key = ((Element) node).getAttribute("key");
 22:                     String value = ((Element) node).getAttribute("value");
 23:                     boolean hide = "true".equals(((Element) node).getAttribute
("hide")); //  【TODO 8007】 <dubbo:parameter hide="" /> 的用途
 24:                     if (hide) {
 25:                         key = Constants.HIDE_KEY_PREFIX + key;
 26:                     }
 27:                     parameters.put(key, new TypedStringValue(value, String.cla
ss));
 28:                 }
 29:             }
 30:         }
 31:         return parameters;
 32:     }
 33:     return null;
 34: }
```

- 第 13 行：解析的参数集合 `parameters` 。
- 第 20 至 27 行：添加 `"key"` `"value"` 到 `parameters` 。

666. 彩蛋

稍显啰嗦的一篇文章，希望胖友能理解。

关于这块内容，在推荐下肥朝写的 《Dubbo源码解析 —— 简单原理、与 Spring 融合》 。