第3章

自定义标签

在第1章中对Spring的简单使用进行了说明，简单介绍了bean标签的使用。bean标签属于Spring的原生标签，在Spring中除了原生标签以外还能够支持自定义标签，本章将介绍SpringXML配置文件中的自定义标签如何进行自定义、如何使用自定义标签，并对SpringXML的自定义标签相关的内容进行源码分析。

3.1 创建自定义标签环境搭建

在开始自定义标签分析之前，需要先编写自定义标签解析相关的测试用例，编写自定义标签需要执行下面四个步骤。

(1) 编写XSD文件或者DTD文件。

(2) 编写NamespaceHandler实现类。

(3) 编写BeanDefinitionParser实现类。

(4) 编写注册方式，向Spring中注册。

接下来对上述四个步骤做详细说明。

3.1.1 编写XSD文件

首先编写一个Java对象用来存储自定义标签解析后的数据，编写UserXsdJava对象，代码信息如下。

```
//省略 getter&setter
public class UserXsd {
    private String name;
    private String idCard;
}
```

完成 XSD 文件解析结果的存储对象后进一步编写 XSD 文件，该 XSD 文件名为 user.xsd，文件内容如下。

```
<?xml version = "1.0" encoding = "UTF - 8"?>
< schema xmlns = "http: //www.w3.org/2001/XMLSchema"
        targetNamespace = "http: //www.huifer.com/schema/user"
        elementFormDefault = "qualified">

    < element name = "user_xsd">
        < complexType >
            < attribute name = "id" type = "string"/>
            < attribute name = "name" type = "string"/>
            < attribute name = "idCard" type = "string"/>
        </complexType >
    </element >
</schema >
```

3.1.2 编写 NamespaceHandler 实现类

完成 XSD 文件编写后进一步编写 NamespaceHandler 接口的实现类，Spring 提供了 NamespaceHandlerSupport 对象让开发者更加简单地使用，开发者只需要重写 init 方法即可向 Spring 注册标签和标签的解析对象，编写 UserXsdNamespaceHandler 类，详细代码如下。

```
public class UserXsdNamespaceHandler extends NamespaceHandlerSupport {

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

}
```

3.1.3 编写 BeanDefinitionParser 实现类

在编写 NamespaceHandler 实现类的时候引入了一个新的 Java 对象 UserXsdParser，该对象是 BeanDefinitionParser 接口的实现类，在 Spring 中可以通过继承 AbstractSingleBeanDefinitionParser 类重写 getBeanClass 和 doParse 方法即可完成 BeanDefinitionParser 的实现，下面是 UserXsdParser 的代码内容。

```
public class UserXsdParser extends AbstractSingleBeanDefinitionParser {
    @Override
    protected Class <?> getBeanClass(Element element) {
        return UserXsd.class;
    }
```

```
    @Override
    protected void doParse(Element element,BeanDefinitionBuilder builder) {
        String name = element.getAttribute("name");
        String idCard = element.getAttribute("idCard");
        builder.addPropertyValue("name",name);
        builder.addPropertyValue("idCard",idCard);
    }

}
```

在这段代码中通过提取 Element 对象的 name 和 idCard 属性将其设置到 BeanDefinitionBuilder 对象的属性表中。

3.1.4 编写注册方式

下面编写注册方式。注册自定义标签解析能力需要编写两个文件，一个是 spring.handlers 文件，另一个是 spring.schemas 文件。在这个测试用例中需要向 spring.handlers 文件中填写下面这段内容。

```
http\://www.huifer.com/schema/user=com.source.hot.ioc.book.namespace.handler.
UserXsdNamespaceHandler
```

对于 spring.handlers 文件可以分成两部分来进行理解，第一部分是等号前面的内容，等号前的内容是指命名空间和 XSD 文件中 schema 中的 targetNamespace 属性之间的关系；第二部分是等号后面的内容，它是指接口 NamespaceHandler 实现类的完整类路径。

完成 spring.handlers 编写后进一步编写 spring.schemas 文件，向 spring.schemas 文件中添加下面这段代码。

```
http\://www.huifer.com/schema/user.xsd=META-INF/user.xsd
```

对于 spring.schemas 文件可以分成两部分来进行理解，第一部分是等号前面的内容，它是指 schemaLocation 的一个链接地址；第二部分是等号后面的内容，它是指 schemaLocation 对应的 XSD 文件描述路径。

3.1.5 测试用例的编写

完成了各项基本准备工作后进一步编写一个自定义标签处理的 Java 程序，首先需要编写 SpringXML 配置文件，文件名为 custom-xml.xml，向 custom-xml.xml 文件中填写下面的内容。

```
<?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
">

    <myname:user_xsd id="testUserBean" name="huifer" idCard="123"/>

</beans>
```

完成SpringXML配置文件的编写后再编写一个测试类，测试类名称为CustomXmlTest，CustomXmlTest中代码如下。

```
class CustomXmlTest {

    @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();
    }
}
```

完成测试类及测试方法的编写后自定义标签测试环境即搭建完成。

3.2 自定义标签解析

下面将进入自定义标签解析相关源代码分析，首先需要找到自定义标签解析的源码入口，该入口的方法签名为org.springframework.beans.factory.xml.BeanDefinitionParserDelegate#parseCustomElement(org.w3c.dom.Element,org.springframework.beans.factory.config.BeanDefinition)，详细代码如下。

```
@Nullable
public BeanDefinition parseCustomElement(Element ele,@Nullable BeanDefinition containingBd) {
   //获取命名空间的URL
   String namespaceUri = getNamespaceURI(ele);
   if (namespaceUri == null) {
      return null;
   }
   //命名空间处理器
   NamespaceHandler handler
= this.readerContext.getNamespaceHandlerResolver().resolve(namespaceUri);
   if (handler == null) {
      error("Unable to locate Spring NamespaceHandler for XML schema namespace [" +
namespaceUri + "]",ele);
      return null;
   }
   return handler.parse(ele,new ParserContext(this.readerContext,this,containingBd));
}
```

3.2.1 NamesapceHandler 和 BeanDefinitionParser 之间的关系

parseCustomElement 方法中体现了 namespaceUri 的一些关系，在测试用例中 namespaceUri 和 spring.handlers 中的文件存在关系，命名空间对应命名空间处理器，通过这个关系可以确认 NamespaceHandler 是 UserXsdNamespaceHandler 对象，在 UserXsdNamespaceHandler 类中有 init 方法来注册标签和标签的解析能力提供类的关系。具体关系如图 3.1 所示。

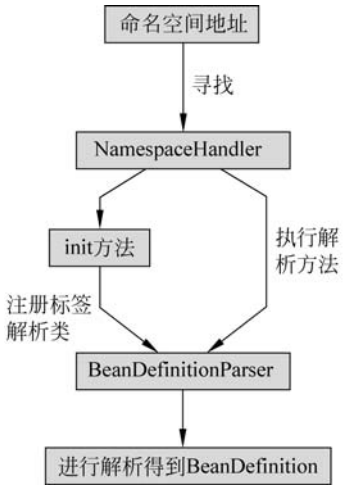

图 3.1 NamespaceHandler 关系图

3.2.2 获取命名空间地址

接下来将介绍命名空间地址的获取，命名空间地址及 namespaceUri 属性，获取该属性的方法是由 org.w3c.dom 提供，具体细节不做展开，获取命名空间地址后的数据是 http://www.huifer.com/schema/user。

3.2.3 NamespaceHandler 对象获取

通过前文获得了 namespaceUri 数据信息后会通过该信息寻找到对应的 NamespaceHandler 对象，具体处理方法如下。

```
NamespaceHandler handler
 = this.readerContext.getNamespaceHandlerResolver().resolve(namespaceUri);
```

在这段方法中负责命名空间解析的对象是 NamespaceHandlerResolver 接口，NamespaceHandlerResolver 接口定义如下。

```
@FunctionalInterface
public interface NamespaceHandlerResolver {

   /**
    * 解析命名空间的 URL 获得命名空间处理器
    */
   @Nullable
   NamespaceHandler resolve(String namespaceUri);

}
```

在 Spring 中它只有一个实现类 DefaultNamespaceHandlerResolver，在 DefaultNamespaceHandlerResolver 中有一个成员变量 DEFAULT_HANDLER_MAPPINGS_LOCATION，成员变量详细信息如下。

```
public static final String DEFAULT_HANDLER_MAPPINGS_LOCATION =
"META-INF/spring.handlers";
```

在 DEFAULT_HANDLER_MAPPINGS_LOCATION 成员变量中定义了默认的命名空间处理器存储路径，spring.handlers 中存储的内容是(key，value)结构，key 是命名空间，value 是命名空间处理器的类全路径，接下来查看 resolve 方法，详细代码如下。

```
//删除异常处理和日志
public NamespaceHandler resolve(String namespaceUri) {
    //获取 Namespace Handler 映射表
    Map < String,Object > handlerMappings = getHandlerMappings();
    //从映射表中获取 URI 对应的 Handler
    //字符串(名称)
    //实例
    Object handlerOrClassName = handlerMappings.get(namespaceUri);
    if(handlerOrClassName == null) {
        return null;
    } else if(handlerOrClassName instanceof NamespaceHandler) {
        return(NamespaceHandler) handlerOrClassName;
    }
    //其他情况都做字符串处理
    else {
        String className = (String) handlerOrClassName;
        Class <? > handlerClass = ClassUtils.forName(className,this.classLoader);
        //通过反射构造 namespaceHandler 实例
        NamespaceHandler namespaceHandler =
(NamespaceHandler) BeanUtils.instantiateClass(handlerClass);
        //初始化
        namespaceHandler.init();
        //重写缓存
        handlerMappings.put(namespaceUri,namespaceHandler);
        return namespaceHandler;
    }
}
```

在 resolve 方法中整体处理流程如下。

(1) 获取命名空间映射关系，即命名空间地址(namespaceUri)对应命名空间解析对象类全路径，这个关系是一个 map 集合。

(2) 从 map 集合中获取命名空间地址(namespaceUri)对应的数据。

(3) 根据提取数据的不同情况进行处理。

① 通过命名空间地址获取的对象是 NamespaceHandler 类型，直接返回使用。

② 通过命名空间地址获取的对象不是 NamespaceHandler 类型。当类型不是 NamespaceHandler 对象时，它的类型只可能是字符串类型，字符串是没有办法直接调用 NamespaceHandler 所提供的方法的，因此需要将字符串转换成 NamespaceHandler 对象。

3.2.4 getHandlerMappings 获取命名空间的映射关系

在前文讲到字符串转换为 NamespaceHandler 的内容在本节会对其做补充，负责这部分处理的方法是 getHandlerMappings，详细代码如下。

```
//删除异常处理和日志
private Map < String,Object > getHandlerMappings() {
    //设置容器
    Map < String,Object > handlerMappings = this.handlerMappings;
    if(handlerMappings == null) {
        synchronized(this) {
            handlerMappings = this.handlerMappings;
            if(handlerMappings == null) {
                //读取资源文件地址
                Properties mappings =
PropertiesLoaderUtils.loadAllProperties(this.handlerMappingsLocation,this.classLoader);
                handlerMappings = new ConcurrentHashMap < > (mappings.size());
                //数据合并,将 mappings 数据复制给 handlerMappings
                CollectionUtils.mergePropertiesIntoMap(mappings,handlerMappings);
                this.handlerMappings = handlerMappings;
            }
        }
    }
    return handlerMappings;
}
```

在这段方法中首先需要关注的是下面这段代码。

```
Properties mappings =
      PropertiesLoaderUtils.loadAllProperties(this.handlerMappingsLocation,this.classLoader);
```

这段代码是用来提取 META-INF/spring.handlers 文件中的数据内容，Spring 将 META-INF/spring.handlers 文件当作拓展名为 properties 类型的文件进行处理，处理之后得到的是 Map 对象，在完成 META-INF/spring.handlers 文件的读取后进行了合并操作，分别将历史的 handlerMappings 和本次读取得到的 handlerMappings 进行合并。图 3.2 为本次读取得到的 handlerMappings 数据。

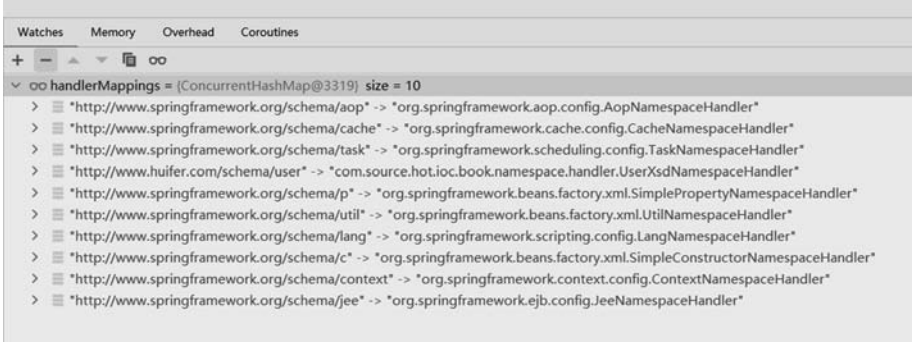

图 3.2 handlerMappings 数据信息

通过图 3.2 可以看到一条这样的信息：http://www.huifer.com/schema/user -> com.source.hot.ioc.book.namespace.handler.UserXsdNamespaceHandler，这段信息的来源是前文在测试用例中文件 spring.handlers 的内容，除此之外，其他的数据是 Spring 项目中存放的，文件依旧还是在 META-INF/spring.handlers 中。

spring-beans 工程下的 spring.handlers 数据信息如下。

```
    http\://www.springframework.org/schema/c = org.springframework.beans.factory.xml.
SimpleConstructorNamespaceHandler
    http\://www.springframework.org/schema/p = org.springframework.beans.factory.xml.
SimplePropertyNamespaceHandler
    http\://www.springframework.org/schema/util = org.springframework.beans.factory.xml.
UtilNamespaceHandler
```

spring-aop 工程下的 spring.handlers 数据信息如下。

```
    http\://www.springframework.org/schema/aop = org.springframework.aop.config.
AopNamespaceHandler
```

spring-context 工程下的 spring.handlers 数据信息如下。

```
    http\://www.springframework.org/schema/context = org.springframework.context.config.
ContextNamespaceHandler
    http\://www.springframework.org/schema/jee = org.springframework.ejb.config.
JeeNamespaceHandler
    http\://www.springframework.org/schema/lang = org.springframework.scripting.config.
LangNamespaceHandler
    http\://www.springframework.org/schema/task = org.springframework.scheduling.config.
TaskNamespaceHandler
    http\://www.springframework.org/schema/cache = org.springframework.cache.config.
CacheNamespaceHandler
```

上述这些文件是 Spring 容器中提供的 NamespaceHandler 数据，除此之外，还有一些其他的 spring.handlers 文件本文不做赘述。

3.2.5 NamespaceHandler 的获取

在前面的操作中已经准备好 handlerMappings 数据对象，下面需要将数据对象进行初始化（实例化，实例化的前提是类型不是 NamespaceHandler）。有关 NamespaceHandler 的获取代码如下。

```
Object handlerOrClassName = handlerMappings.get(namespaceUri);
if(handlerOrClassName == null) {
    return null;
} else if(handlerOrClassName instanceof NamespaceHandler) {
    return(NamespaceHandler) handlerOrClassName;
}
//其他情况都做字符串处理
else {
    String className = (String) handlerOrClassName;
    Class <? > handlerClass = ClassUtils.forName(className,this.classLoader);
    //通过反射构造 namespaceHandler 实例
    NamespaceHandler namespaceHandler =
(NamespaceHandler) BeanUtils.instantiateClass(handlerClass);
    //初始化
```

```
        namespaceHandler.init();
        //重写缓存
        handlerMappings.put(namespaceUri,namespaceHandler);
        return namespaceHandler;
    }
```

上述代码中有以下三种情况需要处理。

(1) 容器中没有当前 namespaceUri 的值。

(2) 容器中有当前 namespaceUri 的值,并且类型是 NamespaceHandler。

(3) 容器中有当前 namespaceUri 的值,但类型不是 NamespaceHandler。

对于上述三种情况需要重点分析的是第三种。在第三种情况中,value 的数据类型是 String,Spring 需要将 String 类型转换为最终的 Java 对象,通过类全路径转换成 Java 对象需要执行以下三个步骤。

(1) 通过 Class.forName 得到 Class 对象。

(2) 通过 Class 对象提取构造函数。

(3) 通过构造函数创建实例。

上述三个操作步骤可以分别对应下面这些代码,第一步对应"Class <? > handlerClass = ClassUtils.forName(className,this.classLoader);",第二步和第三步对应 Spring 项目中的一个工具类 BeanUtils。通过 BeanUtils.instantiateClass 方法即可得到 Java 对象。

3.2.6 NamespaceHandler 的 init 方法

接下来将介绍 NamespaceHandler 的 init 方法,在测试用例中,UserXsdNamespaceHandler 对象继承 NamespaceHandlerSupport 类,接下来需要分析的重点内容都在 NamespaceHandlerSupport 类中。首先需要指出一点,在 Spring 中的 spring.handlers 文件中的实现类都有继承 NamespaceHandlerSupport 类,可见它的重要性。在测试用例中使用了 registerBeanDefinitionParser 方法进行了标签和解析类的注册,下面对 registerBeanDefinitionParser 方法进行分析,先看 registerBeanDefinitionParser 代码。

```
protected final void registerBeanDefinitionParser(String elementName,BeanDefinitionParser parser) {
    this.parsers.put(elementName,parser);
}
```

在这段代码中需要了解以下两个参数的含义。

(1) elementName: XML 标签的名称。

(2) parser: 提供标签解析能力的实际对象,是 BeanDefinitionParser 接口的实现类。

下面介绍 parsers 的数据结构,在 Spring 中对 parsers 的定义如下。

```
private final Map < String,BeanDefinitionParser > parsers = new HashMap <>();
```

key 表示 XML 标签的名称,value 表示 BeanDefinitionParser 接口的实现类。parsers 的数据存储情况如图 3.3 所示。

图 3.3　parsers 数据信息

3.2.7　NamespaceHandler 缓存的刷新

在 NamespaceHandler 的生命周期方法 init 执行完成后会刷新 namespaceUri 对应的 value，此时会产生 handlerMappings 中 value 的多种情况。

（1）value 不存在。

（2）value 存在，且是 NamespaceHandler 实例。

（3）value 存在，且是字符串。

刷新 handlerMappings 变量的操作代码如下。

```
handlerMappings.put(namespaceUri,namespaceHandler)
```

这段代码的执行就是将 namespaceUri 和 namespaceHandler 的关系重新绑定，此时放入的 value 就会变成 NamespaceHandler 实例，后续在需要使用时就会进入下面这段代码。

```
else if (handlerOrClassName instanceof NamespaceHandler) {
   return (NamespaceHandler) handlerOrClassName;
}
```

在这里 Spring 通过了一个 Map 对象的刷新操作来提高了性能，避免了每次从字符串出发进行反射获取实例，获取实例之后再做其他操作。图 3.4 为经过刷新操作后的 handlerMappings 数据情况。

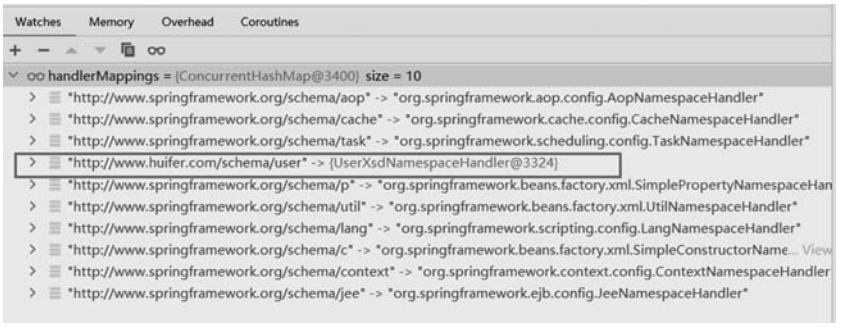

图 3.4　刷新后的 handlerMappings

3.2.8　解析标签 BeanDefinitionParser 对象准备

接下来将进入自定义标签解析的重要环节——BeanDefinitionParser 实现类的准备阶

段。首先阅读 parse 方法：

```
NamespaceHandler handler
= this.readerContext.getNamespaceHandlerResolver().resolve(namespaceUri);
return handler.parse(ele,new ParserContext(this.readerContext,this,containingBd));
```

在这段代码中得到了 NamespaceHandler 对象，这个对象的实际类型是 UserXsdNamespaceHandler，确认实际类型后对于 parse 方法入口的查询有了方向，parse 方法位于 UserXsdNamespaceHandler 的父类 NamespaceHandlerSupport 中，具体代码如下。

```
public BeanDefinition parse(Element element,ParserContext parserContext) {
   //搜索 element 对应的 BeanDefinitionParser
   BeanDefinitionParser parser = findParserForElement(element,parserContext);
   //解析
   return (parser != null ? parser.parse(element,parserContext): null);
}
```

从这段操作代码中可以发现一个重点类 BeanDefinitionParser，在测试用例中编写的内容中有一个与之存在关系，这个类是 UserXsdParser。在测试用例中和 UserXsdParser 对象产生关系的内容是一个注册方法 registerBeanDefinitionParser("user_xsd",new UserXsdParser())，这个方法表示 user_xsd 标签需要通过 UserXsdParser 对象进行解析。在 parse 方法中出现的 findParserForElement 方法目的就是通过标签名称找到对应的标签处理对象。findParserForElement 方法代码如下。

```
private BeanDefinitionParser findParserForElement(Element element,ParserContext parserContext) {
   //获取 element 的名称
   String localName = parserContext.getDelegate().getLocalName(element);
   //从容器中获取
   BeanDefinitionParser parser = this.parsers.get(localName);
   if (parser == null) {
      parserContext.getReaderContext().fatal(
            "Cannot locate BeanDefinitionParser for element [" + localName + "]",element);
   }
   return parser;
}
```

在这段方法中可以看到以下两个处理步骤。

(1) 提取 Element 的名称数据。

(2) 从 parsers 中获取解析 BeanDefinitionParser 对象。

此时得到的 BeanDefinitionParser 是 UserXsdParser 对象。

3.2.9 解析标签 parse 方法调用

在 Spring 中 parse 方法提供者是 AbstractBeanDefinitionParser 对象，详细代码如下。

```
//删除异常处理和日志
```

```
public final BeanDefinition parse(Element element, ParserContext parserContext) {
    //解析 Element 得到 BeanDefinition 对象
    AbstractBeanDefinition definition = parseInternal(element, parserContext);
    if(definition != null && !parserContext.isNested()) {
        //解析标签的 id
        String id = resolveId(element, definition, parserContext);
        if(!StringUtils.hasText(id)) {
              parserContext.getReaderContext().error("Id is required for element '" +
parserContext.getDelegate().getLocalName(element) + "' when used as a top - level tag",
element);
        }
        String[] aliases = null;
        if(shouldParseNameAsAliases()) {
            //标签的 name 属性
            String name = element.getAttribute(NAME_ATTRIBUTE);
            if(StringUtils.hasLength(name)) {
                //别名处理,根据逗号进行字符串切割
                aliases =
StringUtils.trimArrayElements(StringUtils.commaDelimitedListToStringArray(name));
            }
        }
        //创建 BeanDefinitionHolder
        BeanDefinitionHolder holder = new
BeanDefinitionHolder(definition, id, aliases);
        //注册 BeanDefinition
        registerBeanDefinition(holder, parserContext.getRegistry());
        //是否需要触发事件
        if(shouldFireEvents()) {
            //组件注册事件
            BeanComponentDefinition componentDefinition =
new BeanComponentDefinition(holder);
            postProcessComponentDefinition(componentDefinition);
            parserContext.registerComponent(componentDefinition);
        }
    }
    return definition;
}
```

在这段代码中比较难以查看到 UserXsdParser 对象的踪迹,UserXsdParser 类图信息如图 3.5 所示。

接下来看第一段代码。

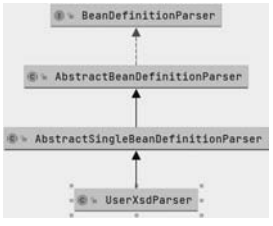

图 3.5 UserXsdParser 类图

```
AbstractBeanDefinition definition = parseInternal
(element, parserContext);
```

parseInternal 方法在 AbstractBeanDefinitionParser 是一个抽象方法,真正的实现在 AbstractSingleBeanDefinitionParser 中,下面是 AbstractSingleBeanDefinitionParser # parseInternal 方法代码。

```
@Override
protected final AbstractBeanDefinition parseInternal(Element element,ParserContext parserContext) {
    //BeanDefinition 构造器
    BeanDefinitionBuilder builder = BeanDefinitionBuilder.genericBeanDefinition();
    //获取 parent 属性
    String parentName = getParentName(element);
    if (parentName != null) {
        builder.getRawBeanDefinition().setParentName(parentName);
    }

    //获取 class 属性
    Class<?> beanClass = getBeanClass(element);
    if (beanClass != null) {
        builder.getRawBeanDefinition().setBeanClass(beanClass);
    }
    else {
        String beanClassName = getBeanClassName(element);
        if (beanClassName != null) {
            builder.getRawBeanDefinition().setBeanClassName(beanClassName);
        }
    }
    //设置源
    builder.getRawBeanDefinition().setSource(parserContext.extractSource(element));

    //获取已存在的 BeanDefinition,该对象仅用来设置 scope 属性
    BeanDefinition containingBd = parserContext.getContainingBeanDefinition();
    if (containingBd != null) {
        builder.setScope(containingBd.getScope());
    }
    if (parserContext.isDefaultLazyInit()) {
        builder.setLazyInit(true);
    }
    //真正的调用
    doParse(element,parserContext,builder);
    //BeanDefinition 构造器中获取 BeanDefinition
    return builder.getBeanDefinition();
}
```

在 UserXsdParser 类中重写了 getBeanClass 和 doParse 方法,测试用例中所编写的 doParse 方法的调用需要通过一层外部调用才可以抵达测试用例中 UserXsdParser 类的 doParse 方法,具体调用过程的代码如下。

```
//AbstractSingleBeanDefinitionParser#parseInternal 中调用
protected void doParse(Element element,ParserContext parserContext,
BeanDefinitionBuilder builder) {
   doParse(element,builder);
}
//需要重写的方法
protected void doParse(Element element,BeanDefinitionBuilder builder) {
}
```

AbstractSingleBeanDefinitionParser#parseInternal 方法处理的细节如下。

(1) 准备基本数据,基本数据包含 parentName、beanClass、source 和 scope。

(2) 执行自定义的 doParse 方法。

(3) 通过 BeanDefinitionBuilder 来获取 BeanDefinition。

在 parse 方法中首先需要进行 id 的处理,代码如下。

```
String id = resolveId(element,definition,parserContext);
```

在这段代码中对于 id 属性的获取其本质是提取 XML 标签中的 id 属性。完成 id 数据获取后需要执行的事项是针对别名的处理,相关代码如下。

```
String[] aliases = null;
if (shouldParseNameAsAliases()) {
   //标签的 name 属性
   String name = element.getAttribute(NAME_ATTRIBUTE);
   if (StringUtils.hasLength(name)) {
      //别名处理,根据逗号进行字符串切割
      aliases =
StringUtils.trimArrayElements(StringUtils.commaDelimitedListToStringArray(name));
   }
}
```

在别名处理阶段会判断是否需要处理别名,默认是都需要处理的。别名处理方式是将 alias 标签中的 name 属性根据分隔符(逗号)切分,关于切分其本质为 name.split(“,”),具体的处理方法是 StringUtils.commaDelimitedListToStringArray。在数据信息准备完成之后需要进行 BeanDefinition 对象的注册和事件发布,相关代码如下。

```
BeanDefinitionHolder holder =
new BeanDefinitionHolder(definition,id,aliases);
//注册 BeanDefinition
registerBeanDefinition(holder,parserContext.getRegistry());
//是否需要出发事件
if (shouldFireEvents()) {
   //组件注册事件
   BeanComponentDefinition componentDefinition =
new BeanComponentDefinition(holder);
   postProcessComponentDefinition(componentDefinition);
   parserContext.registerComponent(componentDefinition);
}
```

在这部分代码处理中,关于事件发布相关内容是一个预留方法,暂时是一个空处理,对于 BeanDefinition 对象的注册会在后续章节中进行详细分析,在这仅需要了解 BeanDefinition 对象的存储容器:

```
private final Map<String,BeanDefinition> beanDefinitionMap =
new ConcurrentHashMap<>(256);
```

当 BeanDefinition 对象被放置到容器后这段方法的处理就完成了。

小结

通过本章的阐述，相信读者对于自定义标签的处理流程有了更加详细的认知。在本章了解了 NamespaceHandler、NamespaceHandlerSupport、AbstractSingleBeanDefinitionParser、AbstractBeanDefinitionParser 和 BeanDefinitionParser 之间的关系，这些内容在 Spring 中是一个十分重要的技术点，Spring 后续的一些内容都强依赖于这一门技术（自定义标签解析），常见的有 SpringMVC、Spring 事务、SpringAOP 等，它们都是基于此作为一个拓展实现了更强大的功能。